RUN TIME SUPPORT FOR THE TUTOR LANGUAGE
ON A SMALL COMPUTER SYSTEM

by

Douglas Warren Jones

B.S., Carnegie-Mellon University, 1973

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1976

Urbana, Illinois

ACKNOWLEDGEMENTS

Professor Thomas T. Chen deserves special credit for providing the initial impetus behind this project and for his continued support and encouragement without which this project would never have been completed. I would also like to thank A. B. Baskin and Thomas Szolyga for the part they played in the implementation phase of this project.

TABLE OF CONTENTS

1. INTRODUCTION

Possibly the largest single collection of computer based instructional material in the world is written in the TUTOR programming language [11,19] fully supported only by the PLATO system [1] developed by the Computer-based Education Reasearch Laboratories (CERL) of the University of Illinois. The PLATO system and the associated TUTOR language have evolved together over the last 10 years, and combine a number of interesting and powerful features that are not widely available in other systems. Though intended primarily for the structuring of instructional material for computer presentation, the TUTOR language has been used for information system development [24], simulation of physician-patient interaction [13,14], and a host of other applications. Subsets of TUTOR have previously been implemented by others, most notably in the MULTITUTOR- HYPERTEXT system at Northwestern University [17,18].

This paper deals with the design and implementation of a run time support system for TUTOR in a small machine environment. The entire system is described elsewhere [2,3,7,8,22].

1.1. APPLICATIONS FOR A SMALL TUTOR SYSTEM

The PLATO system uses a large dual processor machine with two million words of 60 bit Extended Core Storage and a complex multiplexor system to drive to up to 1000 terminals. Startup costs for such a system are quite high, as are communications costs if the user community is geographically dispersed. In addition, the size of the user community on such a system can raise serious managerial and security problems. Because of this, there may be many applications for an alternative in the form of an inexpensive system of 20 to 30 terminals with the capability to tie into networks with similar systems.

The feasibility of such a system has previously been investigated with favorable results [6], although TUTOR has since evolved, invalidating some of the design decisions of this earlier work.

1.2. THE POSSIBILITY OF OTHER LANGUAGES

Currently the PLATO system supports only the TUTOR language for the development of interactive programs, and there is no commitment to support other languages for any but background or batch processing. Alternatives to TUTOR need to be explored, particularly in the area of control structures. At least one such language has been developed and implemented on PLATO by translation to TUTOR [10]. It is hoped that the interpretation technique described in this paper will considerably simplify such experiments.

1.3. PROJECT GOALS

The principal goal addressed here is the demonstration that TUTOR can be supported for a single user on a small to medium scale machine in a manner compatible with extension to multiple users. This involves an analysis of the different features of TUTOR showing either that they are compatible with such an environment, or if not, that the cost of incompatibility is reasonable. An interpreter and compiler have been implemented based on the ideas presented here, and a limited number of TUTOR programs have been transferred from PLATO.

Because of the speed at which TUTOR on PLATO is evolving, an important subgoal is to design the implementation with maximum flexibility so that changes can be incorporated with a minimal amount of new work.

The notation and terminology used here are largely adopted from the PLATO project; for instance, command names are enclosed in dashes (-dot-), and names of special characters are capitalized (FONT); however, TUTOR lessons are referred to as programs and students as users in order to emphasize that TUTOR is a general puropse language and is in no way limited to instructional applications.

The following chapter discusses the general nature of a TUTOR program and the different constraints on how it may be represented and interpreted on a small machine. The remaining chapters deal similarly with the special areas where TUTOR is most different from conventional programming languages, specifically the areas of input analysis or judging, program segmenting or units, and input/output. This should not be considered as a complete description of an implementation of TUTOR, as many conventional facilities of the language are only briefly covered, if at all.

2. BASIC DESIGN CONSTRAINTS

The constraints governing the way TUTOR is implemented fall into three large classes: Those introduced by the target hardware, those introduced by the TUTOR language, and those introduced by the implementor for other reasons. In the last class are such things as the desire to eventually support other programming languages, and the desire to support user terminals other than the present PLATO IV terminal.

One of the greatest barriers to implementing TUTOR in any new environment is that TUTOR is defined in terms of the hardware on which it is implemented at CERL, with little or no attention to the possibility of machine independence. For this reason, part of the problem of implementing TUTOR on a small machine lies in deciding which aspects of the original machine require emulation, and which are nonessential to the definition of TUTOR.

2.1. DOMAINS OVER WHICH TUTOR EXECUTION IS DEFINED

It is convenient to discuss the design constraints imposed by the TUTOR language in terms of the resources that must be brought together for a program to be run. In this context, the current PLATO implementation as well as the available implementation alternatives can be meaningfully discussed, and related to the problems of a small to medium scale machine environment.

2.1.1. PROGRAM SPACE

The program to be executed may be shared by many different users, all executing different parts of it in a time-shared manner. On PLATO, the programs are stored in Extended Core Storage (ECS), and when a user wishes to execute a portion of a program, logical segments of that program are brought into central memory for execution. These segments, called 'units' on PLATO, provide a virtual memory mechanism that allows full use of two million words of ECS by a central processor with a relatively small memory of its own.

An alternative way of segmenting TUTOR programs has been proposed that involves link editing of unit clusters [6]. This scheme relies on the assumption that all units called from any unit are known at compile time; thus, it should be possible to build clusters of units such that inter-cluster linkage is minimized. This scheme may be attractive in environments where large segment sizes are acceptable, but constructs such as -imain- [section 4.1.1] may invalidate the above assumption.

The TUTOR units on PLATO are compiled into two components: The instruction part of a unit is a list of 60 bit entries containing encodings of the command names and parameters or pointers to other information. The second part of each unit holds the additional information required by the commands in the first part. The consecutive list of commands in the first part of the unit simplifies the linear search for certain commands used by the judging mechanism [appendix A].

Because many TUTOR commands invoke complex system functions at run time, or amount to complex service subroutine calls, TUTOR programs will always consist largely of the computation of parameters to system subroutines. Directly compiling TUTOR code to machine code could consume unreasonable amounts of time and space because of the expense of conventional calling sequences. On the other hand, given the paged virtual memory capabilities that are becoming common on todays medium scale machines, some de-emphasization of the TUTOR unit would be desirable.

If TUTOR is to be supported in a demand paging environment, it would be useful to merge the two components of each unit in order to improve program locality. The MULTITUTOR implementation [17,18] has achieved this merger by including in the 60 bit code for each command a pointer to the next command, allowing the continued interpretation of judging semantics in terms of a scan of the (linked) command list. In fact, the run time scan can be fully eliminated from the interpretation process [chapter 3].

2.1.2. STUDENT OR USER VARIABLES

Each user of a TUTOR program has a unique array of 150 words that may be used for user dependent computation. These user variables retain their values when a user changes programs, and they are saved when a user is not on the system; thus, they may be used for inter-program parameters and user related historic data so long as all programs involved agree on the use of each location. There is no automatic storage allocation mechanism nor is there any dynamic storage management system for user data, though it is hoped that some form of both will be introduced in the future.

One of the basic assumptions inherent in the design of TUTOR is that all data types will fit in one machine word. As a consequence, TUTOR does not need or have any data type enforcement or checking; instead, the data type must be provided with each memory reference. For example, 'n1' refers to the first available location as an integer, 'v1' refers to it as a floating point number, and alphanumeric data must be stored packed in integers. This assumption is acceptable in the CERL implementation where the machine word size is 60 bits, but on the majority of available machines in the small to medium size range the common 16 or 32 bit word size could pose a problem.

The word length must be comparable to 60 bits in order to allow simple program interchange with PLATO; on small to medium machines representing a TUTOR word as 64 bits is reasonable as has been previously proposed [6]. Many available machines in the target class have standard hardware available to do 64 bit floating point arithmetic, but the integer representation poses some problems as 32 bit integer arithmetic is the largest commonly available on such machines.

Representing integers by the 32 most signifigant bits has been proposed elsewhere [6] but could introduce many problems in program conversion because of the amount of explicit bit manipulation that TUTOR encourages. For the same reason, the sign bit must be the most signifigant bit, so integers may be represented either by the least signifigant 32 bits of a word with sign extension, or by the full 64 bits using software simulation. Both of these alternatives would add greatly to the size of programs compiled into machine code and signifigantly increase execution times in any implementation. Thus again, some form of interpretation seems preferable.

2.1.3. COMMON OR COMMUNICATION VARIABLES

An important capability of multiple user on-line computer systems that is frequently not well supported is inter-user or inter-program communication. The UNIX system [16] provides a special type of logical file called a pipe which may be read from by one task after being written on by another, but most timesharing systems allow user tasks to communicate only by shared disk files or other slow and inelegant means.

TUTOR provides common variables as a solution to this problem. If a program accesses common variables, then all users of that program will share the same copy of them (as opposed to user variables which are unique to each user). In addition, common variables are also preserved when no users are attached so they may be used by the program to record historic information and constant data as needed. Critical section management for access to common data is provided by the -reserve- and -release- commands which operate on semaphores associated with each named common block.

On PLATO, common variables are stored in ECS; to use them a mapping must be established between the ECS copy and a 1500 word central memory buffer which the program can actually access. This mapping is implicit if the size of the common is less than 1500 60 bit words, but must be explicitly stated for larger commons, and may be changed at run time. Given paged virtual memory hardware instead of word addressable ECS, these arbitrary mappings may be quite difficult to support. Fortunately, only about one fourth of all existing TUTOR programs use common variables at all, and of these only a fraction use more than the 1500 word limit, so the translation costs imposed by incompatibility in the support of large common regions should not be objectionable.

2.1.4. STORAGE OR EXTENDED USER VARIABLES

The 150 word limit on the number of user variables poses a severe restriction on the utility of TUTOR for solving many classes of problems. Because the 150 user variables retain their values between sessions for any given user, it was not considered practical to expand this limit on PLATO; instead a new data space called "storage" was introduced. Storage is statically allocated for each program, but the system must allocate it dynamically to handle transfers of control from one program to another. Unlike common or user variables, storage is always deallocated when the user leaves the system.

Storage is allocated on PLATO in ECS, and the same mapping mechanism is used to gain access to it as is used for common variables. On PLATO, all disk input output must take place to ECS (storage or common). Because of this, the most frequent use of storage on PLATO is for disk buffering; incompatibilities in the support of this will probably be tolerable.

2.1.5. VARIABLE SEGMENTING

As was previously mentioned, the TUTOR language relies on the assumption that any machine word may hold any data type. Even on a 60 bit machine, this forces great inefficiency in the storage of such things as character strings or bit arrays. Early in the evolution of TUTOR a partial solution to this was provided by special commands that manipulate character strings packed 10 six bit codes per 60 bit word. This partial solution proved inadequate, and a general solution was provided in the ability to segment blocks of physical words into arrays of smaller logical words.

It is the intent of TUTOR that a reference to an element of a segmented array should be equivalent to a reference to a machine word in all contexts. Because of the basic design of the PLATO implementation, this intent is not yet fully supported, but given the foreknowledge that it should be, any new implementation should do so from the beginning.

2.2. PROGRAM REPRESENTATION

Because access to segmented variables and integer arithmetic must both be subroutines on a 32 or 16 bit machine, and because almost all of the TUTOR commands are subroutine like, with many parameters and complex side effects, the time overhead of intermediate code interpretation as compared to direct machine code execution should not be too excessive.

Execution by interpreting the original source would be prohibitive because of the cost of lexical analysis and run time symbol table maintenance (TUTOR makes no restrictions analogous to those of BASIC on variable print names). Given that some kind of source compression is required, compilation of expressions to postfix form, and interpretation by means of a virtual stack machine provide an obvious choice. The stack in such a scheme can also be used for action routine argument passing and for user procedure linkage, as well as for temporary storage needed by any of the action routines; this greatly simplifies the problem of storage allocation for the interpreter.

On the target class of machines, an eight bit instruction syllable is reasonable, with 16 bit branch addresses, and 16 bit address fields when needed. Because of the existence of user, common, storage, and segmented variables, address fields must also contain an indication of which data space or subspace is to be addressed as well as the word size and characteristics of that space. The use of 16 bit branch addresses limits the program size to 65536 bytes, comparable to the 8000 60 bit word limit that used to exist on PLATO.

Most of the information about each of the many address spaces and segmented subspaces can be stored in a table of space characteristics. With this scheme, each memory address must contain an index (8 bits) into this segment table as well as the 16 bit offset into the desired addressing space. It should not be difficult to extend this scheme to multi-dimensional arrays as has recently been done on PLATO. This scheme should be contrasted with the hardware data descriptor schemes of some machines [15].

The assignment of values to entries in the segment table would be the responsibility of the compiler if the PLATO definition of TUTOR is retained, however it is a simple extension to allow run time redefinition of table entries. Thus this memory addressing mechanism can easily be extended to support dynamic allocation of temporary variables on the stack.

Interpretation of an intermediate code similar to that outlined above can be quite fast. The code accomplishing the action specified by any particular instruction can be prefixed to code that fetches the next byte from the instruction stream, increments the program counter, and branches through a jump table indexed with the byte fetched to the next action routine [4].

As mentioned earlier, all parameters to TUTOR commands can be passed on the interpreter stack. Given an alternative to the run time scan of the command list that some commands require, the various commands may then be represented in the intermediate code by a prefix byte followed by an 8 bit command code. This allows the definition of 256 commands per available prefix. Parital specifications for such an interpreter are presented in appendix B; examples of commands compiled into this code are presented in appendix C.

2.3. DATA REPRESENTATION

The alternative representations for integers have already been discussed [section 2.1.2], and as was mentioned, the reasonable choices are interpretive simulation of full 64 bit integers or use of available hardware 32 bit integers with sign extension to 64 bits provided interpretively. The latter alternative is probably preferable because there are few problems requiring the use of 64 bit integers and the simulation of multiplication and division to the full precision can be quite slow. The PLATO hardware only supports integer operations for the 48 least signifigant bits of the 60 bit word, and it is not likely that the the difference between 48 and 32 bits will cause many problems.

Because of the practice of dealing with character and bit strings packed into words as integers, the integer comparison and bit manipulation operations must always work over the entire word size.

The largest remaining data representation problems occur with character data. Many programs on PLATO make explicit use of the packing of ten 6 bit character codes per machine word, and until recently, it was common practice to make explicit use of the specific 6 bit codes for various characters. For the latter reason, it has been suggested [6] that the PLATO 6 bit codes be preserved and packed into machine bytes with high order bits unused. Since that suggestion was made, use of quoted charater literals has been strongly encouraged on PLATO, so any character set will probably be acceptable so long as it is extensive enough.

Though many simple CAI programs may survive the change from ten 6 bit to eight 8 bit characters per word, this change may be responsible for the greatest conversion costs for many of the more interesting programs on PLATO. On the other hand, preservation of the current PLATO character set on machines with natural addressing to 8 bit bytes would introduce what may be an unacceptable execution overhead on smaller machines, as well as making PLATO software incompatable with other software already existing on the host machine.

One problem that may invalidate the assumptions about the use of the natural addressing capabilities of some machines is that TUTOR programs frequently make explicit use of the left to right storage of bytes and segmented variables in a machine word. This will require interpretive intervention if TUTOR is to be supported on any of the large family of machines that store bytes right to left in memory.

3. JUDGING OR INPUT MANIPULATION

The response judging capabilities in TUTOR serve two separate purposes. First, they provide the user with well structured access to a set of powerful primitives for requesting terminal input, rejecting that input, or requesting that the input be modified. Secondly, they provide access to a powerful set of character string and numeric expression analysis facilities for the evaluation of the terminal input.

The response judging subset of TUTOR was originally conceived as a mechanism to be used for computer quiz administration or similar applications, where the computer would present a question, and decide which of the responses anticipated by the program author most closely resembled the answer given by the user. After having decided which response to the question was given, the program had the alternative of accepting the response or rejecting it. If the response was rejected, then the program could provide appropriate feedback to the user, after which the computer would automatically request that the user modify the response before re-submiting it to the program.

3.1. THE RESPONSE JUDGING MECHANISM

Once the input has been accepted from the terminal, there are a number of ways that it can be analyzed. TUTOR provides analysis routines that will compare the input with a character string for an exact match, evaluate the input as a simple numeric quantity, evaluate it as an expression, parse it into words, or both parse and compare it with a predefined vocabulary with allowance for simple spelling errors.

All of these capabilities exist on other systems, though they are not commonly all made available in one bundle. The required analysis methods are disjoint, and can be individually implemented by conventional means. The actual input analysis mechanism for use on a small machine is a separate problem [26], and the mechanism used on PLATO has been described in [23]. The input output requirements of TUTOR judging are considerably more complex than conventional unit record approaches [chapter 6], but the complexity is not outside the range of adaptability of some vendor supplied systems [2].

The remainder of this chapter deals with alternatives for the implementation of the program control structures defined by the TUTOR judging mechanisms.

3.2. EFFECTS OF JUDGING ON PROGRAM STRUCTURE

The control structures that TUTOR provides for response judging are described by the PLATO project in terms of a number of program execution states, where each command may have a different meaning in each state [11,19]. These states are summarized in appendix A. The description of the judging control structure in terms of execution states, markers, and searches for various commands obscures the underlying control structures to the extent that most begining and many experienced TUTOR programmers never fully understand it.

The KAIL selector [9] was developed as an alternative syntactic representation for the TUTOR judging control structure. The KAIL selector actually represents only part of the capabilities, with the block exit capabilities of the post -specs- states not supported; however, it represents an important step towards the interpretation of the judging mechanism in terms of traditional control structures.

3.2.1. THE TUTOR JUDGING BLOCK

From the description in appendix A, it can be deduced that a region of TUTOR code involving judging always begins with an -arrow-, and is ended by a new -arrow-, -endarrow-, or -unit- command. Furthermore, termination by an -arrow- or -unit- command is equivalent to termination by an -endarrow- immediatly preceding the -arrow- or -unit-. Since a compiler can always generate the appropriate code for an implicit -endarrow- before -unit- and -arrow- commands if there was a previous -arrow- command, it is safe to consider judging only in terms of -arrow- -endarrow- pairs or judging blocks (excluding for the moment the problem of subroutines).

Furthermore, the -arrow- is merely a prefix to a loop that begins with the first judging command after the -arrow- and ends at the -endarrow-, with termination occuring when judging state ends with an "ok" judgment. Given the above observations, an -endarrow- can be compiled as a conditional branch to the first judging command after the most recent -arrow-, where the branch will be followed if the last judgment was "no" [appendix D.1].

When subroutines and local -branch- commands are included in this description, it becomes much more complicated. There are only rare uses made on PLATO of the more pathological interactions between these language components, and in fact the addition of a few simple rules to the TUTOR language allows this simple description to be retained: First the -endarrow- for each -arrow- must be required to be in the same unit or program segment as the -arrow-. Second, all simple branches into and out of the range of a judging block must be forbidden. These rules correspond closely to the traditional structured programming rules where unrestrained branching is limited and control structures may be nested but not arbitrarily overlaped.

3.2.2. CONDITIONS FOR RESPONSE EVALUATION

The judging commands in TUTOR fall into two classes: Those that are considered judging commands because they require the input to be ready before executing, and must therefore be executed in judging state, and those that actually perform some judgment, ending the judging state with an "ok" or "no". The judging commands may be interspersed with regular commands but in judging state only the judging commands are executed; this suggests linking each judging command to the next in a sequence by branches that are conditional on judging state. This may be considered to be merely the removal of the regular commands from the linked command list implementation of MULTITUTOR [17,18].

When an -endarrow- is encountered in judging state, the judgment is "no", so the interpretation of all -endarrow-s as being preceded by an implicit -no- command is safe (the -no- command is defined as always ending judging with a "no" judgment).

3.2.3. EXECUTION OF BLOCKS CONDITIONAL ON JUDGMENT

With the addition of the branches from endarrow to the first judging command, and from each judging command to the next, it is rather easy to follow through to the end: Any judging command after the first one in a sequence that is preceded by regular commands must be preceded by a conditional branch back to the location after any previous -specs- command, or to the -endarrow-. There is an exception to this rule when the previous judging command was -specs-, in which case the branch is always to the -endarrow-. These branches are conditional on the state being postjudging regular, and they accomplish the eventual transfer of control to the -endarrow- where the decision is made whether or not to loop.

The majority of TUTOR judging blocks are actually quite well structured; typically consisting of a loop terminating on an "ok" judgment containing an input request and a series of blocks of code executed conditionally on the results of the various judging commands. Appendices D.1 and D.2 illustrate the reduction of a tutor judging sequence to a flowchart, and from there to code in a well structured form; note that the inclusion of post -specs- regular commands requires the use of Zahn's event driven block exit [5,12,27].

3.2.4. JUDGING AND SUBROUTINE CALLS

One of the difficulties of compiling TUTOR that will be discussed in more detail in chapter 4 is that it is not possible to specify that a unit is only to be used as a subroutine, as an extension to another unit, or as a main program; in fact, it is perfectly possible to use a single unit as all three though this may be considered bad practice.

For the purposes of this discussion, it is sufficient to know that there are two kinds of subroutine calls in TUTOR: The first of these, the -do- command, is a regular command, and is the most commonly used. The second is the -join- command; it is a somewhat confusing command because it is defined as being executed in all of the TUTOR execution states.

If a unit is entered while in judging state (via -join-), the search for a judging command must continue. This may be accomplished by following the unit entry with a branch on judging state to the first judging command (if any) in the unit, or to the end of the unit if none.

When judging is ended within a unit that has been attached as a subroutine by -join-, control must somehow return to the appropriate place. In that the compiler does not know how the unit is to be executed, it will be attempting to create a branch to the previous -specs- command, or some later -endarrow-. One alternative is to define -endarrow- (even if implicit) differently if it is encountered before an -arrow- in a unit or if there is no -arrow- in the unit; the compiler may then link all regular blocks controlled by judging commands to a possibly virtual -endarrow-, with the appropriate state marker being set before the return.

The execution of -join- while in search state, that is while searching for the (possibly implicit) -endarrow- after an "ok" judgment, is not a widely used feature of TUTOR, and it was not considered necessary to consider it here. There has even been even recent discussion of eliminating search state on PLATO and using a compiled branch scheme where each -arrow- would have a pointer to its corresponding -endarrow- or equivalent.

The problem is then to differentiate all the different execution states that may exist on return from a unit attached by -join- with conditional branches to the appropriate places: One branch should be to the next judging command if still in judging state, the other to the next -endarrow- if in post 'pseudo' -endarrow- state, and finally no branch if in any of the regular states.

3.3. INTERPRETER MECHANISMS AND THE COMPILER

The adoption of the division of control structure semantics and interpretation as outlined above provides the necessary flexibility needed to support alternate source languages that share the same execution package. This places considerably more burden on the compiler than the PLATO scheme, but considerably broadens the range of alternative execution schemes.

The various states that have been mentioned so far for judging purposes may be reduced to a 4 bit 'nibble' appropriate for testing with simple conditional branches. The required bits are listed in appendix B.1.1 and the test and branch commands in B.3.1. Appendix D outlines the compile time expansion process by which various TUTOR commands are converted into assortments of conditional branches on status, and calls to unconditional utility routines.

4. UNITS OR UNIVERSAL PROGRAM SEGMENTS

As well as being the basis of the virtual memory scheme on PLATO, TUTOR units may be used as subroutines by the -do- or -join- commands. Units may also be attached as logical continuations of other units by the -goto- command, or they may be used as new main program segments by the jump commands (not only -jump-, but also 'key arming' commands).

4.1. THE DIFFERENT USES OF UNITS

4.1.1. AS MAIN PROGRAM SEGMENTS

A unit to which control is transfered from another unit by the -jump- command, by one of the interrupt facilities, by sequential entry from the previous unit, or as the first named unit of a new program is executed as a main unit. On entry to a main unit, the subroutine linkage stack is cleared, the screen is erased, and if the feature is armed, the unit named as an imain unit is called as an initializing routine by a call equivalent to the -do- call.

The most obvious solution to providing the special effects on entry to a main unit is to have the interpreter code for branching to a main unit cause these side effects. This is the approach used on PLATO, and has the undesirable result that the interpreter code used for entry to a unit would not be usable for a language where the type of a program segment is bound not by how it is reached but by how it is defined.

An alternative method for compilation of main units exists in which each unit begins with a special command responsible for all of the side effects of main unit entry except the -imain- call (which must take place after the resolution of parameters). All of the commands that enter the unit as a main unit take the address of the side effect command, and all others, such as subroutine entry and -goto- take the address following it. This provides for a complete separation of the side effects from the control structuring commands at the interpreter level at the expense of one or two extra bytes per unit.

When control reaches the end of a main unit, (including passing the implicit -endarrow- of the previous chapter), execution holds until the user directs it to continue (by pressing the NEXT key). When ready, control transfers either to the next unit in sequence or to the unit specified in the last -next- command if one was encountered since the begining of the last main unit; in either case, the new unit is entered as a main unit. These functions can easily be accomplished by a simple command at the end of each unit.

4.1.2. AS SUBROUTINES

TUTOR provides two kinds of subroutine calls which differ only in their relationship to TUTOR judging [section 3.2.4]. Clearly, an essential requirement for a subroutine is that it return when completed. This may be accomplished by having the end unit command mentioned in the previous section execute a subroutine return if the current unit was entered as a subroutine; but to allow future experiments with alternatives to units, it is simple to explicitly compile a conditional subroutine return just before the implicit end unit command.

4.1.3. AS INTERRUPT PROCESSING ROUTINES

In an interactive environment, it is important to allow the user some way of easily altering the flow of program execution. It is easy to think of this ability in terms of allowing the user to interrupt one process and initiate another, possibly with a return to the first when the second is finished. TUTOR allows a special group of keys on the keyboard to be armed with unit names to provide a good approximation of this; whenever input is requested from the terminal and one of these armed keys is struck, a -jump- like transfer of control takes place to the associated unit.

There are two types of interrupt like branches allowed in TUTOR. The first is simply a user initiated -jump-; this is associated with keys such as BACK and STOP, and provides no real implementation problems, given that a key arming mechanism can be made to work, and that a main unit entry mechanism exists.

The second type of interrupt like -jump- allows a return in addition; this type, most frequently associated with the HELP and TERM keys on the keyboard, poses the major implementation problems. The ideal behavior of a HELP type branch would be for the units attached by it to be executed as subroutines, with the entire execution status being restored on return to the point where the interrupt occured, and with allowance for nesting of HELP interrupts. Unfortunately the status that would need to be saved includes the entire contents of the terminal display and input line editing buffers at the time of the interrupt. On the PLATO system with its 512 by 512 dot addressable screen, this would entail the storage of over 256000 bits of information per level of nesting.

The solution adopted in TUTOR, which is probably a reasonable one, is to return to a designated point before the point where the interrupt occured, allowing the contents of the screen to be regenerated, and allowing the reinitiation of any input transaction that had been in progress when the interrupt occured. The restart point in TUTOR is the start of the most recent main unit entered; this is called the base unit, and when a HELP type branch occurs the BACK key is armed in the new main unit to jump back to the base unit.

TUTOR does not allow true nesting of HELP sequences, rather it allows the arming of the HELP key within such sequences to branch to new -help- sequences without changing the base unit pointer. Though the base pointer may be manually cleared or set, it is normally cleared by following the specially armed BACK branch.

Experiments are being made on PLATO in the support of other interrupt like branching abilities, for instance the -helpop- key arming mechanism, but these are too new to cover here.

The PLATO mechanism for supporting TUTOR -help- type branches and returns involves the use of two pointers, one to the current main unit entry point, and one to the current base unit. This scheme is not only adequate for a small machine implementation, but easily lends itself to experimentation with nesting of interrupts by saving and restoring the base unit pointer during procedure linkage.

4.1.4. AS CONTINUATIONS OF OTHER UNITS

The -goto- command in TUTOR simply transfers to the named unit without any change to any of the program status, including memory of the location of the previous -arrow- if there was one. The resultant interaction of -goto- with the judging mechanism includes such things as returns to the last -arrow- and state changes; because of these pathological cases, -goto- poses problems to a new TUTOR implementation.

The -goto- command is incompatible with the compilation of branches proposed in chapter 3 as a solution to implementing TUTOR judging sequencing. Fortunately, the use of this aspect of the TUTOR language is not encouraged, as it is difficult to explain the unexpected results. Because of this, and in keeping with the rules mentioned in section 3.2.1, the use of -goto- from within judging sequences may be forbidden (in almost all cases, -do- can be used to achieve the same function much more clearly anyway).

4.2. PARAMETERS TO UNITS

Somewhat late in the history of TUTOR, in fact after much of the initial work on this project had been completed, the ability to pass parameters with any direct control transfer to a unit was introduced. Previous programming practice in TUTOR had generally included the allocation of fixed groups of user variables to each unit that needed parameters, and then assigning values to each parameter before each call. When the passing mechanism was introduced, it was made completely compatible with that approach; that is, parameters in TUTOR involve no temporary storage or local variables.

Given a stack based interpreter, the obvious implementation is to push all of the parameters onto the stack before a call and pop them into the appropriate locations after entry into the new unit. There are two complications to this scheme, one involving main unit entry which must pop the subroutine linkage stack, the other involving the fact that with any call, a subset of the parameter list may be passed, with only the corresponding locations being changed in the called routine, the other parameter locations retaining their previous contents.

Because parameters may be passed not only with -do- -join- and -goto- but also with -jump- and with trivial extensions -nextnow-, the first problem occurs. If parameters are passed on the stack then the commands that start a new main unit must copy any parameter block down the stack when they undo the procedure linkage. The alternative is the setting aside of a temporary special data area only for parameter passing, an unpleasant alternative, though the one used on PLATO.

The second problem may be solved by passing as an additional parameter a bit vector indicating which parameters are present. This bit vector must also give the types of each parameter (integer or floating) because TUTOR performs automatic conversion of parameter types, and the requirements of fast compilation preclude a global first pass to work out the parameter types expected with each unit. Given a 64 bit wide stack and 2 bits for each parameter, a total of 32 parameters may be passed, an acceptable limit.

This scheme is compatible with the eventual implementation of temporary local data allocation for units using the single stack. The same calling sequence could be used, and the receiving sequence would set up some special segmented variable [section 2.1.5] to allow addressing of the parameter list and new local storage instead of copying the parameters into fixed locations.

4.3. INTERPRETER MECHANISMS AND THE COMPILER

4.3.1. PROGRAM STATUS INFORMATION

The unit sequencing mechanisms outlined above require that the program status contain at least a main and base unit pointer, and four more status bits that may be tested in a manner similar to those used for judging. One status bit would be required for the parameter passing mechanism indicating that parameters exist on the stack and must be stored. The second and third status bits indicate respectively that the current unit is being executed as an attached unit by -do- or -join-. The fourth bit is needed to differentiate -help- from -helpop- type unit attachment when the base pointer is non zero. A count of the number of parameters currently on the stack must also be maintained so that the main unit entry routine can copy them down the stack properly.

In addition to this, the program status must contain space for a return address for -do- and -join- and some mechanism for nesting these calls. The minimum information that must be saved for nested calls consists of the return address and do/join status bits. Saving and restoring the other linkage bits would not cause conflicts, but the judging status bits must not be saved and restored, as they are used to return results of judging operations in joined units. Saving and restoring the base pointer would allow future experiments with -help- type interrupt nesting while not conflicting with the current TUTOR definition.

The use of a single stack for the return linkages as well as intermediate results, parameters, loop control blocks, and future procedure local variables requires that the program status also contain a special linkage pointer to the previous program status block on the stack.

4.3.2. STANDARD CALLING AND RECEIVING SEQUENCES

The actual call generated by a -do- or -join- command must consist first of reserving space on the stack for linkage, then placing the optional parameters on the stack, followed by the bit vector giving the types and positions of the parameters. After this is the actual subroutine call or branch, which must have as in line parameters both the address of the unit to be executed and the number of arguments so that the linkage can be correctly placed in the stack.

If a unit expects parameters, it must begin with a branch conditional on the absence of parameters around the parameter recieving code. The parameter recieving code consists first of the computation of all of the parameter addresses followed by the execution of the parameter resolution command which stores values in addresses with optional floating or fixing (as indicated by the parameter presence bit vector and the type information supplied with the addresses). The process of parameter resolution pops the values and addresses, which is why the linkage must be before the parameters on the stack. Only after all of the parameters have been resolved can the imain unit be called; this should be the responsibility of a special command that performs the appropriate linkage when executed in 'main unit state'. At the end of each unit, after the (possibly virtual) -endarrow- (if any), a return must be inserted conditional on -do- or -join-.

5. TUTOR CONDITIONAL AND LOOPING COMMANDS

All but the most trivial of programs must make choices and repeat various sections of code. TUTOR provides a number of mechanisms for this ranging from simple conditional branch commands to conditional and looping variants of other commands.

5.1. CONDITIONAL COMMANDS

A large number of commands in TUTOR have conditional variants where the command is executed with respect to one of a group of parameter lists dependent on an integer selector value. Commands with conditional variants include flow of control commands such as -branch-, -goto-, and -do-; key arming commands such as -help-; display generation commands such as -writec-; and computational commands such as -calcc- and -calcs-. There may be any number of choices in these conditional forms, the first one being selected when the selector is negative, the second on zero, and so forth.

These conditional variants spread to a large number of TUTOR commands at a time when they were the only control structures embedable in the body of a unit besides those associated with judging and subroutine calls. On PLATO, conditional and simple commands are not compiled into variants of one intermediate code command, but into different commands, where the format of a conditional command may have no relation to the format of the non conditional one with the same name in the source text.

It would greatly simplify interpretation if the conditional aspects of a command were compiled out so that from the point of view of the interpreter all commands would have a fixed format; this requires that there be a compiler generated directive to select amoung a list of commands and parameter lists, analogous to the way -case- statements are handled in many languages with a table lookup branch instruction. Appendix C.3 contains an example of the expansion of a typical TUTOR conditional in terms of the instruction set of appendix B.

5.2. LOOPING COMMANDS

The looping variants of the -do- and -join- subroutine calls were for a long time the only way that an arbitrary block of code could be repeated except by the use of conditional -goto- or -jump- commands. These looping variants take an index variable, initial value, final value, and step size, and may either increment or decrement the index variable. If the looping and conditional capabilities are used at the same time, then the loop includes the conditional list of units within it.

As with the conditional commands, PLATO implements these looping variants as distinct intermediate code commands. Again, to simplify interpretation, the alternative of compiling the loop into a test and branch instruction, a simple command, and an increment and branch back instruction is preferable. The TUTOR loops all are defined as looping zero or more times, so there must be a pre-loop check instruction, as well as a post loop increment and branch.

On PLATO, the increment value and bounds may be arbitrary expressions and may be changed during the execution of the loop. Execution of the loop control commands may be considerably simplified if these values are fixed once the loop is entered, so that they are only calculated once, and saved on the stack during loop execution (allowing nesting of loops). This incompatibility should not introduce too many problems as most applications make no use of the variable increment and bounds, and those that do may be easily reprogramed with the -branch- instruction. The instruction set of appendix B.3.1 lists the precheck and postindex instructions, and an example of the expansion of a TUTOR looping instruction is given in appendix C.4.

5.3. LOCAL CONTROL STRUCTURES

Commands for building control structures within the confines of a single unit were introduced somewhat late in the history of TUTOR. These commands are now generally useable though they were originally limited in scope to single extended -calc- statements. Structured flow of control commands are still being discussed for introduction into TUTOR, but these should easily be compiled into the simple commands listed here.

The -doto- command provides the looping ability of -do- and -join-, repeating not a single command but a group of commands. These loops may be nested and may contain -branch- commands to exit the loop on exceptional conditions.

The -branch- command and its conditional variant allow transfer of control to an arbitrary label in a unit. If the -doto- command is implemented with the use of the loop control block and commands introduced in section 5.2, then special provisions must be made for branches interacting with loops.

Branches into the range of a loop can be prohibited with few ill effects on program transferability. Branches out of a loop must remove the loop control block from the stack before exiting, thus requiring a stack modify instruction. Because the compiler can not easily determine when generating code for a branch whether the branch exits a loop or not, it is necessary that branches pop all loop control blocks from the stack before branching, and all labels inside loops must recover them. Appendix C.4 contains an example of this.

5.4. INTERPRETER MECHANISMS AND THE COMPILER

The above outlined approach allows the separation of the control structures from the other components of the language at compile time, again providing for the implementation of alternate control structures in other languages using the same interpreter. This is quite important because of current discussion on PLATO about the implementation of an entire new set of control structures for TUTOR similar to those of PASCAL [25]. The interpreter is also greatly simplified by the elimination of redundant mechanisms inherent in this compile time separation of control structures and commands.

6. TERMINAL INPUT AND OUTPUT

One of the most unique aspects of the PLATO environment is that all of the terminals are graphics oriented with excellent peripheral support for user interaction. The TUTOR language evolved in this environment, and because of this the input/output features of the language differ in many ways from those devised for batch card or teletype oriented systems.

The PLATO IV terminal supports no natural unit record such as the line or page for output; because of this, TUTOR must use stream output formatting commands. A wide variety of these are provided, including a set of graphics utilities that provide for rotation and scale modification of line drawn figures and text.

TUTOR provides two classes of terminal input management facilities, both of which are somewhat record oriented. For the simple input operations, a fixed record size of one or more characters or external input codes is supported, with no features outside the capabilities of conventional unit record processing. Commands that manipulate such simple input include -pause- and -collect- as well as certan control structure side effects such as those associated with the -unit- command [section 4.1.1].

The most complicated input/output capabilities of TUTOR are associated with the input judging mechanism. Here input is initiated by the first judging command after the -arrow- [section 3.1, appendix A], with a number of state variables changing exactly how that input is collected. The input record is a line or block of text; however, many non conventional input/output operations may be performed with respect to these records.

6.1. THE PLATO IV TERMINAL

All TUTOR terminal input/output is defined in terms of the PLATO IV terminal [20], this terminal has a 512 by 512 dot addressable screen with an 8 by 16 character matrix giving 32 lines of 64 characters on the screen. The terminal has in addition to a hard wired character set a programmable one, dot and vector generation, and the ability to selectivly write and erase individual dots, vectors, or characters.

The MULTITUTOR system [18] supports a number of other terminal types, and the experience gained there indicates that conversion of programs to use 24 line by 80 character alphanumeric CRT terminals is not an unreasonable task. The most important terminal characteristic required for the support of many TUTOR programs appears to be a character addressable terminal with the ability to selectively erase or modify the display contents on a character by character basis.

6.2. TUTOR SUPPORT OF THE TERMINAL

6.2.1. OUTPUT CAPABILITY

TUTOR's output facilities may be divided into three categories: Those of text output, graphics output, and special device output. Text is normally sent straight to the terminal with only minimal system intervention except to handle overlength lines and to establish a left margin. Before sending text to the screen, the program must specify where on the screen the text is to be shown; the system then maintains information so that the program may always determine the location of the last character displayed.

TUTOR allows all display coordinates to be specified either as character and line numbers (coarse grid) or as dot coordinates (fine grid). To simplify interpretation, a special interpreter instruction to convert coarse to fine grid coordinates is included [appendix B.3.7] so that all instructions may be defined only in terms of fine grid. The compiler is then responsible for inserting this conversion instruction when the source program uses coarse grid.

Text output from TUTOR may optionally be converted to line drawn output by the use of either system or user defined linesets. The line drawn text is processed through the graphics output package, thus allowing the text to be rotated and its size changed relative to an arbitrary origin.

The graphics output capabilities of TUTOR range from simple plotting of points and lines between absolute physical screen coordinates to a general two dimensional graphics ability. This includes the ability to display complex figures with respect to a logical origin, and to scale and rotate these figures for display. The ability to display graphic information through a window or mask is also included.

Most TUTOR applications use only the simple graphics in terms of absolute screen coordinates, so a new TUTOR implementation supporting only those should not be too restrictive. Implementation of PLATO like extended graphics capabilities poses no conceptual problems given sufficient computational power, and was not done in the implementation described here merely because of time and labor constraints.

In addition to the above, TUTOR allows program access to the writable dot matrix character set and other PLATO terminal parts such as a rear projection slide selector, audio response unit, and other devices. Support of these features should pose no new problems.

6.2.2. SINGLE KEYSTROKE INPUT

The -pause- command in TUTOR provides the basis for all single keystroke processing on PLATO. The -pause- command has two special capabilities that make it more than just a stream input instruction: First it provides the ability to set a timer on the input so that the program will resume on either timer expiration or a keystroke at the terminal, with the return to the program indicating how the -pause- ended. Second, it is possible to specify which possible inputs will be accepted and which ignored by a -pause- command.

The input filtering capabilities of the -pause- command are easily implemented at the interpreter level, with a simple 'loop until valid input' calling a simple stream input routine. The time limit on input is more difficult, requiring interaction between two different and normally disjoint operating system components; in some systems this may require signifigant system level changes.

Other TUTOR commands may be defined in terms of -pause-, such as -collect- which is equivalent to a loop with a counter and -pause- in it, or the -nextnow- and -unit- commands, which are equivalent to -jump- commands preceded by -pause- commands with only the NEXT key armed as a valid input. Given a working -pause- mechanism, it should be simple to implement these commands.

6.2.3. TEXT INPUT FOR JUDGING

There are four aspects of TUTOR text input that go beyond the normal unit record capabilities. The first of these is an extension to the normal input line editing capability that is present in some form on most interactive systems (backspace keys etc). Whenever a TUTOR program expects a line of input, it is possible to specify a text buffer from which that line may be constructed by copying characters or words interspersed as necessary with the new input. This facility allows simple but powerful text editors to be constructed as well as allowing CAI lessons to bring up an old student response and ask that it be modified for resubmission.

The second important text input capability required is that a program must be able to examine the contents of the input buffer and perform extensive computations while allowing the input operation to be resumed at a later time. The program may even generate output and make use of the single keystroke input facilities while some text input operation is suspended. This capability is required by the judging mechanism [section 3.2.1, appendix A] where the same input line may be repeatedly modified and reentered for judgment until it is judged "ok".

If output is generated between a temporary termination and the reopening of an input request, the output must be erased from the screen in the process of backtracking to the state that existed before the temporary input termination. On PLATO it was decided that total erasure of anything written was infeasable, so only the last -write- or other output command is erased by the system, and an -eraseu- unit may be armed to be attached as if by -do- after the automatic erasure so that the program may do the rest if needed.

The above PLATO solution is good enough that the -eraseu- mechanism is rarely needed; therefore, incompatibility here can probably be tolerated. An alternate solution for instance would be to erase the most recently displayed N characters after the termination of the most recent input operation, if N is large enough, few programs would be effected. This alternate approach has the advantage that it enforces a separation between the source language control structures and the input management system software.

The last important capability of TUTOR text input is the ability of the user to specify and change the internal code sequences associated with the keys on the terminal keyboard ('micro' substitution). This ability should be considered a function of the terminal or the device driver rather than of the interpreter.

6.2.4. RESPONSE TIME

The facilities listed above could easily be supported at the interpreter level using the same stream input mechanism used by the -pause- command and using the stream output to echo to the display; however, the response time for such a scheme could be considerably degraded because of the expense of activating the interpreter for each character of input.

A preferable alternative would be to place the line editing functions at a high priority as a distinct task, or even at a direct interupt priority level. This solution requires that some way be found to communicate all of the desired information between the interpreter and this high priority task. The best solution would be the use of some extension of the input/output protocols supported under the host operating system.

6.3. UNIFORM INTERNAL CHARACTER CODE POSSIBILITIES

One problem with PLATO input/output is caused by the six bit character code used for the internal text representation. This internal code requires two prefix codes (ACCESS and SHIFT) to represent the entire character set used on PLATO; in conjunction with the -pause- command this causes difficulties because one keypress or external input from the terminal may not in general be represented as one internal character code.

On PLATO the result is that the PAUSE command returns the last input in a character code different from the one used internally. It would seem preferable to use a single universal character code for both input and output; however, the PLATO terminal itself does not do this so software character code conversion would be requred. If the internal code is to be some extension of ASCII then conversion on both input and output would be required. These conversions can probably best be put in the line editing mechanism or even closer to the terminal, as is described elsewhere [22].

The introduction of a new universal character code will introduce some incompatibility in the handling of external device and touch panel input because these must still be manipulated as explicit bit patterns on PLATO, but the advantages of all of these changes seem to outweigh the problems in the long run.

6.4. POSSIBLITIES OF SUPPORTING OTHER TERMINALS

The use of an ASCII compatible character set opens the way for support of many other types of terminals. Support of terminals with only a subset of the PLATO capabilities would require filtering of output to eliminate functions that are not supported; this filtering process may be either an interpreter function, in which case the interpreter must always be aware of the terminal characteristics, or a function of the output driver.

It is even more important to consider future support of the various microporcessor based 'intelligent' terminals that are now proliferating, as many of these should be able to support many if not all of the functions of the PLATO terminal; furthermore, the character set used and transmission protocols of such terminals are all under internal sortware control and should be easily matched to any host system [21].

6.5. DIVISION OF INPUT/OUTPUT RESPONSIBILITIES

Because of the above listed considerations, the following breakdown of responsibility for input/output seems best, and is the one that was implemented. Like common variables, the way that linesets and micro tables are supported is highly dependant on the facilities for inter task sharing of information supported by the host hardware and operating system; as such, these are not covered here.

6.5.1. THE INTERPRETER

The interpreter is responsible for formatting output into dot, line, positioning, and text generation commands. Communication between the interpreter and the terminal control program takes place via packed buffers which contain on input either unit records for judging input or single characters for the -pause- family of commands. On output, the buffers contain either packed data for display on the screen or data to be interpreted by device handler to modify future transactions. These buffers are passed to the virtual device support program under the input/output protocols of the host operating system.

6.5.2. THE VIRTUAL DEVICE HANDLER

The virtual device handler is a piece of software that communicates with the interpreter or other user level programs via the system supported input/output buffer passing mechanism. The virtual device handler is responsible for character code translation, input line editing, and key echoing, as well as filtering output to the terminal, and input timer maintenance.

Appendix E outlines the buffer types and message meanings that the handler must respond to as well as an appropriate extension of ASCII. If the -size- and -rotate- directives are to apply to the echoing of keyset input during line editing, then the entire responsibility for handling these should be placed in the virtual device handler instead of the interpreter. This is reasonable because these are logical functions of future intelligent terminals.

Because the virtual device handler is the terminal from the point of view of the interpreter, it is proper to consider the interpreter as being written assuming an ideal intelligent terminal. Virtual device handlers compatible with the interpreter have been written to handle two different terminals to date. A PLATO IV terminal has been supported using handler software distributed between the mainframe and a remote microprocessor entirely responsible for code translation [2,22]. Also, a simple video CRT terminal with no graphics capacity has been supported [2].

7. CONCLUSION

An interpreter was implemented on the basis of the considerations outlined here. A medium scale machine with a maximum memory capacity of one million 8 bit bytes was used. This machine has a virtual memory mechanism based on 256 word pages, and a nominal word size of 16 bits, though the instruction set allows direct addressing and manipulation of bits, bytes, words, double-words, and quad-words.

The interpreter was written to take advantage of machine facilities for reentrant coding, and has been tested with two users sharing it. The virtual memory mechanism has not yet been exploited to its fullest extent but it is anticipated that support of demand paging of the user program space should not be difficult.

7.1. IMPLEMPLEMENTATION RESULTS

Bench marks run on the interpreter indicate that it can support 17 compute bound users while providing the same response characteristics as provided by PLATO for compute bound foreground users during prime time (with about 400 users). Using the assumption that no more than half of the users will be compute bound at any time, a system based on this interpreter should be able to support about thirty on line users at a time.

The problem of core sharing becomes critical if thirty users are to be supported on such a system. A compact user program representation will significantly reduce the swapping or page fault overhead, and if the interpreter can be made to run without significant overlay use, the load on the backing store will be even further reduced. The performance of the interpreter is highly encouraging with respect to these considerations.

The estimates previously published [6] concerning the storage requirements of a small computer based PLATO like system are considerably greater than the requirements experienced here (both for the user program and for the interpeter). The previous estimates were based on extrapolation from the PLATO implementation. The elimination of redundant mechanisms in the interpreter as outlined here is responsible to a great extent for these savings.

The interpreter, as currently implemented, supports around 60 TUTOR commands as well as their conditional and looping variants. To support these commands, as well as the basic computational ability requires 6800 16 bit words of reentrant program space as compared with the previous estimate [6] of 18000 16 bit words to support only the 20 most frequently used commands with overlay processing being used for the remainder.

Aside from the timing benchmark already mentioned, only one PLATO TUTOR lesson has been transfered from PLATO to the new interpreter to date. For the lesson transfered ['a game of GO' by R. Blomme] the compiled code required 9500 8 bit bytes versus 2000 60 bit words on PLATO, a savings of 37%.

7.2. INCOMPATIBILITIES WITH PLATO

The interpreter design proposed here is not fully compatible with PLATO. The most basic incompatibilities are those of data representation introduced either by the hardware, for instance a word size of 64 instead of 60 bits and two's instead of one's complement arithmetic, or by the software, such as an 8 bit extended ASCII character set instead of an extended 6 bit display code.

Incompatibilities in the support of common and storage introduced by a change from ecs to disk based backing storage or by the use of a paged virtual memory also fall into this first category. However, these areas are highly system dependent and as such are not within the realm of this paper.

Other incompatibilities have been introduced in order to obtain a greater degree of freedom in the choice of implementation approach. These include restrictions on changing the increment and bounds of a -doto- loop and limitations on the use of -goto- and -branch- with respect to judging blocks.

Further problems with compatibility are sure to arise in the future as the TUTOR language and PLATO system evolve. In that the TUTOR language as currently designed does have shortcomings, this evolution can only be encouraged. In the light of this, it can be asked what value there is in trying to support TUTOR on a small machine if it is not possible to maintain compatibility with the only major implementation of the language. The most important justification is probably that a small implementation allows a degree of experimentation that is not possible on the large central PLATO system which is bound to compatibility by its large user community.

7.3. EPILOGUE

Though exact compatibility between the PLATO implementation of TUTOR and one on a small to medium scale machine may well be impossible, all of the important features of the language can be supported in a manner flexible enough to be quite useful. It is hoped that the demonstration of this will open the way for numerous experiments in the support of TUTOR like abilities on smaller systems, as well as encourage the application of highly interactive graphic computing in new areas.

LIST OF REFERENCES

[1] Alpert, D. and Bitzer, D. L., "Advances in Computer Based Education," SCIENCE, vol. 167 (20 March 1970) pp. 1582-1590.

[2] Baskin, A. B., personal communication.

[3] Baskin, A. B. and Bloomfield, L., personal communication.

[4] Bell, J. R., "Threaded Code," CACM, vol. 16, no. 6 (June 1973).

[5] Bochmann, G. V., "Multiple Exits from a Loop Without GOTO," CACM, vol. 16, no. 7 (July 1973).

[6] Brackett, J.: Principal Investigator, "Final Report - A TUTOR Minicomputer System," Item no. 0002AD, Contract no. MDA903-74-C-0288, ARPA Order no. 2517, Softech Inc., Waltham, Mass. (7 February 1975).

[7] Chen, T. T., et. al., "A Depository Health Computer Network," to be published.

[8] Chen, T. T., personal communication.

[9] Embley, D. W. and Hansen, W. J., "The KAIL Selector - a Unified Control Construct," Sigplan Notices, vol. 11, no. 1 (January 1976).

[10] Embley, D. W., "Experimental and Formal Language Design Applied to Control Constructs for Interactive Computing," Ph. D. Thesis, University of Illinois (April 1976).

[11] Ghesquiere, J., et. al., "Introduction to TUTOR," PLATO Publications, Computer-based Education Research Laboratory, University of Illinois (June 1975).

[12] Knuth, D. E., "Structured Programming with GO TO Statements," Computing Surveys, vol. 6, no. 4 (December 1974), pp 261-301.

[13] Lackmann, J., "Simulation and Self-Assessment: The Physician Self-Assessment Study Final Report," American Medical Association Department of Continuing Medical Education (1970).

[14] Lackmann, J., "Physician Self Assessment: The Role of Self Assessment in Medical Information Systems," Proceedings of The First Illinois Conference on Medical Information Systems, Instrument Society of America, Pittsburgh, Pa. (1974).

[15] Lonergan, W. and King, P., "Design of the B 5000 System," Datamation, vol. 7, no. 5 (May 1961), pp 28-32. [republished in: Bell, C. G. and Newell, A., "Computer Structures: Readings and Examples," McGraw-Hill Book Company, New York, N. Y. (1971)]

[16] Ritchie, D. M. and Thompson, K., "The UNIX Time-Sharing System," CACM, vol. 17, num. 7 (July 1974) pp 365-375.

[17] Schuyler, J. A., "Student Response Matching Algorithms for Computer-Based Learning Systems," Ph. D. Thesis, Northwestern University, Evanston, Illinois, (August 1973).

[18] Schuyler, J. A., "The Complete Guide to HYPERTUTOR," Computers and Teaching, Northwestern University (June 1974).

[19] Sherwood, B. A., "The TUTOR Language," PLATO Publications, Computer-based Education Research Laboratory, University of Illinois (June 1975).

[20] Stifle, J. E., "The PLATO IV Student Terminal," Proceedings of the Society for Information Display, vol. 13 (1972). [paper by the same name also published by: PLATO Publications, Computer-based Education Research Laboratory, Univeristy of Illinois (November 1974)]

[21] Stone, M., et. al., "An Intelligent Graphics Terminal with Multi-host System Compatability," Digest of Papers, IEEE COMPCON'74, Washington D. C. (September 1974) pp. 37-40.

[22] Szolyga, T. H., "Design and Implementation of an ASCII Interface for a PLATO IV Terminal," M. S. Thesis, Department of Computer Science, University of Illinois (May 1976).

[23] Tenczar, P. and Golden, W., "Spelling, Word, and Concept Recognition," CERL Report X-35, Computer-based Education Research Laboratory, University of Illinois (October 1972).

[24] Williams, B. T., et. al., "PLATO-Based Medical Information Systems Overview," Proceedings of the First Illinois Conference on Medical Information Systems, Instrument Society of America, Pittsburgh, Pa. (1974).

[25] Wirth, N., "The Programming Language PASCAL," Acta Informatica, vol. 1 (1971) pp 35-63.

[26] Wu, Vincent H. L., personal communication.

[27] Zahn, C. T., "A Control Statement for Natural Top-down Structured Programming," presented at Symposium on Programming Languages, Paris (1974).

APPENDIX A - SUMMARY OF THE EXECUTION OF TUTOR

The following material is reproduced with permission from the AIDS information system on (and about) PLATO. This is not a complete summary in that some states are not clearly described and the interaction of the various states with subroutines is largely omitted.

PSO Author Group ---- CERL
Univ of Illinois, Urbana

regular state
judge state

Only regular commands are executed in regular state, TUTOR skips all other commands. Regular state ends when a judging command is encountered or when the end of the main unit is reached. In judge state TUTOR executes only judging commands, skipping all other commands. Judge state ends when a response is matched or when the judging region ends (at an -endarrow-, another -arrow-, or the end of the main unit).

search state
condense time

During search state, only -join- is executed. All other commands are skipped. Search state ends when an -endarrow-, another -arrow-, or the end of the main unit is reached. Encountering an -endarrow- or another -arrow- in search state causes TUTOR to switch to regular state. TUTOR then continues processing in regular state; executing regular commands. The non-executable TUTOR commands set pointers, make lists, and other functions when a lesson is condensed. The non-executable are skipped when TUTOR is executing.

#1 When a lesson is condensed, the non-executable TUTOR commands set pointers, make lists, and other functions. When TUTOR is executing in regular state, judge state, or search state, all non-executable commands are skipped.

TUTOR starts execution of a lesson in regular state. The commands before the first -unit- command (the i.e.u.) are executed. TUTOR enters the first main unit in regular state; TUTOR proceeds to #2.

#2 Regular commands are executed in a sequential manner. If an -arrow- was one of the regular commands executed, TUTOR remembers the location of that command in the unit (sets the -arrow- marker).

If the -arrow- marker was set, execution in regular state ceases when a judging command is reached and TUTOR proceeds to #4

If no -arrow- command was executed (no -arrow- marker was set), execution in regular state ceases when the end of the main unit is reached and TUTOR proceeds to #3.

#3 The main unit named in the tag of the -next- is branched to when the NEXT key is pressed. If there is no -next- command, TUTOR branches to the unit which physically follows the current unit when the NEXT key is pressed. The student may also exit this "completed" main unit by pressing an active function key, e.g. HELP; or executing a -jump- or -jumpout-.

The new main unit is started; TUTOR proceeds to #2.

#4 TUTOR waits for a student response. Judge state begins when the student presses the NEXT key, an active -jkey-, after the first character with a -long 1- in effect, or when the length limit is reached and a -force long- is in effect.

When judge state begins, TUTOR starts at the command after the -arrow- and executes judging commands in a sequential order until a match is found or the region of judging ends. The region of judging ends when an -endarrow-, another -arrow-, or the end of the main unit is reached.

If a -specs- was one of the judging commands executed, TUTOR remembers the location of that command in the unit (a -specs- marker is set). If more than one -specs- was executed, the last -specs- executed serves as the "specs marker". TUTOR proceeds to #5.

#5 If a judging command was matched, TUTOR executes regular commands following that matched judging command until another judging command, an -endarrow-, another -arrow-, or the end of the main unit is reached. The automatic response mark up is displayed and the judgment is set (either "ok" or "no"). TUTOR proceeds to #6.

If a judging command was not matched, TUTOR judges "no" and the automatic response mark up is displayed. TUTOR proceeds to #6.

#6 If a -specs- marker was set for the current -arrow-, TUTOR begins at the command following the last -specs- executed in regular state. All regular commands are executed until a judging command, -endarrow-, another -arrow-, or the end of the unit is reached.

If the -arrow- was judged "no", the student must erase all or part of the response by pressing NEXT, EDIT, shift-EDIT, ERASE, shift-ERASE, or an active -jkey-. The auto erasing of the last comment, the judgment, and the automatic response mark up also occurs (and the erasing by an active -eraseu-) when the student erases part or all of the response. TUTOR return to #4.

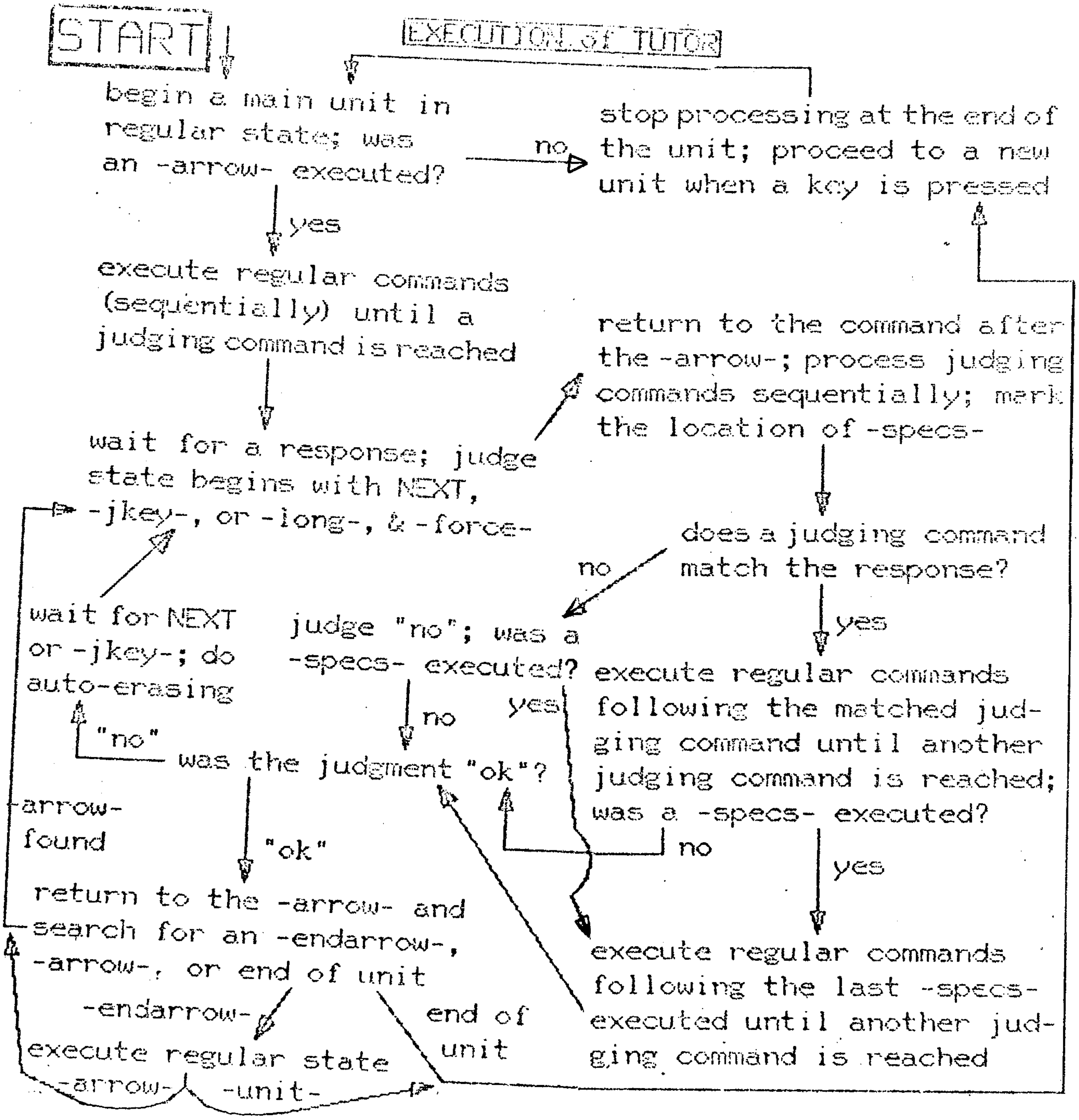
If the -arrow- was judged "ok", TUTOR proceeds to #7 in search state.

#7 After the response was judged "ok", TUTOR returns to the command after the -arrow- to search (search state) for an -endarrow-, another -arrow-, or the end of the main unit (-join- is executed in search state, as well as in regular state and judge state).

If an -endarrow- is reached, TUTOR changes from search state to regular state and executes regular commands just as if a new main unit was entered (except there is no clearing of the main unit pointers or automatic panel erasure). TUTOR proceeds to #2.

If another -arrow- is reached, TUTOR changes from search state to regular state. The new -arrow- marker is set and TUTOR continues to execute regular commands until a judging command is reached. TUTOR proceeds to #4.

If the end of the main unit was reached in search state, TUTOR proceeds to #3.


START
EXECUTION of TUTOR
begin a main unit in regular state; was an -arrow- executed?
no
stop processing at the end of the unit; proceed to a new unit when a key is pressed
yes
execute regular commands (sequentially) until a judging command is reached
return to the command after the -arrow-; process judging commands sequentially; mark the location of -specs-
wait for a response; judge state begins with NEXT, -jkey-, or -long-, & -force-
does a judging command match the response?
no
wait for NEXT or -jkey-; do auto-erasing
judge "no"; was a -specs- executed?
yes
execute regular commands following the matched judging command until another judging command is reached; was a -specs- executed?
yes
no
"no"
was the judgment "ok"?
no
-arrow- found
"ok"
yes
return to the -arrow- and search for an -endarrow-, -arrow-, or end of unit
execute regular commands following the last -specs- executed until another judging command is reached
-endarrow-
end of unit
execute regular state
-arrow-
-unit-

APPENDIX B - SPECIFICATIONS FOR THE INTEPRETER

B.1. PROGRAM STATUS

The elements of the program status defined here are those necessary for the intermediate code interpreter approach to TUTOR execution; they are in addition to those required by the various TUTOR commands, which are adequately discribed in other references [11,19].

B.1.1. 4 JUDGING STATUS BITS, NOT SAVED IN LINKAGE

0000 - Regular state.
0001 - Post -arrow- regular state.
1000 - Judging state (searching for judgment).
0100 - Post judging regular state, "ok" judgment.
0010 - Post judging regular state, "no" judgment.
1100 - Searching for -specs- with "ok" judgment.
1010 - Searching for -specs- with "no" judgment.
0101 - Post -specs- regular state, "ok" judgment.
0011 - Post -specs- regular state, "no" judgment.
1101 - Searching for -endarrow- with "ok" judgment.
1011 - Searching for -endarrow- with "no" judgment.

B.1.2. 4 UNIT TYPE BITS, SAVED IN LINKAGE

1xxx - Parameters on stack to be resolved.
x1xx - Unit entered by -do-.
xx1x - Unit entered by -join-.
xxx1 - Reserved for future experiments with -helpop-.

B.1.3. OTHER PARTS SAVED IN PROCEDURE LINKAGE

PC or INTERPRETER PROGRAM COUNTER; this is a 16 bit pointer into a vector of 8 bit bytes (the program space).

BASE UNIT POINTER; this is a 16 bit pointer into the program space for interrupt return applications.

LINK or STACK LINKAGE POINTER; a 16 bit pointer into a vector of 64 bit elements (the stack). This pointer is used to allow recursive and nested subroutine calls.

B.1.4. OTHER PARTS NOT INVOLVED IN LINKAGE

MAIN UNIT POINTER; a 16 bit pointer into the program space copied from the program counter on main unit entry.

SP or STACK POINTER; a 16 bit pointer into the stack used by all computational instructions and all parameter passing mechanisms. The push operation increments the SP before storing a value, and the pop operation decrements after recovering a value.

PARAMETER COUNT; an 8 bit count of the number of parameters currently on the stack, set by the parameter passing branch and call commands and used by the main unit entry command.

B.1.5. SEGMENT DEFINITION TABLE

This table has 32 entries, containing:
Segment base address, 16 bits.
Segment entry size, 6 bits specifying number of bits per entry.
Sign extender, 1 bit specifying that entries are signed.
Floater, 1 bit specifying entries in floating point format.

The first four entries in the segment table are predefined to:
0) System wide status variables (a special common block).
1) User dependant status variables.
2) User variables.
3) Common and storage share this.
4) Router variables (if implemented).

B.2. DATA FORMATS

B.2.1. SIMPLE DATA TYPES

Integers are stored as 64 bit two's complement values. Integer arithmetic operations use only the least significant 32 bits and sign extend their results to 64 bits, comparison operations work over the entire 64 bits as do bit manipulation operators.

Floating point data is stored in the 64 bit floating point format supported by the host machine.

Alphanumeric data is stored in 8 bits per character, with the least significant seven bits interpreted as ASCII when the most significant bit is zero, and as additional PLATO characters when the most significant bit is one. When alphanumeric data is packed into integers, the last or rightmost character occupies the least significant bit position.

Boolean data, which is produced as the result of comparisons, and is interpreted by the branch on false instruction, is compatable with PLATO and the table lookup branch: true is -1 or negative and false is zero or positive.

B.2.2. MEMORY ADDRESS FORMAT

Memory addresses may be provided either as the least significant three bytes of a 64 bit stack entry or as three consecutive bytes from the instruction stream in some cases. The format is as follows:

XXSSSSSF 00000000 00000000

where X is ignored, S indicates which segment, and 0 specifies an offset from the base of that segment in terms of the word size of that segment. F indicates the type of the data that the location is expected to hold (0=integer, 1=real).

B.2.3. LOOP CONTROL BLOCK FORMAT

STACK [SP] = increment value.
STACK [SP-1] = limit value for index.
STACK [SP-2] = initial value for index.
STACK [SP-3] = memory address of index.

B.2.4. LINKAGE WORD FORMAT

The current linkage word occupies one stack entry and is pointed to by the LINK register when executing within a subroutine. There are fields in this word for all the data of B.1.2 and B.1.3.

B.2.5. PARAMETER TYPE WORD FORMAT

The parameter type word occupies one stack entry and consists of 32 entries giving information about up to 32 parameters on the stack below it; the least significant entry corresponds to the topmost element on the stack below. The two bit fields for each parameter have the following meaning:

0x - parameter missing (no corresponding stack entry).
10 - the corresponding word is an integer.
11 - the corresponding word is floating point.

B.2.6. MASK BYTES FOR STATUS TESTING

A number of interpreter instructions test the bit patterns of B.1.1 and B.1.2 with the aid of a mask. All such masks have the common format: 11112222, where the bits marked 1 are those of B.1.1, and those marked 2 are B.1.2.

B.2.7. SPECIAL DATA IN THE PROGRAM SPACE

When constants requiring more than 8 bits are stored in the program space, they are stored most significant byte first. This holds for 16 bit program space address constants, 24 bit data address constants, and 16, 32, and 64 bit integer constants.

B.3. INTERPRETER INSTRUCTIONS

The instructions listed here provide the needed support for TUTOR, and the required computational ability. The actual TUTOR action routines are not listed here.

Each instruction is identified by its first 8 bits, the instruction number. The action routine for each instruction may consume additional bytes containing constant data or addresses, but almost all instructions are fixed format; if they may consume an extra byte of information, they will almost always consume it. Most instructions also have a fixed effect on the stack, always reading and popping or creating and pushing a fixed number of entries.

Instructions that need variable numbers of parameters are implemented with a well defined fixed part containing the specifications of the variable parts.

8.3.1. EXECUTION SEQUENCE CONTROL INSTRUCTIONS

#1 <16 bit address>: branch to that address.

#2 <16 bit address>: branch on false and pop stack.

#3 <8 bit count> <16 bit address>: table lookup branch. If the integer value of the stack top is greater than the count, the count is used as a value. If the value is negative, -1 is used. A branch is made to the address specified by the word addressed by twice the value plus the given address, and the value is poped from the stack.

#5 <16 bit address>: post index check and branch. Uses a loop control block on the stack. If after incrementing, the index value is out of bounds, then the stack control block is poped from the stack; otherwise the branch is taken. This instruction works for both floating and fixed point on the basis of the type indicated by the address in the stack control block.

#6 <16 bit address>: precheck loop and branch. Uses a loop control block on the stack. Stores the initial value in the index, then checks if it is within bounds. If not, the branch is taken and the stack control block poped.

#7 <8 bit mask> <16 bit address>: branch on status set.

#8 <8 bit mask> <16 bit address>: branch on status clear.

#10 <8 bit count> <8 bit type> <16 bit address>: call. The count is stored in the parameter count. The program status save word is stored in stack location SP-count, and the LINK word is pointed to that location. The least significant 4 bits of the type replace the linkage status bits, and control is transfered to the address.

#11 <8 bit mask>: return conditional on any of the judging or linkage status bits matching those set in the mask. The return consists of fetching the program status save word from the location pointed to by the link.

#12 <8 bit count> <16 bit address>: branch with parameters. The count is stored in the parameter count and control is transfered to the address.

#13 <8 bit signed value> <8 bit count> <list of 16 bit addr>: special table lookup branch with loop exit capabilities. The value on the stack top and the count combine as in #3 to index from the second entry in the list. If the selected address is zero, execution continues from the location after the end of the table, otherwise, the value is used as in #50 and execution resumes at the new address.
#14 <8 bit mask> <8 bit mask> <16 bit address>: special status checking branch. If any status bit matches a bit set in the first mask and no bits match the second mask, then branch.
#15 <8 bit mask>: set bits in status.
#16 <8 bit mask>: clear bits in status.

B.3.2. DATA FETCH AND STORE

#20 <24 bits of data>: push immediate address onto stack.
#21 <8 bits of data>: convert integer to address by providing segment information.
#24 <64 bits of data>: push immediate 64 bits onto stack.
#25 <32 bits of data>: push immediate 32 bits sign extended.
#26 <16 bits of data>: push immediate 16 bits sign extended.
#27 <8 bits of data>: push immediate 8 bits sign extended.
#28: replace address on stack with the data it points to.
#29 <24 bit address>: push data from immediate address.
#32: store data through address below it on stack. Replace the address with the data, and pop the origional copy of the data from the stack.
#33 <24 bit address>: store data in immediate address.
#48: pop word from stack.
#49: replicate word on stack top.
#50 <8 bit signed value>: modify SP by adding value.

B.3.3. BINARY INTEGER OPERATORS

#55: replace the top two stack elements by their sum.
#56: subtract stack top from the value under it.
#57: multiply.
#58: divide the stack top into the value under it.
#59: produce the remainder as in division.
#60: (SP-1)=(SP) replaces the elements compared.
#61: (SP-1)\=(SP)
#62: (SP-1)>(SP)
#63: (SP-1)>=(SP)
#64: (SP-1)<(SP)
#65: (SP-1)<=(SP)

B.3.4. BINARY FLOATING POINT OPERATORS

#67: replace the top two stack elements by their sum.
#68: subtract stack top from the value under it.
#69: multiply.
#70: divide the stack top into the value under it.
#72: (SP-1)=(SP) boolean comparison for equality.
#73: \=
#74: >
#75: >=
#76: <
#77: <=

B.3.5. BOOLEAN AND BIT OPERATORS

#80: boolean and of the top two stack elements.
#81: boolean or.
#82: boolean exclusive or.
#83: bitwise and, mask, or intersection operator.
#84: bitwise or, merge, or union.
#85: bitwise exclusive or, or difference.
#86: logical left shift (SP-1) by (SP), pop count.
#87: logical right shift.
#88: arithmatic right shift.
#89: circular left shift.

B.3.6. COMPLEMENTATION OF STACK TOP

#90: INTEGER NEGATE.
#91: FLOATING POINT NEGATE.
#92: BOOLEAN NOT.
#93: BITWISE NOT OR ONES COMPLEMENT.

B.3.7. ODDS AND ENDS

#95: convert integer to floating point on stack top.
#96: convert floating point to integer by truncation.
#97: integer part floating point number in floating point.
#98: fractional part of floating point number.
#99: convert floating point to integer by rounding.
#100: replace stack top with integer count of 1 bits in it.
#105: convert coarse to fine grid (replace top by two words).

B.3.8. FUNCTION LIBRARY

#128: replace stack top with sine of stack top (floating point).
#129: cosine (radians).
#130: tangent.
#131: arc sine.
#132: arc cosine.
#133: arc tangent.
#140: log10.
#141: loge.
#142: antilog10.
#143: antiloge.
#146: exponentiate.
#147: square root.

B.3.9. SPECIAL TUTOR COMMANDS

#251: begin new main unit, reset all but parameter bit in program status, copy any parameters down the stack, erase terminal screen and reset mode.
#252: if the unit type bits [B.1.2] are 0000 and there is a currnet -imain- unit, call it.
#253: end main unit, wait for any armed keypress or NEXT. If NEXT is pressed and there is a designated next unit to branch to do so, otherwise, fall through to the next unit when NEXT is pressed.
#255: prefix for other TUTOR commands

APPENDIX C - INTERMEDIATE CODE COMPILATION EXAMPLES

C.1. EXPRESSIONS

TUTOR CODE: calc n1:=n1+n(n2:=n2+1)

```
INTERMEDIATE CODE, COMMENTARY:
 1       #20        push address onto stack
 2-4     <n1>           24 bit address of n1
 5       #49        replicate address
 6       #28        fetch n1
 7       #20        push address onto stack
 8-10    <n2>           24 bit address of n2
 11      #49        replicate address
 12      #28        fetch n2
 13      #27        push constant onto stack
 14      =1             8 bit constant 1
 15      #55        integer add n2+1
 16      #32        store n2:=n2+1
 17      #21        convert the sum into an address
 18      <n>            make it n(n2:=n2+1)
 19      #28        fetch n(n2:=n2+1)
 20      #55        integer add n1+n(n2:=n2+1)
 21      #32        store n1:=n1+n(n2:=n2+1)
 22      #48        pop the stack to its origional state
```

C.2. SIMPLE COMMANDS

TUTOR CODE: at 1010

```
INTERMEDIATE CODE, COMMENTARY:
 1       #26        push constant onto stack
 2-3     =1010        16 bit constant 1010
 4       #105       convert coarse (1010) to fine (72,352)
 5       #255       TUTOR command follows
 6       <at>         the code for -at- (pops 2 parameters)
```

C.3. CONDITIONAL COMMANDS

TUTOR CODE: calcs   n1,n2:=n3,0,-n3

INTERMEDIATE CODE AND COMMENTARY:

```
 1      #20         push address in which to store result
 2-4    <n2>            the address of n2
 5      #29         push data from immediate address
 6-8    <n1>            address n1
 9      #3          table lookup branch
 10     =1              the maximum index is 1
 11-13  =36             the 0 entry of the jump table is at 36
 14     #29         push data from immediate address
 15-17  <n3>            address n3
 18     #1          branch to end of conditional
 19-20  =40             address of end
 21     #27         push immediate constant
 22     =0              zero
 23     #1          branch to end of conditional
 24-25  =40             address of end
 26     #29         push from immediate address
 27-29  <n3>            address n3
 30     #90         integer negate -n3
 31     #1          branch to end of conditional
 32-33  =40             address of end
 34-35  =14         table address for negative index
 36-37  =21         table address for zero index
 38-39  =26         table address for positive index
 40     #32         store result in address below it (n2)
 41     #48         pop result value off of stack
```

C.4. LOOPING COMMANDS

```
TUTOR CODE: do      unit1,n1:=1,100,5

INTERMEDIATE CODE, COMMENTARY:
 1      #20        push index address onto stack
 2-4    <n1>           address n1
 5      #27        push initial value onto stack
 6      =1             value 1
 7      #27        push final value onto stack
 8      =100           value 100
 9      #27        push increment value onto stack
 10     =5             value 5
 11     #6         loop control block precheck and setup
 12-13  =22            address beyond end of loop
 14     #10        call
 15     =0             no parameters
 16     =4             type is do (binary 0100)
 17-18  =<unit1>       address of entry point
 19     #5         loop control block post index and branch
 20-21  =14            address of top of loop
```

(C.4. LOOPING COMMANDS continued)

```
TUTOR CODE: doto     1lab,n1:=10,2,-1
            branch   n(n1),2lab,x,1lab
                     n(n1):=-1
            1lab
            2lab
```

```
INTERMEDIATE CODE, COMMENTARY:
 1      #20         push address
 2-4    <n1>            address of n1, loop index
 5      #27         push immediate constant
 6      =10             initial index value
 7      #27         push immediate constant
 8      =2              final index value
 9      #27         push immediate constant
 10     =-1             increment value
 11     #6          loop control block precheck and initialize
 12-13  =48             address beyond end of loop
 14     #29         push from immediate memory address
 15-17  <n1>            address of n1
 18     #21         convert integer to address of n(n1)
 19     <n>             specify student variables (n)
 20     #28         fetch n(n1)
 22     #13         special branch that allows loop exit
 23     =-4           pop 4 levels before branching
 24     =1            maximum index value is 1
 25-26  =48           address for index negative, 2lab
 27-28  =0            no branch on index zero
 29-30  =43           address for 1 or positive, 1lab
 31     #29         push data from immediate address
 32-34  <n1>          address of n1
 35     #21         convert n1 to the address n(n1)
 36     <n>           space attribute for stack top
 37     #27         push immediate 8 bits of data
 38     =-1           constant
 39     #32         store n(n1):=-1
 40     #48         pop back to the loop control block
 41     #50         prepare for label inside loop
 42     =-4             by temporarily popping loop control block
 43     #50         1lab is here, recover loop control block
 44     =4              by restoring stack pointer
 45     #5          postcheck loop control block and loop back
 46-47  =14             address of top of loop
```

APPENDIX D - COMPILATION OF UNITS AND JUDGING SEQUENCES

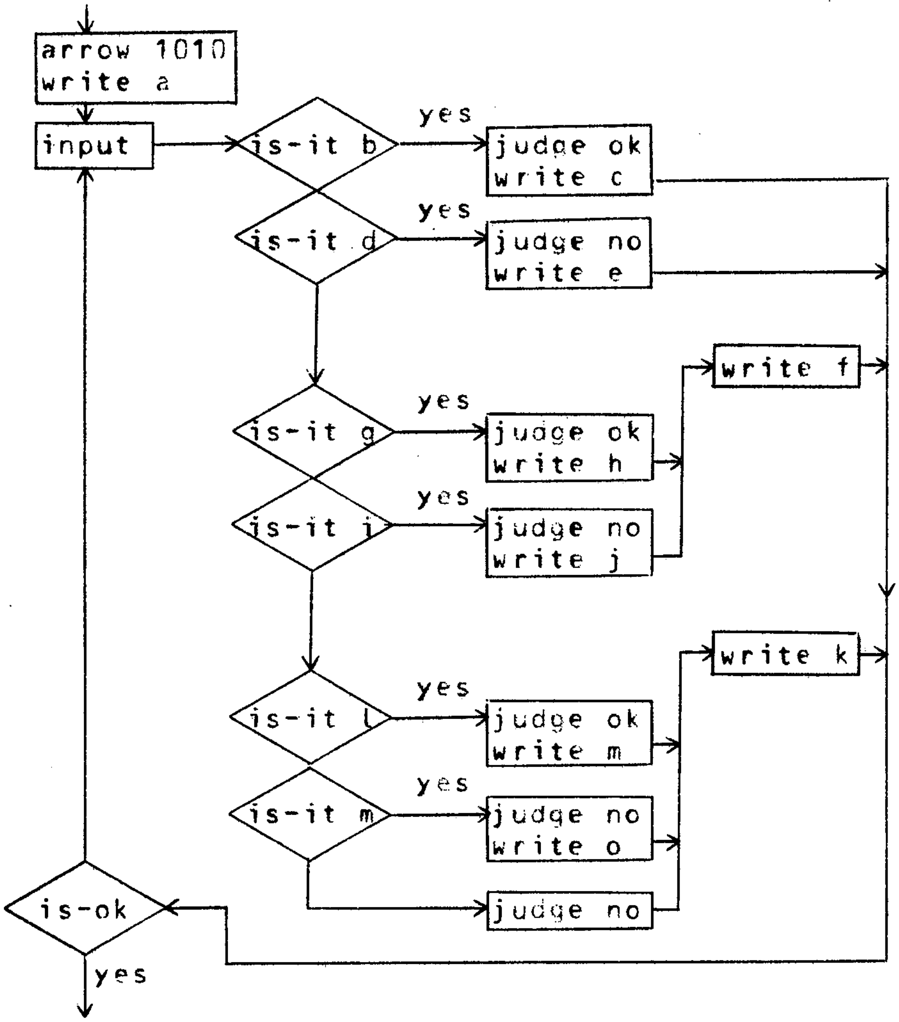
The first part of this appendix is an example of the structure of an -arrow- -endarrow- block in TUTOR. This is useful as a basis for understanding the remainder of this appendix which is a listing of the 'templates' used in compiling the special judging statements of TUTOR into the instruction set of appendix B. Throughout this appendix, the aspects of TUTOR judging relating to its control structure are emphasized at the expense of the data manipulation and input/output side effects of the commands.

D.1. REDUCTION OF A JUDGING BLOCK TO A FLOWCHART

TUTOR CODE:

```
arrow    1010
write    a

answer   b
write    c

wrong    d
write    e

specs
write    f

answer   g
write    h

wrong    i
write    j

specs
write    k

answer   l
write    m

wrong    n
write    o

endarrow
```

FLOWCHART:



D.2. A STRUCTURED REDUCTION OF THE SAME CODE

```
arrow(1010);
write("a");
REPEAT
     input;
     BEGIN UNTIL specs1 OR specs2;
          IF      it-is("b")
               THEN {ok;write("c")}
          ELSEIF it-is("d")
               THEN {no;write("e")}
          ELSEIF it-is("g")
               THEN {ok;write("h");specs1}
          ELSEIF it-is("i")
               THEN {no;write("j");specs1}
          ELSEIF it-is("l")
               THEN {ok;write("m");specs2}
          ELSEIF it-is("n")
               THEN {no;write("o");specs2}
     WHEN
          specs1: write("f");
          specs2: write("k");
     END
UNTIL judgedok;
```

Note that the UNTIL clause is a declaration of the exit blocks that actually occur in the WHEN clause. This notation is an adaptation of that employed by Knuth [12].

D.3. EXPANSION OF THE TUTOR -unit- COMMAND

1) Generate an endarrow if there is not one between the last -unit-, judging command, or -entry- and the current location.

2) Gather all branches to the end of unit to this point, for instance from -goto q- or step (10) of (D.8).

3) Generate code:

   ```
   #11       Retrun conditional on status bits set.
   =6          bits -do- and -join- (binary 0000 0110).
   #253      End of main unit command.
   ```

4) Gather all branches to the new unit as a main unit to this point (-jump-, -help-, -back-, -nextnow-).

5) Generate code:

   ```
   #251      Start of main unit command.
   ```

6) Gather all branches and calls to the new unit as an attached unit (-goto-, -do-, -join-, -helpop-).

7) Generate code only if the new unit expects parameters:

   ```
   #8        Branch if bits not set in status.
   =8          parameter bit (binary 0000 1000).
   <addr>      to the location defined in step 8.
   ```

8) If this unit can receive parameters, generate code to do so.

9) Gather branch from step (7) to this point if needed.

10) Generate code:

   ```
   #14       Branch on bit set.
   =128        judging state bit set (1000 0000).
   <addr>      address of next judging command as
                 specified by (6) in (D.5).
   ```

D.4. EXPANSION OF THE TUTOR -arrow- COMMAND

1) Generate an endarrow if there was not one between the last -unit-, judging command, or -entry- and the current location.

2) Generate the code to compute the parameters to the -arrow- command (X and Y at which to echo), then generate the command itself. This command changes the state to (0001 xxxx) and must only be entered in state (0000 xxxx), as guaranteed by the -endarrow- generated in step (1).

D.5. EXPANSION OF TUTOR JUDGING COMMANDS

1) Generate code:

```
#8        Branch on all bits reset in status.
=240        all of the judging bits (1111 0000).
<addr>      to the address in step (10).
```

2) Generate code if the last judging command could end judging or if the last judging command was -join- or if this is the first judging command in this unit before an -arrow-

```
#8        Branch on all bits reset in status.
=144        "ok" and "no" bits set (1001 0000).
<addr>      to the previous -specs- step (4) of (D.7),
            or if there is none, to the -endarrow-
            step (4) of (D.8).
```

3) Generate code if the last judging command was -specs- or -join- or if this is the first judging command in this unit before an -arrow-

```
#8        Branch on all bits reset in status.
=96         the postarrow bit may be set (0110 0000).
<addr>      to the first judging command after the
            -arrow- as defined by step (4).
#15       Set bit in status.
=128        enter search for endarrow state (1000 0000).
#1        Branch.
<addr>      to the -endarrow- step (7) of (D.8).
```

4) If this is the first judging command after an -arrow- or after the start of a -unit- before an -arrow-, gather branches from step (3) above and step (8) of (D.8) here.

5) If step (4) was followed, generate code to request input from the terminal and enter judging state (1000 xxxx) if necessary.

6) Gather branches to the next judging command to here from -unit- step (10) of (D.3), steps (8) and (9) here and step (3) of -join- (D.6).

7) Generate code to compute the parameters to and call the appropriate judging command.

8) If the specific judging command never ends judging then generate code:

```
#1        Branch.
<addr>      to the next judging command as specified
              by part (6) here or part (9) in (D.8).
```

9) If the specific judging command sometimes ends judging then generate code:

```
#7        Branch on bits set in state.
=128        still in judging state (1000 0000).
<addr>      to the next judging command as in (8).
```

10) Gather branch from step (1) above.

D.6. EXPANSION OF THE TUTOR -join- COMMAND

1) Include parts (4) and (6) from (D.5).

2) Generate -join- command complete with any code for conditional or looping parts.

3) Generate code:

```
#14       Branch on special status test.
=128        judging state not ended (1000 0000).
=112        regular bits all reset (0111 0000).
<addr>      to the next judging command as in part
              (8) of (D.5).
#14       Branch on special status test.
=128        some search state (1000 0000).
=16         for -specs- (0001 0000).
<addr>      to the previous -specs- part (4) of
              (D.7) or to the -endarrow- part
              (4) of (D.8).
#7        Branch on bit set in status.
=128        judging bit set (1000 0000).
<addr>      to step (6) of (D.8), the -endarrow-.
```

D.7. EXPANSION OF THE TUTOR -specs- COMMAND

1) Include steps (1) through (6) from (D.5).

2) Generate -specs- command and parameters.

3) Generate code:

```
#1        Unconditional branch.
<addr>      to the next judging command.
```

4) Setup so branches to previous -specs- will arrive here.

5) Generate code for the special command that changes to post -specs- state and outputs markup.

6) Include step (10) from (D.5) here.

D.8. EXPANSION OF THE TUTOR -endarrow- COMMAND

1) If the last judging command was not -ok- or -no-, and there is an -arrow- between the last -unit- and here, generate the -no- judging command with all of the steps of (D.5).

2) Generate code:

```
#8        Branch on status bits clear.
=240        regular state (1111 0000).
<addr>      to step (11).
```

3) If there is no -arrow- between the last -unit- and here, generate code:

```
#11       Retrun conditional on bit set in status.
=6          -do- and -join- bits (0000 0110).
```

4) Gather branches to -endarrow- from step (2) of (D.5) and step (3) of (D.6) to this point.

5) If there was an -arrow- between the last -unit- and here, then include step (5) from (D.7).

6) If (5) was not done, generate code:

```
#15       Set bit in status.
=128        enter search for -specs- state (1000 0000).
```

7) Gather branches to -endarrow- from step (3) of (D.5) and step (3) of (D.6) to this point.

8) Generate code if there was an -arrow- between the last -unit- and the current point to test and change the state and branch back to the first judging command as specified by step (4) of (D.5); the branch back to the first judging command is executed in judging state after re-requesting input from the terminal.

9) If (8) was not done, gather any branches to the next judging command to this point.

10) If (8) was not done, generate code:

```
#1        Branch.
<addr>      to the end of this unit.
```

11) Gather branch from step (2) to here.

APPENDIX E - TERMINAL TRANSMISSION CODES AND BUFFERS

E.1. THE TRANSMISSION CODE, AN EXTENSION OF ASCII

E.1.1. PRINTING CHARACTERS

First Digit

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0			sp	0	@	P	`	p							´	π
1			!	1	A	Q	a	q							α	
2			"	2	B	R	b	r							β	ρ
3			#	3	C	S	c	s						Σ		σ
4			$	4	D	T	d	t					Δ		δ	
5			%	5	E	U	e	u								
6			&	6	F	V	f	v								
7			'	7	G	W	g	w			¯				θ	ω
8			(	8	H	X	h	x			∩					
9			)	9	I	Y	i	y			∪					
A			*	:	J	Z	j	z			×					
B			+	;	K	[	k	{						←		⋖
C			,	<	L	\	l	|			⇕	≤		↓	λ	≡
D			-	=	M	]	m	}				≠		→	μ	⋗
E			.	>	N	^	n	~				≥		↑		~
F			/	?	O	_	o				÷	≫		⇦	°	

E.1.2. CONTROL CHARACTERS (COLUMNS 0 AND 1)

HEX:	NAME:	USE:
00	NUL	ignored
01	SOH	home (X,M set 0, Y set 512)
02	STX	use normal character set
03	ETX	use user provided charset
04	EOT	end of transmission
05	ENQ	*** enquire
06	ACK	*** acknowledge
07	BEL	ring bell
08	BS	backspace (X set X-8)
09	HT	tab
0A	LF	linefeed (Y set Y-16)
0B	VT	vertical tab (Y set Y+16)
0C	FF	formfeed
0D	CR	carriage return
0E	SO	superscript (Y set Y+5) keyboard SUPER1
0F	SI	subscript (Y set Y-5) keyboard SUB1
10	DLE	data block prefix (binary data)
11	DC1	graphics - at mode
12	DC2	graphics - dot mode
13	DC3	graphics - line mode
14	DC4	graphics - extended graphics mode
15	NAK	*** negative acknowledge
16	SYN	*** synchronise
17	ETB	*** end of transmission block
18	CAN	cancel line of input
19	EM	set mode control
1A	SUB	substitute - set 8th bit in next character
1B	ESC	escape to special controls
1C	FS	touch control
1D	GS	external device prefix (text mode)
1E	RS	*
1F	US	*

*** communications control
*   unassigned

E.1.3. ESCAPE CHARACTERS (COLUMNS 8 AND 9)

ESCAPE KEY:	HEX:	NAME:	EDITING FUNCTION:
@	80	STOP	
A	81	DATA	
B	82	LAB	
C	83	BACK	
D	84	NEXT	
E	85	HELP	
F	86	ANS	
G	87		
H	88	COPY	copy one word
I	89	EDIT	restore word from line buffer
J	8A	ERASE	erase character
K	8B		
L	8C	(square)	copy one character
M	8D	MICRO	
N	8E	SUPER	
O	8F	SUB	
P	90	STOP1	
Q	91	DATA1	
R	92	LAB1	
S	93	BACK1	
T	94	NEXT1	
U	95	HELP1	
V	96	TERM	
W	97		
X	98	COPY1	copy rest of line
Y	99	EDIT1	copy rest of edit buffer
Z	9A	ERASE1	erase word
]	9B		
\	9C	(square)1	
[	9D	FONT MICRO1	
^	9E		{ SUPER1 is 0E }
_	9F	TIMEUP	{ SUB1 is 0F }

E.2. BUFFER TYPES BETWEEN INTERPRETER AND DEVICE HANDLER

Note that all transactions are initiated by the interpreter; the device handler has an entirely passive role except that it must abort the interpreter task when the STOP1 key is pressed.

1) Read single character from terminal without echo or micro table translation, used for -pause- type input. An optional time limit may be specified after which the read will terminate with the TIMEUP code (9F) as the input character.

2) Read 2 characters from terminal without echo or micro; used to read 2 character suffix of TOUCH code.

3) Write buffer of data (all kinds may be mixed) to the terminal. Buffer is terminated with one or more null bytes. This kind of output is used for all normal TUTOR output generation commands such as -write- or -draw-.

4) Write buffer of special data to handler; any of:
   a) Size of future output.
   b) Rotation for future output (origin specified in (3)).
   c) Micro table specification for input echoing.
   d) Lineset specification for output and echoing.
   e) Charset specification for output and echoing.

5) Write buffer of special line editing setup information to terminal driver.
   a) May contain characters that are to be "jkeys".
   b) May contain initial screen location for echoing.
   c) May contain terminal mode for echoing.
   d) May contain markup "ok" or "no" type message.
   e) May contain "copy" buffer for editing from.

6) Read one block or line of input, With limit of <N> char's specified and conditions from (5) parts {a,b,c,e} used to modify the behavior in conjunction with the read type modifiers which may also be included:
   a) Re-open the most recent transaction.
   b) Inhibit the use of the EDIT key on the input line.
   c) Force immediate return when N char's have been read.
   d) if (a) then erase last output when editing starts.
   e) if (a) then start editing immediatly.
   f) if (a) then start editing when the user presses a key.
   g) if (a) then erase old line when editing begins.