WIS-CS-73-179
COMPUTER SCIENCES DEPARTMENT
The University of Wisconsin
1210 West Dayton Street
Madison, Wisconsin 53706

IMPLEMENTATION OF A GENERATIVE COMPUTER-ASSISTED INSTRUCTION SYSTEM ON A SMALL COMPUTER

by

Rick LeFaivre

Technical Report #179

April 1973

Received April 17, 1973

This research was supported in part by NSF Grant GJ-36312.

ABSTRACT

This paper gives a brief overview of several "traditional" approaches towards computer-assisted instruction (CAI). A new CAI language is then described which allows the user to write programs which generate both questions and answers with a minimum of programming effort. An implementation of the described system is discussed in detail, followed by a description of a sample program which drills the student in German-English translation by generating sentences whose complexity is a function of performance.

IMPLEMENTATION OF A GENERATIVE COMPUTER-ASSISTED INSTRUCTION SYSTEM ON A SMALL COMPUTER

I. INTRODUCTION

Computer-assisted instruction (CAI) has long been visualized as a viable means of increasing the level of individualization in the educational process. It has also been regarded disparagingly as having a dehumanizing rigidity which makes meaningful interaction between "teacher" and student impossible. One of the major goals of current research in CAI is to find ways to reduce this rigidity by making the computer more responsive to the needs and desires of the individual student. This paper consists of a brief review of several different major thrusts in the development of CAI, followed by a detailed description of a generative system developed for the Digital Equipment Corporation PDP-12 Mini-Computer.

Traditional CAI

The traditional CAI system consists of a data base of frames to be presented to the student and an executive program to select the particular frames to be presented and check the student's replies. The sequence of frames may be predetermined (linear) or may be determined dynamically as a function of the student's replies and past history (branching). The frames of information, the anticipated answers, and the detailed branching structure (if any) must all be specified by the teacher before the

student may use the program. Clearly, if the instructional sequence is to be at all "tailored" to the individual student, a tremendous amount of effort must be expended in frame preparation and answer anticipation. For example, consider an instructional network layed out as a binary tree--each node in the tree constituting one frame, and each frame having two possible replies. In this setting, a modest instructional sequence of 20 questions would necessitate the preparation of 1,048,575 frames! Of course, such highly individualized teaching sequences are rarely implemented using conventional CAI techniques. Normally there are from one to several major "strands", with provision made only for branching from strand to strand or up and down a strand.

In what we are calling a "traditional" CAI system, the role of the computer is limited to four major tasks: selecting a frame to present (using some pre-supplied selection algorithm); presenting the frame to the student; accepting the student's reply; and comparing the reply to a list of pre-supplied answers. Note that nowhere in this select-present-accept-compare cycle is the computer required to do any actual computing. The power of the computer as a computational device is essentially being wasted, while the teacher is required to spend hours preparing a sufficient variety of frames to assure an interesting teaching sequence. [Note that "interesting" is typically defined in terms of the "normal" student. The particularly slow (or advanced) student is apt to find that not enough easy (or hard) problems were provided to tailor the instructional sequence to his needs.]

Generative CAI

The recognition of the limitations of traditional CAI led to the development of generative CAI systems--systems where the computer is used to generate instructional material as a function of student interaction. Instead of preparing a sequence of frames such as:

```
'ADD 1 + 1' <'2'>
'ADD 5 + 8' <'13'>
      .
      .
      .
```

the teacher may prepare a generative "program":

```
'ADD' NUM1 '+' NUM2 <SUM>
```

[where NUM1 and NUM2 might be functions which generate increasingly harder (or easier) problems as a function of student performance]. Thus in generative CAI, the data base is expanded to a potentially infinite number of frames, any one of which may be "selected" for an individual student as a function of his unique history.

Historically, generative CAI as described above has tended to be used with quantitative problem areas where simple algorithms for question and answer generation are known (e.g., arithmetic, algebra and set theory). Note in particular the work of Uhr [1, 2], Hoffman and Seagle [3], Peplinski [4], Wexler [5], Uttal, et. al. [6], Siklossy [7] and Koffman [8]. Many of the high-level CAI languages developed to assist teachers in CAI program preparation contain at least a minimal generative capability. (Perhaps the most common element in generative systems is a random number generator coupled with the ability to do simple arithmetic.) Zinn [9, 10]

has published several excellent reviews of these CAI languages.

"Intelligent" CAI

A related, but much more complex and possibly more important, form of generation is that present in the systems of Wexler [11, 12] and Carbonell [13, 14]. These systems are "intelligent" in the sense that they have knowledge of the particular subject being taught. This knowledge is stored in the form of a structured network of facts and relationships. Carbonell's system, in particular, relies heavily on the concept of a semantic network as developed by Quillian [15]. The network is used both to generate questions for the student (and check his replies), and to answer questions asked by the student. The systems are able to converse in (restricted) natural language, and allow for comparatively realistic student/"teacher" dialogues. The major effort required on the part of the teacher is initially building the semantic network. As research in related areas such as computer question-answering, natural language processing, and automatic construction of semantic networks progresses, this approach promises to have a major effect on CAI.

An Experimental CAI Language

The purpose of the research reported in this paper was to investigate the use of generative CAI techniques on small (mini) computers. Although the author feels that "intelligent" systems such as Wexler's and Carbonell's ultimately offer the most promise to the development of CAI, such a system was not attempted here. The storage space required to build up a semantic memory of interesting

complexity would seem to preclude the use of anything but a relatively large computer for such a system. Indeed, there are still many important artificial intelligence problems which must be solved before such systems may be seriously considered for large-scale CAI use. Instead, the system described here provides an interpreter for a CAI language which offers a blend of simplicity of use and generative power. This language, called GLEE (Generative Language for Educational Experimentation), allows complex instructional sequences to be generated with a minimum of programming effort. It demonstrates that interesting and useful generative CAI systems may be developed on small (and therefore relatively inexpensive) computers.

II. OVERVIEW OF THE GLEE LANGUAGE

Facilities for Branching

The two major statement types in the GLEE language are frame definition statements and function definitions. A frame definition statement is composed of three basic segments: a label; a message expression; and a list of answer groups. For example, consider the following:

```
#PROB1  'ADD 2 + 2'  <'4,FOUR':RIGHT,PROB2><:WRONG,PROB1>
```

Execution of this statement, which is identified as PROB1 to the rest of the program, will cause the string 'ADD 2 + 2' to be presented to the student, after which the system will wait for the student's reply. If the student answers by typing either '4' or 'FOUR', a branch will be made to the statement labeled RIGHT, followed by a branch to PROB2. Any other answer will cause a branch to WRONG, followed by a repetition of the current problem. WRONG, for example, might be a simple feedback statement:

```
#WRONG  'INCORRECT.  TRY AGAIN.'
```

In general, when a branch is made to a statement, the message expression is evaluated (see below) and the resultant string is presented to the student. The answer expression in each answer group is then evaluated and the alternative answers (separated by commas) are compared with the student's reply (the answer-matching algorithm will be discussed in detail in Section IV). If a match is found, the labels present in the answer group (following the ':') are put on the top of the branching

stack and a branch is made to the first label on the stack. If there are no answers in an answer group (e.g., <:WRONG,PROB1>) the student's reply is ignored and the associated labels are placed on the branching stack as if a match were found. Such an answer group is termed a default answer group.

Facilities for Generation

Given just what has been described thus far, a GLEE user could write a branching CAI program in the traditional sense (i.e., with all the frames (messages) and expected answers pre-programmed). A generative capability, however, may be added in a relatively painless manner through use of variables and functions. Consider the following statement:

```
#ADDGEN 'ADD ' #NUM1 ' + ' #NUM2 <#SUM:RIGHT><:WRONG>
```

The number sign (#) is a unary operator which causes the value of its operand to be converted to a string (e.g., the result of evaluating the expression #(43) is the string '43'). [Note: The '#' preceding ADDGEN above is not an operator, but serves to identify ADDGEN as the label of this statement.] When the above message expression is evaluated, a string is built up consisting of the characters 'ADD' followed by the value of NUM1, followed by '+' followed by the value of NUM2. Evaluation of the answer in the first answer group results in a string representing the value of SUM. There is no way to know at this point whether NUM1, NUM2 and SUM are variables or functions. If they are variables, they could be initialized at some point earlier in the sequence by calling a function like the following:

```
$SET     NUM1 = RAND
         NUM2 = RAND
         SUM = NUM1+NUM2
         !EXIT = 0
```

where SET calls a random number generator RAND. [As can be seen, the convention used to return from a function is to assign a value (which will be the value of the function) to the system variable !EXIT.] If in the above example NUM1, NUM2 and SUM are functions, they might be defined as follows:

```
[THIS IS FUNCTION NUM1
$NUM1    N1 = RAND
         !EXIT = N1
[THIS IS FUNCTION NUM2
$NUM2    N2 = RAND
         !EXIT = N2
[THIS IS FUNCTION SUM
$SUM     !EXIT = N1+N2
```

[Note the use of comments to annotate the GLEE program.]

Expression Evaluation

It might be well to pause at this point and reflect upon the relationship between the evaluation process and the string ultimately used as a message or answer. An expression may appear in a GLEE program as a message, an answer, or, as will be seen, in assignment and if statements in function definitions. The expression (consisting of variables, function calls, and the operators '+', '*', '-', '/' and '#') is evaluated as would be expected (except that operator precedence must be indicated explicitly through use of parentheses) with the following extensions:

1) A quoted string (e.g., 'ABC') has the value 0. When encountered during the evaluation process, the characters between the quotes are added to the current message or answer string being built.
2) The unary operator '#' returns as a value the value of its operand; it also has the side effect of converting the value of its operand to a string of digits and adding it to the current message or answer string.
3) When two operands appear next to one another, the addition operator is inferred (e.g., A B is equivalent to A+B).

Using these rules, and assuming that the next two calls to RAND return the numbers 4 and 7, the actual value of the message expression in ADDGEN is 0+4+0+7, or 11. This value is incidental, however, as the reason for evaluating the expression is actually to build up the message string 'ADD 4 + 7'. To further belabor the point, the following GLEE program segment would have exactly the same effect as the above:

```
#ADDGEN NUMS <SUM:RIGHT><:WRONG>
$NUMS    N1 = 'ADD ' #RAND  [BUILD UP LEFT HALF OF STRING
         N2 = ' + ' #RAND   [BUILD UP RIGHT HALF
         !EXIT = 0          [RETURN (WITH VALUE OF 0)
$SUM     !EXIT = #(N1+N2)   [BUILD UP ANSWER STRING
```

Summary of Basic Syntax

At this point the basic syntax of the GLEE language should be relatively clear. Each frame definition statement must be labeled, with the label preceded by a '#'. The message segment may be any expression. This expression will be evaluated, the value will be discarded, and the resultant string (built via evaluation of quoted strings and use of the '#' operator) will be presented to the student. A list of answer groups appears next, each answer group consisting of a '<', an answer

expression, a ':', a list of labels to be branched to in case a match is found, and a '>'. Functions are defined by placing an entry point on the first line of the function definition, where an entry point consists of a '$' followed by the function name. A function is evaluated by successively executing the statements in its definition, with the value being the value assigned to the system variable !EXIT. Functions may not have parameters, but may be recursive. All variables are global, and may take on any integer value in the range -8,388,607 to +8,388,607. Labels, entry points and variable names consist of 1 to 6 alphanumeric characters, with the first character alphabetic. A single GLEE program may contain up to 204 labels, entry points and variables. Comments, initiated with a '[', may appear almost anywhere in a GLEE program.

Function Definition Statement Types

There are three basic types of statements which may appear in function definitions. Any of the three may be labeled with an entry point. The first is the assignment statement:

VARBLE = EXPR

This statement causes the expression (EXPR) to be evaluated, with the (integer) value stored in the variable (VARBLE).

The second major statement type is the goto statement:

!GOTO NAME

This statement causes evaluation of the current function to continue with the statement labeled with the entry point (NAME).

The other major function definition statement type is the if statement:

```
!IF (EXPR1 OP EXPR2) STMNT
```

EXPR1 and EXPR2 may be any expressions. OP may be one of the relational operators <, =, >, <=, >= or <> (i.e., not equal). STMNT may be an assignment statement, a goto statement, or another if statement. STMNT will be evaluated only if the value of EXPR1 is in the proper relation to the value of EXPR2. If not, STMNT is skipped and execution continues with the next statement, For example, the following function returns the factorial of the number stored in N:

```
$FACT   !IF (N<=1) !EXIT=1
        N = N - 1
        !EXIT = (N+1)*FACT
```

III. OVERVIEW OF THE GLEEFUL SYSTEM

The system described in this paper was implemented on a Digital Equipment Corporation PDP-12 computer with 8192 12-bit words. This computer is somewhat unique in that it has a built-in CRT and two random-access LINCtape drives. The CRT and Teletype are used for interactive I/O, while the tapes are used for bulk storage of GLEE programs and student data. The system exists as a program (called GLEEFUL--GLEE For Useful Learning) which may be loaded by LAP6W, an interactive editor/filer/assembler system for LINC-class computers [16]. The editing and filing capabilities of LAP6W are utilized to prepare GLEE programs, file them on tape, and list them on the Teletype or line printer. Once given a name and filed on tape, the GLEE program is accessible to the GLEEFUL interpreter. GLEEFUL is written in WISAL, LAP6W's LINC assembly language, and occupies approximately 7K of core (the remaining 1K being reserved for LAP6W's resident trap processor).

Loading Procedure

The GLEEFUL system is started by loading a LAP6W system tape on unit 0, placing a tape containing GLEEFUL and the GLEE program(s) to be run on unit 1, and typing

```
→LB GLEEFUL,1
```

After striking RETURN, the following display appears on the CRT:

(1)

```
ENTER NAME OF COURSE MANUSCRIPT.
TERMINATE ALL DISPLAYS BY STRIKING 'RETURN'.
```

The name of the GLEE program to be run is entered, appearing as typed at the bottom of the scope. RUBOUT may be struck at any time to delete the last character entered. After striking RETURN, the named program is located in the file of the tape on unit 1 (if not present, the above display reappears).

Student History File

After locating the desired program, the following display appears:

(2)

```
ENTER PRE-ASSIGNED STUDENT NUMBER IF
REPLIES ARE TO BE ADDED TO YOUR FILE.
STRIKE 'RETURN' IF NOT.
```

A file may optionally be maintained for each student giving a trace of the student's activities during the course of the instructional sequence. This file is assumed to exist on unit 1 as a manuscript filed under the name *STU*NNN, where NNN is the pre-assigned student number (000 to 999). A student is "signed on" to the system by creating (using LAP6W) a manuscript containing his name and other pertinent data and saving this manuscript on unit 1. A cumulative record may then be maintained giving the name of the GLEE program executed, the label of each frame displayed to the student, and his reply. [The question presented to the student may also be retained by placing sense switch 2 on the computer console in a down position. This feature is quite useful when the questions are being generated randomly, and need to be seen to understand the student's replies.] The student file may later be printed out by the teacher,

and is accesible to data analysis and compression programs for summary.

Program Execution

After answering display (2), a pass is made over the GLEE program to be executed to build up a label/entry point table. Execution then commences with the first frame definition statement in the program. Eight blocks (4096 characters) of the GLEE program are kept in core at once, with additional blocks read in when required. This reading of overlay blocks, and the writing of the student history file (LAP6W's manuscript working area on unit 0 is used for scratch storage), constitute the only tape accesses during the instructional process. Since a tape access takes only about one-half of one second, occasional delays are considered tolerable. GLEEFUL may also be run from a disk, so that even these minor delays are eliminated. In order to facilitate the use of GLEEFUL from a remote Teletype, all questions and answers will be typed out if sense switch 0 on the console is down.

Error Handling

If an error is detected during execution of the program (e.g., branching to a non-existent label, syntax errors, etc.) a display of the following type appears:

(3)

```
***AN ERROR HAS BEEN DETECTED.
CONTEXT:
!IF (A .LT. B) !GOTO C
```

The last line of the display shows the segment of the program where the error was detected, with a solid block inserted immediately following the error. Strike RETURN to return to LAP6W, at which time the error can be corrected.

Student Control

Using only the facilities described up to this point, the student is essentially a slave to the program. He can cause only those branches to occur which the teacher specifically allowed for in the GLEE program. It is felt, however, that often it is desirable to give the student somewhat more flexibility as to when he starts a new sequence of steps, and exactly what sequence is to be performed. To allow for this, there are two special commands which may be entered by the student in lieu of a normal answer. The first command consists of a '↑' followed by the label of a frame definition statement in the GLEE program. This causes the branching stack to be re-initialized to contain only that label, and an immediate branch to the corresponding statement is made. Thus by giving him the labels of several major sub-sections of the program, the teacher may allow the student to branch at any time without having to add a list of possible branches to every statement. If the student attempts to branch to a non-existent label, the display

(4)

```
***THERE IS NO SUCH LABEL
```

appears and the attempted branch is ignored.

The second special command which the student may enter consists of a '!' followed by the name of a GLEE program. This command causes the current program to be terminated and the named program to be entered. Thus to branch to program "NAME", the student may enter '!NAME' at any time. If program "NAME" is not found on unit 1, display (1) appears requesting a program to execute. If only the single character '!' is entered, a return is made to the LAP6W system on unit 0.

IV. ADDITIONAL SYSTEM FEATURES

When implementing a system with limited resources available, it is inevitable that certain arbitrary decisions must be made regarding the features which are to be included. The GLEEFUL system is certainly not lacking in such decisions. Desirable features which were omitted from the implementation because of space limitations include parameterized functions, arrays, and string-valued variables. There are, however, some additional features which adhance the usefulness of the system. These extensions will be described in this section.

Answer-Matching Mechanism

Perhaps the most important feature yet to be described is the answer-matching mechanism. The answer string entered by the student is matched character-for-character with the generated answer string, with the following exceptions:

1) If the character '↑' appears in the generated answer, it will match any single character in the student's reply. For example, 'R↑N' matches any three-character string starting with 'R' and ending with 'N'. Thus the strings 'RAN' and 'RUN' would match, while 'RN' and 'RAIN' would not.

2) If the character '!' appears in the generated answer, it will match an arbitrary string of zero or more characters in the student's reply. For example, 'R!N' matches any string starting with 'R' and ending with 'N'. Thus the strings 'RN', 'RAN', 'RAIN' and 'RAGAMUFFIN' would all match. This feature may be used to "unanchor" the match from either the left or right side. For example, 'A!' matches any string which starts with an 'A', and '!A' matches any string ending in 'A'. When a '!' is appended to both the left and right of a generated answer, we obtain a keyword search. For example, '!WORD!' will match any reply containing the string 'WORD'. Several ordered keywords may be searched for, as in '!MADISON!WISCONSIN!',

which matches any reply in which the string 'MADISON' appears, followed (eventually the string 'WISCONSIN'.

3) If the character '.' appears in the generated answer, it matches the entire remaining reply and re-sets the scan pointer to the first character in the reply. For example, '!WASHINGTON.ADAMS.JEFFERSON!' will match any reply containing (in any order) the three specified strings. After a '.' is encountered, the ensuing search will be unanchored on the left, i.e., 'A.B' is equivalent to 'A.!B'. This special character may be thought of as an "and" operator, in that all substrings separated by '.'s must match for success.

4) An "or" operator is also available via the character ','. If any substring delimited by ','s matches, the answer match is considered successful. For example, '!GERMANY!,!DEUTSCHLAND!' will match an answer in which either of the two substrings appears. Note that the '.' operator has precedence over the ',' operator. Thus '!LEWIS.CLARK!,!ROGERS.HAMMERSTEIN!' will match any string which contains both 'LEWIS' and 'CLARK', or any string which contains both 'ROGERS' and 'HAMMERSTEIN'.

5) Finally, if the character '"' appears in the generated answer, the next character is taken "as is". This allows the above special characters to be searched for in an answer. A double-quote character is represented by the pair '""'.

Although these conventions are relatively simple, the author feels they give the GLEE user a rather powerful answer-matching capability.

Setting Time Limits

Another important feature which has not yet been mentioned is the ability to specify the amount of time available to the student to answer a question. The timing feature is enabled by setting sense switch 1 on the

computer console into a down position. Once this is done, the alloted time may be controlled by setting the system variable !TIME to the desired time in hundredths of a second. For example, calling a function which contains the statement

```
!TIME = 1000
```

would cause the time limit to be set to 10 seconds. If the student does not complete his answer in the specified time, the system proceeds as if he answered with the character '\'. This character (which may be used in place of 'RETURN' in a message, and cannot be entered directly by a student) may be used in an answer group to explicitly check if the alloted time was exceeded. For example, the following GLEE segment gives the student 20 seconds to type his name:

```
#NAME   SETIM  'TYPE YOUR NAME'      <'\':TIMOUT><:OK>
#TIMOUT CLRTIM 'YOU TOOK TOO LONG!' <:NAME>
$SETIM  !TIME = 2000
        !EXIT =
$CLRTIM !TIME = 0
        !EXIT =
```

The computation of the alloted time is performed modulo 4096, so that the maximum alloted time is 40.95 seconds. However, if !TIME is set to 0, or if sense switch 1 is not down, the student may take as long as he likes to answer the display.

Program Branching

Another useful feature provided to the GLEE user is the ability to branch to another GLEE program. This is done by preceding the program name with a '!' and placing it at the end of the label list of an answer

group. For example, <:GOODBY,!ARITHTIC> would cause a branch to be made to the frame labeled 'GOODBY', followed by a branch to the program 'ARITHTIC' on tape unit 1. The branching stack will be cleared, and all labels, entry points and user-defined variables in the original program are lost. However, the system variables !TIME, !REM and !VAL (see below) retain their values and may be used to pass information to the called program (these variables, along with all user-defined variables, are initialized to zero when entering the system from LAP6W). Execution continues with the first frame definition statement in the new program. If the specified program is not found on unit 1, display (1) of Section III will appear, asking for a program name. Executing a branch to the single character '!' will cause a return to the LAP6W system on unit 0 to be made. If student replies are being saved, it is recommended that a return to LAP6W be made (either via branching to '!' or via the special student command '!') to assure that the student history file is properly closed.

Additional System Variables

In addition to !TIME and !EXIT, which have already been discussed, there are several other system variables which are available to the GLEE user:

!REM: Always contains the remainder of the last integer division.

!VAL: Contains the integer conversion of the last answer. If the answer was non-numeric, !VAL contains zero.

!ANS: The value of this variable is the same as !VAL. When !ANS appears in an expression, however, the string entered as the last answer is added to the current message or answer being built. For example, the following always displays the answer to the previous question:

```
#DLAST  'LAST ANSWER: \' !ANS <:DLAST>
```

!LSW: Contains the value currently set in the left switches.

!RSW: Contains the value currently set in the right switches.

All system variables (as well as the system command !GOTO) may be abbreviated to their first three characters (!EX, !GO, !RE, etc.).

Missing Statement Segments

The only remaining system conventions to be discussed concern what is done when required statement segments are missing. For example, it has already been stated that when there are no answers in an answer group (e.g., <:GO>), any reply is considered a match and the associated branches are made. [We now know that this is equivalent to including the answers '!' or !ANS in the answer list. Note that this is not the same as the null answer '', which matches only if the student struck RETURN with no answer string.] If no labels are present in an answer group (e.g., <'A':>) and the answer matches, the label of the next frame to be executed is taken off the branching stack (if the branching stack is empty, the error display appears). If there are no answer groups present in a frame description statement, it is as if the null answer group <:> were present. If there is nothing present where an expression is required (e.g., !EX=), a value of zero is assumed.

Finally, if the message in a frame description statement evaluates to the null string, nothing is presented to the user, but answer checking and branching

proceeds as usual. If the message expression evaluates to a value of zero (including the case where no expression is present), the student's reply to the previous frame is used for answer checking. If the value of the message expression is non-zero, it is converted to a string, preceded with a '\' (for uniqueness), and used for answer checking. This is a rather ad hoc method of providing a computed branch ability. Branches may be made on the basis of a computation rather than being solely based on the student's answers. For an example of the use of this feature, consider the following program segment:

```
#BRANCH TEST <'\1' :LOW> <'\2' :AVRAGE> <'\3' :HIGH>
$TEST   !IF (RIGHT<10) !EX=1
        !IF (RIGHT<20) !EX=2
        !EX=3
```

BRANCH will cause a branch to LOW, AVRAGE or HIGH to occur as a function of the current value stored in the variable RIGHT.

V. AN EXAMPLE

Several sample GLEE programs have been written to illustrate various aspects of the system. They include an arithmetic drill which uses the time feature to continually "push" the student; a drill routine in the parsing of simple English sentences (generated randomly); a program that leads the student step-by-step through the solution of (randomly generated) simultaneous linear equations; and a drill in English-to-German and German-to-English translation which generates sentences whose complexity is a function of student performance. This last program was selected for inclusion in this paper to illustrate the relative ease with which simple generative programs may be written even in non-numeric problem domains.

The program, which appears in Appendix A, generates either a German or English sentence and asks for a translation into English or German. If the reply is correct, a positive feedback comment is displayed to the student. If the reply is wrong, the correct translation is given. The simple grammar given below is used for sentence generation:

English:

Sentence:	S	→ NP VP
Noun Phrase:	NP	→ AR N / AR A N
Verb Phrase:	VP	→ PAD IV / PAD TV NP
Possible Adverb:	PAD	→ (NULL) / AD
Article:	AR	→ 'THIS' / 'THE' / 'EVERY'
Adjective:	A	→ 'FUNNY' / 'BIG' / 'LITTLE' / 'PRETTY'
Noun:	N	→ 'WOMAN' / 'CAT' / 'TEACHER' / 'COW'
Adverb:	AD	→ 'ALWAYS' / 'OFTEN' / 'SELDOM' / 'NEVER'
Intrans. Verb:	IV	→ 'SLEEPS' / 'RUNS' / 'PLAYS'
Trans. Verb:	TV	→ 'KISSES' / 'SEES' / 'HITS'

German:

```
Verb Phrase:     VP → IV PAD / TV PAD NP
Article:         AR → 'DIESE' / 'DIE' / 'JEDE'
Adjective:       A  → 'LUSTIGE' / 'GROSSE' / 'KLEINE' / 'SCHONE'
Noun:            N  → 'FRAU' / 'KATZE' / 'LEHRERIN' / 'KUH'
Adverb:          AD → 'IMMER' / 'OFT' / 'SELTEN' / 'NIE'
Intrans. Verb:   IV → 'SCHLAFT' / 'RENNT' / 'SPIELT'
Trans. Verb:     TV → 'KUSST' / 'SIEHT' / 'SCHLAGT'
```

This grammar is probably overly simple for realistic use (e.g., umlauts on German words are ignored, and the gender problem has been suppressed by selecting all feminine nouns), but it serves to illustrate what can be done with such a technique. [I might note that even with this simple grammar, we have moved well beyond the limits of "tabled" teacher-supplied problems--the program can generate 109,800 different sentences to be translated.]

The program generates a random number for each decision which must be made during the generation process, including the choice of which language to initially generate for a given frame. The choices for NP, VP and PAD in the above grammar are constrained by the level of student performance. Simple sentences will ge generated initially (e.g., "The cat plays."), with the sentences gradually becoming more complex as performance improves (e.g., "This funny teacher kisses every pretty cow."). Once these parameters have been determined, a call to the function 'S' will cause a sentence to be generated and output. The type of sentence generated is controlled through the variable ENGERM (-1 for English, 1 for German). Once the above grammars had been worked out, the translation to GLEE code was a purely mechanical task. The entire program took about one hour to code.

It should be noted that the answer-checking performed by this program is minimal. The answer must be entered exactly as generated for a match to occur. (Actually, since the answer is surrounded by a '!...!' pair, the student could type something like "I THINK IT IS <ANSWER> BUT I'M NOT SURE" and still have a correct reply). The answer-checking could be extended to check for keywords and partially correct answers, but it was felt that this would obscure the generative aspects of the program.

A sample run of the program (with sense switch 0 down for Teletype output) is included in Appendix B. The corresponding student history file which was generated is given in Appendix C (sense switch 2 was placed down so that the questions appear in the file along with the answers).

VI. CONCLUSIONS

The author feels the described system represents a viable demonstration of the feasibility of generative computer-assisted instruction on a small computer. As mentioned earlier, it has several limitations (a humbling, though by no means unique, situation). In addition to the lack of parameterized functions, arrays and string-valued variables, the major remaining problems have to do with the rather small number of labels, entry points and variables permitted (204), and a relatively slow response time when a lot of computation must be done (up to several seconds). The first problem is somewhat alleviated by the ability to chain to another program--a large program may be sub-divided into several smaller segments. The occasional slow response time is due to the fact that the code is interpreted. This could be helped somewhat by writing a semi-compiler version of GLEEFUL which would perform all symbol table lookups before execution begins.

Even with limitations such as these, it is felt that a system of the kind described in this paper could be a useful tool for the small computer user. Small computers often lack a usable high-level language capability, and even when a FORTRAN or BASIC-like language is available, the necessary text storage requirements for CAI usage usually call for an inordinate amount of overlay programming. The advantage of using a system like GLEEFUL is that the user needn't worry about such programming matters as answer-matching and accessing mass storage directly. In this sense, the described system is, indeed, "gleeful" (after all, the language wasn't called the Generative Language for Use on Machines, was it?).

VII. ACKNOWLEDGEMENT

I would like to thank Professor Leonard Uhr of the University of Wisconsin Computer Sciences Department, both for initially introducing me to the possibilities of generative CAI languages, and for gently prodding me into writing this paper.

VIII. REFERENCES

[1] Uhr, L., "The compilation of natural language text into teaching machine programs," Proceedings Fall Joint Computer Conference 1964, 35-44.

[2] Uhr, L., "Teaching machine programs that generate problems as a function of interaction with students," Proceedings 24th ACM National Conference, 125-134, 1969.

[3] Hoffman, R. and Seagle, J., "A program oriented computer-based instructional procedure," Proceedings 24th ACM National Conference, 97-110, 1969.

[4] Peplinski, C., "A generating system for CAI teaching of simple algebra problems," Technical Report No. 24, Computer Sciences Department, University of Wisconsin, 1968.

[5] Wexler, J., "A self-directing teaching program that generates simple arithmetic problems," Technical Report No. 19, Computer Sciences Department, University of Wisconsin, 1968.

[6] Utall, W., Pasich, T., Rogers, M. and Hieronymus, R., "Generative computer assisted instruction," Communication No. 243, Mental Health Research Institute, University of Michigan, 1969.

[7] Siklossy, L., "Computer tutors that know what they teach," Proceedings Fall Joint Computer Conference 1970, 251-255.

[8] Koffman, E., "A generative CAI tutor for computer science concepts," Proceedings Spring Joint Computer Conference 1972, 379-389.

[9] Zinn, K., "A comparative study of languages for programming interactive use of computers in instruction," EDUCOM, 1969.

[10] Zinn, K., "Programming conversational use of computers for instruction," in Atkinson, R. and Wilson, H. (eds.), Computer-Assisted Instruction, Academic Press, 1969, 253-268.

[11] Wexler, J., "A generative remedial and query system for teaching by computer," Ph.D. Dissertation, University of Wisconsin, 1970.

[12] Wexler, J., "Information networks in generative computer-assisted instruction," IEEE Transactions on Man-Machine Systems, Vol. MMS-11, No. 4, Dec. 1970, 190-202.

[13] Carbonell, J., "Mixed-initiative man-computer instructional dialogues," Bolt Beranek and Newman Report No. 1971, 1970.

[14] Carbonell, J., "AI in CAI: an artificial intelligence approach to computer-assisted instruction," IEEE Transactions on Man-Machine Systems, Vol. MMS-11, No. 4, Dec. 1970, 181-189.

[15] Quillian, M. R., "Semantic Memory," in Minsky, M. (ed.), Semantic Information Processing, M.I.T. Press, 1968.

[16] LAP6W Manual, Laboratory Computer Facility, University of Wisconsin, 1972.

APPENDIX A

A Sample GLEE Program

```
    PM OF ENG-GERM   PAGE 01
LN=0001

0001 [SAMPLE DRILL IN ENGLISH-GERMAN / GERMAN-ENGLISH TRANSLATION.
0002 [USES A SIMPLE GRAMMER TO GENERATE SENTENCES TO BE TRANSLATED.
0003 [PROBLEM DIFFICULTY A FUNCTION OF PERFORMANCE.
0004
0005 #START   'THIS IS A DRILL IN ENGLISH-GERMAN TRANSLATION.
0006         UMLAUTS ON GERMAN WORDS SHOULD BE IGNORED.'
0007         <:MAIN>
0010
0011 #MAIN    'TRANSLATE FROM ' LANG1 ' TO ' LANG2 ':\\' S '.'
0012         <TRANS:RIGHT,MAIN> <:WRONG,MAIN>
0013
0014 [THIS ROUTINE GETS THE RANDOM PARAMETERS AND
0015 [  OUTPUTS THE NAME OF THE FIRST LANGUAGE.
0016 $LANG1   RLIM=4       [(I.E., GENERATE RANDOM #'S IN RANGE [0,3])
0017         RA1=RAND     [(2 VALUES ARE SAVED FOR ALL NP PARAMETERS
0020         RA2=RAND     [  SINCE THERE MAY BE TWO CALLS TO NP)
0021         RN1=RAND
0022         RN2=RAND
0023         RAD=RAND
0024         RLIM=3       [(SET RANGE TO [0,2])
0025         RAR1=RAND
0026         RAR2=RAND
0027         RIV=RAND
0030         RTV=RAND
0031         RLIM=2       [(SET RANGE TO [0,1])
0032         RNP1=0       [(IN CASE NOT READY FOR MORE COMPLEX NP)
0033         !IF (COUNT>1) RNP1=RAND
0034         RVP=0
0035         !IF (COUNT>2) RVP=RAND
0036         RNP2=0
0037         !IF (COUNT>3) RNP2=RAND
0040         RPAD=0
0041         !IF (C0UNT>4) RPAD=RAND
0042         ENGERM=2*RAND-1    [(-1: ENGLISH FIRST; 1: GERMAN FIRST)
0043         !IF (ENGERM=-1) !EX='ENGLISH'
0044         !EX='GERMAN'
0045
0046 [THIS JUST OUPUTS THE NAME OF THE SECOND LANGUAGE.
0047 $LANG2   !IF (ENGERM=1) !EX='ENGLISH'
0050         !EX='GERMAN'
0051
0052 [THIS GENERATES THE ANSWER (COULD BE EXPANDED TO CHECK FOR
0053 [  PARTIALLY CORRECT ANSWERS).
0054 $TRANS   !EX = SWITCH '!' S '!'
0055
0056 [THIS SIMPLY SWITCHES FROM ONE LANGUAGE TO ANOTHER.
0057 $SWITCH  ENGERM=-ENGERM
0060         !EX=
0061
0062 [COME HERE FOR A RIGHT ANSWER.
0063 #RIGHT   'RIGHT.' UPWT
0064
0065 [COME HERE FOR A WRONG ANSWER.
0066 #WRONG   'NO,\' SWITCH S '.\TRANSLATES TO\' SWITCH S '.' DOWNWT
0067
```

```
    PM OF ENG-GERM   PAGE 02
LN=0070

0070 [THIS UPWEIGHTS OR DOWNWEIGHTS THE COUNTER WHICH
0071 [  CONTROLS SENTENCE COMPLEXITY.
0072 $UPWT    COUNT=COUNT+1
0073          !EX=
0074 $DOWNWT  COUNT=COUNT-1
0075          !EX=
0076
0077
0100 [THE FOLLOWING IS THE SENTENCE GENERATOR FOR ENGLISH (ENGERM=-1)
0101 [  AND THE EQUIVALENT GERMAN (ENGERM=1) SENTENCES.
0102
0103 [SENTENCE:
0104 $S       RNP=RNP1          [(SET PARAMETERS FOR 1ST NP)
0105          RAR=RAR1
0106          RA=RA1
0107          RN=RN1
0110          !EX = NP ' ' VP
0111
0112 [NOUN PHRASE:
0113 $NP      !IF (RNP=0) !EX = AR ' ' N
0114          !EX = AR ' ' A ' ' N
0115
0116 [VERB PHRASE:
0117 $VP      RNP=RNP2          [(SET PARAMETERS FOR 2ND NP)
0120          RAR=RAR2
0121          RA=RA2
0122          RN=RN2
0123          !IF (ENGERM=1) !GO VPGERM
0124          !IF (RVP=0) !EX = PAD IV
0125          !EX = PAD TV ' ' NP
0126 $VPGERM !IF (RVP=0) !EX = IV PAD
0127          !EX = TV PAD ' ' NP
0130
0131 [POSSIBLE ADVERB (EITHER NULL OR A LEADING OR TRAILING ADVERB):
0132 $PAD     !IF (RPAD=0) !EX =
0133          !IF (ENGERM=-1) !EX = AD ' '  [(TRAILING ' ' FOR ENGLISH)
0134          !EX = ' ' AD                  [(LEADING ' ' FOR GERMAN)
0135
0136 [ARTICLE:
0137 $AR      !IF (ENGERM=1) !GO ARGERM
0140          !IF (RAR=0) !EX = 'THIS'
0141          !IF (RAR=1) !EX = 'THE'
0142          !EX = 'EVERY'
0143 $ARGERM !IF (RAR=0) !EX = 'DIESE'
0144          !IF (RAR=1) !EX = 'DIE'
0145          !EX = 'JEDE'
0146
0147 [ADJECTIVE:
0150 $A       !IF (ENGERM=1) !GO AGERM
0151          !IF (RA=0) !EX = 'FUNNY'
0152          !IF (RA=1) !EX = 'BIG'
0153          !IF (RA=2) !EX = 'LITTLE'
0154          !EX = 'PRETTY'
0155 $AGERM   !IF (RA=0) !EX = 'LUSTIGE'
0156          !IF (RA=1) !EX = 'GROSSE'
```

```
    PM OF ENG-GERM   PAGE 03
LN=0157

0157          !IF (RA=2) !EX = 'KLEINE'
0160          !EX = 'SCHONE'
0161
0162 [NOUN:
0163 $N       !IF (ENGERM=1) !GO NGERM
0164          !IF (RN=0) !EX = 'WOMAN'
0165          !IF (RN=1) !EX = 'CAT'
0166          !IF (RN=2) !EX = 'TEACHER'
0167          !EX = 'COW'
0170 $NGERM   !IF (RN=0) !EX = 'FRAU'
0171          !IF (RN=1) !EX = 'KATZE'
0172          !IF (RN=2) !EX = 'LEHRERIN'
0173          !EX = 'KUH'
0174
0175 [ADVERB:
0176 $AD      !IF (ENGERM=1) !GO ADGERM
0177          !IF (RAD=0) !EX = 'ALWAYS'
0200          !IF (RAD=1) !EX = 'OFTEN'
0201          !IF (RAD=2) !EX = 'SELDOM'
0202          !EX = 'NEVER'
0203 $ADGERM  !IF (RAD=0) !EX = 'IMMER'
0204          !IF (RAD=1) !EX = 'OFT'
0205          !IF (RAD=2) !EX = 'SELTEN'
0206          !EX = 'NIE'
0207
0210 [INTRANSITIVE VERB:
0211 $IV      !IF (ENGERM=1) !GO IVGERM
0212          !IF (RIV=0) !EX = 'SLEEPS'
0213          !IF (RIV=1) !EX = 'RUNS'
0214          !EX = 'PLAYS'
0215 $IVGERM  !IF (RIV=0) !EX = 'SCHLAFT'
0216          !IF (RIV=1) !EX = 'RENNT'
0217          !EX = 'SPIELT'
0220
0221 [TRANSITIVE VERB:
0222 $TV      !IF (ENGERM=1) !GO TVGERM
0223          !IF (RTV=0) !EX = 'KISSES'
0224          !IF (RTV=1) !EX = 'SEES'
0225          !EX = 'HITS'
0226 $TVGERM  !IF (RTV=0) !EX = 'KUSST'
0227          !IF (RTV=1) !EX = 'SIEHT'
0230          !EX = 'SCHLAGT'
0231
0232 [RAND - RAL, 1/72
0233 [GENERATES RANDOM #'S IN RANGE [0,RLIM-1]
0234 [   WHERE RLIM < 10001
0235
0236 $RAND    RSAVE = (821*RSAVE + 2113)/10000
0237          RSAVE = !REM
0240          !EX = RSAVE/(10000/RLIM)
0241
0242 [END RAND
0243
0244 [END ENG-GERM (7/72)
```

APPENDIX B

Output from the Program in Appendix A

```
ENTER NAME OF COURSE MANUSCRIPT.

TERMINATE ALL DISPLAYS BY STRIKING 'RETURN'.
>ENG-GERM


ENTER PRE-ASSIGNED STUDENT NUMBER IF
REPLIES ARE TO BE ADDED TO YOUR FILE.

STRIKE 'RETURN' IF NOT.
>123


THIS IS A DRILL IN ENGLISH-GERMAN TRANSLATION.
UMLAUTS ON GERMAN WORDS SHOULD BE IGNORED.
>


TRANSLATE FROM ENGLISH TO GERMAN:

THE TEACHER PLAYS.
>DIE LEHRERIN SPIELT.


RIGHT.
>


TRANSLATE FROM GERMAN TO ENGLISH:

DIE KUH RENNT.
>THE COW RUNS.


RIGHT.
>


TRANSLATE FROM GERMAN TO ENGLISH:

JEDE LEHRERIN RENNT.
>EVERY LEARNER RUNS.(?)


NO,
JEDE LEHRERIN RENNT.
TRANSLATES TO
EVERY TEACHER RUNS.
>OH, SO 'LEHRERIN' MEANS 'TEACHER', NOT 'LEARNER'!


TRANSLATE FROM GERMAN TO ENGLISH:

DIE KUH RENNT.
>THE COW RUNS.


RIGHT.
>I KNOW, I KNOW.
```

TRANSLATE FROM GERMAN TO ENGLISH:

DIESE KLEINE LEHRERIN RENNT.
>THIE LITTLE TEACHER RUNS<----------S LITTLE TEACHER RUNS.

RIGHT.
>

TRANSLATE FROM GERMAN TO ENGLISH:

JEDE SCHONE KUH SPIELT.
>HOW ABOUT 'EVERY PRETTY COW PLAYS'?

RIGHT.
>

TRANSLATE FROM GERMAN TO ENGLISH:

JEDE KUH SCHLAGT DIE FRAU.
>EVERY COW HITS THE WOMAN.

RIGHT.
>

TRANSLATE FROM GERMAN TO ENGLISH:

DIE GROSSE LEHRERIN SPIELT.
>THE BIG TEACHER PLAYS.

RIGHT.
>

TRANSLATE FROM ENGLISH TO GERMAN:

EVERY LITTLE TEACHER SEES EVERY LITTLE WOMAN.
>TOO HARD FOR ME!

NO,
EVERY LITTLE TEACHER SEES EVERY LITTLE WOMAN.
TRANSLATES TO
JEDE KLEINE LEHRERIN SIEHT JEDE KLEINE FRAU.
>OH

TRANSLATE FROM GERMAN TO ENGLISH:

DIESE FRAU SCHLAFT.
>!

APPENDIX C

Sample Student History File

```
    PM OF *STU*123   PAGE 01
LN=0001

0001 [SAMPLE STUDENT HISTORY FILE (STUDENT NUMBER 123).
0002
0003 [STUDENT INFORMATION GOES HERE.
0004
0005 [EVERYTHING AFTER THIS LINE IS GENERATED BY GLEEFUL.
0006
0007
0010 [ENG-GERM
0011
0012 #START   'THIS IS A DRILL IN ENGLISH-GERMAN TRANSLATION.
0013          UMLAUTS ON GERMAN WORDS SHOULD BE IGNORED.'
0014          ''
0015 #MAIN    'TRANSLATE FROM ENGLISH TO GERMAN:
0016
0017          THE TEACHER PLAYS.'
0020          'DIE LEHRERIN SPIELT.'
0021 #RIGHT   'RIGHT.'
0022          ''
0023 #MAIN    'TRANSLATE FROM GERMAN TO ENGLISH:
0024
0025          DIE KUH RENNT.'
0026          'THE COW RUNS.'
0027 #RIGHT   'RIGHT.'
0030          ''
0031 #MAIN    'TRANSLATE FROM GERMAN TO ENGLISH:
0032
0033          JEDE LEHRERIN RENNT.'
0034          'EVERY LEARNER RUNS.(?)'
0035 #WRONG   'NO,
0036          JEDE LEHRERIN RENNT.
0037          TRANSLATES TO
0040          EVERY TEACHER RUNS.'
0041          'OH, SO 'LEHRERIN' MEANS 'TEACHER', NOT 'LEARNER'!'
0042 #MAIN    'TRANSLATE FROM GERMAN TO ENGLISH:
0043
0044          DIE KUH RENNT.'
0045          'THE COW RUNS.'
0046 #RIGHT   'RIGHT.'
0047          'I KNOW, I KNOW.'
0050 #MAIN    'TRANSLATE FROM GERMAN TO ENGLISH:
0051
0052          DIESE KLEINE LEHRERIN RENNT.'
0053          'THIS LITTLE TEACHER RUNS.'
0054 #RIGHT   'RIGHT.'
0055          ''
0056 #MAIN    'TRANSLATE FROM GERMAN TO ENGLISH:
0057
0060          JEDE SCHONE KUH SPIELT.'
0061          'HOW ABOUT 'EVERY PRETTY COW PLAYS'?'
0062 #RIGHT   'RIGHT.'
0063          ''
0064 #MAIN    'TRANSLATE FROM GERMAN TO ENGLISH:
0065
0066          JEDE KUH SCHLAGT DIE FRAU.'
0067          'EVERY COW HITS THE WOMAN.'
```

```
    PM OF *STU*123   PAGE 02
LN=0070

0070 #RIGHT   'RIGHT.'
0071          ''
0072 #MAIN    'TRANSLATE FROM GERMAN TO ENGLISH:
0073
0074          DIE GROSSE LEHRERIN SPIELT.'
0075          'THE BIG TEACHER PLAYS.'
0076 #RIGHT   'RIGHT.'
0077          ''
0100 #MAIN    'TRANSLATE FROM ENGLISH TO GERMAN:
0101
0102          EVERY LITTLE TEACHER SEES EVERY LITTLE WOMAN.'
0103          'TOO HARD FOR ME!'
0104 #WRONG   'NO,
0105          EVERY LITTLE TEACHER SEES EVERY LITTLE WOMAN.
0106          TRANSLATES TO
0107          JEDE KLEINE LEHRERIN SIEHT JEDE KLEINE FRAU.'
0110          'OH'
0111 #MAIN    'TRANSLATE FROM GERMAN TO ENGLISH:
0112
0113          DIESE FRAU SCHLAFT.'
0114          '!'
```

BIBLIOGRAPHIC DATA SHEET	1. Report No. WIS-CS-73-177	2.	3. Recipient's Accession No.
4. Title and Subtitle IMPLEMENTATION OF A GENERATIVE COMPUTER-ASSISTED INSTRUCTION SYSTEM ON A SMALL COMPUTER			5. Report Date April 1973 6.
7. Author(s) Rick LeFaivre			8. Performing Organization Rept. No.
9. Performing Organization Name and Address Computer Sciences Department The University of Wisconsin 1210 West Dayton Street Madison, Wisconsin 53706			10. Project/Task/Work Unit No. 11. Contract/Grant No. GJ-36312
12. Sponsoring Organization Name and Address National Science Foundation Washington, D. C. 20550			13. Type of Report & Period Covered 14.

15. Supplementary Notes

16. Abstracts

This paper gives a brief overview of several "traditional" approaches towards computer-assisted instruction (CAI). A new CAI language is then described which allows the user to write programs which generate both questions and answers with a minimum of programming effort. An implementation of the described system is discussed in detail, followed by a description of a sample program which drills the student in German-English translation by generating sentences whose complexity is a function of performance.

17. Key Words and Document Analysis. 17a. Descriptors

Computer-Assisted Instruction
Generative CAI
Mini-computers

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

18. Availability Statement	19. Security Class (This Report) UNCLASSIFIED	21. No. of Pages 39
	20. Security Class (This Page UNCLASSIFIED	22. Price