KNEJI

An Interactive Interpreter that the User can Extend

Philip L. Karlton
Joseph M. Newcomer

Department of Computer Science
Carnegie-Mellon University
September 9, 1974

ABSTRACT

KNEJI is an interpreter intended primarily for use by programmers. It uses a SAIL-like syntax, but the semantics are somewhat more constricted. An editor is inside KNEJI and allows the creating, editing and saving of programs. KNEJI only exists as a .REL file that can be loaded with the user's program. One of its chief functions is to act as an expression evaluator for External (in the SAIL sense) routines.

This research was supported in part by the Advanced Research Projects Agency of the Department of Defense under contract no. F44620-73-C-0074 and monitored by the Air Force Office of Scientific Research.

TABLE OF CONTENTS

I. Introduction

KNEJI is a system of SAIL procedures which allows the user to write programs in a simple but reasonably powerful language and run them -- either from direct TTY commands or through program (SAIL) control. The system is designed to allow fast and easy circuit through the program-compile-execute loop, particularly in situations where the user is not sure at the outset what form his program will take. More importantly it allows the user to interactively call SAIL routines that he has linked to the interpreter and evaluate the expressions that he is passing as arguments.

One version of KNEJI (CALC) can be used as a super desk calculator with no knowledge of SAIL required. Another version (IGRAPH) that exists can be used for developing image files for the XGP. Documentation for it already exists.

The system consists of a line-oriented editor-monitor (called, henceforth, the "editor"), a compiler (almost invisible to the user) and an interpreter for executing (running) the programs.

The compiler and interpreter are designed so that extensions can be made easily, thus allowing for individual needs. The interface is explained in Section V.

Acknowledgements: Lee Erman wrote the original interpreter (EOI) on which KNEJI was loosely based. Some of this documentation has been bodily stolen from EOIMAN: the manual for that interpreter. Richard Johnsson was indispensible in debugging and providing many of the string hacking functions which allow KNEJI to run at other than a snail's pace. Joe Newcomer extended the system to allow dynamic data typing, local variables, parameterized program calls, sticky line numbers, and several other improvements.

II. The Language of KNEJI

A **program** consists of a set of **lines**. Each line consists of one or more **statements** and is terminated by a CR. Every statement is terminated with a semi-colon, except the last one on a line which has an implicit semi-colon. There may be (currently) twenty (20) programs, each with up to (currently) twenty (20) lines. These limits are easily changed, but they do require a recompilation of the system. Ask, and it might be done.

When a program is executed, execution begins with the first line of the program and continues sequentially except when changed by GOTO's or RETURN's. After the last line is executed, the program exits and returns to whatever program called it (either another KNEJI program, the editor, or an external procedure).

II.1. Data types

There are only two data types internally, real and string. Whenever an explicit integer value is needed, it is obtained from a real by truncation of the fraction part, and from string by using the ASCII value of the first character of the string (this is consistent with SAIL usage). A string is converted to real by first coercing it to an integer and then to real; a real is converted to a string by first converting it to an integer and then using that integer as the ASCII value of a one-character string.

II.2. Statements

Currently, these are the statements that KNEJI recognizes.

II.2.a. Setformat

When the system displays the value of a real expression, it does it essentially in the G format from Fortran. (The "whatever is right" format.). Use this statement to change the precision of the output display.

II.2.a.i. Syntax
SETFORMAT <expression>

II.2.a.ii. Semantics
The value of the <expression> truncated to an integer becomes the number of digits to the right of the decimal point on output.

II.2.b. Print

This statement can be used to display the value of variables on the console.

II.2.b.i. Syntax
PRINT <variable list>

II.2.b.ii. Semantics

The name and value of each variable in the list is displayed on the user's console.

II.2.c. Conditional

The IF-THEN statement allows conditional execution of one statement. It is not yet possible to use compound statements.

II.2.c.i. Syntax

IF <expression> THEN <statement>

II.2.c.ii. Semantics

If <expression> has a non-zero value then <statement> will be executed.† The results of using the boolean operators on <expression>s are zero or non-zero. Using boolean values as arithmetic (real) operands is undefined.

II.2.d. Iteration

The WHILE-DO statement allows iterative execution control of one statement. It is not yet possible to use compound statements.

II.2.d.i. Syntax

WHILE <expression> DO <statement>

II.2.d.ii. Semantics

As long as <expression> has a non-zero value then <statement> will be executed. Note: The execution of a loop may be interrupted; see Section IV.

II.2.e. Assignment

This statement is used to assign a value to a variable.

II.2.e.i. Syntax

<variable> ← <expression>‡

II.2.e.ii. Semantics

The value of <expression> is computed and the result is assigned to <variable>. If the result is of type string then the variable takes on a string value; if it is of type real then the variable takes on a real value.

---

† Expressions are converted to integer by truncation from type real before being tested for zero in a boolean context. This means that if the absolute value of <expression> is less than 1, then <statement> will not be executed.

‡ <expression> stands for either a real or string expression throughout this documentation.

II.2.f. Go to

To begin executing a different line (not statement) in the same program.

II.2.f.i. Syntax
GOTO <expression>

II.2.f.ii. Semantics
<expression> is evaluated and truncated to an integer. The result is the line number to next be executed. Execution will begin with the first statement at the destination line number.

II.2.g. Execute

This statement is used to execute a program.

II.2.g.i. Syntax
EX <expression> (parm1, ...)

II.2.g.ii. Semantics
<expression> is evaluated and converted to an integer. If the result equals zero then the editor is called as a subroutine. If the result equals -1 then the current KNEJI program returns (as if the last line had been executed). Otherwise, the interpreter is recursively called to execute program number <expression>. If parameters are specified in the call, they are made available for LAMBDA binding in the called program; if no parameters are specified, the parentheses must be absent. If the executed program returns an error flag, then the current program also returns immediately with an error flag. See Section IV.

II.2.h. Editor

The ED statement provides a shorthand statement for calling the editor-monitor to edit a specific program.

II.2.h.i. Syntax
ED <expression>

II.2.h.ii. Semantics
The expression is evaluated and coerced to an integer; the editor is called to edit that program.

II.2.i. Return

II.2.i.i. Syntax
RETURN

II.2.i.ii. Semantics

Execution of the current program is terminated and control returns to its caller.

II.2.j. Block structure

The LOCAL statement allows the user to declare variables local to the current incarnation of the program.

II.2.j.i. Syntax
LOCAL <variable list>

II.2.j.ii. Semantics
Each of the variables named in the list is declared local to the current incarnation of the current program. The previous (outer) value is copied into the new (inner) variable. Upon return from the program, the previous (outer) values for all declared variables are restored. Note: LOCAL is an executable statement; it is perfectly possible to have a statment "IF <cond> THEN LOCAL A". If the program attempts to declare the same variable LOCAL more than once in the same program incarnation, the second attempt is ignored (typical case: the LOCAL is within a loop). If a LOCAL statement is executed in immediate mode, it declares the variables LOCAL in the previous program (if this occurs at the top level it is meaningless, but you can do it anyway).

II.2.k. Parameter binding

The LAMBDA statement (the symbol λ may also be used) allows formal parameters to be specified and bound to the actual parameters passed to the program.

II.2.k.i. Syntax
LAMBDA <variable list>

II.2.k.ii.
λ <variable list>

II.2.k.iii. Semantics
Each time the LAMBDA statement is executed, each variable in the <variable list> is declared local (see "Block structure") and its value is bound to the actual parameter corresponding to it. If there are more actuals than formals, the extra actuals are ignored. If there are fewer actuals than formals, an error condition is reported. Note that LAMBDA is an executable statement, and may be executed conditionally; also, two LAMBDA statements in succession act independently, each binding the formals to the actuals. With clever programming it is possible to create programs which take a varying number of parameters and other, much stranger, sorts of programs.

II.2.l. Comment

II.2.l.i. Syntax
! <string without ;>

II.2.l.ii. Semantics
Ignored.

II.2.m. Value

II.2.m.i. Syntax
← <expression>

II.2.m.ii. Semantics
<expression> is evaluated and the result is displayed on the user's console. No type conversion is performed.

II.2.n. String Value

II.2.n.i. Syntax
\ <expression>

II.2.n.ii. Semantics
<expression> is evaluated and coerced to a string, and the result is displayed on the user's console.

II.2.o. Read

This statement is used to set variables at execution time.

II.2.o.i. Syntax
READ <variable>

II.2.o.ii. Semantics
Execution is suspended until the user types a line on the keyboard. No prompt is issued; the user should issue one with the \ or ← command. The expression is evaluated and the result is assigned to <variable>.

II.2.p. String read

This statement allows the user to read in strings without having to put quote marks around them.

II.2.p.i. Syntax
STRINGREAD <variable>

II.2.p.ii. Semantics
Execution is suspended until the user types a line on the

keyboard. No prompt is issued. The line is effectively surrounded by double quotes (and each internal quote is doubled) and assigned to <variable>.

II.2.q. Extension

II.2.q.i. Syntax
<external procedure name> [(<parm1>, <parm2>, ... <parmn>)]

II.2.q.ii. Semantics
The external procedure is called after evaluating each of the expressions being passed as parameters, i.e. call by value. Where the external procedure expects a string parameter, a string expression must appear, and similarly for real parameters.

II.2.r. Untrace

Should you be so unfortunate as to get the debugging version of KNEJI, you will have to turn off the internal tracing features. Should you be a KNEJI debugger, you are also interested in these commands.

II.2.r.i. Syntax
UNTRACE <expression>

II.2.r.ii. Semantics
The expression is evaluated and determines which implementor tracing aids to turn off. A value of 0 turns off all tracing. A negative value prints a list of the trace options.

II.2.s. Trace

This turns on various tracing features in the debugging version of KNEJI. Ordinary users will find that standard KNEJI does not understand this statement.

II.2.s.i. Syntax
TRACE <expression>

II.2.s.ii. Semantics
Evaluates the expression and coerces it to an integer, which determines which debugging trace features to turn on. A value of 0 turns on all tracing. A negative value prints a list of the trace options.

II.2.t. Dump

If you have the debugging version of KNEJI, this allows you to dump certain random pieces of (presumed) useful information. Standard KNEJI does not understand this statement.

II.2.t.i. Syntax
DUMP <expression>

II.2.t.ii. Semantics
Evaluates the expression and coerces it to an integer, which tells what internal information to disgorge. A negative value prints a list of the dump options.

II.2.u. DDT

This statement is only available in the debugging version of KNEJI. It allows direct access of DDT.

II.2.u.i. Syntax
DDT

II.2.u.ii. Semantics
Passes control directly to DDT. You are instructed how to get out when this occurs.

II.3. Operators

In order to evaluate expressions, it is nice to have operators to operate on the variables. All binary operators are infix. The unary operators with one exception are prefix. Operators will coerce their operands to the indicated data type.

Operator	Prec.	Coercion	Meaning
SQRT	4	real	Square Root
CVS	4	real	Convert to String. This function takes an <expression> as an argument and converts it as it would for output to the console.
ABS	4	real	Absolute Value
SIN	4	real	Sine (argument is in degrees)
COS	4	real	Cosine
ASIN	4	real	Arcsine (value is in degrees)
ACOS	4	real	Arccossine
ATAN	4	real	Arctangent
RANDOM	4	real	$0 \leq$ random value ≤ 1
LOG	4	real	Natural Logarithm
EXP	4	real	power of e (2.71828...)
+	4	real	Unary Plus
-	4	real	Unary Minus
NOT (¬)	4	real	Not
↑	3	real	Exponentiation
*	3	real	Multiplication
/	3	real	Division
&	3	string	Concatenation
MIN	3	real	Minimum

MAX	3	real	Maximum
MOD	3	real	Modulo (arguments are coerced to integer before computing result, which is then coerced to type real)
+	1	real	Addition
-	1	real	Subtraction
=	1	real	Equality (For all the following boolean operators, the value is non-zero if true)
<	1	real	Less Than
>	1	real	Greater Than
GEQ (≥)	1	real	Not Less Than
LEQ (≤)	1	real	Not Greater Than
NEQ (≠)	1	real	Not Equal
EQU	1	string	String equality
AND (∧)	0	real	And
OR (∨)	0	real	Or
[]	4	string	Substring. This is a postfix operator that actually has two forms. ([<expression> TO <expression>] and [<expression> FOR <expression>]). The expressions are evaluated, and truncated to integers. The first <expression> is the position in the string that the substring starts. The second <expression> is either the position of the last character of the substring or it is the length of the substring. For more information see the SAIL manual. Example: "ABCDEFG"[2.8 TO 4] has a value of "BCD".

All of the above operators of precedence 4 are unary operators; the rest are binary operators.

II.4. Operands

Operands can be either constants or variables, and may be of type string or real. Identifiers and constants follow the conventions of SAIL, and of most other languages; the symbol "@" in a numeric constant indicates "times ten to the power", e.g., 1@2 is equivalent to 100.

String constants are delimited by double quotes, e.g., "ABC"; as might be expected, to get a double quote inside you use two, e.g., "Type ""This is a string""".

The special symbol π (pi) is already set up in the symbol table, and has the value 3.1415927.

III. The Editor-Monitor

The editor-monitor (called the "editor") allows the user at the TTY to easily [sic]:

1) write and modify programs in the language of KNEJI,
2) save and restore KNEJI programs on the disk,
3) save and restore variables on the disk.

While in the editor, there is always a current program. When the editor is called via the EX statement or ED with a program number of 0 it prompts for a program number:

PNUM←

Legal responses are:

CR	the lowest numbered program which is undefined (no lines in the program) becomes the current program for editing.
<positive integer>	becomes the current program.
E	Exit.

The editor then responds with *, the signal that it is waiting for a command.

III.1. Line Changing Commands

In the following discussion, <lnum> is a line number; <pnum> is a program number; <v> is a variable; <name> is a filename. Brackets around a term indicate it is optional. In any of the following commands a line number may be given in the form <pnum>.<lnum> which has the additional effect of changing the "current" program to <pnum>. Where <lnum> is optional, typing in the line number as <pnum>. will change the current program and act as if a null line number had been typed in.

III.1.a. I[<lnum>[,<inc>]] Insert a line in the current program

The editor prompts with a line number (in the form <pnum>.<lnum>) and will accept everything typed at it until a CR. If an existing line number is given, the editor will use the current increment to determine a new line number. The increment is set to 100 every time the editor is entered, and can be changed by providing a second argument to I.

If <lnum> is omitted, the new line number will be the last line of the program plus the current increment.

Whenever a new line is inserted into a program, it is immediately compiled into the object language of KNEJI (with which the user need not be concerned). If the line contains a syntax error, the compiler will complain and print the line with !...! immediately following the character recognized as being illegal. Following this, there will be a message of the form:

COMPILE: <message>.

You will now be put in alter mode for this line, and may correct it. KNEJI will insist upon a syntactically correct line before allowing you to

do anything else. In order to defeat this wonderful feature, you may do a "Q" (Quit) command to alter mode.

Use an <altmode> to escape from Insert mode.

Note: More than one statement (See Section II) can appear on a line.

III.1.b. D<lnum>[:<lnum>] Delete a line or range of lines in the current program

The <lnum>'th line is deleted. If a range of lines is specified, all lines in that range (inclusive) are deleted; note that in this case neither the lower nor upper lines need exist, since they only specify limits.

III.1.c. R<lnum> Replace a line in the current program

The <lnum>'th line is replaced. <lnum> must be present.

III.1.d. A<lnum> Alter a line in the current program

Enter SOS-Alter mode on the <lnum>'th line. See the SOS manual for details. The commands that involve breaking accross lines have no function in KNEJI. The line is recompiled just as in Insert.

III.2. I/O Commands

The comments about <lnum> and <pnum> from Section III.1 also apply to this section.

III.2.a. P[<lnum>[:<lnum>]] Print a line or range of lines

The <lnum>'th line is printed on the user's console. If <lnum> is omitted, then the entire program is printed. If a range is specified, all lines in that range (inclusive) are printed; note in this case that the lines specified in the range need not exist, since these line numbers specify only limits.

III.2.b. L[<pnum>] List a program on the line printer

Title: is prompted for, and any characters up to a CR are accepted as a heading for the listing. The current time and date will also appear on the listing.

The <pnum>'th program is listed (with line numbers). If <pnum> is omitted then all defined programs will be listed.

III.2.c. >[<pnum>][<name>] Output source to the disk

The source program is output to the disk as a text file with the name <name>. If the file already exists then a warning is issued.

If <pnum> is omitted then the entire program space is written out. This file also contains markers so that if it is read in it can determine in which program space to put back the source lines. If <name> is omitted, then the output file name becomes KNEJnn.KNJ where nn equals the program number. (If <pnum> has been omitted then nn is 0.)

III.2.d. <[<pnum>][<name>] Input source from the disk

The file <name> is read from the disk and the lines on that file are added to the end of the program <pnum>.† If you want to replace a program, first clear it (See Section III.4.d), and then read in your new program. If <pnum> is omitted then the input routine expects to find indicators in the file to put the input lines into the correct program space. It is possible to use LINED or TECO on the file and edit it outside of interpreter. If <name> is omitted then the same naming conventions described for the > command are used.

As each line is read in from the disk, it is compiled. If a line has a syntax error, the compiler gives its message and prints the line that is in error. Correct these errors before attempting to execute your program.

Note that lines which are read in are renumbered with the current increment.

III.3. Variable Changing Commands

KNEJI supports two types of variables: real and string. When an identifier is first encountered it is automatically assumed to be real. If it is subsequently assigned a string value, it becomes type string.‡

III.3.a. V← Set all Variables

This command allows the user to set each of the variables that have been declared so far. KNEJI gives the name of each variable and a prompt of "←" and expects a new value from the terminal. If the user responds only with a CR then the value will remain unchanged. A response of <altmode> while setting the variables will cause the rest of the variables to be skipped.

The only legal response to setting variables is a constant. String constants must have surrounding quote marks.

---

† Nothing is ever deleted from a program while doing input; the lines that are read are added to the program.

‡ There actually exists a third type of identifier (the External SAIL routines) that is discussed more fully in Section V.

Of course, individual variables can be examined and set in Immediate Mode (See Section III.4.f.) by executing the KNEJI statements that accomplish display and assign to variables.

III.3.b. V>[<name>] Output the Variable Values to Disk

The value of each of the variables is output the the file <name>. If <name> is null then the filename becomes KNEJV.KNJ. This file is a text file and hence can be edited with LINED or TECO.

III.3.c. V<[<name>] Input Variable Values from the Disk

The file <name> (KNEJV.KNJ if <name> is null) is read and the variables that are defined there are set to the new values. Variables that already have values will take on new values from the input file.

III.4. Control Commands

These are the commands that do not fit under the previous sections.

III.4.a. Z<pnum> Change the current program number

The current program number becomes <pnum>. If <pnum> is out of bounds then the PNUM← prompt is repeated.

III.4.b. E Exit from the Editor

The Editor can be entered recursively or even be called from one of the programs.

III.4.c. <altmode> Exit from the Editor

Identical to the E command. This means that inside the editor one can change the program out from underneath the one that is being executed. Another way to get into the Editor is by typing any character while it is executing (See Section IV). When the editor is exited it returns to the calling context. If the outer level is exited, it asks if you have remembered to save your program.

III.4.d. C Clear Work Space

KNEJI will prompt with N,P,<n>: . Give the appropriate response.

III.4.d.i. N(one)
Forget it; I have no intention of wiping out all my hard work.

III.4.d.ii. P(rogram)
Clears all the programs.

III.4.d.iii. <n> Clear Program <n>

Program <n> is cleared. If <n> happens to be the current program then a prompt will be given asking what program you want to be editing after clearing the current program. This command comes in handy just before reading a program into a particular program space.

III.4.e. @<pnum> Execute a Program

Program <pnum> is executed. If <pnum> is omitted then the current program is executed. This command is only included for convenience since one can execute any program in immediate mode.

III.4.f. ! Enter Immediate Mode

KNEJI will prompt with a <tab>. Each line that is typed in will be compiled and executed immediately. There is no need to type the semi-colon (See section II.) at the end of the line. For multiple statements on one line it is necessary to put the semi-colon between each statement.

To escape from immediate mode, type an <altmode>.

IV. Asynchronous Breaks

While any KNEJI program is being executed, typing any character at the keyboard causes execution to be suspended after execution of the current line is terminated. A message of the form:

"Break before <pnum>.<lnum>.

is typed to indicate the next line to be executed. The user may respond with:

H Halt. The current program terminates with an error return.

C Continue. Continue execution of the KNEJI program.

anything else

Call the editor. After returning from the editor, the prompt is given again.

If "C" is given but no line <lnum> exists (due to a deletion while in the editor) then the message "Line gone. Can't Continue" is displayed and the options re-prompted.

V. User Extensions

V.1. What to Write

Since KNEJI lives as a REL file (currently on [A330PK01]/B) with unresolved external references, it is necessary for the user to define these references.† He does so by including the following:

```
entry;
begin "<block name>"
define MAXROUTINES=<number of external routines>;

preload!with <name1>, <name2>, ..., <nameMAXROUTINES>;
internal safe string array ENAME[1:MAXROUTINES];

preload!with <caseval1>, <caseval2>, ..., <casevalMAXROUTINES>;
internal safe integer array ROUTINES[1:MAXROUTINES];

preload!with <parm1>, <parm2>, ..., <parmMAXROUTINES>;
internal safe string array PARMS[1:MAXROUTINES];

internal integer EXTERNPOINTER;

external integer procedure SYMLOOKUP(string NAME);
external real procedure GET!REAL(integer INDEX);
external string procedure GET!STRING(integer INDEX);
external procedure SET!REAL(integer INDEX; real VAL);
external procedure SET!STRING(integer INDEX; string VAL);
external integer procedure GET!TYPE(integer INDEX);
external string procedure COMPILE(string SOURCE);
external recursive integer procedure EXECUTE(string OBJECT);

procedure STARTUP;
begin "STARTUP";
EXTERNPOINTER←MAXROUTINES;
<any initialization needed>
end "STARTUP";

require STARTUP initialization [2];

<your procedure declarations here>
```

† If the user plans to use break tables (See the Sail manual for more information) then he must include BRKSER.REL as a load file. The functions of GETBRK and RELBRK defined in there will allow the user to use break tables without interfering with those defined inside of KNEJI.

```
internal integer recursive procedure EXTEND(
        integer ROUTINE; real array A; string array S);
begin "EXTEND"
if not (1≤ROUTINE≤MAXROUTINES) then
        begin
          USERERR(0,1,"EXTEND called with illegal parameter"
                        &CRLF);
          return(0);
        end;
case ROUTINE of
      begin "PICK"
        ;       ! 0 IS NOT A LEGAL CASE;
        <case values in ROUTINES should index into this case statment>
      end "PICK";
return(1);
end "EXTEND";

end "<block name>"
```

V.2. Why It is Written

ENAME is used by KNEJI to find out the names of the routines that the user has defined. At compilation time, the external reference is matched against the list of names in this array. The index that is found for the match is used to pick values out of the next two arrays. ROUTINES should have the index in the case statement (or some other device that the user wants to use) of the procedure EXTEND. When her routines are called, this is how the particular routine wanted is identified. In PARMS is the information that KNEJI needs to compile code to evaluate the parameters that want to be passed to the user's routines. The information that is preloaded should be a string of S's and R's.† When KNEJI is compiling a statement, it uses this string to determine what type (and how many in what order) of parameters to expect for each external routine. Care should be taken when filling the array PARMS. EXTERNPOINTER is used by KNEJI to know the lengths of the Arrays.

The procedures GET!REAL and GET!STRING provide the user's routines with the values of the variables that KNEJI has defined. If the name of the variable is known, then the index into the symbol table can be found by SYMLOOKUP. If the variable name does not already exist, SYMLOOKUP will cause it to be entered into the symbol table. This is a handy way to have the external routines make values available to the KNEJI programs and vice versa. The procedures SET!STRING and SET!REAL store the value of their second argument into the variable indexed by the INDEX argument. Although the symbol table is available as an "external itemvar array VALS[1:'177]" the user should not manipulate this unless she is aware of the dire consequences which may ensue. The procedure GET!TYPE returns the current type of the variable of INDEX, so

† These must be in upper case.

that the appropriate value may be obtained without coercion (GET!REAL and GET!STRING always coerce the value to the indicated type). The integer returned is that obtained from TYPEIT; see the LEAP section of the SAIL manual. The procedures COMPILE and EXECUTE will do just that on their arguments. COMPILE expects a statement in string form; if the value returned is not equal to the one character string conaining rubout, then that value can be passed to EXECUTE.

The routine EXTEND is what KNEJI calls to have external (to KNEJI) routines executed. The first parameter is the case value the user provided in ROUTINES. The other two parameters contain the results of evaluating the expressions that were the parameters to the external routine. The values are packed into the arrays A and S starting from one, e.g. if the value of PARMS[ROUTINE]="RSR" then A[1] will contain the value of the **first** parameter; A[2] will contain the value of the **third** parameter; and S[1] will contain the value of the **second** parameter.

V.3. What has been Written

Currently, two systems have been written based on KNEJI.

V.3.a. IGRAPH

IGRAPH is an interactive system that allows the user to create pictures on the GDP's, and to create an image file for the XGP. See the documentation: IGRAPH.XGO[A810KG00]/B (which must be printed using the LOOK command file @IGRAPH.XMD[A810KG00]) or for the line printer use the file IGRAPH.DOC[A810KG00]/B.

V.3.b. CALC

This is a super desk calculator that uses KNEJI for expression evaluation. It contains almost the minimum code needed in order to loaded with KNEJI. It looks essentially like immediate mode in the interpreter except for two differences. The prompt is "<> " and it puts a "←" in front of the first statement on a line if it does not contain a "←", "\" or call one of the built in statements. This has the side effect of typing back at you the value of any expression you type in. Of course, you still can assign values to variables, SETFORMAT, etc.

In order to get to the editor type "EX 0" or "ED n" to CALC. Now you can start writing programs that can be called from CALC. Get back to CALC the normal way, by giving the E or <altmode> command to the editor. Of course, you could call the calculator by executing the statement "CALC." Now you are in the calculator recursively which will not hurt you, but why?

An <altmode> to CALC will exit it back to the monitor. Do not forget to save the programs you wrote.

INDEX