```
************************************************
************************************************
***                                          ***
***  Name:                                   ***
***                                          ***
***  Project:    1       Programmer:  MWK    ***
***                                          ***
***  File Name: LEAP.WRU[DOC,AIL]            ***
***                                          ***
***  File Last Written:  21:08 3 Jul 1973    ***
***                                          ***
***  Time: 19:39         Date: 15 Jul 1973   ***
***                                          ***
***           Stanford University            ***
***       Artificial Intelligence Project    ***
***        Computer Science Department       ***
***           Stanford, California           ***
***                                          ***
************************************************
************************************************
```

# A USERS GUIDE TO THE LEAP RUNTIME ROUTINES AND STRUCTURES

by K. Pingle

Modified FALL 1972 to reflect changes by Jim Low

WARNING: This document is rated X and is only for the use of adults with very strong stomachs. It is provided for people who have to debug leap programs so they have some idea of what is being done to them and the data structures they might want to look at. The facts provided here are NOT sufficient to allow hackers to modify things from their programs. The information provided may change, or become incorrect, at any time.

## 1. THE USER TABLE

When initialized, SAIL creates a user table in your core image with information for the runtime routines. This table, whose address is contained in the cell named GOGTAB, is normally placed in an AC('15) when leap is called and indexed into. The global model's table is always in a fixed location starting at GLUSER. If you are inside LEAP, or have just left it, a pointer to the user table is in AC '15. If it is, or was, a global model operation (see bits in section 2), a pointer to GLUSER is in AC 7. Below is a list of the more interesting entries in the user table. Be warned that the index may change at any time. Those entries with *** following the index also have meaning in the global user table

| INDEX (octal) | NAME | DESCRIPTION |
|---|---|---|
| **** | ***** | ******************************************** |
| 0 | UUO1 | This is the return address for the last call of LEAP, which was cleverly removed from the stack so you couldn't find it. |
| 302 | PDL | IOWD SIZE,BASE - the initial system pdl |
| 303 | SPDL | IOWD SIZE,BASE - the intial string pdl |
| 305 *** | MAXITM | The current top item number (low number for global items) |
| 306 *** | OLDITM | A linked list of deleted items of the form XWD item #,pointer to next word of list |
| 307 *** | INFOTAB | Points to a table with information on each item. A more complete description will be given later. |

| | | |
|---|---|---|
| 310 *** | DATAB | Points to a table with the datums of each item, indexed by item number. The entry for an item contains a numerical value, array descriptor, a pointer to a set, or zero if there is no datum. A pointer to the table can also be found in cell DATM in your core image. GDATM contains the pointer for global datums. |
| 311 *** | HASTAB | Pointer to a 512 word long hash table for associations. More will be said about it later. |
| 312 *** | FP1 | One word free list with right half of each cell pointing at the next one. FP1 is of the form XWD end of list,start of list. Used for sets and various other one word free cells. |
| 313 *** | FP2 | Pointer to two word free list for associations. The right half of the first word of each pair points to the first word of the next pair. |
| 315 | HASHP | XWD list of free string descriptors,,pointer to printname hash table. More about this later. |
| 316 | MKBP | Address of make - breakpoint procedure or 0 if none. |
| 317 | ERBP | Address of ERASE - breakpoint procedure or 0 if none |
| 323 | LEABOT | A 86 word long array search control block, or SCB, used for retrieving associations by the derived set, association existance test, bracketed triple item retrieval, and erase operations. The SCB will be described later. |
| 324 | FRLOC | Points to the current SCB (for the FOREACH statement we are currently executing) or zero if we are not in a FOREACH. The left half points to a variable named SCB... of the procedure in which the FOREACH resides. SCB... is used as a flag to the block exit routine (BEXIT) which signals whether a FOREACH wil have to be exited before a GO TO out of the block is done. FRLOC is only valid if there are no processes. If there are processes the information normally in FRLOC is contained in the process variable CURSCB. |
| 325 | SCBCHN | Points to a list of abandoned SCB's. |

## 2. LEAP CALLS

Except for CVIS, CVSI, NEW.PNAME, DEL.PNAME, IFGLOBAL, TYPEIT, LISTX, SUCCEED, FAIL,
all calls to leap are MOVE 5,control_word, followed by PUSHJ 17,LEAP. The right half of the control word contains the dispatch number of the routine to be executed. The left half may contain one or more of the following bits. Ignore any other bits - leap does.

400000 This is a bracked triple search in a foreach specification (i.e., in the 'such that' clause)

200000 This is a GLOBAL model operation.

20000 This is a set operation in a foreach specification.

400,40,4 Attribute/Object/Value (of A⊗O≡V) has been bound locally in a foreach specification. The argument here is the index into a table in the SCB containing the bound value.

200,20,2 Attribute/Object/Value is being bound by this search in a foreach specification. The result, if the search succeeds, will be put in the SCB.

Some special routines such as NEW, and others use the left half for other information. The exact usage of the left half will be included in the routine descriptions.

Below are the (octal) dispatch numbers, all '140 of them, and what they mean. Unless otherwise noted all routines return to the location following the PUSHJ '17,LEAP.

The contents of ACs upon exit from leap is given. This is subject to change at any time.

| NUMBER | LABEL | DESCRIPTION |
|---|---|---|
| ***** | ****** | ********************************************************* |
| 0 | FOREC | The associative searches for the foreach specification. A, O, and V are in the stack in that order at entry. Parts of the triple not globally bounded are represented by table indicies. ANY is represented by a zero. An example of a foreach statement compilation is given later. If the search fails, control is passed internally (inside the procedure) to the FOREC search immediately preceding this one in the foreach statement. If this is the first one, control goes to the fail exit (see routine 12). If it succeeds, it will return normally with the current bindings in the SCB in use. Currently AC 14 will point to this SCB on exit. To determine which search LEAP is actually going to perform, check for the BINDING bits in the left half of the control word and the presence of ANY('0) in the stack. |

| | | |
|---|---|---|
| | | ?⊗?≡? As this search is not yet implemented this will only give an error message |
| 1-7 | | RESERVED for future use. |
| 10 | | 10-11 are the set searches in a foreach specification. The item, or index, and set pointer are in the stack.<br>A∈S |
| 11 | | ?∈S |
| 12 | FORGO | Start a foreach statment. Call+2 is a JRST which is executed when the foreach fails. The next cell (call+3) is the number of unbound variables and it is followed by one cell for each unbound variable containing the itemvar's address. It returns with a pointer to the SCB in AC 14. |
| 13 | FRPOP | Put the current bindings from FOREC into core for the user at the end of the searches, or before a boolean in the foreach specification. Unbound variables will get random values. |
| 14 | DOAG | This call is at the end of a foreach statement and returns control internally to FOREC for the next group of bindings. This also saves the current values of the foreach locals, so that they may be restored to the last successful binding if future searches fail. |
| 15 | FRFALSE | Called by the FALSE result of a boolean expression in a foreach specification. It is identical to routine 14 except that the current values of the locals are not saved. |
| 16 | MAKE | Make an association. A, O, and V are in the stack when called. On exit, AC 11 points to the two word block containing the association. |
| 17 | BMAKE | Make a bracketed triple. A, O, V are in the stack. It returns the item it has associated with the triple on the top of the stack. |
| 20 | ERASE | Erase an assocaition. A, O, V are in the stack when called. The search routines are used. |
| 21-27 | | RESERVED for future use. |
| 30 | ISTRIPLE | ISTRIPLE test. The item is in the stack when called. Answer returned in AC 1. (-1 TRUE, 0 FALSE). |
| 31 | SELECTOR | 31-33 select a part of a bracketed triple. The item associated with the triple is in the stack.<br>FIRST |
| 32 | | SECOND |

| | | |
|---|---|---|
| 33 | | THIRD |
| 34 | CORPOP | inverse of routine 12. Not currently used in compiled code. |
| 35 | LD1 | 35-37 generate derived sets inside foreach specifications. The two items are in the stack. It leaves the a dummy item containing the next element of the set at the top of the stack. (A⊗O) |
| 36 | LD2 | (A'V) |
| 37 | LD3 | (O≡V) |
| 40 | D1 | 40-42 generate normal derived set. Same arguments as 35-37. All leave a temporary set descriptor on top of the stack. (A⊗O) |
| 41 | D2 | (A'V) |
| 42 | D3 | (O≡V) |
| 43 | DELETE | Delete the item in the stack. |
| 44 | NEW | A new item with no datum is put on the top of the stack. Left half of control word contains type code of new item (1) and global bit if a global NEW. |
| 45 | NEWART | A new item with the arithmetic value in the stack as its is put on the top of the stack. The type code of the new item is contained in the left half of the control word. Left half contains global bit ('200000) if a global NEW. NOTE if a new string item then the value is on top of the string stack not the arithmetic stack. |
| 46 | NEWARY | A new item with a copy of the array whose descriptor is in the stack as its datum is put on the top of the stack. Type code and global bit in left half of control word. |
| 47 | FDON | Release the current foreach statement for DONE or GO TO jumping out of foreach. |
| 50 | PUTIN | PUT the item in the stack into the set pointed to by AC 14 on entry and exit. |
| 51 | REMOV | REMOVE the item in the stack from the set pointed to by AC 14 on entry and exit. |
| 52 | SIP | For making up sets from lists of items {A,B,C,D}. The next item to insert is on the top of the stack. The set being built is next in the stack and is left on the top of the stack. |
| 53 | STIN | Test if the item on the top of the stack is in the set or list which is next in the stack. |

| | | |
|---|---|---|
| 54 | COUNT | Returns in AC 1 the length of the set or list on the top of the stack.(Often compiled in-line). |
| 55 | UNIT | Returns on top of stack the first item of the set or list on the top of the stack at entry (COP) |
| 56 | UNION | The union of the two sets in the stack is left on the top of the stack. |
| 57 | INTER | The intersection of the two sets in the stack is left on the top of the stack. |
| 60 | SUBTRA | Set subtraction left on top of the stack. The subtrahend is on top of stack at entry, other set below it. |
| 61 | STORITM | Store the set or item on the top of the stack in the cell pointed to by ac 14, which has a -1 in the left half if storing a set. If the thing is an item you should never get this call since the compiler now generates a 'POP' in line. If it stored a set, it reclaimed the old set, if any. |
| 62 | | Same as 61 but also leaves the thing on the top of the stack. |
| 63 | RESERVED for future use. | |
| 64 | POPSET | Same as 61 but puts a set in AC1. |
| 65 | SETEST | 65-72 are set relationals. Both sets are in the stack<br>A<B |
| 66 | | A>B |
| 67 | | A=B |
| 70 | | A≠B |
| 71 | | A≤B |
| 72 | | A≥B |
| 73 | ISIT | Test for the existance of an association using the search routines. The three items are in the stack. |
| 74-102 | RESERVED for future use. | |
| 103 | BRITM | Retrieve a bracketed triple, given A, O, V in the stack and put its item on the top of the stack. |
| 104-112 | RESERVED for future use | |
| 113 | ITMRY | Initialize the array item on the top of the stack unless the global bit is set. Then, if bit 1 is |

also on in the control word it is a global array item; otherwise it is just a global array with nothing in the stack.

| | | |
|---|---|---|
| 114 | ITMYR | Initialize a compiled in array item.You shouldn't see this as all array items are now dynamically allocated. |
| 115 | STLOP | Apply LOP to the set or list in AC 14 and put the item on the top of the stack. |
| 116 | BNDTRP | Associative boolean of form BIND x ⊗BIND y ≡ BIND z where any of the BINDs may be omitted. |
| 117 | SETCOP | Copy the set in AC 14 for use as a value parameter to a procedure. New set put into loc. pointed to by AC 14. |
| 120 | SETRCL | Reclaim the set pointed to by AC 14 which was created by 117. |

121 CATLST concatenate the list on the top of the stack to the list below it on the stack. Return result on top of stack.

122 PUTAFT searches the list pointed to by AC 14 for the item(1) on the top of the stack and places the item(2) below it on the stack inside the list after the first instance of item(1) or at the end of the list if item(1) is not present.

123 PUTBEF searches the list pointed to by AC 14 for the item(1) on the top of the stack and places the item(2) below it on the stack inside the list before the first instance of item(1) or at the head of the list if item(1) is not present.

124 SELFET index on top of stack, list below index on stack. Fetches the n th (index) element of the list and leaves it on the stack.

125 TSBLST preforms the sublist operation LIST[I TO J]. J on top of stack I below that and list below I. Returns sublist on top of stack.

126 FSBLST same as 125 except preforms FOR sublisting operation.

127 SETLXT takes the list on the top of the stack and returns a set containing the same elements on the top of stack.

130 RPLAC preforms LIST[N]← it. AC 14 points to LIST. it on top of stack, N immediately below it.

131 REMX performs REMOVE n FROM list. list pointed at by AC 14, n on top of stack.

132 REMALL performs REMOVE ALL it FROM LIST. LIST pointed to by AC 14, it on top of stack.

133 PUTXA performs PUT it IN LIST AFTER n. LIST pointed to by AC 14, n on top of stack, it immediately below n.

134 PUTXB same as 133 except BEFORE.

135 LSTMAK same as 52 except makes list

136 CALMP call a matching procedure. on stack is a zero followed by parameters to matching procedure with the procedure descriptor at the top of the stack.

137 STK4VL stack a ? local. On top of stack is XWD routine_increment,,satis. no.∫ If satisfier unbound adds routine_increment to INDEX4 of SCB which is used as added to dispatch in FOREC.

140 STK4LC stack a ? local as a matching procedure ? parameter.

## 3. AN EXAMPLE OF A FOREACH COMPILATION

Below is the actual code generated (on JUNE 10,1973) by the following statement:

```
FOREACH X,Y|A⊗B≡X∧(DATUM(X)=1)∧X⊗ANY≡Y∧(ISTRIPLE(Y))∧(X≠Y) DO Z←X;
```

The parts of the statement are enclosed in {} in the listing. Notice that, in the comments below, when control is transfered to L2 or L6, it is transfered inside leap to the code called by those calls. Breakpoints at those locations would not win.

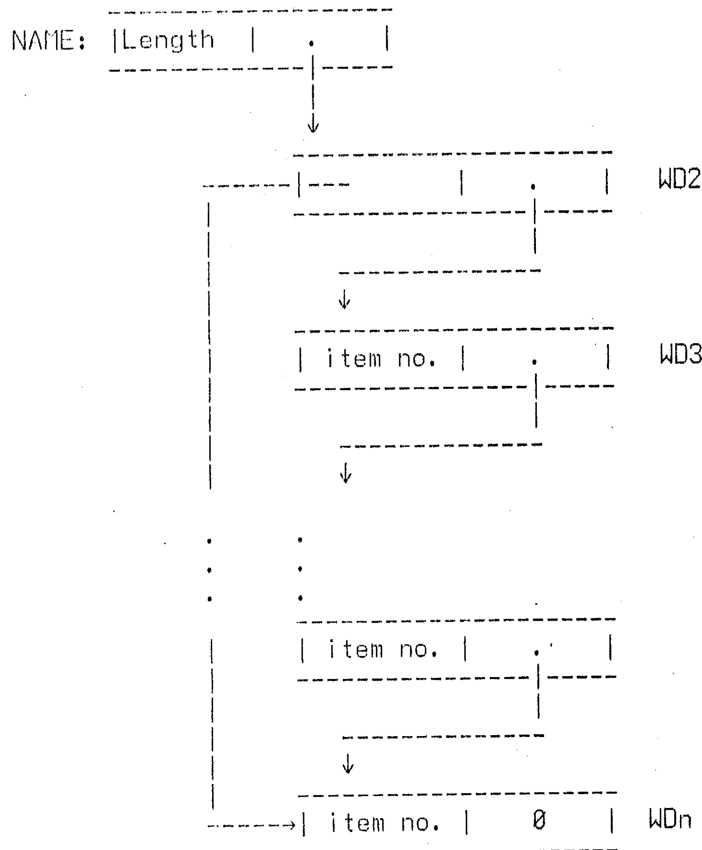
```
{FOREACH X,Y|}

{A⊗B≡X∧}

{(DATUM(X)=1∧}

{X⊗ANY≡Y∧}

{(ISTRIPLE(X))∧}

{(X≠Y)∧}

{DO Z←X}

{;}
```

If you want to know what leap is doing internally during all this, read on, and on, and on.

## 4. SETS and LISTS

Sets and lists are composed of one word blocks linked as follows:

```
NAME:   XWD number of elements, WD2     The set or list descriptor
WD2:    XWD WDn,WD3
WD3:    XWD item number,WD4
WD4:    ...
        ...
        ...
        XWD item number,WDn
WDn:    XWD item number,0
```




The words come from the one word free list (FP1) and are returned there when the set or is deleted. With sets, the items are ordered by item number, with the lowest first. This means that the earliest declared or created item will be first for local items and the most recent for global items, whose numbers start at 4096 and come down. The order for lists is completely program dependant.

There are two kinds of sets, permanent and temporary. The former are created by "PUT X IN SET1" or by assigning a set to a set

variable. They stick around until deleted by the program by storing PHI or another set into the variable. PHI, the null set, causes a zero to be stored into the set variable. Temporary sets are created by all other set operations and are indicated by a negative count in the first word. For example, if you have the statement:

IF X∈(A∩B)∪{A1,A2} THEN ...;

then A∩B generates a temporary set, {A1,A2} generates a second one, the union generates a third and deletes the first two, and the inclusion test deletes the third one. If the statement is inside a loop, this happens every time. You should assign the set expression to a variable, if possible, to make it permanent. Sets passed by value to subroutines are copied, only if they are permanent, and the copy, which looks like a permanent set, is deleted upon exit from the procedure. Temporary sets should be pointed to only by accumulators and the stack; they should never be stored in variables.

There are similarly two kinds of lists, permanent and temporary which behave much as the corresponding kind of set.

5. ASSOCIATIONS

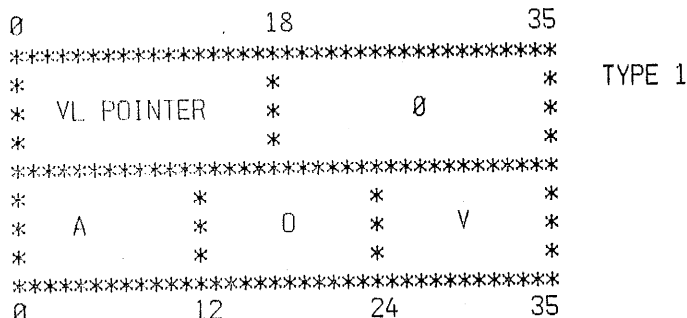
Describing the way associations are stored can be done only with some difficulty. We will start with some definitions to save me writing (remember these for section 6 also). WD1 is the first word of a two word association block. WD2 is the second word. LH is the left half of the word specified. RH is the right half. A, O, and V refer to the three items of an association (A⊗O≡V).

To start the description we look at INFOTAB (from Section 1), an array which has an entry for each item, both local and global items in the case of a lower segment, indexed by the item number. The LH of each entry contains the start of the value list [VL], which links together all associations with this item as V. It points to WD1 of the association. In fact, all pointers to associations point to WD1. The RH of each entry contains a 12 bit filed for PROPS, and a 6 bit field for the type whose value is returned by TYPEIT. Two byte pointers exist called PROPS and INFTB which correspond to these fields. Simply load AC 3 with the item number and the do a LDB ac,INFTB and ac will now magically contain the type code.There are two similar byte pointers GPROPS and GINFTB for the global model.

Associations are stored as two word blocks in a bucket hash table. To get the table index of the bucket we perform an operation called hashing. There are many ways of doing this but here we hash A and O by shifting A left one bit, exclusive ORing O into it, and ANDing the result with a mask to truncate the result to the size of the table. The contents of this bucket is a pointer to the first of a list of things which hash to the same value (known as the conflict list).

We may have several associations with the same A and O, but different V's (there is of course only a single copy of any association so we never have the case of two associations in the store containing the exact same A, O, and V). This is called multiple hits.

First let us consider the easy case where there are no multiple hits and there are no two associations which hash to the same bucket.




The association fits in one word since the maximum item number is twelve bits long. The VL pointer points to the next association on the value list for V or is zero if this is the last one.

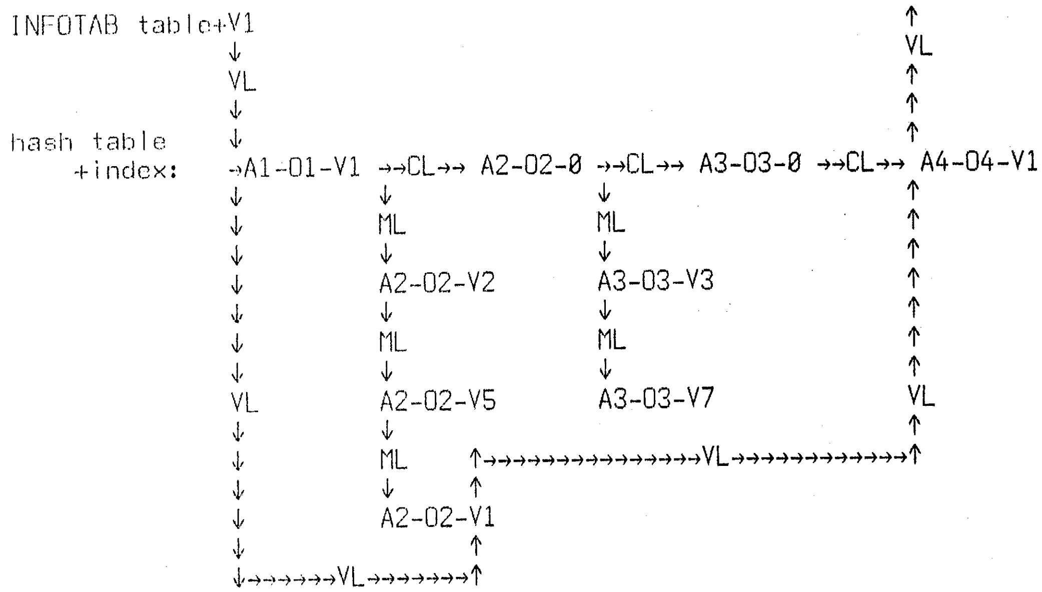
If there are multiple hits, then the entry on the conflict list looks like this:

```
0                  18                 35
****************************************
*                  *                  *    TYPE 2
*  MH POINTER      *        0         *
*                  *                  *
****************************************
*            *            *           *
*    A       *     O      *     0     *
*            *            *           *
****************************************
0            12           24          35
```

The zero in the V part of WD2 indicates multiple hits [MH]. This block is not an association, it is the header block for a list of associations with this A and O. The LH of WD1 points to the first association on the list, all of which [type 3] are the same as type 1 except that the RH of WD1 points to the next association on the MH list, or is zero for the last one. The blocks for associations on the MH list are taken from the two word free storage list (FP2) and are returned there if the association is erased.

If there are conflicts, the RH of WD1 each element of th conflict list, which is a block of either type 1 or 2, points to the next association on the conflict list [CL], which may be of either type 1 or 2, depending on whether or not there are multiple hits for that A and O. The conflict list continues through the RH of WD1 of all associations which hash to this index, with a zero for the last one. This structure is expanded and collapsed as necessary when associations are made and erased. Note that when a multiple hit list contains only two associations and one is then erased, we do not erase the multiple hit list header but wait until there are no associations with that A, O pair.

For those who prefer pictures with lots of spaghetti, this mess can be represented by the picture below, showing the multiple hit list [ML] and the conflict list [CL] for this hash table entry, and part of one of the value lists [VL] linking into it.




If you do not understand the above description and picture, you are welcome to read the code.

Before leaving this facinating subject, there is one more complication, which I left until last so I would not have to include it in the above picture.

When a bracketed triple is created, a normal association is made and linked into the hash table. The high order bit of the LH of WD1 is complemented from its normal value (it is now 1 for a lower segment association and Ø for an global association) to indicate a bracketed triple. The next thing in the value list through the association is a one (1) word block with the value list pointer in the LH and the RH containing the item representing the bracketed triple. The RH of the item's entry in DATAB points to the original association block.

To do fast associative searches, two more hash tables are needed, one hashing O and V, with an attribute list (corresponding to the value list for this hash table), and the other hashing A and V, with an object list. Then, given two items, hashing into the proper table gives all possible third items, and, given one item, the list for that item given all possible pairs of items in the current associations. Since we only have one table and list, mainly to save core, some searches are slower than others, as hinted at in section 2. The associative searches are done like this:

A⊗O≡? hash A and O to get a triple, or a set of them (the multiple hit list).
A⊗O≡V hash A and O to get V, searching the multiple hit list if

necessary. There is only one possible match.
?⊗O≡V and A⊗?≡V search Vs value list for all instances of the given A or O.
?⊗?≡V Vs value list is the set of associations requested.
?⊗O≡? and A⊗?≡? Try using all possible A's(O's)and then use the A⊗O≡? search for each possible A (O).

## 6. FOREACH STATEMENTS

Foreach statements use the structures described in the last two sections and retrieve from them items which fit the conditions of the foreach specification. This section describes the foreach search control blocks (SCBs) which enable the leap routines to keep track of the status of each search for when it is necessary to continue it. Each foreach statement generates a new SCB when first called and releases it when the statement is exited. Each SCB is 87 words long and contains the following:

WD1 push down pointer to the top of the stack for this block, which starts at WD16. The PDL initially points to WD17.

WD2 if you load AC 3 with the index of an unbound variable and execute WD2, you get the current satisfier in AC 1 from the table at WD6.

WD3 Same as WD2 except satisfier appears in AC2.

WD4 DPB X,WD4 stores the item in AC X in the table as the satisfier for the variable whose index is in AC 3.

WD5 minus the number of unbound variables as obtained from the second word after the call of leap routine 11.

WD6-WD15 a 10 word table of satisfiers. The LH of each word is the current item. The RH is the address of the itemvar the satisfier is bound to. These are filled by the search routines and are stored in your program by routine 12.

WD16 start of a two word dummy SCB entry below the start of the stack. It contains XWD 0,-1 which stops searches in this block when the routines try to use it as an index into a table of search routines.

WD17 the JRST for the failure exit for this foreach statement.

WD18-WD86 The rest of the SCB is used as a pushdown stack containing one 8 word SCB entry for each active search for a triple or set inclusion specification in the associative context of this foreach statement (i.e., one is set up by the initial call, for each foreach statement, of routines 0-10). The PDL pointer in WD1 points to the end of the SCB entry currently being used in a search.

The 8 word SCB entry looks like this:

WD1 satisfier index, if unbound, or item number for the value of this association.

WD2 satisfier index or item number, for object of associaton if search routine 0-6, or the set descriptor for routines 7-10.

WD3 index or item number for attribute of association, or for

set test.

WD4 compare mask for associative searches. It contains ones in the parts of the word containing bound portions of the triple and zeros in the remainder.

WD5 -1 if no search yet (WD6 not set up), else ≥0

WD6 pointer into set or associative structures (ML or VL lists) where search is to continue. If it is zero search will fail if called again.

WD7 control word from call to this search from program, so we can branch back internally when a search fails. The left half contains the bits and the right half contains the search routine to be executed (actually a number 0-10 which corresponds to the leap routines with those numbers).

WD8 return address from the call to this search, for when we succeed.

WD37 OF SCB -address of SCB... variable and the SCB of the dynamically enclosing foreach.

7. ERROR MESSAGES

Most of the leap runtime error messages are easy to understand. However here is the explanation for all the ones at present anyway.

<INCORRECT ITEM # FOR GLOBAL DATUM> - you have attempted to take the global datum of a non-global item

<LEAP SHOULD HAVE BEEN INITIALIZED> - the LEAP runtime environment has not been initialized properly. Theoretically you can only get this message if you call LEAP directly from an assembly language program or SAIL START_CODE.

<DRYROT-LEAP:ROUTABLE> - the routine index you have given to LEAP is not valid. This is usually caused by having an incompatibility between the version of the compiler and the runtimes. Recompile, reload and try again.

<ASSOCIATIVE SEARCH WITH NOTHING BOUND> - you have specified a search (or erase) with none of the positions bound. As this particular search has not yet been implemented, you lose.

<GLOBAL SEARCH WITH LOCAL ITEM> - one of the elements to a global search(or erase) was a local item

<MAKE WITH UNBOUND ITEM> - an argument to a MAKE statement was either the item BINDIT or the item ANY. As all itemvars are initialzed at load time with ANY this is a common error.

<GLOBAL MAKE WITH LOCAL ITEM> - one of the arguments to a global make statement was a local item

<DRYROT -- ERASE1>- While attempting to erase an association, it was noted that the association was not on the appropriate value list. Report this to your local LEAP expert.

<DRYROT -BRACKET CONFUSION> - while erasing a bracketed triple association, erase blew up. Report to LEAP expert.

<DRYROT -- ERASE2> - while erasing an association it was noted that the association was not on the appropriate conflict list.

<DRYROT -- ERASE3> - the association which was being erased was not on the appropriate multiple hit list

<NOT A BRACKETED TRIPLE>- the argrument to FIRST, SECOND, or THIRD was not a bracketed triple