```
************************************************
************************************************
***                                          ***
***  Name:                                   ***
***                                          ***
***  Project:    1       Programmer:  MWK    ***
***                                          ***
***  File Name: LEAP.TXT[DOC,AIL]            ***
***                                          ***
***  File Last Written:   0:32 28 Jun 1973   ***
***                                          ***
***  Time: 19:15           Date: 15 Jul 1973 ***
***                                          ***
***            Stanford University           ***
***       Artificial Intelligence Project    ***
***         Computer Science Department      ***
***            Stanford,  California         ***
***                                          ***
************************************************
************************************************
```

```
COMMENT ⊗   VALID 00022 PAGES
RECORD PAGE   DESCRIPTION
 00001 00001
 00005 00002    Leap Implementation Documentation
 00006 00003    LEAP COMPILER INITIALIZATION(LEPINI)
 00008 00004    Declarations(compiler)
 00014 00005    COMPILE-TIME STACK
 00020 00006    COMPILE-TIME STACK
 00025 00007    Compiling calls to LEAP runtimes:
 00030 00008    Compiling calls to LEAP runtimes(cont)
 00035 00009    Compiling calls to LEAP(cont)
 00037 00010    Compiling FOREACH'S
 00041 00011    Compiling Foreach's (continued)
 00046 00012    COMPILATION OF FOREACH SEARCHES
 00047 00013    GLOBAL MODEL CONSIDERATIONS (compiler)
 00048 00014    LEAP RUNTIM INITIALIZATION (LPINI)
 00049 00015
 00050 00016    PRINTNAMES(COMPILER)
 00055 00017    PRINTNAMES( RUNTIME )
 00059 00018    PRINTNAME DATA STRUCTURES
 00063 00019                                ____________________
 00067 00020            These item-printname nodes  consist of two words which
 00071 00021    Printname Runtime routines.
 00074 00022    DEL_PNAME (item) - delete printname. Called to disassociate
 00083 ENDMK
⊗;
```

Leap Implementation Documentation

The following will hopefully (someday?) include all the important implementation information necessary for someone else to maintain the LEAP compiler routines and the LEAP runtime routines.

LEAP COMPILER INITIALIZATION(LEPINI)

Every time the compiler is started, or restarted the routine GENINI is called to initialize the state of the world, including zeroing out all the variables declared in ZERODATA areas. GENINI calls LEPINI to do various initializations for LEAP.

LEPINI does the following:

1. Initializes Q-stacks (LOCST,MPSTAK,ITMSTK,MPQSTK) by either pushing a zero argument (MPSTAK,MPQSTK) to mark the bottom of the stack (to prevent underflow), or pushes and pops a dummy argument onto the stack(LOCST,ITMSTK) so that the address of the first stack entry can be kept so that the Q-stack may be used as a FIFO queue.

2. Gets dummy semblks for NIL, PHI, and NULL_CONTEXT(see backtracking)

3. Initializes the ITEMNO and GITEMNO cells. (Used to keep track of how many local and global items have been declared).

4. Initializes LEAPSK to be an empty stack.

LPNAME: This is called from the productions to insert the predeclared items such as MAINPI, EVTYPI etc into the symbol table. This really should be done as part of RTRAN, but as RTRAN cannot allocate integer constant semblks, there is really no alternative.

Declarations(compiler)

I. Declaration of items, itemvars.

Items and itemvars are declared in the standard way: the type bits are collected by TYPSET and the symbol is inserted into the symbol table by ENTID. Whereas itemvars are normal variables and thus will have storage associated with them, items are considered to be constants. An integer constant is created for each item declared. The value of the constant is determinehd by the following algorithm.

1. If the item is local (not global model item), which is determined by the GLOBL bit not being on in BITS, increment ITEMNO and use the current value of ITEMNO for the item's number.

   Note: to reserve space for predeclared items such as MAINPI, EVTYPI, the routine LPINI (called before every compilation) initializes ITEMNO to '10).

2. If the item were global (GLOBL bit on in BITS) then decrement GITEMNO and use that value for the item's number. GITEMNO is initialized to '7777

3. With the above number as parameter(currently in ac A) call CREINT to get a integer constant semblk. Store the semblk pointer in the $VAL2 in the semblk for the item.

There is no actual storage allocation done for items as they are constants, though of course the corresponding integer constants may be placed out if refered to by other than "immediate" instructions. Itemvars are allocated in the same manner as real or integer variables.

II. Declaration of sets, lists.

Again the declaration is pretty standard. Type bits are collected into BITS by TYPSET and the symbol is inserted into symbol table by ENTID. Note that there is only a single parse token for both lists, and sets. The type list is indicated by both the SET bit and the LSTBIT being on in the TBITS entry. Also when stacked on the compile-time stack (LEAPSK: se below), lists are denoted by having both the LPSET and LPXISX bits on in the left half of the compile-time stack entry.

Allocation at the end of procedure. Because the SAIL runtime initialization must be able to find all statically allocated sets and lists these are allocated together and a loader "LINK" is put out so that the runtime leap initializer(LPINI) can find all static sets and zero them out.During initialization when the SAIL program is initialized (by SAILOR) these variables will be zeroed (set to NIL or PHI).

Value set parameters must be copied on entry to a procedure(SETCOP) and must also be deallocated(SETRCL) on procedure exit. A better practice would seem to be that SETCOP be called by the calling procedure since it can often be determined that no copying is necessary as the set is a temporary. On exit from a recursive procedure sets must also be deallocated. This is done from inside the block exit code (runtime routine BEXIT) or (runtime STKUWD) used in goto's out of procedures), both routines use the procedure descriptor (also called PD) to determine where the set locals and parameters are located on the stack.

When any LEAPish declaration or construct is seen the cell LEAPIS is made non-zero. This will be used later to determine if LEAP will have to be initialized at runtime.

COMPILE-TIME STACK

I. STITM (STSET)

Whenever an item or itemvar(SET OR LIST) is scanned in an expression (not the left side of an assignment) the routine STITM stacks the semblk on an internal stack whose current top is pointed to the cell LEAPSK. Outside of FOREACH's STITM will also generate code to stack the previous element on LEAPSK if any. The reason we defer generating the code that will stack the current element is two-fold: one this itemvar may be a reference parameter in which case we will not want the value of the itemvar but rather the address; two, sometimes certain expressions can be compiled into more efficient code if we wait(e.g itmvr1←itmvr2 can be compiled into a MOVE, MOVEM rather than a PUSH, POP).

Note that we must keep track of which things in LEAPSK for which code has been emitted to stack them. Therefore in the left half of the LEAPSK entry is a bit STACKED which will be on if code to stack the entity has been emited and off if no such code has been emited. Other left half bits in the LEAPSK entry keep track of such important information as whether this is an item expression(LPITM) or not (LPSET, LPXISX);if this is properly a retrieval(RETRV), or constuction(CNSTR) expression. All expressions are possibly construction expressions at compile time, but NEW cannot properly be part of a retrieval expression, ANY and UNBOUND are not properly construction expressions, but this impropriety is discovered only at runtime.

Other left hand bits, are: BINDING which indicates this is a foreach local which has not yet been bound; BOUND indicating a foreach local which has been bound by a previous search; FIXED which is on for item constants or contents of some non-local itemvar(it is read only in if-expr,and case-expr its exact significance is beyond me); DUMSEM which is on if the semblk is a dummy and is on only if the thing was a temp which has been remopped; LPDMY on if a item from a bracketed triple of a derived set within FOREACH; LPNUL is on only for null sets and lists. Two other bits FBIND and QBIND are not put in by STITM but are in the left half of the LEAPSK entry to indicate a BIND itemvar or a ? itemvar respectively as in:

```
if A⊗BIND x ≡ ? z then ...
```

Within a Foreach associative context(indicated by the LPPROG bit being on in FF), no code is emited by STITM to stack the elements of LEAPSK. This is to enable better code to be emitted for certain searches involving bracketed triples, and derived sets. For example

```
        FOREACH x | x⊗[a⊗o≡v]≡z do
is compiled as if it were:
        FOREACH x,q | [a⊗o≡v] = q∧ x⊗q≡z do
```

We could not simply stack x, then do the bracketed triple, then stack v because the design of the foreach interpreter at runtime does not allow anything to be remembered on the stack(see the difficulties if x were on the stack and then the search a⊗o≡v failed).

NOTE: Some of the runtimes return their values on the top of the stack (as opposed to the normal convention of returning values in AC 1).
E. G. COP(list). The compiler will emit the code to call the routine,

and will push an entry onto LEAPSK with the STACKED bit on in the left half and a zero right half where a semblk would normally appear. (SEE BFIN and BFINA in the compiler).

COMPILE-TIME STACK

II. ITMREL - since STITM outside of FOREACH's causes code to be emited which will cause the previous top of the LEAPSK stack to be stacked on the runtime stack, we often must short-circuit this by removing things from the LEAPSK. One such routine is ITMREL which is used in item relations such as:

```
itmvr1= itmvr2.
```

ITMREL takes the top element of LEAPSK (causes code to be emited to pop it into an ac if STACKED) and puts the semblk into the parse stack for relational operations later.

III. OKSTAC- often we must force the top element of LEAPSK to actually be stacked on the runtime stack. For example:

```
MAKE a⊗o≡itmfn(0,1,a);
```

if we did not stack "o" before starting to process the itmfn, the o would not be stacked until the second a was seen thus the parameters on the runtime stack would be in the order a,0,1,o,a instead of a,o,0,1,a which is the correct order. Calling OKSTAC causes code to be emitted to stack the last operand on the runtime stack if necessary.

IV. STCHK - called by the macro STAKCHECK(#of parms). This makes sure that the #of parms top entries of LEAPSK are stacked. Also makes sure collects the BOUND, BINDING bits to be passed to the runtimes into the left half of the word BYTES, and will return in left half of ac A: FBIND if any of the parms had FBIND or QBIND on; the AND of the RETRV,CNSTR bits so you can check for construction -retrieval failure; and the AND of the LPITM,LPSET,LPXISX bits so that you can check if all args were items, or sets.

This removes the top entries from LEAPSK, and updates ADEPTH since it knows that these stack entries will go away. This updating of ADEPTH is very important. If you have a routine which calls LEAP with leap expressions and arithmetic expressions you must either make sure the arithmetic expressions are calculated first, or you must restore ADEPTH before calculating them.See PUTINL for an example of the later.

STCHK is usually very simple. However certain foreaches cause very complicated things to happen.

For example:

```
foreach x | x⊗cop(set1)≡v do
```

We have the situation when we call STCHK that x and v are not stacked but cop(set1) is (SEE STITM above for reason why x was not stacked). Therefore we must emit code that will pop "cop(set1)" into a cortmp and then push the three arguments onto the runtime stack.

V. LASCHK -called from case-expr,if-expr
As it is known to be called from outside an associative context it does not have to check certain FOREACH dependant properties, but otherwise is equivalent(except for minor ac differences) to STAKCHECK(1). See STCHK above.

VI. BNDITM(BNDLST) - these routines make sure the top entry of LEAPSK is of the appropriate type ITEM (SET or LIST) and make sure the code has gone out that will stack that entity. Then the top entry of LEAPSK is removed. These routines are now called from the APPLY, SPROUT and REMOVE execs.

Compiling calls to LEAP runtimes:

Most calls to leap runtimes are made by loading ac 5 with a word containing some flag bits in the left half and a routine index in the right half, followed by a PUSHJ P,LEAP. Inside the runtimes this index will be used in a branch table calculation.

Within the compiler there is a marvelous macro called RUNTIM which calculates the indices of the various runtime routines. To generate the proper code to call a given routine, simply place the desired lh of the flag word into the lh of BYTES, move the index (RUNTIM defines a symbol which may be used for the index,the symbol is "L" concatenated with the routine name) into ac A. And call either LEAPC1 or LEAPC2. These routines will do an ALLSTO followed by generating code which will load ac 5 with the required flag word, followed by emitting a PUSHJ P,LEAP. LEAPC2 differs from LEAPC1 in that it will add to the index in A the contents of the right half of BYTES.

The routine STCHK described partially above, in addition to making sure that the arguments will get stacked, calculates the BOUND,BINDING bits for the attribute, object, and value positions of the flag word and stores these in the left half of BYTES. A routine offset is also calculated and placed in the rh of BYTES.Currently though, this offset is only used for set or list searches within foreaches.

A macro to call LEAPC1 or LEAPC2 exists and is called LPCALL. LPCALL takes three parameters (the last two being optional). The first is the routine name, the second the location of an amount to be added to the primary routine index (see SETREL for example), and the third if present indicates that LEAPC2 should be called rather than LEAPC1.

For an example of how these routines are used let us look at the compiler's exec routine for MAKE.

```
STAKCHECK (3)      ;MAKE HAS THREE ARGUMENTS
LPCALL(MAKE)       ;GENERATE THE CALL TO LEAP
POPJ P,
```

The macro STAKCHECK calls STCHK with the argument three. And LPCALL calls LEAPC1 with ac a loaded with the value LMAKE.

Other macros often seen are RETCHK and CONCHK which when called immediately after STCHK will verify that the arguments were of retrieval type, or constructive type (Actually this is done merely by chkecking the lh of ac A).

Most of the leap runtime routines leave their result, if any on the runtime stack P. However others , notably LENGTH of set, ISTRIPLE etc. return their result in ac 1. A macro XPREP makes sure this register is available. It also loads ac D with the value 1 so that a call to the MARK routine will mark the value as being in ac 1. Therefore be careful not to destroy D or else restore it before calling MARK.

Compiling calls to LEAP runtimes(cont)

Most of the compiler execs for generating calls to the runtimes are fairly straight forward. Let us briefly go through them.

MAK and ERAS are straight-forward, as are STIN, and ISTRIP. ISIT has to choose between two different alternatives depending on whether any of the arguments were preceded by BIND or "?" indicaed by the presence of the FBIND bit in the left half of A following the STAKCHECK.

STREL must first determine if the arguments are sets or items. if items then a call to ITMREL to move the second parm to the parse-stack followed by a jrst to IREL in the standard relation handler. If the arguments are sets the LPCALL to SETREL is generated.

DELI is straight-forward.
SIPGO, and LIPGO mark a q-stack (LORSET) to determine if a set or list is being constructed. Then cause a ZERO to be pushed on the runtime stack (this position of the stack will be used by the runtimes to collect the set or list).
SIP1 calls either the list-maker(LSTMAK) or the set-maker(SIP) depending on the value of the top entry of the LORSET q-stack.

STCNT-(length of set) is straight-forward. An optimization easily added would be to see if the argument is really a reference set or list, and do the code in-line.

STUNT-(cop of set) is straight-forward.
ECVI- convert arithmetic expression to item
The of the arithmetic expression is "gotten" and then marked as an itemvar temp. The itemvar semblk is then placed on the LEAPSK.

ECVN- convert item to integer
If the item expression is stacked it is popped off into an ac, and the ac is marked as an integer temp. If the item expression is a constant item the integer semblk for the item is placed into the parse stack. Otherwise code to get the item expression into an ac is emitted and the ac is marked as an integer temp.

STLOP -lop of set
This takes a reference set argument (hurray for deferred stacking) and calls the routine FIRREF to emit code to load the address of the set arguent into ac 14.Otherwise it is straight-forward.

STMIN,STINT,STUNI- the set operations are rather straight-forward.

PUTIN -Put item into set
Uses FIRREF to get load the address of the set, otherwise it straightforward.

PUTINL- put item in list before,after
Must compute which of four routines to call depending on wheter last argument is an item or arithmetic expression, and whether BEFORE or AFTER. The routine LISTGT is called to call FIRREF with list argument which was cleverly removed from the LEAPSK by the

exec routine HLDPNT.

REMXD - uses FIRREF but otherwise is straightforward.

REPLCX - is straightforward except for the fact that the list argument was never placed on the LEAPSK because of a hack in the productions.

Compiling calls to LEAP(cont)

CVLS- convert list to set, vice-versa. Set to list merely changes the marking of LEAPSK. list to set requires a LPCALL.

REFINF,LSSUB -are exec routines which set up the necessary information for ∞ onto the q-stack LENSTR. REFINF is called when the list or set argument is by reference so the semblk is stacked, LSSUB, is called when the set or list has been stacked and saves ADEPTH on the stack. A flag in the left half of the LENSTR entry indicates which type, REFINF,LSSUB or string inf is in use. The exec for inf will use this flag to call either the string inf. or LINF which handles the two different inf's for lists.

SELIP, SELSBL, are straightforward except for munging ADEPTH when dealing with STAKCHECK(see STCHK for motivation).

LSTCAT is straight-forward.

NEWNOT, NEWART are straight-forward except that the type code for the new item is stored in the left half of BYTES before the LPCALL. The type code is calculated by the routine ITMTYP.

SELET - FIRST,SECOND THIRD are straightforward.

Compiling FOREACH'S

As an example to be used in the remainder of this section let us consider:

FOREACH ? X,Y,Z SUCHTHAT X⊗Y≡A ∧(Y≠B) ∧Y⊗X≡Z DO

I. Initializing the foreach.

When the FOREACH is parsed, a new block is entered (see productions) and the exec routine EACH4 is called. EACH4 declares a local variable named "SCB...". This variable will be passed to the runtime leap initializer, and is used in the block exit code (see BEXIT, STKUWD in the runtimes) to indicate whether a foreach must be exited. EACH4 causes code to be emitted which will push the address of this variable onto the runtime stack.

II. Collecting the locals.

The local variables to this foreach are X,Y,Z. As the local list is being scanned by the parser, ENTITV or QLOCAL followed by ENTITV are called for each local parsed. These routines do several things:

1. The LPFREE bit in the SBITS entry of the itemvar semblk is turned on. This will remain on until a search is performed or a matching procedure called which will bind this itemvar. STCHK is responsible for turning it off.(Also turned off by ISUCAL for matching procedure actual parameters within FOREACH's).

2. The LPFRCH (for normal locals),FREEBD (for question locals) bit is also turned on in the SBITS word.

3. The count of number of locals seen (LOCALCOUNT) is incremented and the new value is the satisfier number for this local. Note: so that the backup will work efficiently when we are calling an associative search within a foreach we do not pass the address of the local variable that is being bound but rather the satisfier number. This number will be used as an index to a table within the runtime FOREACH interpreter. The satisfier number is also passed to any search following the search which bound the itemvar as the value is not normally moved from the internal table to the core variable (see FRPOP below).
4. A constant semblk for the satisfier number is obtained by calling creint and its pointer is saved in the $val2 entry of the local's semblk.

5. The itemvar's semblk is pushed onto a q-stack of locals called LOCST. This will be used at the end of the foreach execs to determine if searches to bind on the locals have been emitted.

Compiling Foreach's (continued)

III. Emitting the call to start of FOREACH

When all the locals have been scanned and the SUCHTHAT is seen the exec routine FRCHGO is called. This routine does the following:

1. Turns on the LPPROG bit in the FF word. This bit tells everyone that a FOREACH associative context is in progress. This will effect such things as whether the current value of a foreach local is stacked or the satisfier number. See BOOLEAN expressions below.

2. Emits an instruction which will load ac 14 with the address of the FOREACH satisfier block (below) The fixup for this instruction is saved in SATADR.
3. Emits the call to the runtime leap initialzer

IV. The FOREACH satisfier Block
Passed to the FOREACH interpreter at runtime is the address of a block of data about the local itemvars.

The first word is a JRST to the instruction following the FOREACH (For use when the last search fails). A fixup to this instruction is placed in the LOOP block to be filled in later by the LOOP code.

A word containing the value of LOCALCOUNT is emitted (that is the total number of locals and question locals for this FOREACH).

The semblks for each local itemvar (obtained from LOCST) are inspected and a word containing the following is emitted.

In the address portion of the word (index field and RH) the address of the local is placed (This will be picked up by a MOVEI @ at runtime so the indirect bit must be off).

In the left half the bits POTUNB for question locals, and MPPARM if this local is a formal question parameter to a matching procedure (These are necessary since interpretation will be done at runtime on these special itemvars).

We have not been totally truthful when we talked about the address portion of the word to be emitted. Consider the following

```
RECURSIVE PROCEDURE FOO;
BEGIN ITEMVAR BAZ;
        ...
        PROCEDURE ZORK;
        BEGIN FOREACH BAZ | ...
        END;
END;
```

In this case the address of BAZ is not simple to compute as a display register is required (see up-level addressing elsewhere) Therefore in this case, the word we will emit will (in addition to any other LH bits) have the CDISP (calculate display) bit on. In the right half will be the normal stack offset, and in the INDEX field will be the difference between the current display level and the display level where the itemvar was declared (in this case 1).

COMPILATION OF FOREACH SEARCHES

GLOBAL MODEL CONSIDERATIONS (compiler)

LEAP RUNTIM INITIALIZATION (LPINI)

PRINTNAMES(COMPILER)

Until a "REQUIRE n PNAMES" statement is scanned the compiler assumes that no printname initialization is required and any items declared will not receive initial printnames.

When the require statement is encountered it calls the routine PNAM (within the source_file LEAP). PNAM sets the variable PNMSW to 0. PNMSW was originally initialized to be -1 to indicate no printnames were requested. From here on it will contain the number of items with initial printnames. The count n from the require statement is placed in the variable PNAMNO which will later be a part of the space allocation block at runtime. PNAM sets up a Q-stack (a push down list) and records the head (bottom) of this list in the variable PNBEG. The top of this stack will stay in the variable PNLST. This Q-stack will contain entries whose left half is the item number and whose right half is the semblk pointer for the string constant corresponding to the items name.

From then on everytime an item declaration is scanned the following things are done within the ENTID routine:

1. PNMSW is incremented.
2. A string constant the same as the item name is generated.
3. A word containing the item number in the left half and the semblk pointer for the string constant in the right half, is pushed onto the Q-stack PNLST.

At the end of the compilation within the routine DONES the printname initialization block is put out. First the program counter (which corresponds to the address of this block) is placed in PINIT (corresponding to $PINIT in the space allocation block at runtime). Then the number of pnames to be initialized is put out, followed by one word per pname containing the item number in the left half and the address of the string constant for the pname in the right half.

BUG!!! String constants may be used as comments at statement level. To keep from keeping constants used only as comments around in the compiler the string comment handler will try to get rid of unused constants. Therefore if the fixup chain of a string constant used as a string comment is empty the semblk will be reclaimed. This will cause problems if the string was also used as an initial pname, as the reference to the constant is not emitted until the end of the program. Thus we may find the semblk deleted or used for something else. There should be some way of indicating in the semblk that even though the fixup chain is empty the constant is still in use. Note this problem also occurs for blocknames.

PRINTNAMES(RUNTIME)

A string called a printname (often refered to as a pname) may be associated with a given item at runtime. Primitive actions on pnames include dynamically associating a printname with an item (NEW_PNAME), deleting the printname of an item (DEL_PNAME), finding the unique item with a given printname (CVSI), and finding the printname of a given item (CVIS). NOTE: an item may have at most a single non-null printname, and no two distinct items may have the same printname.

Printname initialization:

Items declared following a REQUIRE n PNAMES compiler statement, have initial printnames which are the same as their names. For example if the following declaration followed the require:

```
ITEM X;
```

the printname of item X would initially be "X".

During the initialization of the SAIL runtime environment a LEAP initialization routine(LPINI) is called. This routine in turn calls the printname initialization routine INTNAM. The parameter to INTNAM is contained in AC A and is the address of a printname initialization block. This address was obtained from the space allocation block entry $PINIT. NOTE: for each separately compiled SAIL program there is a space allocation block which includes such information as how many items were "REQUIRED" , the user's estimate of the number of pnames he will use, etc. All the space allocation blocks of programs loaded together are linked by means of the LOADER link command.

The format of the printname initialization block is:

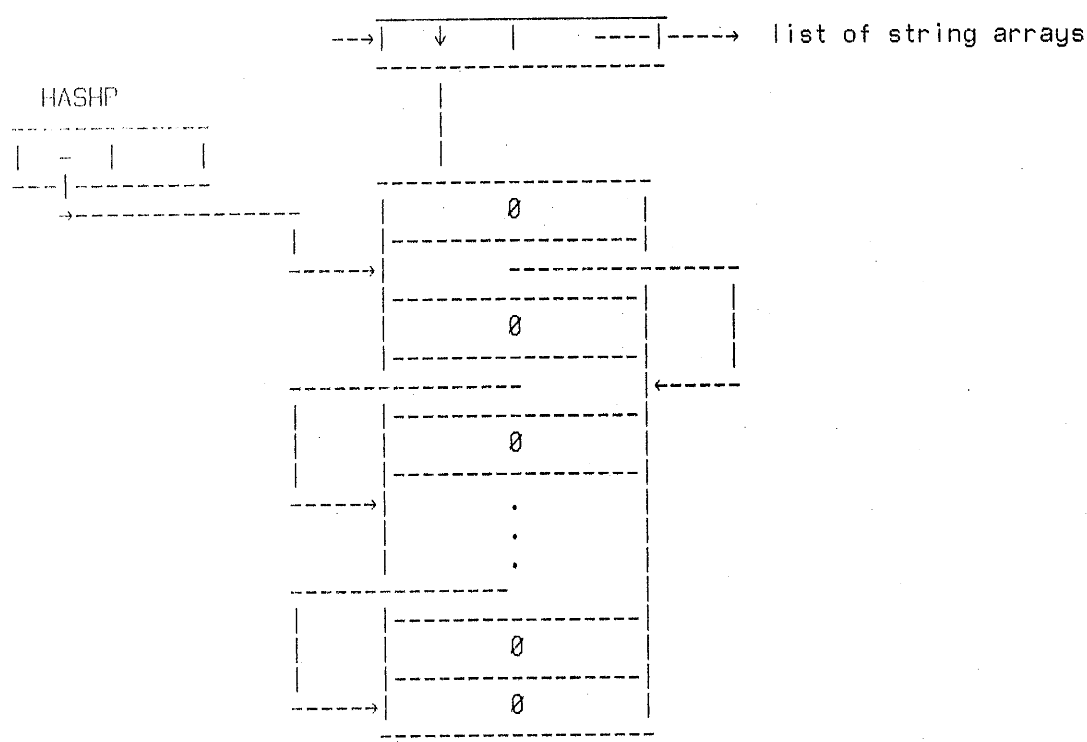
```
                 **********************
(first word)     * number of pnames   *      (may be zero)
                 * to be initialized  *      (<0 if no REQUIRE statement)
                 **********************
                 * itemno, addr(str)  *      (one word for each pname
                 **********************          to be initialized)
                 .                    .
                 .                    .
                 .                    .
```

INTNAM calls NEW.PNAME for each pname to be initialized with the item number from the left half and the string whose address is contained in the right half of the printname initialization block entry.

PRINTNAME DATA STRUCTURES

The strings representing the printnames are stored in the standard SAIL string space (thus unfortunately not allowing sharing, by use of the global segment, of printnames between jobs). A SAIL string variable is represented by a two word string descriptor: the first word containing the information such as the length of the string and the string id; the second, containing a byte pointer pointing to the first character of the actual string. For the string garbage collector to work correctly, it must be able to access all such two-word string descriptors. We could, of course, have a list of all the descriptors in use and have the garbage collector use it to access the printnames. However, such a list would be relatively expensive to maintain on deletion of printnames, as we would have to search down it to find the appropriate descriptor to delete when we removed a printname. It therefore seems better to allocate one or more string arrays from which we will allocate individual descriptors for pnames, and simply have the address of these string arrays on the same list (ARYLS) as the addresses of the string arrays which are the datums of items.

On initialization of the pname structure, we allocate a string array, the size of which is the maximum of the "n" in the "REQUIRE n PNAMES" statements of all the separately compiled SAIL programs which are being loaded together to form a single job (if that number is less that 50, the string array's size is 50). We form a list of the individual string descriptors which are not in use. The link pointing to the next available descriptor is contained in the seconde word of the current descriptor. These string descriptors are also used by datums of STRING ITEMs. NOTE: because the string garbage collector ignores all string descriptors whose first word is zero, we do not have to worry about the garbage collector interpreting the links as byte pointers to actual strings. The address of the first available string descriptor is placed in the left half of the HASHP word of the user table (GOGTAB). Thus we have a the following structure:



initial string array

NOTE: The links to the two-word blocks point to the second word to conform with the SAIL convention that the address of a string variable is the address of the second word of the two word descriptor.

The most common operation to be preformed on items and their printnames is anticipated to be lookup: that is given an item find its printname and vice-versa. To be able to perform this lookup in a reasonable time we use a hash table (scatter storage) technique.

There are actually two hash tables: one for items; one for pnames. It turns out by use the halfword instructions we can have the two tables live in the same block of core. The item hash table is in the left half words of the block, and the string hash table is in the right half words of the block. The address of this block (whose length is PHASLEN, currently set to 128 decimal) is contained in the right halfword of the HASHP entry of the user table (recall that the address of the head of a list of available string descriptors is in the left half of the HASHP entry).

Now suppose that we are given an item and are asked to find its printname. First we compute the hashcode from the item (currently this is simply done by "AND"ing the item number and PHASLEN-1). We then add this hashcode to the base of the hashtable (right half of HASHP). This gives us the location containing the address of a conflict list of nodes of item-printname pairs whose item numbers hash to the same value.

These item-printname nodes consist of two words which are logically divided into 4 different fields: the item number; the link to the next node in the item conflict list; the address of the two word string descriptor for the pname; and the link to the next node in the string conflict list.

```
*************************
*           *           *
* item no.  * item link *
*           *           *
*************************
*           *           *
* addr. str.* str. link *
*           *           *
*************************
```

There is one of these nodes for each printname-item in existence.

Let us continue with our search for a pname given an item number. We now have the address of the first node in the item conflict list. We check to see if the item no. within the node is the one whose printname is desired. If it is the same we return the string pointed to by the lefthalf of the second word of the node. If not we obtain the address of the the next node in the item conflict list from the right half of the first word in the current node under examination. We continue with this technique until we either find the node which corresponds to the given item number or we run out of nodes on the conflict list (this is indicated by a zero link field. If we exhaust the conflict list then we know that there is no printname for the given item.

The search for the item with a given string name is similar. We form a hashcode from the string. Use that code as an index into the string hashtable. From this we obtain the list of nodes whose printnames hash to the same value. We search down this list until we find a node with the same string as we have and return the item number or we exhaust the list and thus know that there is no item with the given string as its printname.

Printname Runtime routines.

CVSI (string,@flag) - Convert string to item.
Forms hashcode from string. Searches down appropriate conflict list for a node pointing to the same string as the parameter string. If found it sets the variable flag to "FALSE" and returns the appropriate item number as found in the left half of the first word of the node. If such a node is not found then CVSI sets the variable flag to "TRUE" and returns a garbage result in accumulator 1. The calling sequence from the user's program is:

```
PUSH SP,word1    ;first word of string descriptor
PUSH SP,word2    ;second word of string descriptor
PUSH P,[FLAG]    ;address of flag variable
PUSHJ P,CVSI     ;call routine
```

CVIS (item,@flag) - Convert item to string.
Forms hashcode from item number. Searches down appropriate conflict list for a node which contains the parameter item number in the left half of its first word. If such a node is found it sets the flag to "FALSE" and pushes the two word string descriptor pointed to the left half of word2 of the node, onto the string stack(SP). If not found the NULL string is pushed onto the SP stack, and the variable FLAG is set to "TRUE" to indicate there was no printname for the given item.

The calling sequence is:

```
PUSH P,[item no.] ;item
PUSH P,[FLAG]     ;address of flag variable
PUSHJ P,CVIS      ;call routine
```

DEL_PNAME (item) - delete printname. Called to disassociate and item and its printname. It is a noop if the item has no printname. First of all, the item hashcode value is formed from the item number. This value is used as an index into the hash table thus yielding the address of the conflict list of nodes whose items hash to the same value. This list is searched for a node whose item entry is the same as the parameter. If no such node is found then DEL_PNAME simply returns. If the node is found, the node is then deleted from the item conflict list.The string pointed to by string entry in the node is temporarily saved on the string stack. The first word of the two word string descriptor is zeroed so that the string garbage collector will ignore that descriptor. The address of the head of the list of available string descriptors (contained in the left half of HASHP(USER)) is placed in the right half of the second word of the newly available string descriptor and the address of this descriptor is placed in the left half of HASHP(USER). Thus we have linked the newly freed string descriptor onto the list of available string descriptors.

We now must remove the node from the string conflict list We do this by forming the hashcode from the string which we have temporarily placed on the top of the string stack. We then chain down the string conflict list thus selected until we find the node we wish to delete. After deletion we now link the node onto the free list of two-word blocks, FP2(USER).

Calling sequence:

```
PUSH P,[item no.]
PUSHJ P,DEL.PNAME
```

NEW_PNAME(item,string) - give an item a new printname. First we check to see if printnames have been initialized. That is, if the hash table has been allocated. If not initialized we call the routine INITNM to do the allocate the table. Next we call the routine CVIS to see if the item already has a printname. If it does then we see if the oldname is the same as the new one in which case we simply return. If the names are different we issue a warning message to the user, then call DEL.PNAME to remove the old pname. Now we know that the item has no printname. We must now make sure that the new name is not the printname of some other item. We do this by calling CVSI. If CVSI tells us that the string is already in use as a pname we give an error message to the user. We now know that the printname does not already exist. We take an available string descriptor by calling the routine SDESCR (which will allocate a new string array and link them up if there are no more available). The string descriptor corresponding to the pname is placed in the string descriptor obtained from SDESCR. We now get a two word free to serve as a node for this pname-item binding. We place the address of the string descriptor into the left half of the second word and the item number in the left half of the first word. We then hash the item number to obtain the address of the item conflict list and link the node onto the list as its first element. Similarly we link the node onto the appropriate string conflict list. Finally we return.

Calling sequence:

```
PUSH P,[item no.]
PUSH SP,word1 of string descriptor
PUSH SP,word2 of string descriptor
PUSHJ P,NEW.PNAME
```