```
**************************************************
**************************************************
***                                            ***
***  Name:                                     ***
***                                            ***
***  Project:    1        Programmer:  MWK     ***
***                                            ***
***  File Name: TUTOR.DOC[DOC,AIL]             ***
***                                            ***
***  File Last Written:  19:30 13 Jul 1973     ***
***                                            ***
***  Time: 19:43         Date: 15 Jul 1973     ***
***                                            ***
***            Stanford University             ***
***       Artificial Intelligence Project      ***
***         Computer Science Department        ***
***            Stanford,  California           ***
***                                            ***
**************************************************
**************************************************
```

LEAP TUTORIAL

By Jim Low

I. INTRODUCTION

SAIL contains an associative data system called LEAP. It is patterned after the LEAP system implemented at LINCOLN LABORATORY by ROVNER and FELDMAN. Features contained in our LEAP but not in the Lincoln LEAP include a data-type called list, and "matching" procedures to be described below.

This document is intended to serve as a companion volume to the SAIL manual and hopefully will be easier to understand than the manual as here we can afford expound more on the various constructs and we also have the space to include more examples.

Other documents which may be of interest to the LEAP user include LEAP.WRU[LEP,JRL], which is a general guide to the leap runtime environment; LEAP.TXT[LEP,JRL], which is a detailed guide to the LEAP parts of the SAIL compiler and the SAIL runtime system; and of course the SAIL manual.

II. ITEMS

The basic entities which LEAP manipulates are called "items". An item is similar in many respects to a LISP atom. It optionally has a printname and a datum. A datum is a scalar or an array of any SAIL data-type other than types "item" and "itemvar". An itemvar is simply a variable whose values are items.

As an example of an item we may consider the following declaration.

```
REAL ITEM RITEM;
```

This declares an item named RITEM whose datum is a scalar real variable.

In addition to declarations of items at compile-time, we may dynamically create new items by calling the function NEW. This function may either have no arguments, (in which case the created item has no datum); or it may have a single argument which is either an expression or an array. This argument is copied and the copy becomes the value of the datum of the new item. We may of course later change the value of the datum or an element of the datum (if the datum is an array) by using standard algolic assignments. The data-type of the datum of the new item is the data-type of the argument to NEW. Thus NEW(1) would create a new integer item whose datum was initially given the value 1.

Items may be assigned to itemvars by standard SAIL assignment statements and assignment expressions.

```
itmvr← itmexpr;
```

Items themselves are considered to be constants and thus may not appear on the left hand side of an assignment statement or expression.

III. DATUMS

Associated with most items are datums which may be treated as standard SAIL variables. To refer to the datum of an item we use the operator DATUM.

Example:

```
    INTEGER ITEM II;
    INTEGER I;
L1: DATUM(II)←5;
L2: I ← DATUM(II)+1;
```

At L1 the datum of the item II is given the value "5". At L2, the value of the datum of II is used in an arithmetic assignment statement which would cause the variable I to receive the value 6.

Datum takes as its argument a typed item expression: Typed item expressions include:

1. Typed items and itemvars (declared with their type followed by ITEM as in:

```
INTEGER ITEM JJ;
INTEGER ARRAY ITEM X[1:5];
INTEGER ITEMVAR ARRAY Y[1:5];
INTEGER ARRAY ITEMVAR ARRAY Z[1:5];
INTEGER ARRAY ITEMVAR Q;
```

   Note the distinctions between the later four declarations. X is declared to be a single item whose datum is an integer array containing five elements. Y is declared to be an array of five itemvars, each of which is claimed to contain an item whose datum is a scalar integer. Z is declared to be an array of five itemvars whose values are claimed to be items with datums which are integer arrays. Q is an itemvar which supposedly contains an item whose datum in an integer array. As is shown above, we do not specify the dimensions of the the array which is the datum of an array itemvar. Thus for example, each element of Z could contain items whose datums were arrays of different dimensions. However for array items we must declare the dimension because otherwise the compiler would not know how much space to allocate for the array. We place the dimensions of the array following the item name. This is somewhat confusing as it appear that we have an

array of items rather than a single item whose datum is an array. SAIL has solved this problem in a very arbitrary way by outlawing declarations of arrays of items. One can get the effect of arrays of items by declaring itemvar arrays and then assigning "NEW" items to the individual elements.

2. Typed itemvar function calls:

```
STRING ITEMVAR PROCEDURE SNEW(STRING X);

    RETURN(NEW(X));
```

Thus we may talk of DATUM(SNEW("anystring"))

3. Assignment expressions whose left hand side is a typed itemvar.

The type of the datum variable is the type of its item expression. Thus, the datum of an integer item expression is treated as an integer variable, the datum of a real array item expression is treated as a real array and so forth.

NOTE: no check is actually made that the item is of the claimed type. Thus, for example disastrous things may happen if one uses DATUM on a string itemvar in which an integer item has been stored. Therefore the user should be careful about storing typed items into different type itemvars. When in doubt about the actual type an item expression he should use the function TYPEIT to verify that the item is of the required type. TYPEIT is a predeclared integer function.

```
INTEGER PROCEDURE TYPEIT(ITEMVAR X);
```

The values returned by TYPEIT are:

0 - deleted or never allocated	1 - item has no datum.
2 - bracketed triple(no datum)	3 - string item
4 - real item	5 - integer item
6 - set item	7 - list item
8 - procedure item	9 - process item
10 - event-type item	11 - context item
12 - refitm item	16 - string array item
17 - real array item	18 - integer array item
19 - set array item	20 - list array item
24 - context array item	25 - invalid type(error in LEAP)

Those codes not mentioned (13-15,21-23) are also invalid and should be considered erroneous.

IV. SETS

A set is an unordered collection of unique items. All set variables are initialized to PHI, the empty set consisting of no items. Set variables receive values by assignment or by placing individual items in them by PUT statements.

SET EXPRESSIONS:

1. Explicit Set - A sequence of item expressions which make up the set surrounded by set brackets.
   E. G.
   a) {item1,item2,item3}
   b) {item2,item1,item3}
   c) {item2,item2,item2,item3}

   Note: Since sets are unordered and a given item may appear at most once within a set, set expressions a,b,and c above all represent the same set.

2. PHI - the empty set. The set consisting of no elements at all is the empty set which may be written as either
   {} or PHI

3. Set Union - written SET1 ∪ SET2.
   The resultant set contains all items which are elements of either SET1 or SET2 or both.
   E.G.
   {item1,item2} ∪ {item2,item3} = {item1,item2,item3}

4. Set Intersection - written SET1 ∩ SET2
   The resultant set contains all items which are elements of both SET1 and SET2.
   E. G.
   {item1,item2,item3} ∩ {item1,item2,item4} = {item1,item2}

5. Set Subtraction - written SET1 - SET2
   The resultant set contains all items which are elements of SET1 but not elements of SET2.
   E. G.
   {item1,item2,item3} - {item2,item4,item5} = {item1,item3}

PUT and REMOVE

1. To place a single item into a set variable we may use the PUT statement:

   PUT itemexpr IN setvar;

   This has the identical effect as:

   setvar ← setvar ∪ {itemexpr};

   However, as the assignment causes the set to be copied, and the PUT doesn't the PUT statement will take less time and space to execute.

2. To remove a single item from a set variable we may use the REMOVE statement

   REMOVE itemexpr FROM setvar;

   This has the same effect as:

   setvar ← setvar - {itemexpr};

   Again, as the REMOVE statement avoids copying the set, it is more efficient than the equivalent assignment statement.

SET Booleans

1. Set membership

   itemexpr ∈ setexpr

   TRUE only if the item is an element of the set.

2. Set equality

setexpr1 = setexpr2

TRUE only if each item in setexpr1 is in setexpr2 and vice versa.

3. Set inequality

setexpr1 ≠ setexpr2

TRUE if setexpr1 or setexpr2 contains an item not found in the other.
Equivalent to

¬(setexpr1=setexpr2)

4. Proper containment

setexpr1 < setexpr2 or setexpr2 < setexpr1

TRUE if every item in setexpr1 is also in setexpr2, but setexpr1 ≠ setexpr2 Equivalent to: ((setexpr1 ∩ setexpr2) = setexpr1) ∧ (setexpr1 ≠ setexpr2)

5. Containment

setexpr1 ≤ setexpr2 or setexpr2 ≥ setexpr1

TRUE if every item in setexpr1 in also in setexpr2.
Equivalent to

(setexpr1 = setexpr2) ∨ (setexpr1 < setexpr2)

COP, LOP and LENGTH

1. COP (setexpr) - returns an item which is an element of the set. As sets are unordered you do not know which element of the set will be returned It is useful most often when we know the set has but a single element in which case it will return that item.

2. LOP (setvar) - same as COP except argument must be a set variable and the item returned is also removed from that set. It is logically equivalent to the following procedure:

```
ITEMVAR PROCEDURE LOP(REFERENCE SET Y);
BEGIN ITEMVAR Q;
        Q ← COP(Y);
        REMOVE Q FROM Y;
        RETURN(Q)
END;
```

LOP is often valuable if we wish some operation to be performed on each item of a set. Consider the example below where we wish the datums of all integer items contained in a set SETI to be incremented by one. Assume that we have declared IITMVR to be an integer itemvar and TSET to be a set variable which we will use as temporaries.

```
TSET ← SETI; "Copy set of interest into temporary"
WHILE (TSET ≠ PHI) DO "loop while TSET has elements"
BEGIN IITMVR ← LOP(TSET); "remove an element from TSET"
   IF TYPEIT(IITMVR) = 5 THEN "check if really integer"
        DATUM(IITMVR) ← DATUM(IITMVR) + 1;
END;
```

NOTE: LOP is compiled into code other than a straightforward procedure call and thus like many other functionals cannot appear as a statement but only as part of an expression. Thus if we just wanted to remove an arbitrary set element and throw if away we would have to say:

```
DMY ← LOP(SETVAR);
```

where DMY is an itemvar whose contents we do not care about, rather than the simpler:

```
LOP(SETVAR);
```

3. LENGTH(setexpr) - returns the number of items within a set. Logically equivalent to following procedure:

```
INTEGER PROCEDURE LENGTH(SET Y);
BEGIN "LENGTH"
   INTEGER COUNT; ITEMVAR DUMMY;
   COUNT ← 0;
   WHILE (Y ≠ PHI) DO
      BEGIN
         DUMMY ← LOP(Y);"remove an element from the set"
         COUNT ← COUNT +1; "step count of elements"
      END;
   RETURN(COUNT);
END "LENGTH";
```

The actual implementation of LENGTH is much more efficient than the above procedure (usually taking only two machine instructions). The most efficient way of determining if a given set is empty is to see if the LENGTH of that set is zero. This is actually much faster that comparing the set and PHI for equality.

V. LISTS

A list is an ordered sequence of items (not necessarily distinct). List variables are initialized to NIL the empty list containing no items. List variables receive values by assignment or by placing individual items in them by PUT statements.

LIST Expressions

1. Explicit List - written as the sequence of items (separated by commas) all surrounded by list brackets "{{ }}". SKIP 1
   a. {{ item1, item2, item3}}
   b. {{ item2, item1, item3 }}
   c. {{ item2, item2, item1, item 3 }}

   Note that since the order and number of times each item appears is important for each lists, the expressions a, b, and c above all represent different list expressions.

   NOTE: we may represent NIL, the empty list, by {{ }}

2. Concatenation - written list1 & list2 This forms a new list containing all the items in list1 followed by all the items in list2.
   E. G.

   {{item1, item2, item3,}} & {{item3, item4, item5 }}
   =
   {{item1, item2, item3, item3, item4, item5}}

   {{item1, item2}} & {{item 4, item4, item5}}
   =
   {{item1, item2, item4, item4, item5}}

3. Sublists - There are two forms of sublist expressions

   a. listexpr [i1 TO i2] - the first integer expression (i1) stands for the position of the first element to be taken and the second (i2) stands for the position of the last element to be taken.
      E. G.

      {{itema,itemb,itemc,itemd}}[2 TO 3]={{itemb,itemc}}

```
{{itema,itemb,itemc,itemd}}[3 TO 3]={{itemc}}
```

b. listexpr [i1 FOR i2] - the first integer expression (i1) stands for the position of the first element to be taken and the second (i2) stands for the number of elements to be taken. E. G.

```
{{itema,itemb,itemc,itemd}}[2 FOR 3]={{itemb,itemc,itemd}}

{{itema,itemb,itemc,itemd}}[3 FOR 1]={{itemc}}

{{itema,itemb,itemc,itemd}}[3 FOR 0]={{ }}= NIL
```

LIST Selectors

It is often useful to think of a list as an untyped itemvar array with a single dimension with lower bound 1 and upper bound variable.

1. Expression selector

   listexpr [i1] - returns the item which is the i1 element of the list. If i1 is less than 0 or greater than the number of elements of the list an error is signaled.
   E. G.

```
{{itema, itemb, itemc}} [1] = itema

{{itemb, itemc, itemd}} [2] = itemc
```

   Note the difference between listexpr[i1] and listexpr[i1 FOR 1]. The former returns an item and the later returns a list containing a single item.

2. Replacement selector

```
listvar[i1] ← itemexpr;
```

   This removes the i1 element of the list and replaces it with the itemexpr i1 must be between 1 and the number of elements in the list + 1.
   E. G.

```
LIST1 ← {{ITEM1}};
LIST1[1] ← ITEM2; "NOW LIST1 = {{ITEM2}}"
LIST1[2] ← ITEM3; "NOW LIST1 = {{ITEM2,ITEM3}}"
LIST1[1] ← LIST1[2]; "NOW LIST1 = {{ITEM3,ITEM3}}"
```

VI. ASSOCIATIONS

The associative power of LEAP comes from the use of associations, also called triples or associative triples. A triple is a 3-tuple of items. We write a triple as:

A⊗O≡V

where A, O, and V are items or item expressions. We call the first component of the triple (A above) the "attribute"; the second component (O above), the "object"; and the third component (V above), the "value". Triples are kept in the "associative store". Triples are inserted into the associative store by MAKE statements and removed from the associative store by ERASE statements.

MAKE

A MAKE statement is of the form:

MAKE itmexpr1⊗itmexpr2 ≡itemexpr3;

If the triple already exists in the associative store, the statement does nothing, otherwise the triple is inserted into the store.

ERASE

To remove a triple from the associative store we execute an ERASE statement:

ERASE itmexpr1⊗itmexpr2≡ itemexpr3;

If the triple is not in the associative store, the statement does nothing, otherwise the triple is removed from the associative store. We often wish to erase all the triples which have specific items as 1 or 2 of their components but we don't care about the remaining components. To do this we may use the token ANY to stand for the unspecified components.

E. G. to erase all associations with item2 as their object we could use:

```
ERASE ANY⊗item2≡ANY;
```

to erase all associations with item1 as their attribute and item2 as their object we would use:

```
ERASE item1⊗item2 ≡ ANY;
```

ANY may be used in 0, 1, 2, or all 3 positions in the triple. Thus,

```
ERASE ANY⊗ANY≡ANY;
```

would get rid of all associations in the UNIVERSE.

ASSOCIATIVE BOOLEANS

We may determine if a given triple exists by using the boolean expression:

itmexp1 ⊗ itmexp2 ≡ itmexp3

which will evaluate to TRUE if the triple containing the items exists in the associative store. As with ERASE 1 or 2 of the components may be ANY. Thus,

ANY ⊗ ANY ≡ item1

will yield the value TRUE if any triple in the associative store contains item1 as its value component.

DERIVED SETS

In order to use the associative nature of triples we must have ways of finding triples which have certain specified components. One way is to use derived sets. The other, FOREACH statements, will be discussed later.

There are three forms of derived sets now implemented: the (') derived set, the (⊗) derived set, and the (≡) derived set.

itmexp1 ' itmexp2

produces the set of all items X, such that

itmexp1 ⊗ X ≡ itmexp2

is a triple currently in the associative store.

itmexp1 ⊗ itmexp2

produces the set of all items Y such that,

itmexp1 ⊗ itmexp2 ≡ Y

is a triple in the associative store.

itmexp1 ≡ itmexp2

produces the set of all items Y such that.

Y ⊗ itmexp1 ≡ itmexp2

is a triple in the associative store

One or both of the item expressions may be the token ANY. Again this means that we do not care about the value of that component. Thus,

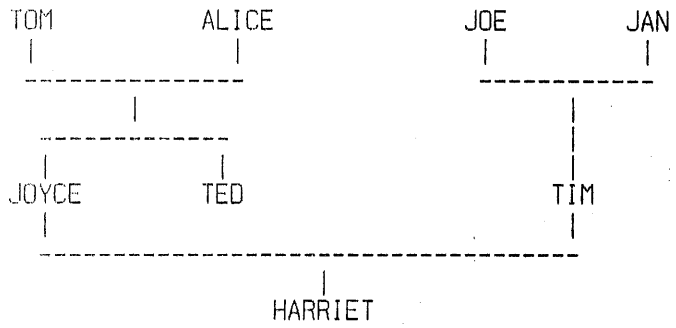
itmexp1 ⊗ ANY

will search the associative store for all associations which have itmexp1 as their attribute component, and will return the set of value components of such associations.

EXAMPLE -DERIVED SETS

Let us represent a family tree using associations. We will have the declared item PARENT and the sets MALE and FEMALE, as well as items representing members of the family: JOE, TIM, TED, JOYCE, JANET, ALICE, and HARRIET.

The familial relationships are represented by a the following tree.



Thus, the parents of HARRIET are JOYCE and TIM; the parents of JOYCE and TED are TOM and ALICE and so forth.

We can represent this tree by making the following associations:

```
MAKE PARENT ⊗ HARRIET ≡ JOYCE;
MAKE PARENT ⊗ HARRIET ≡ TIM;

MAKE PARENT ⊗ JOYCE ≡ TOM;
MAKE PARENT ⊗ JOYCE ≡ ALICE;

MAKE PARENT ⊗ TED ≡ ALICE;
MAKE PARENT ⊗ TED ≡ TOM;

MAKE PARENT ⊗ TIM ≡ JOE;
MAKE PARENT ⊗ TIM ≡ JAN;
```

To keep track of the sexes of the various people we have the two sets MALE and FEMALE.

```
MALE ← {TIM,TED,TOM,JOE};
FEMALE ← {JAN,JOYCE,ALICE};
```

NOTE: The above is merely one possible way we might represent the family tree. For example instead of the MALE and FEMALE sets, we

might have associations of the form: SEX ⊗ person ≡ MALE, where MALE is now an item. One of the interesting difficulties in using LEAP is deciding how to represent a given system of data as LEAP will often allow many different ways of representing the same information. Some of the tradeoffs between the different representations will be discussed later.

Now to use the structure we have built. Let us say that we wished to find the parents of Harriet. We may easily do this by use of a derived set.

```
HARRIET_PARENTS ← PARENTS ⊗ HARRIET;
```

where HARRIET_PARENTS has been declared to be a set variable.

To find Harriet's brothers is a little more complicated.

```
"find one parent"
PARENT_ITMVR ← COP(PARENTS ⊗ HARRIET);

"set of brothers,sisters"
SIBLING_SET ← PARENT ' PARENT_ITMVR;

"brother is a male sibling"
BROTHER_SET ← SIBLING_SET ∩ MALES;
```

The above example illustrates the use of associations as binary relations between items, in this case the relation "parent of". Often we wish to associate several different pieces of data with an item. To do this we may declare items which will be used to name the data and then allocate items which will contain the corresponding data for each item. For example we may wish to record such various attributes of a person such as weight, height, nickname. To do this we will have items WEIGHT, HEIGHT, and NICKNAME which will be used to name the attributes. We will allocate items whose datums are the corresponding values. E.G.

```
MAKE WEIGHT ⊗ JOE ≡ NEW(165);
MAKE HEIGHT ⊗ JOE ≡ NEW (70);
MAKE NICKNAME ⊗ JOE ≡ NEW("JOEY");
```

Then to find the value of an attribute such as weight we would use the expression:

```
DATUM(INT_ITMVR ← COP(WEIGHT⊗JOE))
```

Remember that the assignment of the item to the integer itmvr is required so that the compiler can tell what the data type of the datum is.

By using these operations and set variables we have sufficient power to do any search on the associative data base. However one soon realizes that they are very inconvenient to use in all but the most simple cases. Therefore another technique is provided called FOREACH statements.

A FOREACH statement is similar to a FOR statement in that it causes the iteration of a given SAIL statement to be performed with a control variable receiving various values for each iteration. These values are obtained by searching the associative data base or enumerating the members of a set of items.

The most simple FOREACH statement has a single "local" itemvar and a single "associative context". A local itemvar serves the same purpose as the loop variable in a FOR statement. With each iteration it will receive an item value and a SAIL statement will be executed. A simple example of a FOR statement is:

```
FOREACH X | BROTHER⊗BOY1≡ X DO
                <stmt>
```

This statement is equivalent to the following:

```
listx← BROTHER⊗BOY;
FOR j ← 1 step 1 until LENGTH(LISTX) DO
BEGIN  X←listx[j];
        <stmt>
END;
```