NBSIR 87-3622

Tutorial on Programming in LEMM and MACFOR

David F. Redmiles

U.S. DEPARTMENT OF COMMERCE
National Bureau of Standards
National Engineering Laboratory
Center for Applied Mathematics
Gaithersburg, MD 20899

July 1987

Sponsored by:

U.S. DEPARTMENT OF COMMERCE
National Bureau of Standards
National Engineering Laboratory
Center for Applied Mathematics
Gaithersburg, MD 20899

NBSIR 87-3622

TUTORIAL ON PROGRAMMING IN LEMM AND MACFOR

David F. Redmiles

U.S. DEPARTMENT OF COMMERCE
National Bureau of Standards
National Engineering Laboratory
Center for Applied Mathematics
Gaithersburg, MD 20899

July 1987

Sponsored by:
U.S. DEPARTMENT OF COMMERCE
National Bureau of Standards
National Engineering Laboratory
Center for Applied Mathematics
Gaithersburg, MD 20899

Tutorial on Programming in LEMM and MACFOR

Abstract:

Two FORTRAN 77 libraries called LEMM and MACFOR are described. These libraries were implemented to enable FORTRAN programmers to use list and structure oriented data types like those available in LISP and Pascal. The LEMM library combines ideas from the LISP programming language and the relational database model to provide a structured data type for representing and storing data. The MACFOR library implements many of the functions of the LISP programming language within FORTRAN and supports the implementation of the LEMM subroutines. The two libraries are intended to be used together and their expositions here are interposed with emphasis placed on using LEMM. However, the MACFOR library can be used independently, especially by programmers familiar with LISP. The LEMM and MACFOR libraries codify popular and proven concepts in data representation. Many programming examples are provided to make the use of these libraries understandable and applicable in the field.

Keywords: data structures, list programming, relational database, FORTRAN, LISP

Table of Contents

Tutorial on Programming in LEMM and MACFOR

1. Introducing LEMM and MACFOR

1.1 The LEMM and MACFOR libraries

This report is a tutorial on two FORTRAN 77 libraries called LEMM and MACFOR. These libraries were implemented to enable FORTRAN programmers to use list and structure oriented data types like those available in LISP and Pascal. The LEMM library combines ideas from the LISP programming language and the relational database model to provide a structured data type for representing and storing data. The MACFOR library implements many of the functions of the LISP programming language within FORTRAN and supports the implementation of the LEMM subroutines. The two libraries are intended to be used together and their expositions here are interposed with emphasis placed on using LEMM. However, the MACFOR library can be used independently, especially by programmers familiar with LISP.

The LEMM and MACFOR libraries were developed to implement an interactive graphics program [Notes, 1986] for editing and storing alloy phase diagrams These diagrams are graphs that give information about metallic systems. Many of the examples in the text will therefore relate to metals and elements.

1.2 Introducing LEMM

LEMM is a collection of subroutines for representing complex data structures in FORTRAN 77. LEMM provides a capability like that of the record data type in Pascal [Jensen, 1974] and the structure data type in PL/1 [IBM, 1984]. Specifically, LEMM gives FORTRAN users the ability to create non-homogeneous aggregates of the basic data types: integer, real, and character. Arrays of these aggregates and arrays within the aggregates can be created also. Moreover, the data structure created by the LEMM routines is more flexible than its Pascal and PL/1 counterparts since it can be created and modified at any time during the execution of the calling program; it is not restricted to a compile-time definition.

The data structure used by the LEMM subroutines is called a LEMM object. The form of the LEMM object derives from concepts in the programming language LISP and in relational database systems [Ullman, 1982; BCS, 1986]. Here is a LEMM object which is a data structure collecting pertinent information about the element hydrogen.

```
      (H
        (ako: element
         name: Hydrogen
         atomic-number: 1
         atomic-weight: 1.0079)
      )
```

The object "H" is represented by the values under the tags "ako (a-kind-of)", "name", "atomic-number", and "atomic-weight". These tags correspond to what are generally called attributes in a relational database and hereafter will be referred to by that term. In a relational database, attributes are often displayed as column headings. In LEMM, attributes are immediately followed by a colon to distinguish them from their values. Groups of attributes and their values are collected together as a list, denoted by opening and closing parentheses. This notation comes from LISP. Also, readers familiar with LISP will recognize attributes as "properties" of the object "H". In fact, the LEMM subroutines implement LEMM objects as a variant of the property list idea.

1.3 Introducing MACFOR

MACFOR is a collection of subroutines for using linked list data structures in FORTRAN 77. The implementation of the linked structure and the operators that can be applied to it are patterned after a dialect of MacLISP known as UCI LISP [Meehan, 1979]. MACFOR gives FORTRAN users access to many functions for manipulating sequential lists, association lists, and property lists. Equally important, it provides memory management through a garbage collector.

An example of an ordinary, sequential list follows. It collects four items, the names of four elements.

```
(Hydrogen Nickel Titanium Aluminum)
```

An association list might be used to associate these names with their abbreviated form:

```
((Hydrogen . H) (Nickel . Ni) (Titanium . Ti) (Aluminum . Al))
```

The association list and the property list (which will not be exemplified in this document) are known to advanced users of list programming; however, the user of the LEMM routines will have to understand only the most basic, sequential list.

While at first it may seem easier to implement one's own linked structures when needed, the MACFOR system offers some important advantages. Applications that use linked structures can grow to require a great variety of list operations and require a mechanism for data memory management. A small investment in learning and using the MACFOR routines can avoid a large effort in designing, programming, and debugging routines for linked list operations and memory management and can provide the user with routines using programming methods that would otherwise be unknown to him. All the MACFOR capabilities follow the general model laid out in LISP so that the programmer using MACFOR will be following common terminology and conventions for using lists and related structures. Also, using the MACFOR routines for basic list manipulations opens the door for use of the more powerful LEMM structures.

1.4 Using LEMM and MACFOR

The LEMM routines are intended to provide a means for storing and accessing information in an aggregate data structure; they provide no computation capability. What is unique about LEMM is its data structure, the LEMM object. What FORTRAN does best is computation. There is the division of labor. Use LEMM to store information when aggregate data structures are preferred over ordinary FORTRAN data structures. Use FORTRAN to perform the computation on the LEMM objects.

The LEMM routines represent LEMM objects internally as linked lists in an integer array workspace. This linked list internal representation is what allows LEMM objects to collect mixed data types. It also permits LEMM objects to be allocated and vary in length at any time during the execution of a program.

The linked representation is managed by the LISP-like MACFOR routines. However, the user does need to convert his FORTRAN data into the MACFOR workspace by calling conversion routines. These conversion routines return integer values which are "pointers" into the workspace. It is these pointers which are collected into the LEMM objects and otherwise processed when passed as arguments to the LEMM and MACFOR routines. To the user, they represent lists, object names, attributes for objects, and the attribute values as well.

Recall the hydrogen object listed in the section "Introducing LEMM". In a FORTRAN program, an integer variable would be used to hold the pointer to that object. To pass this object as an argument to the LEMM function which prints objects, the user would give to the print routine the name of the integer variable that holds the pointer. The following example program builds the hydrogen object and passes it to the print function.

```
      PROGRAM HPRT
*   DECLARE VARIABLES THAT WILL HOLD POINTER VALUES.
      INTEGER H,AKO,NAME,ATNUM,ATWGHT
*   INITIALIZE MACFOR; THE VALUE RETURNED FROM THIS
*   FUNCTION CAN BE IGNORED.
      I = MACINI()
*   MAKE AN IDENTIFIER FOR THE HYDROGEN OBJECT
      H = MAKEID('H')
*   MAKE IDENTIFIERS FOR THE ATTRIBUTES USED
*   IN AN ELEMENT OBJECT.  SAVE THEIR POINTERS.
      AKO = MAKEID('ako')
      NAME = MAKEID('name')
      ATNUM = MAKEID('atomic-number')
      ATWGHT = MAKEID('atomic-weight')
*   STORE ON H THE DESIRED ATTRIBUTE-VALUE COMBINATIONS.
      I = LPUT(H,AKO,MAKEID('element'))
      I = LPUT(H,NAME,MAKEID('Hydrogen'))
      I = LPUT(H,ATNUM,MAKEFX(1))
      I = LPUT(H,ATWGHT,MAKEFL(1.0079))
*   PASS THE HYDROGEN OBJECT TO THE PRINT ROUTINE.
      I = LPP(H,-1)
*
      END
```

The output of this program is simply the printed representation of the hydrogen object as shown earlier. That printing is accomplished with the one line invocation of function LPP. The preceding lines, the calls to the Lput function, actually are building the LEMM object for hydrogen. Lput puts attribute-value combinations on an object. Those attributes and values are specified in terms of their LEMM pointer representation. These pointers, in turn, were obtained from calls to the MACFOR "make" routines: MAKEID and MAKEFL.

The "make" routines are half of the conversion routines: they convert FORTRAN data types into pointers. They save into the MACFOR working storage the FORTRAN value they are passed, returning a pointer that can then be collected into a LEMM object or referenced by other LEMM and MACFOR routines. The other half of the conversion routines are the "undo" routines: they convert the MACFOR pointers into FORTRAN values. Their use will be discussed more later. While the use of conversion routines may seem awkward and even unnecessary, it is more efficient to leave the decision of when to convert up to the programmer than to have the LEMM and MACFOR routines do a FORTRAN-pointer conversion every time they are invoked.

Conversion routines make up the majority of function calls in this example but this disproportion should not discourage users: this example is a very simplistic use of LEMM and in other programs, a more balanced proportion of conversion, manipulation, and computation can be expected. In some cases, it is preferable to leave the majority of data in the pointer form and convert only interesting pieces for calculation. In other cases, the conversions are hidden away in subroutines. Examples of these two cases will be given later.

Although this example program has little practical value, it serves to demonstrate the major aspects of using LEMM and MACFOR: the use of LEMM and MACFOR through function calls, the use of integer variables to hold pointers, and the use of MACFOR conversion routines to make pointer data types out of FORTRAN data types.

2. Basic understanding and usage of pointer data types

2.1 LEMM and MACFOR pointer data types

As has been stressed, to create a list data type and a structure-like object, a system of pointers had to be implemented. Lists and objects are represented by pointers and their constituent numbers and strings also have to have pointer forms.

The LEMM and MACFOR routines provide for six pointer data types:

object	the principal LEMM data structure, a tagged aggregate of other data items including other objects;
list	a linked collection of other data items;
string	ASCII characters enclosed in quotation marks (");
flonum	a real number, single precision;
fixnum	an integer;

identifier	a name or word as opposed to a string.

All of these data types are implemented in FORTRAN as integer variables holding pointers. The object data type is implemented and manipulated by the LEMM routines with support from the MACFOR routines. Lists are built up and used through the MACFOR routines. Instances of string, flonum, fixnum, and identifier data types are created from FORTRAN values using the conversion routines of MACFOR. They are the basic building blocks for lists and objects and therefore are referred to sometimes as atoms.

In this document, the list, string, flonum, fixnum, and identifier data types are referred to as MACFOR data types because they are implemented entirely within the MACFOR routines. The integer workspace all the objects are stored in is also maintained entirely by the MACFOR routines so it is referred to as the MACFOR workspace.

2.2 Object

A description of objects was given in the section "Introducing LEMM". An object representation of the element hydrogen was given as the example:

```
(H
  (ako: element
   name: Hydrogen
   atomic-number: 1
   atomic-weight: 1.0079)
)
```

All the pieces of this object are identifiers except the fixnum "1" and the flonum "1.0079". Adding to the data structure a reference, here the fictitious reference "Periodic Table by Smith", gives an example of a string:

```
(H
  (ako: element
   name: Hydrogen
   atomic-number: 1
   atomic-weight: 1.0079
   reference: "Periodic Table by Smith")
)
```

Attribute-value combinations are stored on objects using the LPUT function and are retrieved with LGET:

```
YVAL = LPUT(NOB,NATT,YVAL)
PVAL = LGET(NOB,NATT)
```

In a FORTRAN program, all of the variables shown in these calling sequences would be declared integer: the assigned values, the arguments, and the functions themselves. Remember that all of the LEMM and MACFOR data types are represented by integer pointers and the same is returned through the function call. Hence comes the ability to compose function calls as seen in the first programming example (line 9).

The arguments in these calling sequences are NOB, NATT, and YVAL. NOB is the pointer to the object in question; NATT, the attribute; and YVAL, the value. LPUT stores or "puts" the value YVAL under the attribute NATT on the object NOB. LPUT returns the value it stored, namely the pointer passed in YVAL. LGET is the inverse: it retrieves or "gets" the list of values stored under the attribute NATT on the object NOB. LGET returns a pointer to this list. The notion of a "list" will be explained precisely in the next section.

LGET returns a list because more than one value can be associated with the same attribute on a given object. Assume for example that two references agree on the atomic weight of hydrogen and that is desirable to keep track of both. Assuming two fictitious references "Periodic Table by Jones and Jones" and "Periodic Table by Smith", the hydrogen object would become

```
(H
  (ako: element
   name: Hydrogen
   atomic-number: 1
   atomic-weight: 1.0079
   reference[1]: "Periodic Table by Jones and Jones"
   reference[2]: "Periodic Table by Smith")
)
```

The function call "PVAL = LGET(NOB,NATT)" where NOB points to the hydrogen object and NATT points to the word "reference" would assign to pval a pointer to the list collecting two values, the strings "Periodic Table by Jones and Jones" and "Periodic Table by Smith". In the next section, the way to access values in a list will be explained.

2.3 List

The list data structure is familiar to many programmers. The list data type supported by the MACFOR routines is like that of the LISP programming language. It consists of nodes called "cons cells" which have two pointers, one called the "car" and the other called the "cdr". Cons cells do not have an information field, only the two pointer fields. In simple lists, the cdr points to the next cons cell in the list while the car points to an item such as an identifier, string, flonum, or fixnum.

There are two common ways to depict lists. The first is with a diagram of boxes and arrows. The boxes represent cons cells with the left half depicting the contents of the car and the right half, the cdr. Arrows represent pointers to other cons cells or to datums. The list of the two references would be depicted as follows:

```
+---+---+                                +---+---+
| * | *-+------------------------------->| * | *-+---->NIL
+-+-+---+                                +-+-+---+
  |                                        |

"Periodic Table by Jones and Jones"      "Periodic Table by Smith"
```

It is evident that there are two cons cells, one to link each element in the list. The car of each cell points to an element of the list, the cdr points to the next cons cell. The last cdr points to something called NIL.

NIL is a pointer. Its special significance is that it is used to indicate the end of a list. Also it is used to mean the empty list or no value. If a call to LGET returns NIL then there are no values on a particular object under the specified attribute. The pointer NIL is made available to the FORTRAN user through a global variable in a common block as will be illustrated shortly.

The second common way to represent lists is to write the elements of the list within opening and closing parentheses. The list of two references would be written as follows:

```
("Periodic Table by Jones and Jones"  "Periodic Table by Smith")
```

This concise notation is clearly more practical. The box diagrams are mainly for visualizing the usage of the car and cdr.

The car and cdr of a list are accessed with the MACFOR functions MCAR and MCDR:

```
NVAL  = MCAR(PLIST)
PNEXT = MCDR(PLIST)
```

In these calling sequences, PLIST is a pointer to a cons cell in a list. MCAR returns the pointer in the car portion of that cell. Usually that is a pointer to a literal value as in the reference list example. MCDR returns the pointer in the cdr portion of the cons cell. That value will be a pointer to the next cons cell in the list or the value of the NIL pointer.

Lists are built up by combining car and cdr parts into cons cells using the MACFOR function MCONS:

```
PLIST = MCONS(NVAL,PNEXT)
```

This call may be thought of as the inverse or the recombining of the car and cdr calls illustrated above. PLIST is a pointer to a new cons cell whose car part is the pointer held in NVAL and whose cdr is the pointer held in PNEXT. The following code fragment builds the list of the two references illustrated above.

```
*   ACCESS THE COMMON VALUE OF THE NIL POINTER
      COMMON /MACATS/ NIL
*   INITIALIZE MACFOR
      I = MACINI()
*   MAKE MACFOR POINTER VERSIONS OF THE TWO REFERENCE STRINGS.
      IREF1 = MAKESG('Periodic Table by Jones and Jones')
      IREF2 = MAKESG('Periodic Table by Smith')
*
      PLIST = MCONS(IREF2,NIL)
```

```
           PLIST = MCONS(IREF1,PLIST)
```

Note that the list is built from the end to the beginning so that the current value of PLIST always will point to the current head of the list. Also note the use of NIL from the common block MACATS to terminate the list by being the last cdr. Additional issues pertaining to creating lists are discussed in the sections "Memory management" and "Protecting lists".

Users do not necessarily need to know how to build lists. However, because the multiple values returned by LGET are returned in a list, users do need to understand how to use MCAR and MCDR to access list values. An example of accessing LGET results will be given with the use of the print function.

A list may be printed with the MPRIN function:

```
     PLIST = MPRIN(PLIST)
```

MPRIN prints its argument using the parenthesis notation and returns the pointer it was passed. MPRIN can print all the data types except for objects.

MPRIN does not output a line terminator, i.e., carriage return line feed. The user terminates the line at an appropriate time with the MTERPR function:

```
     P = MTERPR(P)
```

P is any pointer and, as is implied in the calling sequence, MTERPR returns the pointer passed to it. It is not uncommon to see calls of the form:

```
     PLIST = MTERPR(MPRIN(PLIST))
```

Assume that the object NOB points to the version of the hydrogen object with the two references under the attribute "reference". The following code segment would retrieve the list of values and then (1) print the list and (2) print each value in the list on a separate line.

```
*    ACCESS THE COMMON VALUE OF THE NIL POINTER
       COMMON /MACATS/ NIL
*    INITIALIZE MACFOR
       I = MACINI()
*    NOB IS ASSIGNED THE HYDROGEN OBJECT
       ...
*    PREPARE THE ATTRIBUTE "REFERENCE"
       NREF = MAKEID('reference')
*    OBTAIN THE LIST OF VALUES UNDER ATTRIBUTE REFERENCE
       PLIST = LGET(NOB,NREF)
*    PRINT THE ENTIRE LIST POINTED TO BY PLIST
       PLIST = MTERPR(MPRIN(PLIST))
*    PRINT THE CAR VALUES OF THE SUCCESSIVE CONS CELLS
       PNEXT = PLIST
     5 IF (PNEXT .EQ. NIL) GOTO 10
          NVAL = MCAR(PNEXT)
          NVAL = MTERPR(MPRIN(NVAL))
```

```
    11            PNEXT = MCDR(PNEXT)
    12            GOTO 5
    13     10 CONTINUE
    14         ...
```

The loop of lines 7 through 13 is very important: it provides the pattern for accessing the values in the list of values under an attribute. LGET always returns a list of values even if there is only one item in that list. If the user expects only one item or is interested in only the first item of the values under an attribute, he can reduce the loop to the following single line call:

```
     NVAL = MCAR(LGET(NOB,NATT))
```

Once an individual item has been assigned to a variable, such as NVAL in this case, any computation can be performed on it. The pointer value can be converted into a FORTRAN variable, as will be shown later, and FORTRAN arithmetic, print statements, and subroutine calls can be applied to it. Alternatively, it might be left as a pointer so that it can be passed to other LEMM or MACFOR functions.

2.4 Literal values: fixnum, flonum, and string

The data types fixnum, flonum, and string correspond respectively to the FORTRAN data types integer, single precision real, and character. The different terminology helps distinguish between the pointer representation and the FORTRAN value. The fixnum, flonum, and string terms are also borrowed from LISP. Fixnum is a mnemonic for "fixed precision number" and flonum is a mnemonic for "floating-point number".

Values in these classes of data are the most common items used for attribute values. It was explained earlier that in order for the values of FORTRAN integers, reals, and character strings to be stored under attributes, the values first had to be converted to pointers into the MACFOR workspace. The following functions provide for this conversion:

```
     OVAL = MAKEFL(RVAL)
     NVAL = MAKEFX(IVAL)
     SVAL = MAKESG(CVAL)
```

MAKEFL takes a FORTRAN real number, either in the form of a literal or a variable, stores the value in the MACFOR internal workspace, and returns an integer pointer to the MACFOR pointer version of the value. The pointer version is necessary for compatibility with the cons cells. The pointer is the means for passing values as arguments to LEMM and MACFOR functions such as the LEMM function LPUT. MAKEFX converts an integer to a fixnum and MAKESG converts a FORTRAN character string to a MACFOR string. Remember that all the pointer values are FORTRAN type integer, so OVAL, NVAL, and SVAL would be declared integer.

Conversion from MACFOR pointer values back to FORTRAN variables is handled by the following functions:

```
     OVAL = MUDOFL(OVAL,RVAL)
     NVAL = MUDOFX(NVAL,IVAL)
     SVAL = MUDOSG(SVAL,ILEN,CVAL)
```

MUDOFL takes a pointer to a flonum, OVAL, and puts its value in a FORTRAN variable, RVAL; it "undoes" the FORTRAN to MACFOR conversion. MUDOFX undoes a pointer to a fixnum. MUDOFL and MUDOFX always return the pointer they were passed. If the pointer does not actually represent a flonum or fixnum, as the case may be, they supply 0 for the FORTRAN value.

MUDOSG undoes the MACFOR string SVAL into the supplied character variable, CVAL. MUDOSG will not exceed the declared length of CVAL. Also, MUDOSG puts the number of characters taken from SVAL in ILEN. If the pointer passed to MUDOSG is not a string, then 0 is returned for the length and NIL as the function value.

Using the conversion routines, the inner loop of the program segment for printing values in a list can be modified to use FORTRAN print statements:

```
*    PRINT THE CAR VALUES OF THE SUCCESSIVE CONS CELLS
       PNEXT = PLIST
     5 IF (PNEXT .EQ. NIL) GOTO 10
          NVAL = MCAR(PNEXT)
          NVAL = MUDOSG(NVAL,ILEN,CVAL)
          PRINT 100,CVAL(1:ILEN)
   100    FORMAT(1X,A)
          PNEXT = MCDR(PNEXT)
          GOTO 5
    10 CONTINUE
```

IF the type of a pointer is unknown, another routine is available for conversion:

```
     Y = MUDOUK(Y,ITYPE,IVAL,RVAL,ILEN,CVAL)
```

where Y is the pointer of unknown type. MUDOUK will undo the unknown pointer into IVAL if Y points to a fixnum, into RVAL if flonum, and into CVAL with length in ILEN if Y is a string. A type flag is returned in ITYPE: 0 if Y is not a simple data type, 1 if Y is a fixnum, 2 if flonum, and 3 if string. MUDOUK returns Y as its function value if Y is successfully converted, NIL otherwise.

2.5 Identifier

The MACFOR identifier data type is best described as a word or name. An identifier is a sequence of ASCII characters not enclosed in quotations and not consisting entirely of numbers and arithmetic signs.

Though there is no precise correspondent in FORTRAN to an identifier, the notion of a variable comes close. A LEMM object must have a name and only an identifier may serve in this role. In this sense, identifiers are used like variables. H was the identifier which gave a name to the hydrogen object. However, identifiers are used also in other contexts. Attribute names are usually identifiers although this is not a requirement. For users who know LISP, an identifier is the LISP identifier (non-literal) atom.

As with all the other LEMM data types, identifiers are represented by pointers and therefore require conversion routines. To deal with identifiers in FORTRAN, MACFOR treats them as character strings. The following routines make and undo identifiers:

```
     NID = MAKEID(CID)
     NID = MUDOID(NID,ILEN,CID)
```

These conversion routines behave exactly the same way for identifiers as they do for strings. MAKEID returns a pointer to the MACFOR identifier whose name consists of the characters in the FORTRAN character string CID. MUDOID stores in CID the characters in the name of the MACFOR identifier pointed to by the value in NID. The number of characters are stored in ILEN, 0 if NID is not an identifier. MUDOID returns NID if it does point to an identifier, otherwise NIL.

The conversion routine for unknown pointers, MUDOUK, also works on identifiers:

```
     Y = MUDOUK(Y,ITYPE,IVAL,RVAL,ILEN,CVAL)
```

If Y is an identifier, a value of 4 is stored in ITYPE, the number of characters in ILEN, and the character string version of the name in CVAL. For a successful conversion, MUDOUK returns Y; for an unsuccessful conversion, MUDOUK returns NIL.

Note that both identifiers and strings are case sensitive. Some FORTRAN compilers preserve upper and lower case letters in character strings. Thus, it should be assumed that the identifier "Hydrogen" is not equivalent to the identifier "hydrogen" because their first letters differ in case.

The choice of using an identifier or a string is left to the programmer. Sometimes the choice will be determined by programming considerations such as how the datum will be used in the future. In other cases, the choice is based upon a particular programmer's style. However, as a general rule, identifiers are good for naming objects or referring to abstractions and strings are good for specifying data that depends on a particular case. In hydrogen example mentioned earlier, the identifier "element" names the class of objects to which H belongs, is a-kind-of. The string "Periodic Table by Jones and Jones" names the reference from which data was taken for this particular element. The class "element" is general and many objects will belong to it; but a reference is very dependent on the object in question. Both the element symbol "H" and the element name "Hydrogen" are good for naming the object, and hence are made identifiers.

3. LEMM and MACFOR programming

3.1 Conventions for programming style

In the preceding sections, names for FORTRAN variables have included IVAL, RVAL, CVAL, NVAL, OVAL, and SVAL. The choice of the first letter for these variables is not random. Rather, these first letters obey the following convention, which can be implemented easily with a FORTRAN implicit statement:

First letter	implies variable contains
b	FORTRAN logical (b is for boolean)
c	FORTRAN character
i	FORTRAN integer
r	FORTRAN real
n	MACFOR pointer to an identifier
o	MACFOR pointer to a flonum (floating-point)
p	MACFOR pointer to a list
s	MACFOR pointer to a string
x	MACFOR pointer to a fixnum (integer)
y	MACFOR pointer to a datum whose type is not fixed or not known

These first letter conventions may at first seem extravagant. However, with practice, they are not hard to recall. More importantly, they allow the user to distinguish at a glance which variables contain FORTRAN values and which contain MACFOR pointers to values. At a finer level, they give the expected data type of the variable's value, whether actual or pointer.

Function names also follow a first letter convention:

First letter	implies a
L	LEMM function
M	MACFOR function

LEMM functions almost always have as their first argument a pointer to an identifier for an object name and perform object specific operations such as LPUT. MACFOR functions are more general; they perform conversions to the basic LEMM data types and list manipulations.

A uniform use of capitalization also improves the readability of code. The following guidelines are suggested for use with computers that permit mixed case in source files:

- use upper and lower case in comments as is natural for the language used;
- write global (FORTRAN common) variables entirely in upper case;
- write local variables entirely in lower case; and finally,
- make the first letter of a function name upper case and the remaining letters lower case.

Finally, if a function is being invoked only for an effect such as conversion, that is, it is not important to save the return value, make that fact obvious as in the following call:

```
      yeffct = Mudoid(nid,ilen,cid)
```

Future code examples will adhere to the above conventions.

3.2 Levels of attributes: LEMM objects within LEMM objects

The LEMM object for hydrogen in all its variations has only one level of attributes. The attributes in the object representing hydrogen are all followed by simple data types: identifier, string, fixnum, or flonum. However, it is possible for an attribute to have a more complex value, namely, another attribute-value list.

For instance, suppose it was desired to build a data structure to represent a curve of data points. An individual point might be represented as follows:

```
(point-1 (x: 1.0 y: 1.0))
```

Several point objects could be built and then stored on the curve object under the attribute "point":

```
(curve-1
   (ako: curve
    point[1]: point-1
    point[2]: point-2
    point[3]: point-3)
 )
```

Part of the code to build three such points and store them on a curve object follows. Assume that the variables nx, ny, and npoint have been initialized already to the identifiers x, y, and point:

```
1         curvel = Makeid('curve-1')
2         pointl = Makeid('point-1')
3         yeffct = Lput(pointl,nx,Makefl(1.0))
4         yeffct = Lput(pointl,ny,Makefl(1.0))
5         yeffct = Lput(curvel,npoint,pointl)
```

Lines 2 through 5 would have to be repeated for each point substituting "point-2" with x and y values of 2.0 and 4.0 for instance, and then with "point-3" with its values, say 3.0 and 9.0.

However, instead of making names for each point, it is possible to store the essential part of each point, i.e., its attribute and values, directly on the curve object. The curve object would be so improved as follows:

```
(curve-1
   (ako: curve
    point[1]: (x: 1.0 y: 1.0)
    point[2]: (x: 2.0 y: 4.0)
    point[3]: (x: 3.0 y: 9.0))
 )
```

The new object is said to have two levels of attributes: the first level for the curve attributes and a second for the point attributes.

Logically, both data structures are equivalent. The object is built using the same steps except instead of the user supplying a different name for each point, he uses a name generated by a LEMM function. The calling sequence to have LEMM generate a name takes two functions combined:

```
     name = Mntern(Mgensm())
```

The user only needs to know that the value returned can be used as the object for an Lput:

```
     yeffct = Lput(name,nx,Makefl(1.0))
```

The new call to Lput can be put more easily in a loop since a different variable for the name is not needed for each different point:

```
         DO 10 i=1,ipts
            name = Mntern(Mgensm())
            yeffct = Lput(name,nx,Makefl(rxpts(i)))
            yeffct = Lput(name,ny,Makefl(rypts(i)))
            yeffct = Lput(ncurve,npoint,name)
      10 CONTINUE
```

Here, the values for the x and y data points come from the arrays rxpts and rypts. The values could come equally as well from a calculation.

3.3 Writing data access routines

As the data structures for a program become more complex, it becomes convenient to isolate the LEMM Lput, Lget, and conversion operations into "data access subroutines". For instance, suppose a program is building many curve objects from FORTRAN data as described above and suppose that or another program needs to access the data as arrays. Two useful routines could be written: one to build and return LEMM curve objects from FORTRAN arguments, and one to store into FORTRAN arguments data taken from a LEMM object. The code for two such routines follows:

```
        INTEGER FUNCTION Iptput(cname,ipts,rxpts,rypts)
*   Store data points on a curve, cname.  Return the pointer
*   to the LEMM object corresponding to cname.
*
*   (cname
*      ( ...
*       point[i]: (x: rxpts-i y: rypts-i)
*       ))
*
*   Assume naming conventions
        IMPLICIT INTEGER      (I,N,O,X,P,S,Y)
        IMPLICIT REAL         (R)
        IMPLICIT CHARACTER*80 (C)
*
        DIMENSION rxpts(*),rypts(*)
        CHARACTER cname*(*)
*   NPOINT points to the identifier "point",
*   NX to "x", and NY to "y".
      COMMON /ATTS/ NPOINT,NX,NY
*   Get the LEMM pointer equivalent to cname
      name = Makeid(cname)
*   For every x-y pair, generate an attribute-value list
*   to hold x and y, then store that list under the point
*   attribute of name.
      DO 10 i=1,ipts
         ntemp = Mntern(Mgensm())
         yeffct = Lput(ntemp,NX,Makefl(rxpts(i)))
         yeffct = Lput(ntemp,NY,Makefl(rypts(i)))
         yeffct = Lput(name,NPOINT,ntemp)
   10 CONTINUE
*
      Iptput = name
*
      END
*
      INTEGER FUNCTION Iptget(cname,ipts,rxpts,rypts)
*   Retrieve data points from the LEMM object corresponding
*   cname, storing them in rxpts and rypts with the number of
*   points in ipts.  Return the pointer to the LEMM
*   object searched.
*
*   (cname
*      ( ...
*       point[i]: (x: rxpts-i y: rypts-i)
*       ))
*
*   Assume naming conventions
      IMPLICIT INTEGER      (I,N,O,X,P,S,Y)
      IMPLICIT REAL         (R)
      IMPLICIT CHARACTER*80 (C)
*
      DIMENSION rxpts(*),rypts(*)
      CHARACTER cname*(*)
*   NPOINT points to the identifier "point",
*   NX to "x", and NY to "y".
      COMMON /ATTS/ NPOINT,NX,NY
*   Access the NIL pointer
      COMMON /MACATS/ NIL
*   Get the LEMM pointer equivalent to cname
      name = Makeid(cname)
*   The x-y pairs are stored as attribute-value lists under
*   the attribute point on the object.
      pnext = Lget(name,NPOINT)
      i     = 0
    5    IF (pnext .EQ. NIL) GOTO 10
         i      = i + 1
         ntemp  = Mcar(pnext)
         yeffct = Mudofl(Mcar(Lget(ntemp,NX)),rxpts(i))
         yeffct = Mudofl(Mcar(Lget(ntemp,NY)),rypts(i))
         pnext  = Mcdr(pnext)
         GOTO 5
   10 CONTINUE
*
      Iptget = name
*
      END
```

Thus the user can hide all references to LEMM, allowing others to take advantage of his data structures without having to understand the LEMM routines. Further, the user can manage the data in his program in a more modular fashion.

Attributes used by several different programs can be kept on the same object. Several data access routines can be written, each picking out just the attributes of interest to the calling program. Since all attributes are preserved when reading and writing with Lread and Lpp, one master file can be kept for all the programs.

3.4 Reading and writing LEMM and MACFOR data to files

The first code example used the Lpp function to print a LEMM object representing the element hydrogen. The calling sequence for Lpp and for another printing function follow:

```
      nob = Lpp(nob,ilevl)
      nob = Lprin(nob)
```

In both calls, nob is the LEMM object to be printed. It is the value returned for both functions. Lpp stands for "LEMM object pretty-print" and Lprin, "LEMM object print".

Pretty-print implies that the object will be printed in a pleasing format, breaking long output lines at logical points and using indentation. The goal is an aesthetically pleasing format. Lpp also permits the user to control how many levels of attributes are printed. LEMM objects which have more than one level of attributes will be illustrated later. Using a FORTRAN value of -1 for ilevl causes all levels to be printed.

Lprin outputs a lemm object without using indentation. It breaks lines upon reaching the right margin. Lprin automatically prints all levels of attributes.

Use Lpp most of the time; it is the only form easily read by people. Use Lprin only if it is necessary to save the spaces incurred by the indentation used by Lpp.

After using either function, the Mterpr function should be called to force a line terminator. Alternately, the print function call could be surrounded by the Mterpr: e.g., "nob = Mterpr(Lpp(nob,-1))".

LEMM objects are read using the Lread function:

```
      nob = Lread(iccode)
```

Lread reads the next LEMM object from the input unit, stores that object into the workspace, and returns the pointer to the object. A condition code is also returned in iccode. This flag should be compared against one of three common variables: IOKCC, IERRCC, or IDONCC. "IF (iccode .EQ. IOKCC)" then there was no problem reading the returned object. "IF (iccode .GE. IERRCC)" then an error occurred during reading. Finally, the most common use of the condition code is to check if end-of-file has been reached: "IF (iccode .GE. IDONCC)" then there is no more input. Note that a combined value (sum) of IDONCC and IERRCC would be returned if end-of-file were encountered in the middle of an object.

Lread accepts input in free format. The LEMM object can be written entirely on one line (if it fits) or broken however is convenient for the user. The read routine can read objects printed by either of the print functions.

Typically, the user has one program which builds LEMM objects with the Lput and conversion routines, and saves them to a master file calling on one of the print routines. The program might be a modeling or interactive application such as is common in computer aided design. Later other programs or the same program can recover the earlier state of data by reading the master file with the Lread function.

The LEMM input and output units are controlled in the MACFOR routines and by default work from the standard input and output units, commonly an interactive terminal. However, input and output can be redirected to different FORTRAN units using the following MACFOR functions:

```
ilast = Mnunt(iunit)
ilast = Mount(iunit)
```

The function Mnunt changes the input to read from the new unit, iunit. The function Mount changes the output to write to the new unit, iunit. Both functions return the unit number previously set. By saving the previous value, the user can switch back to that file or device when appropriate.

The following example program demonstrates all of the functions related to reading and writing LEMM objects. It is a program to copy one by one the objects out of one disk file into another disk file.

```
       PROGRAM Copy
*
*   Program to copy all the LEMM objects from
*   one file to another
*
*   Assume naming conventions
       IMPLICIT INTEGER      (I,N,O,X,P,S,Y)
       IMPLICIT CHARACTER*80 (C)
*   Include common block with I/O condition codes
       COMMON /MACCDS/ IOKCC,IERRCC,IDONCC
*   Initialize MACFOR
       yeffct = Macini()
*   Get file names
       PRINT *,'Source file?'
       READ  *,cinfil
```

```
      PRINT *,'Destination file?'
      READ  *,coufil
*   Open files
      OPEN (FILE=cinfil,UNIT=8,STATUS='OLD')
      OPEN (FILE=coufil,UNIT=9,STATUS='NEW')
*   Switch MACFOR input and output units accordingly,
*   saving their current values
      inunt = Mnunt(8)
      iount = Mount(9)
*   Loop reading LEMM objects from the source file
*   and writing them to the destination file
*   until no more objects can be read.
    5    nob = Lread(iccode)
         IF (iccode .GE. IDONCC) GOTO 10
         nob = Mterpr(Lpp(nob,-1))
         GOTO 5
   10 CONTINUE
*   Switch input and output units back before closing
*   files.
      yeffct = Mnunt(inunt)
      yeffct = Mount(iount)
*
      CLOSE (8)
      CLOSE (9)
*
      END
```

The "copy until end-of-file" program pattern is common to many applications. Within the read loop, the pointers of the newly read objects could be saved as FORTRAN array elements for later processing or the values of an attribute of particular interest could be extracted, converted to FORTRAN variables, and be used in some calculation. A fun example would be to read a source file containing one object for each of the known elements and calculate the average atomic weight:

```
      natwgt = Makeid('atomic-weight')
      icount = 0
      rtotal = 0.0
    5    nob = Lread(iccode)
         IF (iccode .GE. IDONCC) GOTO 10
         icount = icount + 1
         yeffct = Mudofl(Mcar(Lget(nob,natwgt)),ratwgt)
         rtotal = rtotal + ratwgt
         nob = Mterpr(Lpp(nob,-1))
         GOTO 5
   10 CONTINUE
      PRINT *,'Average atomic weight is ',rtotal/float(icount)
```

Reading and writing MACFOR data types is essentially the same as discussed for LEMM objects. The Mnunt and Mount functions are used to modify the input and output unit numbers and the same condition codes apply when reading.

The print function for MACFOR data, Mprin, was already illustrated in the section on the list data type. Its calling sequence is as follows:

```
     ydata = Mprin(ydata)
```

The parameter ydata may point to any MACFOR data type: list, string, flonum, fixnum, or identifier; Mprin automatically determines the type of its argument and formats it in the minimal space necessary. The call to Mprin is usually surrounded by or followed by a call to the function Mterpr to terminate the line with a line feed and carriage return.

The function for reading MACFOR data types is Mread. Its calling sequence is as follows:

```
     ydata = Mread(iccode)
```

Mread reads the next item from the input unit, stores it in the MACFOR workspace, and returns the pointer to it. In this case, the returned pointer would be stored in the variable ydata. A condition code with the same meaning as for Lread is returned in the parameter iccode. The type of item read does not matter; Mread is designed to distinguish between strings and numbers and identifier names. A space or line break is enough to separate such items. Lists are plainly delimited by their opening and closing parentheses.

To modify the Copy program to read and copy lists instead of LEMM objects, lines 14 through 18 would be replaced with the following:

```
         *   Loop reading MACFOR items from the source file
         *   and writing them to the destination file
         *   until no more objects can be read.
     14      5   ydata = Mread(iccode)
     15          IF (iccode .GE. IDONCC) GOTO 10
     16          ydata = Mterpr(Mprin(ydata))
     17          GOTO 5
     18     10 CONTINUE
```

Sometimes, it is desirable to annotate output or even to separate items with a space. Indeed, all prompting and input in a program can be channeled through the MACFOR routines. MACFOR provides a function for printing FORTRAN character strings directly. Its calling sequence follows:

```
     icstg = Mlpcsg(cstg)
```

Mlpcsg takes the FORTRAN character string in the variable cstg, prints up to the last non-blank character on the current output unit, and returns as a FORTRAN integer, the number of characters printed. Mlpcsg always prints at least one character so the following call would print a space on the output:

```
     icstg = Mlpcsg(' ')
```

Remember to use Mterpr if a new line is needed after the print:

```
      1        ieffct = Mlpcsg('Line of annotation')
      2       yeffct = Mterpr(NIL)
```

Functions for printing real and integer data directly as well as executing tabs are described in the appendices.

3.5 Memory management

New pieces of LEMM and MACFOR data are created with the make routines, the Mcons routine, the Lput routine, and the read routines. All the data is stored in the integer array workspace of MACFOR. As new data enters and old data is discarded by reassignment of variable values, memory gets used up. Eventually, memory would be completely used up unless discarded data could be reclaimed.

One way memory can be reclaimed is by requiring the programmer explicitly to call routines to free obsolete pointers. Another approach is to allow the programmer to overwrite pointer variables freely and later sort through memory, placing discarded data back into the free space. The latter process is known as garbage collection [Standish, 1979]. It is the form of memory management used in the programming language LISP and adopted for MACFOR. It has the advantage that it avoids encumbering an algorithm with explicit memory management calls; very often programs never use up all of the available free space and the garbage collection does not even occur.

Garbage collection is programmed into MACFOR and is triggered automatically when no more free cells are available in the MACFOR workspace. A message printed to the default FORTRAN output unit informs the user when garbage collection is beginning. A second message indicates when garbage collection is finished and how many cells were reclaimed.

MACFOR data created by the conversion routines and LEMM objects built with the Lput routines or read with the Lread routine are all left untouched by a garbage collection. However, these routines generate garbage as a side effect of their use. This garbage may be thought of as scratch space for creating the final product returned to the user's program. Programmers who use the LEMM data structure need not be concerned about the garbage collection process except for the fact that it can cause a 30 second delay in program execution. For many programs, though, garbage collection is infrequent if it occurs at all.

On the other hand, programmers who create lists with Mcons or read them with Mread have to protect the pointers to these lists from garbage collection. Protection is not complicated and is introduced to the programmer interested in using lists in the section, "Protecting lists".

4. Advanced list programming

4.1 Protecting lists

In the section on memory management, it was stated that MACFOR data created by the conversion routines and LEMM objects built with the Lput routines or read with the Lread routine are all left untouched by a garbage collection.

In a sense, the garbage collector of MACFOR "knows" about the instances of the data created by these routines and because it keeps track of them, it can protect them from a garbage collection. What actually happens is that these data are associated with atoms or are themselves atoms and all the atoms are kept track of on a hash table known in some LISP implementations as the "oblist" (for object list). When the garbage collector runs, it assumes all of the items on the oblist are in use and should not be collected.

However, lists that are created in the MACFOR workspace with the Mcons routine or with the Mread routine are not watched by MACFOR. If a garbage collection occurs during the execution of a program segment that assigns the result of a Mcons or Mread to a list pointer variable, the list cells pointed to by that variable would be returned to the free space and the list would be rendered invalid. Therefore a mechanism is provided to the programmer for protecting new lists from collection. This mechanism is known as marking.

Consider again the example of building the list of two references from the section on lists, here rewritten to conform to the programming conventions:

```
    *   Code to build the list:
    *      ("Periodic Table by Jones and Jones"
    *       "Periodic Table by Smith")
    *
    *   Access the common value of the NIL pointer
1         COMMON /MACATS/ NIL
    *   Initialize MACFOR
2         yeffct = Macini()
    *   Make MACFOR pointer versions of the two reference strings.
3         sref1 = Makesg('Periodic Table by Jones and Jones')
4         sref2 = Makesg('Periodic Table by Smith')
    *   Build the list.
5         plist = Mcons(sref2,NIL)
6         plist = Mcons(sref1,plist)
```

The variables sref1 and sref2 are safe from garbage collection; they point to string atoms which are maintained on the oblist. Since the variable plist points to a list which is just being created, its value is not safe from collection. It will be explained shortly how the programmer can mark such a list safe from garbage collection.

It should be emphasized that marking is important only to programmers wishing to create new lists. Variables that point into existing lists, such as the lists returned by an Lget, do not need any special protection; they are already safe because they are part of a structure associated with an atom on the oblist, the identifier atom that names the LEMM object.

Marking is part of the garbage collector already. The garbage collector works in two phases. The first phase is the marking phase in which all items in the working storage known or assumed to be in use are marked by turning on a mark bit reserved for each memory cell. The second phase is the collection phase in which all unmarked cells are reassociated with the free cells of the MACFOR workspace.

During the marking phase, all of the atoms on the oblist and all the cells associated with them are marked. The garbage collector then calls a routine, Msymrk, for marking system variables that are not associated with anything on the oblist. The last step in the marking phase is to call the routine, Murmrk, for marking user variables.

The function Murmrk is the user's opportunity to protect his list pointer variables from garbage collection. A default version of Murmrk is provided; it takes no action. If the user has variables to protect, he supplies an appropriate version of Murmrk. The user's version of Murmrk would parallel functionally the system marking function. Consider as an example the routine that marks the scratch list variables used in the LEMM system:

```
      1        INTEGER FUNCTION Mlemrk(imkbit,itgbit)
        *
        *   This routine protects from garbage collection some local and
        *   global variables in LEMM. All the variables to be protected
        *   are equivalenced to elements of the global array LEMPCT so that
        *   marking its elements, marks the equivalenced variables.
        *
        *   Assume naming conventions
      2       IMPLICIT INTEGER      (I,N,O,X,P,S,Y)
        *   Access the common value of the NIL pointer
      3       COMMON /MACATS/ NIL
        *   The mark and tag bit arrays are for passing to the mark routine.
      4       DIMENSION imkbit(*),itgbit(*)
        *   Include array holding pointers to protect.
      5       COMMON /LEMPCT/ LEMPCT(36)
        *   Mark each element of the array.
      6       DO 100 i=1,36
      7          yeffct = M0mark(LEMPCT(i),imkbit,itgbit)
      8   100 CONTINUE
        *   Return NIL.
      9       Mlemrk = NIL
     10       END
```

Line 1 declares the marking function. The programmer would of course substitute here the name Murmrk. The two arguments are work arrays for the marking and collection processes. They are already allocated by MACFOR. The programmer should pass them to the marking subroutine M0mark as in line 7.

The array variable declared as common in line 5 is the key to protecting user list variables. Its values are the ones passed to the marking algorithm. It is intended that the MACFOR programmer will set up such an array variable in a common block and in the appropriate subprogram modules, equivalence variables needing protection to elements of this array. The array would have one element for each pointer variable needing protection and all elements should be initialized to NIL. When garbage collection is triggered, this array would share the current pointer values of the variables needing protection -- although these variables could be from different program units -- and when Murmrk is called, the variable values would be marked safe. For example, it was noted that the variable plist in the program segment above needed protection. The following two lines inserted after line 2 of the reference list example, along with a Murmrk routine patterned after Mlemrk would suffice to protect this variable:

```
      1       COMMON /PURPCT/ PURPCT(1)
      2       EQUIVALENCE (plist, PURPCT(1))
```

Again, because the variable plist is equivalenced to a common variable, its value can be accessed from the Murmrk subroutine called by the garbage collector. When a program unit finishes with the value of a protected variable, it should set that variable to NIL to free up its old value for the next garbage collection. However, routines that will be nested in function calls should not wipe out their return value. They should keep it protected in case a garbage collection occurs before the result of the nested calls can be protected. Another example is given in the next section, "Simulating recursion".

4.2 Simulating recursion

The basic requirement of recursion is to have available a stack. Stacks are easily implemented with lists and MACFOR provides the standard push and pop stack routines:

```
     yval = Mpush(pstack,yval)
     yval = Mpop(pstack)
```

Mpush puts the value yval at the head of the list pstack and returns yval for its value. Mpop returns the value at the head of the list and sets the list pstack to the remainder of the list. Thus both functions modify their first argument, the stack variable: a variable name therefore should always be used in this position. Stack variables should be initialized to NIL so it is clear when they are empty. Further, stack variables are newly created lists and accordingly must be protected as described above.

With the availability of stacks, recursion can be simulated by judicious use of Mpush and Mpop to save arguments on recursive calls and restore them on returns. The following example is a function to return a copy of an arbitrary list structure. It recurses down the car and cdr of a list until it reaches an atom. As the recursion unwinds, a new list is built with Mcons. The function uses stacks to save pointers when traversing the list, to save values returned at different depths of recursion, and to indicate where a "return" from a recursive call should be. The listing of the function follows:

```
      INTEGER FUNCTION Mcopy(yarg)
*************************************************************************
* MACFOR copy                                                           *
*************************************************************************
*                                                                       *
*   Copy the pointer structure of yarg and return the copy.  This       *
*   function is equivalent to the following LISP function:             *
*                                                                       *
*       (defun Mcopy (y)                                                *
*          (cond ((atom y) y)                                          *
*                (t (cons (Mcopy (car y))                              *
*                         (Mcopy (cdr y))))                            *
*           )                                                          *
*         )                                                            *
*                                                                      *
*************************Implicit Declarations*************************
*                                                                      *
      IMPLICIT INTEGER      (A,G,I,N,O,X,P,S,Y)
*                                                                      *
**********************Local Variable Declarations**********************
*                                                                      *
*   yarg      s-expression to copy                                     *
*                                                                      *
*   nwhr      signals where to return                                  *
* + pretsk    stack of return values                                   *
* + pwhrsk    stack return locations                                   *
* + pysk      stack of s-expr to copy                                  *
*   y         current s-expr to copy                                   *
*   yoret     old return value (result of copy of car)                 *
* + yret      return value                                             *
*                                                                      *
*************************Global Declarations***************************
*                                                                      *
*   NIL,T
      INTEGER                  T
      COMMON /MACATS/ NIL,IPNIL,T
*   MACPCT
      COMMON /MACPCT/ MACPCT(40)
*                                                                      *
*****************************Equivalence*******************************
*                                                                      *
* + pretsk,pwhrsk,pysk,yret
      EQUIVALENCE (pretsk,MACPCT(32)),(pwhrsk,MACPCT(33)),
     +            (pysk,MACPCT(39)),(yret,MACPCT(40))
*                                                                      *
******************************Procedure********************************
*                                                                      *
*   don't overwrite argument
      y      = yarg
*   initialize stacks
      pretsk = NIL
      pwhrsk = NIL
      pysk   = NIL
*   if y is atomic, return immediately
   10 IF (Matom(y) .EQ. NIL) GOTO 20
         yret   = y
         GOTO 50
*   y is a list, recurse with its car
   20 yeffct = Mpush(pysk,y)
      y      = Mcar(y)
      yeffct = Mpush(pwhrsk,T)
      GOTO 10
*   car has been copied, restore the old y and
*   recurse with its cdr
   21 yeffct = Mpush(pretsk,yret)
      y      = Mpop(pysk)
      y      = Mcdr(y)
      yeffct = Mpush(pwhrsk,NIL)
      GOTO 10
*   car and cdr have been copied, cons them together, return
   23 yoret  = Mpop(pretsk)
      yret   = Mcons(yoret,yret)
*     GOTO 50
*   soft return, decide by value on return stack
   50 IF (pwhrsk .EQ. NIL) GOTO 60
         nwhr   = Mpop(pwhrsk)
         IF (nwhr .EQ. T) GOTO 21
            GOTO 23
*   real return (when return stack is empty)
   60 Mcopy = yret
*   clear protected values
      pretsk = NIL
      pwhrsk = NIL
      pysk   = NIL
*
      END
```

5. Summary

The LEMM and MACFOR libraries bring structured data organization to FORTRAN users. The LEMM library combines ideas from the LISP programming language and relational database systems to provide a high-level data structure and file representation for structured data. MACFOR implements many of the functions of the LISP programming language within FORTRAN to support the LEMM subroutines. The MACFOR library can be used independently of LEMM by programmers interested only in list programming. Conversely, the LEMM structure can be used with minimal of list programming or of the MACFOR routines.

The LEMM and MACFOR libraries are implemented in FORTRAN 77 and are available immediately. A small investment in learning and using these routines can avoid a large effort in designing, programming, and debugging one's own routines for linked list operations and memory management. The MACFOR routines are patterned after a proven model for list programming, that of the LISP programming language. If desired, references to the LEMM and MACFOR routines can be relegated to subroutines to insulate the casual or applications end user.

LEMM was designed to benefit applications in which human users need to interact with structured data. The printed form of the LEMM data structure is symbolic. This characteristic allows it to be more easily shared between machines and databases. It also allows viewing and modification by people using only an ordinary text editor. The use of attribute-value combinations makes the stored format more natural to human viewers as well as more compatible with commercial relational database software.

The LEMM and MACFOR libraries codify popular and proven concepts in data representation. Many programming examples are included in this document to make the use of these libraries understandable and applicable in the field.

6. Acknowledgments

The author wishes to thank Anthony Datel for helping to code LEMM and MACFOR; Jim Sims for commenting on early drafts of this report; and Don Orser for putting forward his ideas on representing scientific data.

References

BCS (1986) Boeing RIM User's Manual, Document Number: 20492-0502, Boeing Computer Services, Seattle, Washington, October 1986.

IBM (1984) OS and DOS PL/I Language Reference Manual, Form GC26-3977-1, IBM Corporation, 1984.

Jensen, Kathleen and Niklaus Wirth (1974) Pascal User Manual and Report, Springer-Verlag, New York, New York, 1974.

Meehan, James R. (1979) The New UCI LISP Manual, Lawrence Erlbaum Associates, Publishers, Hillsdale, New Jersey, 1979.

Notes (1986) User's Guide FHAZE, unpublished notes, November, 1986.

Standish, Thomas A. (1979) Data Structure Techniques, Addison-Wesley, Reading, Massachusetts, 1979.

Ullman, Jeffrey D. (1982) Principles of Database Systems, Computer Science Press, Rockville, Maryland, 1982.

Appendix A: Common Blocks

This appendix lists in full the common block declarations used in the example programs in the text. Unless declared otherwise, all the variables adhere to the standard FORTRAN type defaults based on the first letter of their names.

The common block MACATS includes predefined atoms: NIL, the empty pointer; T, which implies "truth" (not NIL); and PEOLCH, the end-of-line character detected by the read routine Mrdeol.

```
      INTEGER                     T
      COMMON /MACATS/ NIL,IPNIL,T,IATMHR,IFXNUM,IFLNUM,ISTRNG,IPNAME
     +               ,IOBJCT,PEOFCH,PEOLCH
      INTEGER                 PEOFCH,PEOLCH
```

The common block MACCDS holds the condition codes returned by all the read routines: IOKCC, fully successful read; IERRCC, an error occurred during the read; and IDONCC, end-of-file was detected during the read.

```
      COMMON /MACCDS/ IOKCC,IERRCC,IDONCC
```

Two variables not discussed in the text but which the user may find useful are the left and right margin variables: IOLMGN and IORMGN. They are found in the common block MACIO and can be set directly by the user to values between 1 and 80.

```
      CHARACTER*80 CNFMT,CNLINE,COFMT,COLINE
      LOGICAL LSLSFY
      COMMON /MACIO/ INUNT,INMXLL,IOUNT,IOMXLL,IEUNT,IEMXLL
     +          ,INBUFF(81),INPTR,IOBUFF(80),IOPTR
     +          ,IOLMGN,IORMGN,IPRTLV,LSLSFY
     +          ,INSNCO,INSCO,IFCSCO,ISCO
     +          ,CNFMT,CNLINE,COFMT,COLINE,ISYSIN,ISYSOU
```

Though not a common block, the implicit statements used for the programming naming conventions discussed in the text are listed below:

```
      IMPLICIT INTEGER      (A,G,I,N,O,X,P,S,Y)
      IMPLICIT REAL         (R)
      IMPLICIT CHARACTER*80 (C)
      IMPLICIT LOGICAL      (B)
```

Appendix B: Categorized List of LEMM Routines

Adding attributes

```
     Lput(nob,natt,yval)
     L0put(nob,natt,yval)
```

Accessing attributes

```
     Lget(nob,natt)
```

Removing attributes

```
     Lremov(nob,natt)
```

Reading (input) LEMM objects

```
     Lread(icond)
```

Writing (output) LEMM objects

```
     Lpp(nob,ilevl)
     Lprin(nob)
```

Appendix C: Descriptions of LEMM Functions in Alphabetical Order

L0put(nob,natt,yval)

LEMM system level put [value]: put value "yval" under attribute "natt" on object [identifier] nid; returns yval; if yval is NIL, only the attribute is added.

nob	object to update
natt	new or existing attribute to put value under
yval	value to be added

Lget(nob,natt)

LEMM get [value]: get the values under attribute "natt" on object "nob" and return them in a list; returns NIL if there are no values under "natt" or if "natt" is not an attribute of "nob".

nob	object to retrieve values from
natt	attribute to look for values under

Lpp(nob,ilevl)

LEMM pretty print: prints object nob to the output unit, slashifying special characters, and indenting different levels of attributes. The depth of levels to print is specified by ilevl; ilevl=-1 implies all levels are to be printed. Lpp returns nob.

nob	object to print
ilevl	cutoff on descending levels of complex values

Lprin(nob)

LEMM print: Prints object nob to the output unit, attributes and values are slashified as needed but not indented.

nob	object to print

Lput(nob,natt,yval)

LEMM level put [value]: put value "yval" under attribute "natt" on object [identifier] nid even if yval is NIL; returns yval.

nob	object to update
natt	new or existing attribute to put value under
yval	value to be added

Lread(iccode)

LEMM read: reads the next LEMM sentence from the input unit. A valid LEMM sentence defines an object in terms of attributes and their values. If the object already exists, it is augmented by the new attributes and values. Lread returns a condition code in iccode, and returns as value, the name of the object.

iccode	returned condition code (c.f. Appendix A)

Lremov(nob,natt)

LEMM remove [attribute]: removes attribute natt from object nob, returns natt if successful, nil if natt not found.

nob	object to remove attribute from
natt	attribute to remove

Appendix D: Categorized List of MACFOR routines

Initialization
- Macini()

Conversion to pointers
- Makefl(ro)
- Makefx(ix)
- Makeid(cid)
- Makesg(csg)

Conversion from pointers
- Mudofl(o,ro)
- Mudofx(x,ix)
- Mudoid(nid,iid,cid)
- Mudosg(sg,isg,csg)
- Murmrk(imrkbit,itgbit)

Atoms
- M0tanp(y)
- M1idap(y)
- Matom(y)
- Mflnmp(y)
- Mfxnmp(y)
- Mgensm()
- Mget(nid,nprop)
- Mgnatp(y)
- Mntern(n)
- Mnumrp(y)
- Mput(nid,nprop,yval)
- Mrempr(nid,nprop)
- Mstrgp(y)

Lists
- Massoc(nkey,psrch)
- Mcar(p)
- Mcdr(p)
- Mcons(y1,y2)
- Mcopy(y)
- Mdrevr(p)
- Mdrmvl(y,plist)
- Memq(nmatch,plist)
- Mequal(yptr1,yptr2)
- Mlast(p)
- Mlngth(plist)
- Mnconc(p1,p2)
- Mnth(p,i)
- Mpop(pstack)
- Mpush(pstack,yval)
- Mrplca(y1,y2)
- Mrplcd(p1,p2)

```
Reading (input)
     M0rdln()
     Mnunt(iunit)
     Mrdeol(iccode)
     Mread(iccode)

Writing (output)
     M1pcsg(cstg)
     M1pngr(ival)
     M1prel(rval)
     M1tab(iotab)
     Mount(iunit)
     Mprin(y)
     Mprinc(y)
     Mterpr(y)

Memory management
     M0mark(y,imkbit,itgbit)
     Mgc2()
     Murmrk(imkbit,itgbit)

Simulated bit arrays (a byproduct of memory management)
     M0btcl(inbits,irows,icols,ibits)   SUBROUTINE
     M0btsr(bset,ival,inbits,irows,icols,ir,ic,ibits)   SUBROUTINE
```

Appendix E: Descriptions of MACFOR Functions in Alphabetical Order

M0btcl(inbits,irows,icols,ibits) SUBROUTINE

MACFOR system level bit [array] clear: zeros the two-dimensional bit array packed into ibits.

inbits	number of bits which can be packed into a word (usually the number of bits in the machine's word minus 1)
irows	number of rows in the bit array ibits
icols	number of columns in the bit array ibits
ibits	the simulated bit array, should be dimensioned in caller to irows*icols/inbits + 1

M0btsr(bset,ival,inbits,irows,icols,ir,ic,ibits) SUBROUTINE

MACFOR system level bits [array] set/read: sets or reads values the simulated bit array ibits.

bset	.TRUE. ==> set value; .FALSE. ==> read value
ival	a variable for setting or reading ibits(ir,ic)
inbits	number of bits which can be packed into a word (usually the number of bits in the machine's word minus 1)
irows	number of rows in the bit array ibits
icols	number of columns in the bit array ibits
ir	row index of ibits
ic	column index of ibits
ibits	the simulated bit array, should be dimensioned in caller to irows*icols/inbits + 1

M0mark(y,imkbit,itgbit)

MACFOR system level mark: mark s-expression y safe from garbage collection; programmed according to the Schorr-Waite-Deutsch method for possibly cyclic lists; c.f. Standish, DATA STRUCTURE TECHNIQUES, p. 215. Returns NIL.

y	s-expression to mark
imkbit	marker simulated bit array; parallels MACFOR memory
itgbit	tab bit simulated bit array; parallels MACFOR memory

M0rdln()

MACFOR system level read line: fetch the next line of input; always returns FORTRAN integer condition code IOKCC.

M0tanp(y)

MACFOR system level test atom name predicate: returns T if y is a fixnum, flonum, string, or NIL; otherwise, returns NIL.

M1idap(y)

MACFOR low level identifier atom predicate: returns T if y is an identifier atom; otherwise NIL.

y	item to test

Mlpcsg(cstg)
MACFOR low level print [FORTRAN] character string: outputs FORTRAN character string cstg up to the last nonblank character; always prints at least the first character; returns the number of characters printed as a FORTRAN integer.

cstg	sting to output

Mlpngr(ival)
MACFOR low level print integer: outputs FORTRAN integer ival; same return as Mlpcsg.

ival	integer to output

Mlprel(rval)
MACFOR low level print real: outputs FORTRAN real rval; same return as Mlpcsg.

rval	real to output

Mltab(iotab)
MACFOR low level tab: updates output buffer pointer so next output will start in column iotab; forces new line if iotab has already been passed; returns updated buffer pointer as a FORTRAN integer.

iotab	column to tab to

Macini()
MACFOR initialize: initialize the global variables; returns the FORTRAN integer, ok condition code.

Makefl(ro)
MACFOR make flonum: returns interned flonum with value ro.

ro	value returned flonum should have

Makefx(ix)
MACFOR make fixnum: returns interned fixnum with value ix

ix	value returned fixnum should have

Makeid(cid)
MACFOR make identifier [atom]: returns interned identifier atom whose pname is the significant characters of cid; if cid is blank, returns NIL.

cid	pname returned atom should have

Makesg(csg)
MACFOR make string: returns interned MACFOR string whose value is the significant characters of the FORTRAN character string csg; if csg is blank, returns NIL.

csg	value returned string should have

Massoc(nkey,psrch)
MACFOR assoc[iation]: returns sublist in psrch whose car EQUALs nkey; returns NIL if no such sublist is found.

nkey	provides the key for the search
psrch	list of lists (or cons pairs) to search

Matom(y)
MACFOR atom [predicate]: returns T if y is an atom or NIL; otherwise, returns NIL.

y	item to test

Mcar(p)
MACFOR car: returns the car of p; loosely, the fist item on the list p; note, the car of NIL is NIL.

p	item to take the car of, usually a list

Mcdr(p)
MACFOR cdr: returns the cdr of p; loosely, the remainder of list p after the first item; note, the cdr of NIL is NIL.

p	item to take the cdr of, usually a list

Mcons(y1,y2)
MACFOR cons[truct]: returns the cons of y1 and y2; loosely, the list p whose car is y1 and whose cdr is y2.

y1	supplies the car, often an atom to become the new head (car) of a list
y2	supplies the cdr, often the rest of the list

Mcopy(y)
MACFOR copy: returns copy of y; i.e. EQUAL structure but different cons cells constructing list.

y	what to copy, usually a list

Mdrevr(p)
MACFOR destructive reverse: returns p, but the order of the items in the list p have been reversed; applies only to first level.

p	list to reverse

Mdrmv1(y,plist)
MACFOR destructive remove 1st [occurrence]: returns plist but the first occurrence of y has been spliced out; works even if y is the first item on plist.

y	item to remove
plist	list to splice y out of

Memq(nmatch,plist)
MACFOR member [test using] eq[ality of pointers, not structure]: returns remainder of plist starting with the first occurrence of nmatch; if nmatch does not occur on plist, returns NIL.

nmatch	item to key on, usually an atom
plist	list to search

Mequal(yptr1,yptr2)
MACFOR equal[ity of structure]: returns T if yptr1 and yptr2 point to equivalent, but not necessary the same, list structures.

yptr1, yptr2	items to compare

Mflnmp(y)
MACFOR flonum predicate: returns T if y is a flonum; otherwise, returns NIL

y	item to test

Mfxnmp(y)
MACFOR fixnum predicate: returns T if y is a fixnum; otherwise, returns NIL

y	item to test

Mgc2()
MACFOR garbage collector 2nd edition: returns the free storage availability list updated to include all MACFOR memory words no longer a part of an active list or atom. Mgc2 considers all interned atoms (i.e. those on the symbol table, also called the oblist) active, along with their property lists. Mgc2 calls Msymrk to mark as active (i.e. safe from garbage collection), lists and atoms used by MACFOR but not on the symbol table. Mgc2 also calls Murmrk, a subroutine supplied by the user when desired, that calls marking routines to protect his special lists and non-interned atoms.

Mgensm()
MACFOR generate symbol: generates and returns a new identifier atom; the atom is not interned.

Mget(nid,nprop)
MACFOR get [property value]: returns the value of the property nprop on identifier atom nid; returns NIL if nprop is not on the property list of nid.

nid	identifier atom whose property list will be searched
nprop	atom naming property to search for

Mgnatp(y)
MACFOR generated atom predicate: returns T if y is a generated (gensym) atom; otherwise returns NIL.

y	item to test

Mlast(p)
MACFOR last [on list]: returns the remainder of list p beginning at the last item.

p	list to find the remainder of

Mlngth(plist)
MACFOR length: returns the fixnum whose value is the number of items in list plist.

plist	list with items to count

Mnconc(p1,p2)
MACFOR end concatenate: returns p1 modified so that its end is joined with p2; will not modify p1 if it is NIL.

p1	list to modify
p2	new tail for p1

Mntern(n)
MACFOR intern: return the atom already on the oblist (symbol table) which is equal to n; if no atom equal to n is yet on the oblist, return n after putting it on the oblist.

n	atom to intern

Mnth(p,i)
MACFOR nth [item]: return the remainder of list p starting at the i'th item; returns NIL if there are less than i items on the list; works by taking i-1 cdr's.

p	list to fetch remainder of
i	position of first item on remainder

Mnumrp(y)
MACFOR number predicate: returns T if y is a fixnum or flonum; otherwise, returns NIL.

y	item to test

Mnunt(iunit)
MACFOR input unit [set]: sets the FORTRAN logical unit from which input is read to iunit; returns the former input unit as a FORTRAN integer.

iunit	new logical unit for reading input

Mount(iunit)
MACFOR output unit [set]: sets to iunit the FORTRAN logical unit to which output is sent; returns the former input unit as a FORTRAN integer.

iunit	new logical unit for sending output

Mpop(pstack)
MACFOR pop [stack]: returns the top element of a stack, implemented here as a list; updates pstack so the next item is the new top.

pstack	stack to pop

Mprin(y)
MACFOR print: prints y with slashifying; returns y.

y	item to be printed

Mprinc(y)
MACFOR print characters: prints y without slashifying; returns y.

y	item to be printed

Mpush(pstack,yval)
MACFOR push [onto stack]: modifies stack pstack, here implemented as a list, so its new top element is yval; returns yval.

pstack	stack to modify
yval	new top value

Mput(nid,nprop,yval)
MACFOR put [property value]: put yval under the property nprop on the property list of identifier atom nid; return yval.

nid	identifier atom to receive property value
nprop	property to store yval under
yval	item to store under nprop

Mrdeol(iccode)
MACFOR read [including] end of line: return next expression read from input and update the condition code argument (c.f. Appendix A). If Mrdeol is called at the end of a line, i.e., no non-blank characters left, Mrdeol will return the end of line atom, PEOLCH, accessed in common block MACATS. Use M0rdln to proceed to a new line.

iccode	receives updated condition code

Mread(iccode)
MACFOR read: return next expression read from input; updates condition code argument (c.f. M0rdat).

iccode	receives updated condition code

Mrempr(nid,nprop)
MACFOR remove property: removes property-value pair on property list of nid; returns the value of that property if it existed; otherwise, returns NIL

nid	atom with property to remove
nprop	property to remove

Mrplca(y1,y2)
MACFOR replace car: replace the car of y1 with y2; return y2

y1	item whose car will be modified
y2	new car of y1

Mrplcd(y1,y2)
MACFOR replace cdr: replace the cdr of y1 with y2; return y2

y1	item whose cdr will be modified
y2	new cdr of y1

Mstrgp(y)
MACFOR string predicate: returns T if y is a string atom; otherwise, returns NIL.

y	item to test

Mterpr(y)
MACFOR terminate print: dumps output buffer followed by newline to current output unit; returns y; usual usage is Mterpr(Mprin(y)) so y is printed and followed by newline or Mterpr(NIL) just to force output buffer.

y	goes along for ride

Mudofl(o,ro)
MACFOR undo flonum: store the value of flonum o into real number ro; returns o; if o is not a flonum, returns NIL.

o	flonum supplying value
ro	variable to receive value

Mudofx(x,ix)
MACFOR undo fixnum: store the value of fixnum x into integer ix; returns x; if x is not a fixnum, returns NIL.

x	fixnum supplying value
ix	variable to receive value

Mudoid(nid,iid,cid)
MACFOR undo identifier [atom]: stores the pname of identifier atom nid in cid and store the number of significant characters of cid in iid; returns pname of nid or NIL if nid does not have a pname.

nid	identifier atom to supply pname
iid	receives length of cid
cid	receives character string version of pname

Mudosg(sg,isg,csg)
MACFOR undo string: stores the value of string sg in csg and the number of significant characters of csg in isg; returns sg; if sg is not a string, returns NIL.

sg	string to supply value
isg	receives length of csg
csg	receives character string version of sg

Murmrk(imrkbit,itgbit)

MACFOR user [variables] mark: marks non-interned user variables safe from garbage collection; returns NIL. The MACFOR library version of this function has no effect; it should be replaced by the user when necessary.

imkbit	marker simulated bit array; parallels MACFOR memory
itgbit	tab bit simulated bit array; parallels MACFOR memory

NBS-114A (REV. 2-80)

U.S. DEPT. OF COMM. **BIBLIOGRAPHIC DATA SHEET** *(See instructions)*	1. PUBLICATION OR REPORT NO. NBSIR 87-3622	2. Performing Organ. Report No.	3. Publication Date July 1987

4. TITLE AND SUBTITLE

Tutorial on Programming in LEMM and MACFOR

5. AUTHOR(S)

David F. Redmiles

6. PERFORMING ORGANIZATION *(If joint or other than NBS, see instructions)*

NATIONAL BUREAU OF STANDARDS
DEPARTMENT OF COMMERCE
WASHINGTON, D.C. 20234

7. Contract/Grant No.

8. Type of Report & Period Covered

9. SPONSORING ORGANIZATION NAME AND COMPLETE ADDRESS *(Street, City, State, ZIP)*

10. SUPPLEMENTARY NOTES

☐ Document describes a computer program; SF-185, FIPS Software Summary, is attached.

11. ABSTRACT *(A 200-word or less factual summary of most significant information. If document includes a significant bibliography or literature survey, mention it here)*

Two FORTRAN 77 libraries called LEMM and MACFOR are described. These libraries were implemented to enable FORTRAN programmers to use list and structure oriented data types like those available in LISP and Pascal. The LEMM library combines ideas from the LISP programming language and the relational database model to provide a structured data type for representing and storing data. The MACFOR library implements many of the functions of the LISP programming language within FORTRAN and supports the implementation of the LEMM subroutines. The two libraries are intended to be used together and their expositions here are interposed with emphasis placed on using LEMM. However, the MACFOR library can be used independently, especially by programmers familiar with LISP. The LEMM and MACFOR libraries codify popular and proven concepts in data representation. Many programming examples are provided to make the use of these libraries understandable and applicable in the field.

12. KEY WORDS *(Six to twelve entries; alphabetical order; capitalize only proper names; and separate key words by semicolons)*

data structures; list programming; relational database; FORTRAN; LISP

13. AVAILABILITY

[✓ Unlimited]
☐ For Official Distribution. Do Not Release to NTIS
☐ Order From Superintendent of Documents, U.S. Government Printing Office, Washington, D.C. 20402.
[✓ Order From National Technical Information Service (NTIS), Springfield, VA. 22161]

14. NO. OF PRINTED PAGES
47

15. Price
$11.95

USCOMM-DC 6043-P80