WORKING PAPER 86

IDEAS ABOUT MANAGEMENT OF LISP DATA BASES

by

ERIK SANDEWALL

Massachusetts Institute of Technology
Artificial Intelligence Laboratory
January, 1975

Abstract. The trend toward larger data bases in A.I. programs makes it desirable to provide program support for the activity of building and maintaining LISP data bases. Many techniques can be drawn from present and proposed systems for supporting program maintenance, but there are also a variety of additional problems and possibilities. Most importantly, a system for supporting data base development needs a formal description of the user's data base. The description must at least partly be contributed by the user. The paper discusses the operation of such a support system, and describes some ideas that have been useful in a prototype system.

Work reported herein was conducted partly at Uppsala University, Sweden, with support from the Swedish Board of Technical Development, and partly at the Artificial Intelligence Laboratory, a Massachusetts Institute of Technology research program supported in part by the Advanced Research Projects Agency of the Department of Defense and monitored by the Office of Naval Research under Contract Number N00014-70-A-0362-0005.

1. Focus on the data base.

LISP is really a programming language for a certain type of data bases, and it is less interesting as a 'list processing' or 'list structure' language. The item in LISP that makes it different from the other major programming languages is not the cons cell, but the atom. Atoms are the carriers of property-lists, which is the primary representation in LISP's data base. Atoms are also essential for the facility to read and write data structures, which is what makes LISP a good interactive language. Similarly, the most significant built-in functions in LISP are not car, cdr, and cons, but get, put, and other functions for accessing property-lists in the LISP data base. The language also contains some functions of secondary significance which can be used to construct and decompose properties, for example cons, maknam, cdr, and explode.

Such an alternative view of our favourite programming language is becoming increasingly useful with the present trend toward larger and more complex data bases in Artificial Intelligence systems. It is also supported by the fact that, with the on-going blurring of the program/data distinction, programs become integrated parts of the data base in a less trivial sense than used to be the case. A number of new aspects of the programming language and its use arise when the focus of interest is changed to the data base, for example:

--- Current programming practice for using LISP's representational primitives. The structure of atoms and lists is quite different from the record structure of other languages 'with data structures', and encourages a different methodology. This methodology already exists, but it should be talked and written about, both for tutorial purposes, and as a basis for developing it further.

--- Block structure in the data base. It is common practice to organize LISP programs into 'blocks', or groups of closely related functions. Such block structure encourages modularity and facilitates maintenance. Both purposes would be worthwhile for the data base as well. However, a number of new and interesting problems arise when one attempts to organize the data base into blocks, for example because data items can be related in multiple ways, so that clustering criteria are not trivial to decide.

--- Self-description in the data base, whereby the data base contains a description of itself, and (in more developed systems) of its relation to the intended application. Such self-description could be made useful both for the user (as a documentation aid), and for programs which use it as parameters, to determine how operations on the data base are to be performed.

--- Utility programs for LISP data bases. By utility programs, I mean general programs which are primarily intended to be called directly by the user, rather than as subroutines from another program, and which do some service operation on a data base. Utility programs for programs are in common use, for example compilers, editors, file grinders, and (to some extent) indexers. Many of these operations generalize to data bases as well. Others can be added, for example consistency checks.

Since programs is a special case of data in LISP, one can often extend the use of programs that were written for operating on programs, to be used for other types of data as well. But it must be better to design such programs right from the start for data bases in general.

With data base techniques in LISP as my present major interest, I have been toying for a while with the idea of a *support system* for data base management in LISP. Such a system would be a collection of programs (and associated data) which support the user in his work with the data base. The term 'data base' is here taken to mean collections of knowledge that are used by one or more programs. It does not refer to temporary data such as hypotheses or sets of subgoals, which are created during a computation and later garbage collected, or discarded at the end of the computer run. Correspondingly, the 'user' whom the system supports, develops not only programs, but also data bases in the given sense. The support system could develop into a 'data base hacker's assistant'.

An experimental system, called DABA, has been instrumental in developing and testing some of the ideas, and hopefully will also serve as an illustration of them. DABA is a MACLISP program. This working paper is an attempt to summarize my ideas at present. For concreteness, it uses some of the notation of the DABA system, but it is not a systematic description of DABA.

The major service that a support system can offer its user is utility operations. Sometimes the user will be writing down parts of a data base directly, much like he writes down a program. He then needs the same kinds of utilities as for program development, which enable him to administrate and update his data easily. At other times, the user will obtain parts of his data base from computation. They may be the accumulated experience of a performing program, or the result of running a utility program on previously existing data (for example a program for shift of representation). In either case, the user needs a program which can administrate the new data, shovel them around, and integrate them properly in his data base.

Unfortunately, the user can not count on obtaining such a service without effort from his part: he has to specify his representation and data base structure to the utility program. (Even if the data base contains information about itself, he at least has to specify what conventions he used for the self-description). Therefore, a support system for utilities must necessarily contain a system for description of data bases. Ideally, one would like to have general-purpose descriptions, which can be used by all utilities, and one would also like to store the description in the data base (which was called self-description above), so that the utilities can be applied recursively to the descriptions.

A user supporting system should of course allow for the variety of different representations that are found in current LISP programming. Some LISP users work directly with the data base primitives provided by the language, but many develop their own higher-level representations, or use available systems such as PCDB, or the data base handlers in Conniver or QA4. A support system should therefore enable one to make a definition of for example Conniver's data bae primitives (such as contexts) in terms of the underlying LISP primitives, so that the user then can talk in Conniver terms when he describes his Conniver data base, and when he calls for a utility operation on it.

A system for describing the representation in a part of a data base, could also be used to describe a program with respect to what representation it assumes in its data. A program which adjusts data to fit given programs, or vice versa, is then a desirable utility.

The requirements of utility programs actually call for two different kinds of self-description. There is *self-description of representation*, where the syntax and perhaps also semantics of the chosen representation are specified. But many utilities can be characterized as *scanners*, in the sense that they scan over a specified segment of a data base, and perform some operation throughout it. Examples of scanner utilities are for saving data bases on files, for presentation (as a generalization of prettyprinting), for checking, and for shifts of representation. Scanner utilities then need a *description of extent* of parts of the data base. It is not yet clear to me how one should properly handle the relationship between description of representation and description of extent, but a tentative model is described in the present paper.

One part of the description of a representation for a data base is the set of procedures which access (in the sense of both 'get', 'put', 'modify', and 'delete') data bases that use the representation. (Too often the access procedures are the only description). The DABA system assumes that such access procedures are part of the self-description. This means that an application program can access the data base through DABA, which knows where in the self-description the access function is located. For efficiency of computation, one will often want to open-compile such calls and eliminate the detour through DABA, but the idea of first storing the access function in the description has advantages, for example that it becomes simpler to generate and retain access procedures from other parts of the description.

Generation and default definition of access functions are achived in the prototype DABA system by a recursive access mechanism: in order to use the data base, an access function is retrieved and used, and at least in principle the access function is retrieved in the same way, using *its* access function, and so on. Such recursive access also provides a simple and elegant basis for handling other features in the system, for instance description of data blocks. It has some obvious efficiency problems, which I think however can be resolved. The next section describes the access mechanism in more detail.

It is attractive to let the descriptions of the data base be in the data base itself, so that the support system can be used recursively. This necessitates a choice of data representation for phrasing the description in, but should not and need not imply a choice or a restriction of what data representations can be described. I have preferred to use an object/property-type structure for the descriptions, rather than for example a relational structure, since the former is the closest to the property-list data base primitives in LISP system, and since the auxiliary systems for alternative and higher-order representations are ultimately defined from such primitives. The object/property representation is augmented with the well-known method of nested property-lists. With this representation, for each pair of *carrier* and *indicator*, the data base may contain a *property*, which may be an arbitrary entity, but in particular may be a set of assignments of *sub-properties* to *sub-indicators*.

A blocking concept as discussed above has also been useful for structuring the descriptions. The data base is viewed as a collection of 'items' (which may be property assignments, relations, some variety of frames, or something else that is an entity), and such items are grouped into *blocks*. Some of the possible connotations of that term are however not intended: no parenthesis-like nesting of blocks and no scope concept for identifiers are being used. The term 'data set' would have been more appropriate if it had not already been taken for another purpose. The primary intended analogy is with the practice to group sets of functions in large

LISP programs into 'files'.

Blocks have some resemblance to contexts in Conniver and QA4, except that contexts are mostly intended for 'scratchpad' data. Like for contexts, the same carrier/indicator pair may have different properties in different blocks.

Blocks proved useful for structuring the data base descriptions, both the descriptions that the DABA system expects the user to provide, and the higher-level descriptons in the system itself. I believe that data base block structure can also be very useful in many applications. One particular usage is for conserving storage in LISP systems which are plagued by space problems in the heap: blocking enables one to keep little-used parts of the data base as text files, and only bring them in when needed.

A question of terminology has to be resolved at this point. The words *structure* and *representation* are easily overused when talking about data bases and data structures. We shall use the term 'representation' for the constructs used in expressing information, for example particular structures of atoms and lists. Property-list representations and relational representations are other examples. The word 'structure' will as far as possible be reserved for block structure, and for relationships between blocks or other groups of data in the data base, for example the relationship between a block of data and the block from which it was generated.

Finally, there is a third concept, for which one might be tempted to use the word 'representation', but where another term is chosen to avoid ambiguity. Different blocks in a system may use different data representations, in the sense just specified. But one block may also appear in several forms which we call *incarnations*. For example, a block using the object/property representation may have one incarnation as a text file of the form

```
(DEFPROP C1 V11 I1)
(DEFPROP C1 V12 I2)
   ...
(DEFPROP Cm Vmn In)
```

A second incarnation of the same block is as a list of sub-lists of length three, in LISP memory, and another obvious incarnation is to store the block in LISP memory on the property-lists of the carriers Ci. The relations between various incarnations of a block, and their relations to the programs which perform the conversion, are an aspect of the data base structure, and as such is a topic of the self-description in the data base.

2. Access functions.

Some DABA notation is necessary for describing its accessing mechanism. Access in a data block with an object/property representation requires a basic function of three arguments, dgetp[c,i,n] which gets the property of the carrier c under the indicator i in the block whose name is n. (We distinguish between a block and a blockname. The block is for example a set of trituples which represents property-list information. The blockname is an atom which denotes the block). In the function name dgetp, d stands for *derived* (for reasons that will be clear), and p stands for *property*.

For some simple purposes, it is sufficient to let each data block maintain its own set of properties, but very often one wants the properties to be implicitly defined. The systematic way of doing that is to associate an access function with each indicator. Since different blocks may use different access functions, the access function shall also be in a data block, which is then the *meta*-block of the block in which access is made. The major case in the definition of dgetp is therefore:

```
        dgetp[c,i,n] = apply[ dgetp[i,ACCESSFN,getp[n,META]], list[c,i,n]]
```

where getp is the ordinary LISP function for getting properties from the property-lists of atoms.

This definition of dgetp begs two questions. One is where the recursion ends, and the other is how the access function is to be written. We need in fact one function xgetp[c,i,n] which looks up the property of c under i in the data block that is immediately associated with n. Thus one could have

```
        dgetp[i,ACCESSFN, ... ] = XGETP
```

in some contexts, meaning that there are no default options for the data block. The function xgetp is then a simple hack which enables one carrier/indicator pair to have different properties in different data blocks as indicated by the third argument.

It follows that each block name is really associated with *two* data blocks: one block of explicit information associated with the block name, and one amended, derived block. The functions xgetp and dgetp make access in these respective blocks.

For example, suppose we have two data blocks named B and B', where the block of B is explicitly specified, and the block of B' is a modification of B. The rule for getting something in B' is first to look if it is explicitly in B', otherwise get the property from B. In other words, the block B' which is a modification of B is the *derived* data block of B', and the explicit block of B' is the difference set between B' and B.

Concretely, if we have

```
        getp[B,META] = M

        getp[B',META] = M'

        getp[B',MODIFOF] = B
```

we define access functions as follows for each indicator in B and B':

```
        dgetp[i,ACCESSFN,M'] = (LAMBDA (C I N)(OR (XGETP C I N)
                                       (DGETP C I (GETP N 'MODIFOF)) ))
```

```
dgetp[i,ACCESSFN,M] = e.g. XGETP
```

The other question is where the recursion ends. The obvious answer is to let it terminate at a fixpoint, so that the exact definition of dgetp is

```
dgetp[c,i,n] = if (n = OMEGA) then xgetp[c,i,n] else
        apply[ dgetp[i,ACCESSFN,getp[n,META]],list[c,i,n]]
```

A simpler solution might have been to look up the ACCESSFN property in the explicit rather than the derived data block of getp[n,META]. However, using the derived block has the advantage that defaults may be defined for access functions. In the above example, it would be a nuisance to have to write out access functions for all indicators in B'. It is however sufficient that each i has the appropriate ACCESSFN property in the derived data block of M', which can be arranged by defining

```
getp[M',META] = MODACC
```

where MODACC contains the access information for data blocks which are updates of other data blocks, phrased as follows:

```
dgetp[ACCESSFN,ACCESSFN,MODACC] =
    (LAMBDA (C I N)(FUNCTION
        (LAMBDA (C I N)(OR (XGETP C I N)
                      (DGETP C I (GETP N 'MODIFOF)) )) ))
```

In order to be complete, the system should of course also include

```
getp[MODACC,META] = OMEGA

dgetp[ACCESSFN,ACCESSFN,OMEGA] = XGETP
```

The last property establishes that the accessfn of accessfn in MODACC can be stored explicitly, which otherwise would not be obvious.

It is the intention that access functions shall be part of the description of all data blocks, both 'system' blocks which themselves are descriptions, and blocks in the application. Access functions have three arguments for carrier, indicator, and blockname (and in the actual DABA program also a fourth argument, which we ignore here). This argument structure implies a bias toward an object/property representation of information, but it does not exclude other representations. This is important if the system shall ultimately be able to describe and support a variety of user-defined representation schemes. For alternative representations, one will often choose to let the carrier and indicator arguments of dgetp be non-atoms. This is acceptable since these arguments are merely passed on to an access function, which can do what it wants with its arguments.

For example, a sub-system for relational data bases could assume the first argument of dgetp to be the list of arguments for the relation, and the second argument to be the relation name. A block analogous to MODACC in the example above, but more complex, would contain appropriate access functions which accept such arguments. As an extreme example, it is trivial to define a block APPLY so that

```
dgetp[l,f,APPLY] = apply[f,l]
```

The access function scheme, if used literally, implies a considerable overhead in access from a data block, which increases with the block's distance from OMEGA. In a practical system, one could speed up the scheme in several ways, for example modify the definition of dgetp so that it recognizes a flag on a block name for 'explicitly stored', and does a lookup in the explicit data block without further recursion if that flag is set. The blockname OMEGA would immediately obtain such a flag. With that convention, it is worthwhile to compute access functions once and for all, and save them for later use (memoization).

The memoization of access functions would be helpful, but it is not quite trivial. Sometimes it must be combined with optimization of the memoized function in order to be effective. For example, if an access function for one data block specifies that one should retrieve and use the access function for the same indicator in another data block, then it is not sufficient to save it -- one also wants to look up the substitute access function at memoization time.

Another complication arises because the computation of an access function may have involved access to some properties which are later changed. Memoization should therefore be combined with a forward deduction scheme, so that whenever a 'sensitive' property is updated, other properties which depend on it are also updated. This requires data blocks for keeping track of the dependency relations, and should therefore be considered as a sub-system, to be implemented once the core of the system is going.

3. Extent of blocks.

Several of the common utility operations on data blocks have to scan the contents of the block, for example presentation, reorganization, saving on text files, etc. The system therefore has to know the extent of each block. This is simple if blocks are stored in separate places, which however is not always the case. When dgetp[c,i,n] is defined for only one n, we may want to store it globally on ordinary property-lists for quicker access. Even if it is defined for several n, we may consider one of the blocks as 'primary' and store its contents on global property-lists. Finally, if we want to perform a block-scanning utility operation on a part of a block, or on the union of several blocks, we should be able to make a description of the extent of a new block in terms of old ones, without actually copying the contents to a new location.

For such reasons, it is desirable to have a catalogue for each block, i.e. a list of carriers in the block, and information about what properties the carriers may have. The catalogue may be explicit, but we also want the option of computing it as needed. This is achieved by arranging that the catalogue is retrieved using the function dgetp, and thereby by access functions which can be appropriately defined to fit each purpose.

The catalogues in DABA assume that carriers are sub-divided into sorts, each sort being represented as an atom (in the simplest case). The extent of the block is then specified in two parts. The NODES property of a blockname is a free property-list of the form

getp[n,NODES] = [s1 {c11 c12 ... } s2 {c21 c22 ...} ...]

where each sk is a sort and the corresponding ckj are property carriers in the sort sk.

The following notation is used here and elsewhere in the memo: angle brackets <...> are used for tuples; curly brackets {...} for sets, and square brackets [...] are used for free property-lists. In principle, a property-list [i1 v1 i2 v2 ...] is considered as a set of assignments of vk to ik, so the square bracket expression is really an abbreviation for

{<i1 v1>, <i2 v2>, ...}

When running LISP, all three kinds of brackets degenerate of course into ordinary LISP parentheses (...). We also use round parentheses in writing out function definitions in LISP.

The other part of the extent specification for a block is located in its META block, which contains CARRPROPS properties for each sort. This property specifies the indicators under which objects in that sort may have properties. Thus we could have

getp[n,META] = m

dgetp[sk,CARRPROPS,m] = {ik1, ik2, ... }

where the latter property is the set of all ikl such that

dgetp[ckj,ikl,n]

is defined for at least some ckj in the sort sk.

So far, we have located access functions for indicators, and carrprops properties for sorts, in the meta block. The meta block needs of course a catalogue as well, which may contain two sorts, called INDIC and SORT, with the obvious intended meaning.

A very simple example may be useful at this point. The data block USCITIES shall consist of some simple facts about cities in the United States, and may contain:

dgetp[BOSTON,INSTATE,USCITIES] = MASS

dgetp[BOSTON,SUBURBS,USCITIES] = {LEXINGTON REVERE ...}

...

dgetp[NYC,INSTATE,USCITIES] = NY

...

dgetp[MASS,HASCITIES,USCITIES] = {BOSTON LEXINGTON ...}

dgetp[MASS,HASCAPITAL,USCITIES] = BOSTON

...

In order to write out the catalogue, we have to make up our mind about what are the appropriate sorts. In this case the decision is simple: we choose CITY and STATE as sort names, and can write

ngetp[USCITIES,NODES] =

[CITY {BOSTON NYC ...} STATE {MASS NY ...}]

The function ngetp gets properties of datablock names, and will soon be defined. Notice that the NODES property needs only include the occurring property-carriers. Thus if LEXINGTON and REVERE do not have any properties in the block USCITIES, there is no reason to include them in the NODES property.

We choose CITIES as the name of the description block for USCITIES, and should therefore have:

getp[USCITIES,META] = CITIES

where the block CITIES contains the following information for the sorts:

dgetp[CITY,CARRPROPS,CITIES] = {INSTATE SUBURBS}

dgetp[STATE,CARRPROPS,CITIES] = {HASCITIES CAPITAL}

and the following information for the indicators:

```
    dgetp[INSTATE,ACCESSFN,CITIES] = XGETP

    dgetp[SUBURBS,ACCESSFN,CITIES] = XGETP

    dgetp[HASCITIES,ACCESSFN,CITIES] = XGETP

    dgetp[CAPITAL,ACCESSFN,CITIES] = XGETP
```

If we have decided

```
    getp[CITIES,META] = OMEGA
```

then the last four properties must be written out explicitly, but we may also choose to refer to a sub-system which imposes defaults for the access fuctions.

The meta block CITIES also needs a catalogue:

```
    ngetp[CITIES,NODES] = [SORT {CITY STATE}
                           INDIC {INSTATE SUBURBS HASCITIES CAPITAL}]
```

The same analysis can be made on the next meta-level, resulting in

```
    getp[CITIES,META] = OMEGA

    dgetp[SORT,CARRPROPS,OMEGA] = {CARRPROPS}

    dgetp[INDIC,CARRPROPS,OMEGA] = {ACCESSFN}

    dgetp[CARRPROPS,ACCESSFN,OMEGA] = XGETP

    dgetp[ACCESSFN,ACCESSFN,OMEGA] = XGETP
```

We then need

```
    ngetp[OMEGA,NODES] = [SORT {SORT INDIC} INDIC {CARRPROPS ACCESSFN}]

    getp[OMEGA,META] = OMEGA
```

whereby OMEGA adequately specifies its own extent.

The goal is that all information in the system shall have a description also in the system. However, we have not yet introduced any description for the properties of blocknames, such as NODES and META properties which are always needed, and properties like MODIFOF which was discussed in the previous section, and which are introduced by sub-systems. One choice might be to form a master block of all information about block names known by the system. However, we sometimes want to scan the properties of the block name together with the properties in the block. For example, if a block is written on a text file, the file name's properties should usually go with it.

It is therefore useful to form a tiny block of the properties of each block name. We shall refer to this as the *catalogue block* of the block name and of the block it names. The catalogue block contains not only the NODES property of the block name, but also the META property, the MODIFOF property, etc.

If B is a block name, what is the name of the catalogue block of B? It would be a waste always to have an atom for that purpose, and the risk of infinite regress is obvious. But the block name is mostly needed for its properties, and we can specify the properties to a first approximation: the NODES property shall be [BNAME {B}], if BNAME is the sort for block names, and the META property shall be a block which knows about the properties of block names. It is convenient to put that knowledge also into OMEGA. With that, the name of the catalogue block of B can be constructed as a list

<☆FPL☆ META OMEGA NODES [BNAME {B}]>

This is a list which pretends to be an atom, in that its cdr is a property list. (☆FPL☆ stands for 'free property list'). Property-list access functions such as getp and ngetp must of course be defined to play this game. Also, the list is a function only of B, and can be set up as such. We define a function catname[n] which does that for a block name n.

To complete the description, we must add to OMEGA the properties

dgetp[BNAME,CARRPROPS,OMEGA] = {META NODES}

dgetp[META,ACCESSFN,OMEGA] = XGETP (or something more suitable)

dgetp[NODES,ACCESSFN,OMEGA] = XGETP

and make the appropriate modification of ngetp[OMEGA,NODES].

The function ngetp used for getting the NODES property can then be defined as

ngetp[c,i] = dgetp[c,i,catname[c]]

This solution works for block names with standard properties. However, we have already seen the need for adding extra properties to block names, and it is also clear that we will want to use non-standard access functions for NODES (namely if the NODES property shall be computed from other information about the block). OMEGA should therefore be the default value rather than the fixed value for the META property in the name of the catalogue block.

How are non-standard META properties then specified? One possibility would be to make it a property of the name of the underlying block (such as B above). But then if that property is to be retrieved with ngetp (which enables us to define an access function for it, assign defaults, etc.), we get into an infinte regress. Suppose the indicator is called CM, and we compute

ngetp[n,CM]

By the definition of ngetp, we need to compute

```
catname[n]
```

which recursively needs ngetp[n,CM] in order to set up the META property in its value.

The problem is solved by keeping that knowledge in the META of n, so that the definition of catname becomes

```
catname[c] = list[ *FPL*,  NODES, list[BNAME, list[c]],
                  META, ngetp[getp[c,META],CMETA]]
```

The META property must always be retrieved using getp, rather than ngetp, since the use of ngetp again would cause an infinite regress. This also means that the META property must always be explicit. All other system properties can however be retrieved using dgetp or its derivative ngetp, and therefore the retrieval method can be manipulated by the system.

While working with the DABA system, it is in fact often useful to specify non-standard access functions for the property CMETA. Consider again the example of the sub-system MODACC which enables one data block B' to be a modification of another block B, where the relationship is stored as

```
ngetp[B',MODIFOF] = B
```

The solution in the previous section involved defining

```
getp[B',META] = M'
getp[M',META] = MODACC
```

It also assumed that the reference from B' to B should be stored as

```
getp[B',MODIFOF] = B
```

In view of the present section, it is of course better to represent it as

```
    ngetp[B',MODIFOF] = B
```

We clearly want the description of the properties of the catalogue name B' to know, that B' has a property under this indicator. For this purpose, catname[B'] should have as its META property a data block MODCAT which knows that

```
    dgetp[BNAME,CARRPROPS,MODCAT] = {NODES META MODIFOF}
```

This holds for every B' that has M' as META, so we can safely set

```
    ngetp[M',CMETA] = MODCAT
```

But since this should be the case for every M' that has MODACC as META, we can define

```
    dgetp[CMETA,ACCESSFN,MODACC] =

        (LAMBDA (C I N) 'MODCAT)
```

or (if we want to enable the user to override this choice):

```
        (LAMBDA (C I N) (OR (XGETP C I N) 'MODCAT))
```

The data block OMEGA must of course be modified to include information about CMETA with respect to CARRPROPS and ACCESSFN.

The reason for introducing the NODES property was for the use of scanning utility programs. It follows circularly that the property should include those carriers which the user wants his utilities to scan over. Usually this will be the explicit data block associated with the blockname, rather than the derived block name, , but it can be decided from case to case by the user.

4. Blocks with molecular names

Blocks need names, of course, and it is natural to use atoms as blocknames. However, it is also sometimes useful to have blocknames which are combinations of atoms, or 'molecules'. For example, for a given block B, we may wish to have also a block of comments about entities in B, or a block which consists of optimized versions of corresponding elements in B. The need for such blocks arises not only for blocks of procedures, but also for blocks of other data. Molecular names for the added blocks could be for example

    <COMMENTS B>

and

    <OPTIMIZED B>

respectively. The molecular names have a mnemonic advantage in that they automatically provide a systematic naming scheme, and also a technical advantage since the components of the molecule are retrievable and may be used for deriving properties of the name. For example, <COMMENTS B> should probably have the same catalogue as B, and this can then be implicit in the operator COMMENTS.

The need for molecular names is of course not unique to data blocks; it arises very frequently in data bases, and has been met by a variety of methods, such as 'relational' data base schemes (e.g. Micro-planner), facilities for associating property-lists with tuples (QA4), 'internization' or 'normalization' procedures which generate a unique copy in the LISP heap of a cons cell or a list, and so on.

The choice between using a molecular name and inventing a new atomic name, is sometimes quite arbitrary. Molecular names tend to be less useful when several alternative blocks with the same purpose are to be named. For example, if there are two different comments blocks for B, one will probably prefer to give them atomic names such as BC1 and BC2, and maintain property-list pointers between the atoms B, COMMENT, BC1, and BC2. On the other hand, if B has been optimized using two different methods M1 and M2, the resulting blocks could best be named with molecular names such as <OPTIMIZED Mi B> or <<OPTIMIZED Mi> B>. The latter naming convention is preferable if one wants to associate properties with <OPTIMIZED Mi>.

When the choice between the two types of naming is arbitrary, one will want a representation for molecular names which adapts as closely as possible to atomic names. In the DABA system, molecular names which intuitively should be written <fn arg>, are internally represented as

<*FPL* META fn BASE arg>

This tuple is computed by a function with the historical name option, defined so that

option[x,fn] = list[*FPL*, META, fn, BASE, x]

Like the catalogue names in the preceding section, this is an entity which pretends to be an atom, since its cdr is a property-list, and which moreover pretends to be an atomic block name since it has a META property. The molecular name contains sufficient information, since the system is arranged so that all properties in all data blocks are retrieved by using the access function in the meta-block.

This representation of molecular names has the advantage that atomic and molecular names can be handled by the same uniform access mechanism. Also, the same meta-block, for example COMMENT, may be used for both atomic and molecular names. The blocks BC1 and BC2 in the example above could be characterized using

```
    getp[BCi,META] = COMMENT
    getp[BCi,BASE] = B
```

although in this case additional references from B to the Bi are required.

The definition of the meta-blocks used in forming molecular names is often somewhat intransparent for the novice, at least if one sticks to the very pure access function design that was described in section 2. Let us work out some of the details for an example. Consider the case of fn = COMMENT, where the catalogue of option[x,COMMENT] shall be the same as the catalogue of x. Thus

```
    ngetp[option[x,COMMENT],NODES] = ngetp[x,NODES]
```

By the conventions in section 3, this is equivalent to

```
    dgetp[option[x,COMMENT],NODES,catname[option[x,COMMENT]]] =
        dgetp[x,NODES,catname[x]]
```

This can be arranged by defining

```
    dgetp[NODES,ACCESSFN,getp[catname[option[x,COMMENT]], META]] =
        lambda[c,i,n]
                comment  c=option[x,COMMENT], and n=catname[c];
                ngetp[getp[c,BASE],NODES]
```

Thus it is easy to specify the access function, and it remains to decide where it is located. We have, still according to section 3,

```
    getp[catname[n],META] = ngetp[getp[n,META],CMETA]
```

and therefore

```
getp[catname[option[x,COMMENT]],META] =
     ngetp[getp[option[x,COMMENT],META],CMETA] =
     ngetp[COMMENT,CMETA]
```

Thus COMMENT must have a CMETA property. It is again a borderline case whether it should be given an atomic or a molecular name, but let us assume for simplicity here that the name is atomic and chosen as COMCM, so that

```
ngetp[COMMENT,CMETA] = COMCM
dgetp[NODES,ACCESSFN,COMCM] =
        lambda[c,i,n] ngetp[getp[c,BASE],NODES]
```

This information specifies that the NODES property of option[x,COMMENT] shall be the same as the NODES property of x. In practice, COMMENT would not need its own CMETA block, since one would probably want several 'satellites' for data blocks, each of which adds one or a few properties to the objects in the 'center' block. COMMENT would be one such satellite, and all satellites could use the same CMETA block. In fact, if all satellites have a common META (so that for example getp[COMMENT,META] = SATELLITE), then the CMETA reference could be implicit in dgetp[CMETA,ACCESSFN,SATELLITE].

The first argument of the function 'option' should always be a blockname, since it will be used for deriving access functions and CMETA references. However, it clearly does not have to be an atomic blockname. The second argument is only there so that the access functions can pick it up and use it, so its structure is arbitrary as long as it accords with the access functions. For example, if one wants to form a new data block which is the union (in some reasonable sense) of several other data blocks whose names are bnl, bn2,..., then the union could be completely specified by its name

option[list[bnl, bn2, ...], UNION]

Data blocks formed by set-theory operations on data blocks are particularly useful in the context of utility programs. To a first approximation, a scanning utility program could work as follows: it accepts a block name as its argument, looks up the catalogue of the block (i.e. the NODES property of the block name), and scans the catalogue. For each sort name in the catalogue, it looks up its CARRPROPS property in the META-block of the given block, to find out which properties objects of this sort may have. It then makes a two-dimensional scan over the objects in this sort according to the catalogue, and the indicators carried by this sort according to the CARRPROPS property, and applies some operation to each combination of these.

Clearly, one sometimes wants to use the utility on a collection of data which are not already a named data block, or in other words, one wants to define the data block for the utility. Molecular names for blocks formed by set operations are useful in such situations.

The definition of the data-block UNION involves two types of problems. First, one must arrange so that access in the block

xuy = option[list[x,y,...], UNION]

attempts access in the blocks x,y, successively, and second, one must arrange that the name xuy obtains the right properties, for example the right NODES property. Let us work out this example as well.

The atom UNION must be the name of a data block UNION which contains the rules for accessing unions of blocks. For example, in order to compute

```
    dgetp[c,i,xuy]
```

a system using the accessing scheme described in section 2, computes

```
    dgetp[i,ACCESSFN,UNION]
```

which must then come back as a function of the type

```
    lambda[c,i,n]
        comment n is xuy when this function is called;
        search the list getp[n,BASE] for some member
            which is a name for a data block in which c
            exists and can have a property under the
            indicator i;
        retrieve and apply the access function for i in that
            data block;
        end
```

Since this is required for all i, it must be put in the accessfn of accessfn. Thus getp[UNION,META] must be a specialized block UNIONMETA, and dgetp[ACCESSFN,ACCESSFN,UNIONMETA] must be a function which always returns the above lambda-expression.

Also, a utility operation which is to scan the catalogue of xuy will first compute

```
ngetp[xuy,NODES]
```

which has to be handled in the same way as the previous example for the COMMENT satellite. In the case of UNION, assume that

```
ngetp[UNION,CMETA] = UNIONCM
```

Then

```
getp[catname[xuy],META]
```

will evaluate to UNIONCM, so that

```
dgetp[xuy,NODES,catname[xuy]]
```

will first compute

```
dgetp[NODES,ACCESSFN,UNIONCM]
```

and apply the result with xuy as its first argument. The primary purpose of UNIONCM is therefore to contain an ACCESSFN for NODES of the form

```
lambda[c,i,n]
    comment c is xuy;
    for each member bn of the list getp[c,BASE], compute
        ngetp[bn,NODES]. Form and return the "union" (in the obvious
        non-trivial sense) of those NODES properties;
    end
```

As another example of the use of molecular names, consider the solution that was given above to the problem of defining one data block B' as a modification of another data block B. That solution assumes that access in the explicit data block of B' is done using the function xgetp, which means it is an access in a data block with an object/property representation. This solution is not useful when one has auxiliary data base systems. Instead, one really wants the following: *if getp[B,META] = M, then getp[B',META] shall be option[M,MODIF]*, which is a meta-block which prescribes that access in B' shall be made by first making access in B' using the access function prescribed for B, and if that yields NIL, by making access in B. This is accomplished by:

```
dgetp[i,ACCESSFN,option[M,MODIF]] =
          lambda[c,i,n]  comment n = B';
                    or[apply[dgetp[i,ACCESSFN,M], list[c,i,n]],
                        dgetp[c,i,ngetp[n,MODIFOF]] ]
```

which in turn is accomplished by:

```
dgetp[ACCESSFN,ACCESSFN,MODIF] =
          lambda[i,a,m]  comment m = option[M,MODIF];
                    return function  lambda[c,i,n]
                              or[apply[dgetp[i,ACCESSFN,getp[m,BASE]],list[c,i,n]],
                                  dgetp[c,i,ngetp[n,MODIFOF]] ]
```

The use of m in the returned function requires that a closure or 'FUNARG expression' is returned. -- With this content, the data block MODIF becomes a general tool for defining modifications to data blocks that use arbitrary representations.

As a final example, consider the case of a 'programmable' utility program, that is a proram which sometimes will look up and give control to procedures associated with its data. Let the utility program be a data block U. (Programs are sets of functions = procedures, which are properties of function names, and therefore programs are good examples of data blocks). U is to operate on a data block B, whose meta is M. Suppose for concreteness that the purpose of U is to check the correctness and internal consistency of a data block B. A reasonable example of programmability is then to associate a 'checking' procedure with each indicator that is used in B. Let the set of such checking procedures be a data block C. The catalogue of C is clearly a subset (in the obvious sense) of the catalogue of M. For this reasons, and also since several Bi using the same M as meta probably need the same checking procedures, it is natural to consider C as a satellite of M. It is therefore formed as

option[M,CH]

for some suitable CH which knows which indicator is used for storing the checking procedure. This CH is simply an encoding of some of the conventions used by the program block U, and it is reasonable to include CH in U (or consider it as a satellite of U, but that would make the example too messy).

In summary then, the following data blocks are involved:

B	the data block that is to be operated on
M	its meta-block
U	the utility program
option[M,U]	a block specifying the behavior of U when operating on blocks like B whose meta is M.

Also involved is getp[U,META], which should be a meta-block for programmable utility programs, and would have the status of a system data-block.

The structure that develops of blocks, meta-blocks, catalogue blocks, satellites, and other blocks with molecular names, seems at first very entangled. One has to get used to it, and one also has to develop a set of useful auxiliary blocks, just as one has to develop a set of auxiliary functions in order to feel at home in a programming language. But the number of such methods and tricks seems to be fairly limited. The block structure has been quite flexible and useful for describing the structure and content of data bases.

5. Where go next: up or down?

The previous sections in this paper have described my present ideas about what data base management in LISP should be like. The ideas have been seasoned through a few iterations of re-programming the DABA system, but neither the ideas nor the system are yet by any means definite.

The present DABA system is partly a straight LISP program, but a large part of it consists of system data blocks, which enables is to be self-describing. It includes a programmable block saver DSAV, which prints data blocks or sets of data blocks on text files, a facility for description of property syntax (whereby one can state e.g. that NODES properties shall be free property-lists which bind sets of carriers to sort names), and a facility for maintaining a data block of all data blocks that are in core at one time. The procedures used by DSAV are kept as satellites, and so are the property syntax descriptions. In general, the DABA system has provided an opportunity to play with various aspects of block structure in data blocks. A reasonably user-friendly version of the system and a user's guide for the novice are available to the curious. See the author for details.

One characteristic of the present system is that it is fairly slow. Single accesses are instantaneous, of course, but the operation of saving the set of main system blocks using DSAV may take more than 10 minutes even at low-load hours. This makes it unattractive to use the system for maintaining itself, although it is sufficiently self-descriptive for making that possible. The efficiency problems are partly because the system's structure has intentionally been kept pure and simple while it is in the development stage. There are plenty of ways to speed it up, by compromising elegance and in other ways, when it develops to the point where it finds usage.

The use of the system may be found in several direction, which can be characterized as 'straight ahead', 'up' and 'down'. The first possibility is that the representation of the application is on the same level as the representation used in DABA itself, that is LISP property-lists with minor extensions. There is a good deal to say for programming in LISP on that level, perhaps even in fairly practical work, such as for writing pilot versions of 'real' data base systems.

The 'up' direction has been mentioned earlier in the paper, and would serve to make utilities available to users who design their own, higher-level representations on top of the LISP system. The major problems to solve then are to develop specifications of higher-level systems in terms of lower levels, which can be used efficiently by the utilities.

The 'down' direction, on the other hand, is to use LISP and utilities of LISP for maintaining structure descriptions of large, simply structured data bases of commercial type. Such structure descriptions are maintained by current supervisory systems for large data bases, for example systems that implement the CODASYL proposal. But in such systems the descriptions are in a rigid, pre-defined representation, and also hard for the user to get at. Sometimes, they are of course unavailable by intention, but it would be nice to enable the user of a data management system to use and extend the structure description, for example for the following purposes:

--- Documentation (like when DABA is used within the LISP context)

--- Advanced query languages, for example in natural language. Query systems relate the language of the user to the conventions and codes used by the system, and therefore need access to a description of the data base.

--- Generation of application programs from specifications that can conveniently be made by the user.

The large-data-base system would then consist of two parts, a production part which administrates the large volumes of data in an efficient way, and a monitor part which contains and manipulates descriptions of the data base, and extends calls to the production part. One would very likely want to use different programming languages and data structures in the two parts. LISP's unique, interactive and flexible data base facilities could make it well suited for use in the monitor part of such systems.