Monocopy and Associative Algorithms in an Extended Lisp

by Eiichi Goto

Information Science Laboratory

Faculty of Science, University of Tokyo

TR 74-03
U. of Tokyo
Apr. 1974

First Manuscript May 16. Rivised on May 31, 1974.

Abstract: An extension of Lisp called HLISP (for Hash-LISP) is described. Two features called monocopy and assoccomp (for Associative Computing) and two data types H-molecules and associators are added to Lisp. The monocopy feature creates H-molecules, which are rewrite protected and are free of duplicating copies among H-molecules. H-molecules are like non-atomic Lisp data in other respects. An associator is a Lisp cell which associates an H-molecular key with a value of any data type. Associators are like atom headers in other respects. Efficient algorithms for copying ordinary lisp data into H-molecules and for performing basic set operations such as union with the number of steps required being proportional to the number of pertinent Lisp cells are given. In the assoccomp feature, the value of prespecified functions is associatively retrieved whenever the value has been evaluated before for the same function for the same argument/s and if the value is remaining in the primary storage. Otherwise it is evaluated as usual. Assoccomp provides a systematic method for increasing the efficiency of some effective algorithms which are inefficient when evaluated by conventional methods. A non-paging virtual memory scheme, enabling a collection of Lisp functions and H-molecular data amounting to several times larger than the primary storage capacity to be used in a single Lisp program, is presented. A hash coding scheme with a dynamic garbage collection capability and some other algorithms crucial for the implementation of HLISP are described in detail.

Key words and phrases: Lisp, Hash Coding, Garbage Collection, Associative Retrieval, Virtual Memory, Effective Algorithm, Efficient Algorithm.

§1. Introduction

At the 1973 Programming Symposium,[1] Sato, Nozaki, Noshita and the author proposed a hash coding scheme for lisp, which will be referred to as monocopy lisp for convenience. In monocopy lisp, the entire free storage area was also to be used as a hashing area so as to eliminate the creation of duplicating copies of S-expressions at a reasonable operating speed without using extra storage space for hash tables. The objectives of monocopy lisp were the saving of storage space and acceleration of checking for equalities.

For example, consider the internal data structures created by the following two lisp functions:
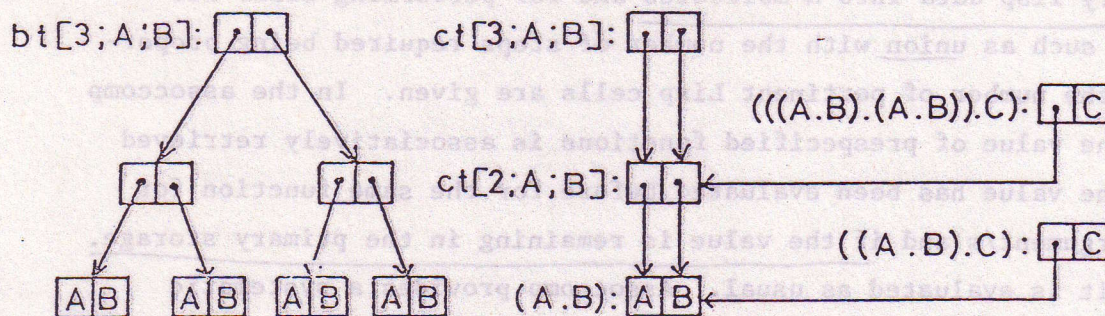
$$bt[n;x;y]=[n\leq 1 \to cons[x;y]; T \to cons[bt[n-1;x;y];bt[n-1;x;y]]]$$
$$ct[n;x;y]=[n\leq 1 \to cons[x;y]; T \to \lambda[[z];cons[z;z]][ct[n-1;x;y]]].$$

Fig.1 A Binary Tree and Monocopy Structures

Fig.1 shows $bt[3;A;B]$ and $ct[3;A;B]$. Although the two internal structures are quite different, the two are regarded equal in lisp. Namely, $equal[bt[n;A;B];ct[n;A;B]]=T$ for all integers n. Note that the recursive definition of equal[2] are referenced $2^{n+1}-1$ times during the evaluation of the above equal. While $bt[n;A;B]$ consists of 2^n-1 cells, $ct[n;A;B]$ consists of n cells (n=3 in Fig.1). While there are 4 duplicating copies of the S-expression (A.B) and 2 copies of ((A.B).(A.B)) in $bt[3;A;B]$, there is no duplication in $ct[3;A;B]$.

In monocopy lisp, the creation of duplicating copies was to be eliminated by modifying the system subroutine for cons, so that bt[n;A;B] would have created the same identical structure as ct[n;A;B] would have. Thus, saving in storage from 2^n-1 cells down to n cells would have resulted in this specific case. The resultant internal data structure of the execution of monocopy lisp functions, bt[3;A;B], cons[bt[2;A;B];C]] and cons[(A.B);C]] would have been as shown in Fig.1 to the right.

The recursively defined equal was to be replaced by the non-recursive basic function eq so that eq[bt[n;A;B];ct[n;A;B]]=T for all integers n. Speeding up of at least a factor of $2^{n+1}-1$ would have resulted in the evaluation of equal in this specific case.

In this paper, ideas incubated by monocopy lisp are further extended and results of an actual implementation are given. Although the schemes to be described would be applicable to other programming languages as well, lisp is used as the base language for clarity and simplicity.

For convenience, the language to be described is called HLISP (for Hash-Lisp). HLISP is an extension of LISP, which is a specific dialect of lisp and is compatible with other lisps[2, [3] in its most essential features. Unless stated otherwise, functions and terms are to be understood as the same as in the Lisp 1.5 manual[2.

Names of functions specific to HLISP shall start from the letter "h". This turns out to be a consistent convention, since no function in Lisp 1.5 starts from "h".

-3-

Definitions of the basic HLISP functions are given in §2 with simple examples. An alphabetically ordered list of HLISP functions is also given in the Appendix, which would serve as a short form manual.
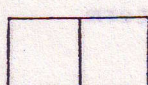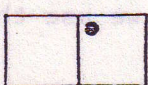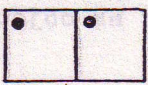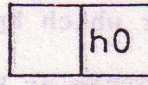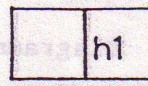
In §3, a copy algorithm to be used frequently in HLISP and fast algorithms for basic set operations are described.

In §4, a system feature called <u>assoccomp</u> (for associative computing) is presented. This feature is believed to give a reasonable compromise between the choice <u>whether a function is to be computed or the value is to be retrieved from a table.</u> This feature would provide a systematic method for speeding up some recursive algorithms which are effective but inefficient when evaluated by conventional methods.

An implementation of HLISP is described in §5 and a non-paging virtual memory scheme is described in §6.

§2. HLISP Objects and Basic Functions

In HLISP, the objects visible from the user are classified into the following five types: L(for Lisp)-molecules, H(for Hash)-molecules, associators, headers and basic integers as shown in Fig.2.

HLISP Term	L-(Lisp-) Molecule	H-(Hash-) Molecule	Associator	Header	Basic Integer	H0- Cell	H1- Cell
Prime Predicates	consp[x]	hconsp[x]	hassocp[x]	headerp[x]	intp [x]	h0p[x]	h1p[x]
LISP object?	Yes	No	No	Yes	Yes	Only for System Use. Invisible from HLISP User.	
atom[x]	NIL	NIL	T	T	T		
molecule[x]	T	T	NIL	NIL	NIL		
hp[x]	NIL	T	T	T	T		
hdotp[x]	NIL	NIL	T	T	NIL		
car/cdr value/key DIAGRAMS	car cdr --- ---	car cdr --- ---	car cdr value key	car cdr value key or A,B,X1A,....	---- ---- ..,-2,-1 0,1,2,...	car cdr --- --- h0 or h1	

Examples of Diagrams

Header of atom NIL:

Associator of (ENGLISH.I) :

Associator of (GERMAN.ICH) :

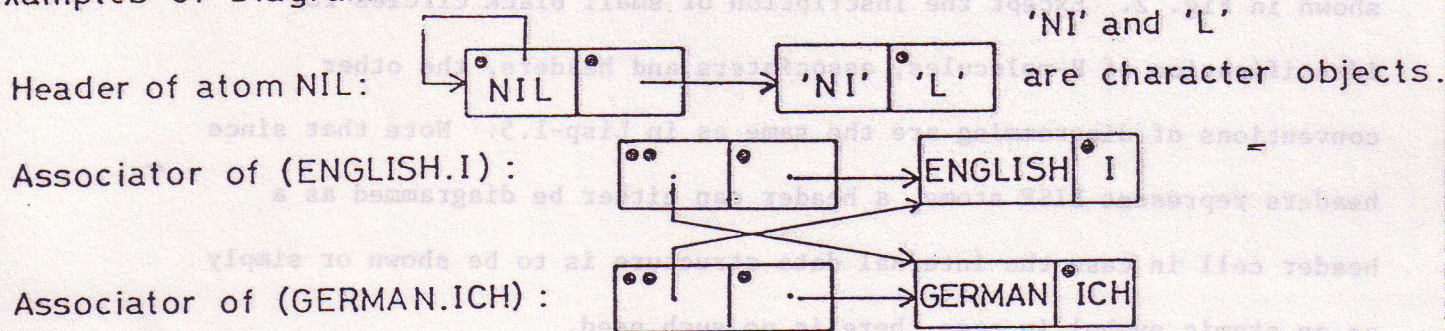'NI' and 'L' are character objects.

Fig.2 Classification of HLISP Objects, Predicates and Diagrams.

As indicated in the top two lines in Fig. 2, each type is respectively
identified by prime predicates consp, hconsp, hassocp, headerp, or intp.
For example, for L-molecular x, consp[x]=T and the values of the remain-
ing four predicates are NIL. The other two types, HO-cells and H1-cells,
are invisible from the user. They represent inactive free storage cells
and are treated in §5.

LISP makes use of L-molecules, headers, and basic integers only.
L-molecules are the same as non-atomic (molecular) data in other lisps.
Headers are functionally equivalent to atom headers of Lisp-1.5.

Truth values of some other type predicates, atom[x], molecule[x]=
not[atom[x]], hp[x] and hdotp[x], are given in Fig. 2. Just like calling
HLISP objects x, for which atom[x]=T, simply atoms, we call objects x,
for which hp[x]=T, HP's or HP-objects. Similarly, we use the term
HDOTP's or HDOTP-objects.

Diagramming of data structures has been very helpful for understanding
the principles of lisp. We shall, therefore, diagram HLISP objects as
shown in Fig. 2. Except the inscription of small black circles for
identification of H-molecules, associaters and headers, the other
conventions of diagramming are the same as in Lisp-1.5. Note that since
headers represent LISP atoms, a header can either be diagrammed as a
header cell in case the internal data structure is to be shown or simply
as an atomic symbol in case there is no such need.

In the present version of HLISP, like in other lisps, the internal
representative of any HLISP object x is an integer, say i and we write
i=m[x] in terms of a function m (for machine). Except for basic
integers, i specifies an HLISP cell which actually consists of two
array elements, mcar[i] and mcdr[i]. Basic integers are represented by
non-existing array addresses. Actually in terms of system constants

maxh, module, minint=maxh+1, izero=maxh+module and maxint=izero+module-1,
i represents a basic integer i-izero, if minint<i<maxint. Hence,
izero=m[0], i.e., izero is the internal representative of the basic
integer 0. The system constant, |module| sets the maximum absolute
magnitude of basic integers which can be handled directly by the system.
Integers of greater magnitudes, rational fractions, floating point
numbers and so on are represented differently from basic integers and
they are to be handled by suitable HLISP functions or subroutines. Only
basic integers will be treated in this paper. The system constant maxh
specifies the maximum address of the arrays mcar and mcdr. Hence,
if i<maxh, i represents an HLISP object accompanying an HLISP cell.
For identifying the types of an HLISP cell, flag bits (actually the
sign bits) in the car and cdr parts of the cell are used as one would
imagine from the diagrams in Fig.2. Another method is given in §5.

   H-molecules are similar to ordinary lisp cells (L-molecules) except
in the following points.

1. cons and functions containing cons (list append etc.) create L-
   molecules but never H-molecules;

2. hcons, defined as the following, is the only basic function which
   can create an H-molecule:

   hcons[x;y]=[hp[x] $\wedge$ hp[y]→mhcons[x;y];T→cons[x;y]].

   A non-HP-object, i.e., an L-molecule, may or may not have a duplicating
L-molecular or H-molecular copy. On the other hand, an HP-object shall
never have a duplicating HP-object. If either x or y represents a non-
HP-object, the value of hcons[x;y] is an L-molecule as a result of cons
to ensure no duplicating HP-object. Only if both x and y represent HP-

objects, the system function mhcons[x;y] gives an H-molecular value to

hcons[x;y]. The following gives an effective procedure for mhcons[x;y]:

Search through all HLISP cells for an H-molecule with address i, such

that m[x]=mcar[i] and m[y]=mcdr[i]. If found i is the result, otherwise

take an inactive free storage cell, the jth say, make a new H-molecule

by the substitutions mcar[j]:=m[x], mcdr[j]:=m[y] and return j as the

result. Actually, the speed of the search through operation in this

procedure is improved by using a hash coding scheme, which is described

in §5.

hconspp[x;y]=[hp[x] ' hp[y] / (hcons[x;y] is present)→

prog2[setq[*aside;hcons[x;y]];T];T→NIL]

This is the presence predicate for hcons[x;y]. In case hcons[x;y] is

present as an H-molecule, the value of *aside is set aside to

the already exsisting hcons[x;y]. hconspp neither creates a new H-

molecule nor a new L-molecule.

The car and cdr parts of an associator will be called respectively

the value and key parts hereinafter so as to stress their meaning rather

than a specific implementation (cf. Fig.2). Any HP-object, say x, can

have at most one associator. If x has an associator, the content of

the key part of the associator is m[x]. Non HP-objects, i.e. L-molecules,

shall never have associators. Hence, the content of the key part of an

associator is necessarily an HP-object.

For an HP-object x, the value of a function hassoc[x] is the

associator of x. Similarly to hcons, the following gives an effective

procedure for hassoc[x]: Search through all HLISP cells for an associator

with address i such that m[x]=mcdr[i]. If found i is the result,

otherwise take an inactive free storage cell, the jth say, make a new

associator by substitutions mcar[j]:=d0 and mcdr[j]:=m[x] and return j
as the result. Speeding up by hash coding is described in §5 and d0 is
a system constant which stands for undefined values. The content of the
value part of an associator can be any HLISP object or d0.

dzerop[x]=[hdotp[x] ∧ (The value of x is d0, i.e., undefined)→T;T→NIL]

    is the predicate to check for d0 valued HDOTP's. The pseudo function

dzero[x]=x, in value, sets the value of x to d0 if x is an HDOTP,
    otherwise it has no effect.

hassocpp[x]=[not[hdotp[x]] ∨ (associator of x absent)→NIL;
    setq[*aside;hassoc[z]] → [dzerop[*aside] → 0;T →1]]
    This is the presence predicate for associator of x.

    For an associator a, evaluation of set[a;v] sets v into the value
part of a. Conversely, hvalue[a] gives the value part of a as its value;
in case a is d0 valued, it is regarded as an error.

    Headers are similar to associators. The above explanations of
associators all hold for headers as well by changing the word "associator"
into "header" and the function names hassoc/hassocp/hassocpp into
header/headerp/headerpp. HLISP utilizes headers for associating the
print names of literal atoms with its value (cf. header of NIL in Fig.2)
and for assoccomp to be described in §4. In this respect headers may be
termed system associators. In the present implementation of HLISP, header
is a pure system function and is not explicitly exposed to the user in
order to avoid possible confusions and conflicts between the system and
the user programs.

    As an example of the use of associators, the data structure with
two associators in Fig.2 can be created by executing the following

LISP program (= stands for "define" and ===, for "the value is"):

```
makedic[x;y;u;v]=set[hassoc[hcons[x;y]];hcons[u;v]] === (MAKEDIC)

makedic[ENGLISH;I;GERMAN;ICH] === (GERMAN.ICH)

makedic[GERMAN;ICH;ENGLISH;I] === (ENGLISH.I)
```

The content of this miniature dictionary can be retrieved associatively
in the following way:

```
readdic[x;y]=[

    hconspp[x;y]→[onep[hassocpp[*aside]]→hvalue[*aside];T→NIL];T→NIL]

    === (READDIC)

readdic[ENGLISH;I]===(GERMAN.ICH), readdic[GERMAN;ICH] === (ENGLISH.I)

readdic[ENGLISH;ICH] === NIL, readdic[ENGLISH;HE] === NIL.
```

Suppose the item introduced by

```
makedic[ENGLISH;YOU;GERMAN;DU] === (GERMAN.DU)
```

is to be cancelled for some reason. This is done in the following way:

```
deletedic[x;y]=dzero[hassoc[hcons[x;y]]]=== (DELETEDIC)

deletedic[ENGLISH;YOU] === (ENGLISH.YOU), readdic[ENGLISH;YOU] === NIL.
```

It would be worth noting that the effect of list structure altering pseudo functions, rplaca, rplacd etc. must be prohibited from rewriting HP-objects to ensure the duplicate copy free (or the monocopy) feature of the HP-objects. In other words, H-molecular cells and the key part of HDOTP's must be regarded as rewrite protected read only storage. The garbage collector only are allowed to reclaim them into the inactive free storage state. This rewrite protection feature has the the following fringe benefit. rplaca and rplacd and their derivatives (nconc etc.) are very useful for writing efficient algorithms. On the other hand, unexpected alteration of data structures including S-expression programs often causes program bugs which are difficult to locate. In HLISP, data not to be rewritten can be represented as HP-objects. Unexpected data alteration would result in the printing of error messages and/or errorset treatments.

F. Motoyoshi pointed out that type predicates hconsp, hassocp and headerp can be defined in terms of the corresponding presence predicates in the following way:

$$hassocp[x]=[hdotp[x]\rightarrow[hassocpp[hkey[x]]\rightarrow eq[*aside;x];T\rightarrow NIL];T\rightarrow NIL].$$

It is impossible, however, to express presence predicates in terms of other functions except hnext[].

§3. Monocopy Algorithms

An important HLISP function hcopy can be defined as:

hcopy[x]=[hp[x]→x;T→hcons[hcopy[car[x]];hcopy[cdr[x]]]].

hcopy gives an HP-object equal to x. One may use LISP to create some data structures by using L-molecules for speed or for some other reason and then switch to hcopied data so as to utilize the speed in checking equalities among the data.

If this hcopy were applied to the data structure ct[n;A;B], consisting of n L-molecules, it would result in $2^{n+1}-1$ recursive calls of hcopy. Thus, the execution of hcopy would become impracticable even for rather small value of n.

In order to remedy such a hazard the following function, rplach ("replace H") could be used instead of hcopy.

rplach[x]=[hp[x]→x;T→hcons[rplaca[x;rplach[car[x]]];

rplacd[x;rplach[cdr[x]]]]].

This pseudo function would replace the contents of L-molecules in (the sub-structures of) x by their "hcopies", so that the execution time for any data structure x would be proportional to the number of L-molecules in x. This replacement would be useful if such side effects were the intention of the user. But if not, another hazard! This can be remedied by keeping records of the replacements made during the execution of rplach and by restoring the original data before completion. For machine language implementations the system push down stack can be used for such recording. In the following definition of hcopyl, a list, pds (for Push Down Stack) is used instead.

hcopyl[x]=prog[[pds, result]; setq[result; rplachl[x]];

A [null[pds]→return[result]];

```
        rplaca[car[pds]]; cadr[pds]]; rplacd[car[pds]; caddr[pds]];

    setq[pds;cdddr[pds]];go[A]]

rplachl[x]=prog[[];[hp[x]→return[x]];

    setq[pds;cons[x;cons[cdr[x];cons[cdr[x];pds]]]];return[

    hcons[rplaca[x;rplachl[car[x]]];rplacd[x;rplachl[cdr[x]]]]]]]
```

The execution time of _hcopyl_ is proportional to the number of L-

molecules in x and there is no hazardous side effects any more.

The function eq[hcopyl[x]; hcopyl[y]] checks the equality of x and

y, with the execution time being proportional to the number of L-molecules

in x and y.

    We shall also make use of the following function which gives the

list of hcopied list elements as its result:

mapcarhcopy[x]=mapcar[x;function[hcopyl]]

        =[null[x]→x;T→cons[hcopyl[car[x]];mapcarhcopy[cdr[x]]]].

    We now discuss the application of the monocopy feature to the basic

set operation _union_ which is defined in Lisp-1.5[2] as:

union[a;b]=(a ∪ b)=[null[a]→b;member[car[a];b]→union[cdr[a];b];

            T→cons[car[a];union[cdr[a];b]]]

member[x;y]=[null[y]→y;equal[x;car[y]]→T;T→member[x;cdr[y]]]

Given two lists a and b with N_a and N_b elements, the execution time of

union is proportional to $N_a \cdot N_b$, since $N_a \cdot N_b$ references are made of _equal_

to check for the equality among the elements. In case many of the elements

are non-atomic(molecular), use of _hcopied_ elements would greatly speed

up the _equality_ checking operations.

    Further improvements can be attained in case the elements are pre-

sorted. Note that the internal representatives of HP-objects establish

a perfect linear order, to be called H-order, among HP-objects. Now, let

<u>align</u> be a function which aligns the HP-objects in a given list in the

ascending H-order. For example, align$[(A_{15}, B_3, (A.B)_9);$function$[$hltp$]]=$

$(B_3, (A.B)_9, A_{15})$ with suffices indicating the internal representatives

at the moment of evaluation of <u>align</u>. For <u>align</u>, the well-known sort-

merge algorithm is to be used, which aligns n elements with the execution

time being proportioned to $n.\log_2 n$ in the worst case. If both elements

in lists a and b are aligned in the ascending H-order, <u>union</u> can be

realized as a merge algorithm, of which the execution time is proportional

to N_a+N_b. Thus, we obtain the following algorithm with the predicate

<u>hltp</u> being given as a functional argument:

union$[$a;b$]=$merge$[$align$[$mapcarhcopy$[$a$]$;function$[$hltp$]]$;

align$[$mapcarhcopy$[$b$]$;function$[$hltp$]]$;function$[$hltp$]]$

merge$[$a;b;p$]=[$null$[$a$]\rightarrow$b;null$[$b$]\rightarrow$a;

   p$[$car$[$a$]$;car$[$b$]]\rightarrow$cons$[$car$[$a$]$;merge$[$cdr$[$a$]$;b;p$]]$;

   p$[$car$[$b$]$;car$[$a$]]\rightarrow$cons$[$car$[$b$]$;merge$[$a;cdr$[$b$]$;p$]]$;

   T$\rightarrow$cons$[$car$[$a$]$;merge$[$cdr$[$a$]$;cdr$[$b$]$;p$]]]$

align$[$a;p$]=[$null$[$a$]\rightarrow$a;T$\rightarrow$car$[$sort$[$sort2$[$a;a;p$]$;p$]]]$

sort$[$x;p$]=[$null$[$x$]\vee$null$[$cdr$[$x$]]\rightarrow$x;

      T$\rightarrow$sort$[$cons$[$merge$[$car$[$x$]$;cadr$[$x$]$;p$]$;sort$[$cddr$[$x$]$;p$]]$;p$]]$

sort2$[$x;y;p$]=[$null$[$cdr$[$y$]]\rightarrow$list$[$x$]$;p$[$car$[$x$]$;cadr$[$y$]]\rightarrow$sort2$[$x;cdr$[$y$]$;p$]$;

         T$\rightarrow$cons$[$sort1$[$x$]$;sort2$[$cdr$[$y$]$;cdr$[$y$]$;p$]]]$

sort1$[$x$]=[$eq$[$x;y$]\rightarrow$list$[$car$[$x$]]$;T$\rightarrow$cons$[$car$[$x$]$;sort1$[$cdr$[$x$]]]]$

In case the given lists a and b are already aligned in the ascending H-

order, <u>align</u> is designed to give the result with the execution time being

proportional to N_a+N_b. Note that the resultant list of merge is aligned

in the ascending H-order.

   Thus, the execution times of this algorithm is proportional to

$N_a + N_b$ in case one can consistently use H-ordered lists throughout the program and is proportional to $N_a \cdot \log_2 N_a + N_b \cdot \log_2 N_b$ in case pre-alignment is necessary.

The H-order may also be termed subjective or relative order, since no objective or absolute order is specified by the user of _union_. The H-order is also relative in that the specific H-order to be established among HP-objects depends upon the entire state of the free storage. Nevertheless, the monocopy feature with the rewrite protection mechanism establishes a consistent H-order during the execution of an HLISP program, and this is just what is needed for the present scheme.

When prealignments are needed, the following algorithm, of which the execution time is always proportional to $N_a + N_b$, is more recomendable:

union[a;b]= λ [[u];bindq[[u;remdup[u];NIL]][mapcar[append[a;b];

$$\text{function}[\lambda[[v];\text{hassoc}[\text{hcopy}[v]]]]]$$

remdup[w]=[null[w]→NIL;hvalue[car[w]]→remdup[cdr[w]];

set[car[w];T]→cons[hkey[car[w]];remdup[cdr[w]]]].

An example: ((..A) is the associator of A. cf. _print_ in Appendix)

union[(Λ,(A.B));(B,(A.B))]

=remdup[((..A),(..(A.B)),(..B),(..(A.B)))]=(A,(A.B),B)

The initial argument to _remdup_ is the list, u of _associators_ of _hcopied_ elements of append[a;b]. The values of the associators are initially all NIL and are set to T when the pertinent elements are enlisted in the result so as to _remove duplicating_ enlistments.

Other basic set operations such as intersection[x;y]=x∩y, setdifference[x;y]=x∩¬y, subsetp[x;y]=[x⊆y→T;T→NIL], etc. can be implemented similarly.

§4. The Assoccomp (<u>Assoc</u>iative <u>Comp</u>uting) Feature

This system feature of HLISP may best be explained by examples.

Many mathematical functions are characterized, defined or evaluated in

terms of recurrence formulas. These formulas can be directly translated

into lisp functions such as

(4.1) fb*[n]=[n=0→fb0;n=1→fb1;T→u[fb*[n-1];fb*[n-2];n]] and

(4.2) c*[n;m]=[m=0→c0;m=n→c1;T→v[c*[n-1;m];c*[n-1;m-1];n;m]].

These algorithms are effective, i.e., the results are obtained eventually

for all integers $n \geq m \geq 0$, provided that u and v always yield some values

(symbolic or numerical) for given arguments. In the special cases

fb0=fb1=c0=c1=1, u[x;y;z]=v[x;y;z;w]=x+y, (4.1) and (4.2) give Fibonacci

numbers fb[n] and binomial coefficients c[n;m] respectively. These

algorithms, however, are known to be notoriously inefficient. The reason

for this can be explained simply by the diagrams shown in Fig.3.
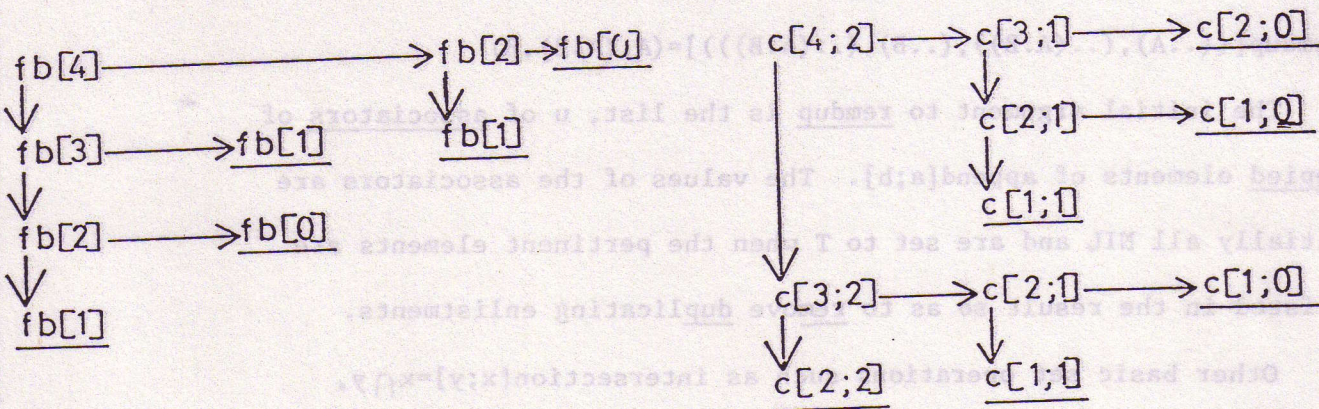

Fig.3 Recursive References in the Evaluation of
       a Fibonacci Number and a Binomial Coefficient.

These diagrams show the recursive reference made by the functions for the given argument/s. For example, fb[4] refers fb[3] and fb[2], fb[3] refers fb[2] and fb[1] and so on. For both fb and c the reference diagrams are binary trees, ending at leaves where the values of the function evaluate to 1. Since, fb and c are defined so as to sum up the 1's on the leaves, there must be fb[n] and c[n;m] leaves respectively. Since there are one less branches than leaves in any binary tree, the total number of recursive references and their asymptotic forms for large n and m=n/2 (which maximizes c for given n) are given respectively by

$$rfb[n]=2 \cdot fb[n]-1 \approx 0.894 \cdot (1.62)^{n+1}, \quad rc[n]=2 \cdot c[n;m]-1 \approx 2^{n+2}/\sqrt{2 \pi n}.$$

These results are independent of the choice of functions u and v and of the values fb0, fb1, c0 and c1. Because of such exponential explosions in the execution time, the recursive algorithms (4.1) and (4.2) are regarded impractical and faster (recursive or non-recursive) algorithms, such as

(4.3)   fb[n]=fb3[1;1;1], fb3[i;j;k]=[n≤i→j;T→fb3[i+1;j+k;j]],

are usually used instead.

Such explosions are actually caused by the repeated evaluation of the same function for the same argument/s, as in the cases of fb[2] and c[2;1] in Fig.3. In hand calculations, the human computer would avoid such repetions by memorizing the previous results. The assoccomp feature may be regarded as a mechanized version of such a computational procedure.

The pseudo function assoccomp is syntactically the same as trace and takes arbitrary number of function names as its arguments. It may be regarded as an instruction for the system to tabulate functions and make use of the tabulated values whenever possible. Actually, execution of

assoccomp[fb;c] === (FB C) sets an assoccomp flag on the definitions of fb and c, similarly to the case of trace[fb;c] === (FB C).

In case of a traced function, the system would print "THE ARGUMENT OF FB IS 2" upon entering the evaluation of fb[2], and would print "THE VALUE OF FB IS 2" upon completion. The assoccomp works similarly. Upon entering the evaluation of fb[2] the system makes an associative search for whether fb[2] has be evaluated before. If this is the case the previous result is retrieved. Otherwise the system evaluates fb[2] by making use of the effective procedure (4.1) and stores the information necessary for possible future retrieval upon completion. For assoccomped n argument function g, the system upon entering the evaluation of g with actual (evaled) arguments a_1^*, ..., a_n^*, first makes k=hcopy[(g*, a_1^*, ... a_{n-1}^* a_n^*)]. Headers are used in assoccomp for associative retrieval. Namely, the system next does the following:

[onep[headerpp[k]]→hvalue[*aside];zerop[headerpp[k]]→error[

$$CYCLIC DEFINITION ASSOCCOMP$];T→set[header[k];apply[g;(a_1^*, ...a_n^*)]]

Note that programming errors caused by cyclic definitions, such as

fb[n]=[n<1→1;T→fb[n]+f[n-1]] (fb[n] in the definition of fb[n]),

are detected by making use of d0 valued headers.

Thus, the assoccomp feature would remedy the exponential time explosion hazards often encountered in recursive algorithms such as (4.1) and (4.2). Actually in the Fibonacci case, fb[n] on the present HLISP system, the first evaluation of fb[20], which has to compute fb[19], fb[18] ... fb[1] and fb[0], was made 200 times faster and re-evaluations of fb[20], 4000 times faster by using assoccomp. These figures will be further improved by level ups to be made in the system, especially by those in the hash coding subroutine.

In the binomial case (4.2), the assoccomp feature would also have the following merits in comparison to other methods. Firstly, the use of the close form formula $c[n;m]=n \cdot (n-1) \cdots (n-m+1)/m!$ for $m \leq n/2$ could yield very large intermediate results and a sophisticated multiprecision arithmetic functions or subroutines would be needed, while there is no such need in assoccomped (4.2). Secondly, in many application of the binomial coefficients many different coefficients would be used repeatedly. The assoccomp would soon provide a complete table for all such coefficients.

A. Nozaki suggested that for many numbers in combinatorics, Stirling's number and the partition number for example, no closed form formulas are known and they are to be computed by recurrence formulas. The assoccomp scheme would be useful in such cases.

The assoccomp scheme should not be regarded as a universal remedy for speeding up any arbitrary computations. First of all for many basic operations, it would be faster to compute rather than trying to retrieve the previous results. In the present HLISP, therefore, assoccomp on basic built in functions are all neglected. For user defined functions, it is the programmer's prerogative to select the functions to be assoccomped for the best result.

There can be a storage explosion hazard in the assoccomp scheme, since headers used in the scheme consumes free storage. In the present HLISP, in case the ordinary garbage collector (GBC) fails to reclaim enough free storage space, a subroutine called the grand garbage collector (GGBC) is called for. GGBC cancels all headers used in assoccomp and return them to the free storage. When this takes place, the assoccomped functions are computed according to the effective but not necessarily efficient original algorithm. If there is a reasonable space in the

storage, the headers of the frequently used assoccomped functions would be reconstructed and the system would soon regain the speed. If there is not enough space for assoccomp headers, the computation would proceed mainly by using the effective (but inefficient) original algorithm.

Methods used for reducing the volume of mathematical tables would also apply to assoccomp. For example, the following definition for c[n;m] would halve the storage requirements, in most cases, by subjecting d only to assoccomp.

$$c[n;m]=[m{\leq}n-m{\rightarrow}d[n;m];T{\rightarrow}d[n;n-m], \quad d[n;m]=[m=0{\rightarrow}1;T{\rightarrow}c[n-1;m]+d[n-1;m-1]].$$

Reading of files (drums or disks) would provide another interesting example. Let file[x] be a pure function of which the value is the S-expression read from the file from the entry specified by x. The assoccomp feature applied to file would bring the most frequently and recently used data into the high speed storage enabling them to be accessed faster upon reuse.

Among numerous other possibilities, application to association lists would be worth noting. Association list schemes, provided by functions[2] assoc[x;a] and pairlis[x;y;a] or the similar, are frequently used in lisp programs, including the Lisp-1.5 system itself. The assoccomp applied to assoc or the similar would provide a logically consistent scheme for speeding up the retrieval of data from association lists. This would be particularly efficient when the association list is deep. It would be interesting to apply this scheme to the A-list of Lisp-1.5 and to compare and evaluate the performances.

§5. An Implementation of HLISP

The most crucial feature of HLISP consists in the hash coding
scheme for accelerating the storage search procedures. Besides the flag
bit method (cf. Fig.2), L-molecules can be identified by separating the
storage areas. Namely, in terms of a system constant minh, x is an L-
cell (L-molecule), if $m[x]=i<minh$, otherwise x is an HP-object. Inflexi-
bility in the number of L- and H-cells available to the user is the draw-
back of this method. It seems impossible to shift the value of minh
dynamically during the execution of HLISP programs, especially if the
H-order feature (cf. §3) is to be used. Nevertheless, this method was
employed in the first version of HLISP and will also be explain here
because of its simplicity.

The scheme employed is essentially a rehashing method. It makes
use of two types of inactive cells H0 and H1 (cf. Fig.2) to allow for
dynamical cancellations invoked by the GBC (garbage collector).

H0-cells are inactive cells without conflict and H1-cells are
those with conflict, i.e., cells which has been reclaimed by GBC which
had been active cells in conflict. The following gives an algorithm
for mhcons (cf. §2). As seen from mhcons, H1-

mhcons[x;y]=prog[[i;j;k];i:=hashcons[x;y;k];

   A   [h0p[i]→go[D];h1p[i]→go[B];eqhconsp[x;y;i]→return[i]];

       i:=rehash[i;k];go[A];

   B   j:=i

   C   j:=rehash[j;k];[h0p[j]→go[D];eqhconsp[x;y;j]→return[j]];go[C];

   D   mcar[i]:=m[x];mcdr[i]:=m[y];nh0=nh0-1;[nh0≤mh0→gbc[]];return[i]]

eqhconsp[x;y;i]=hconsp[i] ⋁ (mcar[i]=m[x]) ⋀ (mcdr[i]=m[y])

, where hashcons maps the two HP's, x and y, onto the H-area (i.e., i such that minh≤i≤maxh);

rehash maps i in H-area onto H-area with the maximum periodicity, i.e., the period is equal to the size of the H-area, nhash=maxh-minh+1 and k is a variable of which the value is initialized by hashcons and advanced by rehash in order to improve the clustering characteristics.

The followings are the simplest hashing and rehashing functions with rather poor clustering characteristic because of negligence of k:

hashcons[x;y;k]=remainder[m[x]+m[y];nhash]+minh

rehash[i;k]=[i=maxh→minh;T→i+1]

The speed and other performances critically depend upon the choice of these functions. The best choice, however, largely depends on the repertoire and the relative speeds of the machine instructions.

hassoc and header can be implemented similarly to mhcons. For hassoc, hashcons in mhcons is to be replaced by hashassoc[x;k] which maps x onto the H-area; eqhconsp, by

eqhassocp[x;i]=hassocp[i]∧(hkey[i]=m[x]) and the line starting from D, by substitutions described in §2. As seen from mhcons, H1-cells can be reused but they can not serve as terminals of the rehashing cycle. Only H0-cells serve as terminals. Hence, the average length of hash-rehash cycle is $1/p0=nhash/nh0$, where nh0 is the number of H0-cells and p0 is their fraction among H-cells. When nh0 becomes less than a system constant, mh0=minp0·nh0, with minp0=0.2 in the present HLISP, GBC is called so as to preserve the speed of the hashing process by keeping $1/p0≤5.0$.

The GBC proceeds in the following steps. The effect of GBC on L-

-22-

molecules are the same as in other lisps.

G1. Mark (as in other lisps) all cells referred from active system
   registers, from the system pushdown stack and from non-d0-valued
   HDOTP's.

G2. Change all unmarked H-cells (including H0 and H1-cells) into H0-cells.

G3. Unmark all H-cells.

G4. Execute the following, which changes H0-cells in conflict into H1-cells.

h0toh1[]=prog[[*hnext;i];

   A [null[hnext[]]→return[NIL]];i:=hashall[*hnext;k];

   B [i=*hnext→go[A];h0p[i]→sethl[i]];i:=rehash[i;k];go[B]]

   , where hashall[j;k] gives the hash number of HP-object j and

   sethl[i] changes the H0-cell i into an H1-cell.

G5. A step to be added in §6 to improve performance.

G6. Call GGBC if the number of H0-cells reclaimed is insufficient.

   An important feature of this dynamic hash coding scheme consist in
the fact that the fraction of H1-cells p1 does not grow beyond a certain
limit which is about 0.16 in the present system. This was first verified
by simulation and a more detailed analysis will be given in a separate
report[4]. Thus, about 64% of H-cells are available as active cells
and about 36% are reserved for the hashing process. The actual saving
in storage space effected by the monocopy feature is about 30 to 40%
depending upon the nature of the program. Thus, this saving is approximately
offset by the hashing overheads.

   A few comments would be sufficient for the rest of implementation.

As indicated in the appendix, similarly to Lisp-1.6[3], the present HLISP
does not use the A-list of Lisp-1.5[2]. The BCP (Binding contex pointer)

mechanism in Lisp-1.6 is not used either for the sake of speed and
simplicity, which is rather essential for virtualizing the pushdown stack
to be described in §6. The two features in which the use of the A-list
is essential in Lisp-1.5 are treated as in the following. Optional
freezing is used for quoting functional arguments and a single variable
*fexpr is reserved for FEXPR's (cf. Appendix). The library functions
syntax or scompile would give warnings to appearance of unfrozen free
variables in functional arguments and to the use of the variable *fexpr
in places other than in the definition of FEXPR's. The header of an atom
does not have a P-(Property-) list (cf. Fig.2). P-list functions (attrib,
prop, get, remprop) are implemented similarly to makedic, readdic and
deletedic of §2, but the system does not make use of them for the sake
of speed. Each SUBR or FSUBR is given a unique code number, which is
used as an index of a computed goto jump (in FORTRAN terminology) for
deciphering it in a single step. EXPR's are identified by LAMBDA or
LABEL and FEXPR's, by FEXPR. In case of pure interpretation, SUBR- and
FSUBR-name atoms are supposed to have the corresponding code numbers in
their value parts. syntax would give warnings to the use of these names
as λ- or prog-variables. scompile would translate these names into the
code numbers and would make various short cuts. The form (EQ (CAR X)
(QUOTE PLUS)) would be scompiled into (-9503(-9499 X)(-9466 PLUS)), which
short cuts three references to the array mcar (cf. Fig.2). In case
idiomatic combinations of basic functions, such as eqq, atomcar, eqqcar
(eq with the 1st argument quoted and the 2nd argument cared), etc., are
built in, the form above would be scompiled into (-9570 PLUS X).

ELISP is fully coded in FORTRAN for the sake of machine independence.
Hand recompiling of the object codes was very rewarding for a small

-24-

machine with a poorly optimized compiler (in FACOM 270/20 with 16 KW of
16 bit words, the speed was doubled and code length, halved). For a
larger machine with an optimized compiler, (FACOM-230/75 with 192 KW of
36 bit words), hand recompiling was hardly meaningful.

§6. A Non Paging Virtual Memory Scheme in HLISP

By virtual memory in this section, it is meant schemes, logically
invisible from the user, which utilize the secondary storage (drum or
disc to be called "file" hereinafter) as an extension of the primary
storage (core or IC, to be called "core" hereinafter). Core paging
schemes, occasionally regarded synonymous to virtual memories, will not
be considered here.

Virtualization of the system pushdown stack is simple and almost
trivial. When the stack in the core overflows, the stack is "pushed
down" halfways in the core and the bottom half is saved in the file.
When the stack in the core underflows (or is over-popuped), the bottom
half of the stack in core is restored from file. The only parameter the

system designer has to select is the size of the stack to be placed in core so as to keep the file read/write overhead times below a reasonable level. 1KW would be sufficient in most cases.

Storage for H-molecules is virtualized in the following way: An HDOTP is called a FILEP if the value part contains a special number called the file address. Actually in terms of system constants maxf and minf=maxint+1, a system predicate filep is defined as:

$$filep[x]=hdotp[x] \bigwedge (minf \leq mcar[x] \leq maxf).$$

A file address represents the (direct or indirect) address in the file, where an H-molecule has been filed, in a S-expression format, by the GGBC. Whenever a FILEP is evaled (by the eval part of the system or by hvalue), the H-molecule corresponding to the file address is reconstructed in core and the value part of the FILEP is replaced by a pointer to the H-molecule reconstructed. This file address and the pointer are registered in a table called the "file record", which is used to prevent the GGBC from spending unnecessary file space and time in repetitively refiling the same H-molecule. (Note that the contents of H-molecules are never altered.)

The GGBC, called from the last step G6. of GBC, proceeds in the following steps:

GG1. Cancel all headers used in assoccomp (cf. §4).

GG2. For each item in the file record, exchange the file address and the content of the car part of the H-molecule.

GG3. Replace the value parts of HDOTP's and values in the pushdown stack by the file addresses, if the values represent H-molecules with file addresses in the car parts. This step efficiently refiles H-molecules which have been filed before.

GG4. Restore the car parts of H-molecules from the file record.

GG5. Cancel all file records.

GG6. File H-molecules in the value parts of HDOTP's and in the pushdown
stack and replace them by the corresponding file addresses. This
step creates new files of H-molecules which have not been filed
before.

GG7. Call GBC. If still not enough H0-cells are reclaimed the execution
of the HLISP program is terminated.

G5. In the fifth step G5. in GBC, items in the file record with pointers
to H0- or H1-cells are (and must be) cancelled.

Since sequential addressing is sufficient for the functioning of file
records, they are stored in the file with a small buffer in core.

In order to fully utilize the virtual memory scheme, the standard
definitions of functions or library functions are loaded in the file
together with the HLISP system. When a d0-valued atom is _evaled_ or _hvalued_,
the system searches for a standard definition in the file. If found, the
definition is written in the core as an HP. (If unfound, an error.) The
standard definition of _define_ or _scompile_ gives _hcopied_ results.

The virtual memory has enabled the use of collection of functions
amounting to several times larger than the free storage capacity in a single
HLISP program without any difficulty or complexity at a reasonable operating
speed[5]. This would be meaningful for application areas of lisp, wherein
programms become very large to obtain results of any significance whatsoever.

The monocopy and the rewrite protected features of H-molecules are
essential for the present virtual memory scheme. Design of a logically
consistent and reasonably efficient virtual memory scheme for lisp data
subjected to arbitrary use of rplaca and rplacd (L-molecules in HLISP
term) seems to be an open problem within the author's knowledge.

In the present HLISP, if the L-molecular data explodes during the execution of a program, the program has to be revised. Making hcopies of some L-molecular data not subjected to rplaca or rplacd would work in some cases.

The present GGBC takes all conceivable measures to reclaim the greatest number of free storage cells in a single stage. It may be a better strategy to break it down into three stages as follows and proceed to the next stage only if it fails to reclaim enough HO-cells: The 1st the refiling (GG2-4.), the 2nd the new filing (GG6.) and the 3rd the cancellation of assoccomp headers (GG1.) stages.

§7. Concluding Remarks

Hash Coding is practiced in many language processors, including lisp[6], for matching strings of symbols in identifiers. HLISP may be regarded as a straightforward extension of such practices.

Use of the simplest possible data structure--the binary trees-- and the ability to define or bootstrap the system in terms of a very few basic functions are the characteristic features of lisp. In order to comply with this philosophy of lisp, the binary association of keys with values has been employed in HLISP because it seems to be the simplest. It should be possible to efficiently represent more complex associative structures, such as those used in LEAP[7] for example, in terms of binary trees and binary associators.

The present HLISP functions do not allow for circular list (loop or ring) arguments. Endless recursion would result in most cases as in many lisp functions. Generalization of the present HLISP scheme to treat more general types of internal data structures in a logically consistent manner would be an interesting theme.

References

1) H. Sato, K. Noshita, A. Nozaki and E. Goto, "Some Remarks on Data Structures of Lisp", Proceedings of the Programming Symposium, Jan. 1973, Information Processing Society of Japan. (in Japanese)

2) J. McCarthy et al., "LISP 1.5 Programmer's Manual", 1965, The MIT Press.

3) L. H. Quam, "Stanford Lisp 1.6 Manual", Stanford Artificial Intelligence Laboratory Operating Note No. 28.4

4) T. Gunji and E. Goto, "Analysis of a Dynamic Hash Coding Scheme", to be published.

5) E. Goto et al. "Performance Evaluation of a Non-Paging Virtual Memory in an Extended Lisp", to be published.

6) D. G. Bobrow, "Storage Management in LISP." in Symbol Manipulation Languages and Techniques, North-Holland (1968), pp.291-301.

7) J. A. Feldman and P. D. Rovner, "An Algol-Based Associative Language" CACM 12, pp.439-449 (1969).

APPENDIX

The following is the list of basic HLISP functions and LISP functions subjected to extension or modification in HLISP.

apply[fn;x] The third argument (A-list) of Lisp-1.5 is deleted, since the present HLISP does not use an A-list, similarly to Lisp-1.6.

assoccomp[f_1; ...; f_n] f_1, ..., f_n are function names. cf. §4.

atom[x] associators are regarded as atoms. cf. Fig.2

bind[x;y;z]=apply[list[LAMBDA;x;y];z]. x must represent a list of HDOTP's, y a form and z a list of HLISP objects. If list z is shorter than list x, z is regarded as if there were extra list elements NIL. Associators and headers are treated equally.

bindq[x;y;z] Same as bind except in that y is quoted.

bindq[(A,B,C);plus[a;b;c];(1,2,3)] is the same as

λ[[a;b;c];plus[a;b;c]][1;2;3]=6

These functions are used to λ-bind the values of HDOTP's resulting from the evaluation of x with elements of z, during the evaluation of form y. The former values of the HDOTP's are restored (from the pushdown stack) upon completion.

car[x], cdr[x] x must be a molecule, otherwise an error. Unlike in Lisp 1.5 car and cdr can not go beyond (atom) headers and associators. To go beyond, use hkey, hvalue or dzerop instead.

cons[x;y] x and y can be any HLISP-object. The result is always an L-molecule.

consp[x] Type predicate for L-molecules. cf. Fig.2.

dzero[x], dzerop[x] Undefine and predicate for undefined. cf. §2

eval[e] The second argument is deleted for the same reason as in apply.

eq[x;y]=[m[x]=m[y]→T;T→NIL] Checks equality of internal representatives.

equal[x;y]=[hp[x]∧hp[y]→eq[x;y];atom[x]∨atom[y]→NIL;

equal[car[x];car[y]]→equal[cdr[x];cdr[y]];T→NIL]

In case both x and y are HP's, equal reduces to non-recursive eq

by virtue of the first conditional.

fexpr FEXPR's are defined as in the following example for or.

define[[[or;fexpr[[null[*fexpr]→NIL;eval[car[*fexpr]]→T;

T→apply[or;cdr[*fexpr]]]]]]]

Namely, fexpr[<form>] is syntactically regarded as a <function> and

the value of *fexpr is λ-bound to the unevaled list of arguments.

cf. §5.

function[fn;x_1; ...; x_n] x_1, ..., x_n (optional) are the names of free

variables in the functional argument fn to be frozen. cf. §5.

hassoc[x]/hassocp[x]/hassocpp[x] \ creater/type predicate/presence predicate

hcons[x;y]/hconsp[x]/hconspp[x;y] | for associators, H-molecules

header[x]/headerp[x]/headerpp[x] / and headers. cf. §2 and Fig.2.

hcopy[x], hcopyl[x] cf. §3.

hdotp[x]=hassoc[x]∨header[x] cf. Fig.2.

hkey[x]=[hdotp[x]→(key part of x);T→error[HKEY]]

Extracts the key part of HDOTP's. cf. Fig.2.

hltp[x;y]=[m[x]<m[y]→T;T→NIL]

Similar to lessp. It works on internal representatives. For

basic integers, the result is the same as lessp.

hnext[]=*hnext in value. Advances the value of *hnext to the next H-

molecule, associator or header in storage. All existing headers

except NIL, for example, are printed out by executing: prog[[*hnext];

A [null[hnext[]]→return[NIL];headerp[*hnext]→print[*hnext]];go[A]].

hp[x] cf. Fig.2.

hvalue[x] Extracts the value of HDOTP's.   cf. §2 and Fig.2.

m[x]=(Internal representative of x).   Is used to express system

  functions in M-expressions. cf. §2.

mapcarhcopy[x] cf. §3.

print[x], printl[x] Print with or without line termination.

  The key part of headers are printed as the p-name of literal atoms as

  in other lisps. Associators, L-molecules and H-molecules are

  printed as in the following examples:

  print[hassoc[hcons[ENGLISH;I]]] prints (..(ENGLISH.I)) with dots;

  print[cons[ENGLISH;I]] prints (ENGLISH . I) with extra blanks;

  print[hcons[ENGLISH;I]] prints (ENGLISH.I) without extra blanks.

rplaca[x;y], rplacd[x;y] x must be an L-molecule, otherwise an error.

              y can be any HLISP object. cf. §2.

set[x;y]=y in value. x must be an HDOTP. y can be any HLISP object.

  In HLISP set and cset; setq and csetq are synonyms. cf. §2.