SYSTEM.DOC

```
 -- STANFORD HIGH SPEED BATCH MONITOR --
//J778 JOB 'J778,130,01,15,99','MASINTER                    JOB  423
***SERVICE LOG=NO CLASS=Q EXEC=S
***PRINT CHAIN=B COPIES=10
// EXEC PGM=WYLIST
//SYSPRINT DD SYSOUT=A
//SYSPUNCH DD SYSOUT=B
//SYSUDUMP DD SYSOUT=A
//SYSIN DD DATA
//
          NORMAL JOB END

 RALPH JOB STATISTICS --     904 CARDS READ --      836 LINES PRINTED --      0 CARDS PUNCHED --   0.02 MINUTES EXECUTION TIME
```

NEQ
LEQ
& GEQ

NIL, NUMBERS, ATOMS, SEXPRESSIONS

UNDEFINED FUNCTIONS
REFERED TO ONLY
BY COMPILED FUNCTIONS
ARE REMOVED FROM
OBLIST BY CLEANUP()
AND 'FUNCTION NOT
DEFINED' MESSAGE
IS NOT GENERATED
BUT INSTEAD SOME
CRAZY MESSAGE!

CLEANUP (MACRO COMPILE LAP ARRAY) works though

TIMETOT:
ADDS
2 CONSES AND
.44 MS TO EACH
FUNCTION CALL.

```
10.      J778.SYSTEM DOC on SYS05
20.      This is data-set J778.SYSTEM.DOC on SYS05, last updated 2/8/72.
30.
40.      This is a description of a set of LISP features designed to aid the
50.      LISP programmer.
```

This is data-set J778.SYSTEM.DOC on SYS05, last updated 2/8/72.

This is a description of a set of LISP features designed to aid the LISP programmer.

********** (1) SPECIAL and COMMON variables:**********

VARIABLE	VALUE	PROPERTY
COMMA	$$','	SPECIAL
BLANK	$$' '	SPECIAL
PERIOD	$$'.'	SPECIAL
EQSIGN	$$'='	SPECIAL
LPAR	$$'('	SPECIAL
RPAR	$$')'	SPECIAL
PLUSS	$$'+'	SPECIAL
DOLLAR	$$'$'	SPECIAL
STAR	$$'*'	SPECIAL
CENT	$$'¢'	SPECIAL
QMARK	$$'?'	SPECIAL
EXPNT	$$'!'	SPECIAL
OBLIST	-----	COMMON

**********(2) Extended operation codes for LAP functions.**********

These operation codes ahave been added by putting their values as MC properties on the specivied atom. the operation codes available are: *MXFXAT, *MKLGAT, *MKFLAT, *GEVAL, BE, BL, BNL, BH, BNH, BO, BNO, B, BTYPE, and DCN.

FORM	DESCRIPTION
(*MKFXAT Rn)	Create a fix-point numeric atom with the value of register Rn. Leaves the result in register A.
(*MKLGAT Rn)	Like *MKFXAT, execpt that *MKLGAT creates a logical numeric atom.
(*MKFLAT FPRn)	Makes a floating-point atom with the value contained in the floating point reegister FPRn. The result is left in register A.
(Bx label)	Conditional branch to "label" depending on the previous setting of the condition code. the options and their meaning are:

```
(BE label)          Branch if equal
(BNE label)         Branch if not equal
(BNL label)         Branch if not low
(BL label)          Branch if low
(BH label)          Branch if high
(BNH label)         Branch if not high
(B label)           Branch unconditionally

                ATOM            Branch to "label" if the
                FWD             S-expression addressed by
(BTYPE <NOT> NIL Rn label)      register Rn is/isn't an
                FLOAT           atom, fullword, NIL,
                LOGIC           floating-point number,
                NUMBER          logical number, or numeric,
                                respectively.


  (DCN n val)       Expands to (DC val)(DC val)...
                    repeated n times.
```

**********(3) Macro expanding functions.**********

A macro function is defined by putting a lambda-expression on the property-list of the function with one of four properties *-MACRO, *-MACWEISS, *-MACEXPR, *-MACFEXPR, depending on the type of macro. DEFINE has been rewritten to first call a function *-MACSRCH, which examines the function for calls to functions with one of the above four properties. If such a function is encountered, *-MACSRCH expands the macro and replaces the expanded code in the function definition.

The four types of macros are explained below:

(a) MACWEISS. A macro-function of the type MACWEISS is defined by calling the top-level function MACWEISS, similar to DEFINE. This type of macro works exactly like the macros described in Weissman's LISP 1.5 Primer, and is expanded in the same manner. This feature has been included for compatability with other LISP macro features.
The function MACWEISS takes a list of doublets (name lambda-expression) just as DEFINE does, but it puts the lambda-expression on the property list of "name" with the property "*-MACWEISS". The lambda-expression should be a function of one argument which is the form in which "name" occurs.
For example, if one defines *SETQ as a MACWEISS with:

```
MACWEISS(((*SETQ (LAMBDA(X)
  (COND ((NULL (CDDR X))(CADR X))
        (T (LIST (QUOTE SETQ) (CADR X)
                 (CONS (QUOTE *SETQ)(CDDR X)))))))))
```

when *-MACSRCH encounters the form (*SETQ A B C D E), it applies the above lambda expression to the entire list "(*SETQ A B C D E)"; that is, "(*SETQ A B C D E)" is bound to x and "(COND ((NULL (CDDR ...."

is EVALuated.
The result of this application is the form:
(SETQ A (*SETQ B C D E))
*-MACSRCH then re-examines this form for further names with

is EVALuated.
The result of this application is the form:
 (SETQ A (*SETQ B C D E))
 *-MACSRCH then re-examines this form for further names with
macro properties. In the examination of (SETQ A (*SETQ B C D E))
*-MACSRCH again finds **SETQ, this time in the form (SETQ B C D E).
This is expanded to (SETQ B (*SETQ C D E)), so that the original
form now looks like (SETQ A (SETQ B (*SETQ C D E))). This process
continues until *-MACSRCH applies the above lambda-expression to
(*SETQ E), which expands to "E". The final form then is:
 (SETQ A (SETQ B (SETQ C (SETQ D E)))).

 Of the four types of macros, MACWEISS and MACRO are reexamined to see
if the expansion contains any macros.

(b) MACEXPR. To define a macro function of the type MACEXPR, use the
top level function MACEXPR, which again takes a list of name,
lambda expression pairs. The function MACEXPR again puts the
lambda expressions on the property lists of the corresponding
name, this time with the property *-MACEXPR.
In this case, *-MACSBCH binds each of the arguments of the
macro in the form in which it is encountered to the corresponding
lambda variable in the lambda expression under the *-MACEXPR
property.
An example might clarify:

```
MACEXPR(((PRINTE(LAMBDA(A B)
   (LIST(QUOTE PROG2)
        (LIST (QUOTE PRINT)
              (LIST (QUOTE QUOTE) A))
        (LIST (QUOTE PRINT) B))))))
```

 then when *-MACSRCH sees (PRINTE (THE VALUE IS) (CAR X)), it
replaces it with:
  (PROG2 (PRINT (QUOTE (THE VALUE IS))) (PRINT (CAR X))).

Note that the expanded form is not re-examined by *-MACSRCH.
An example taking advantage of this fact is as follows.
Suppose that, contained within one's code are expressions of the
form (SETQ A B), but that one actually wishes the SETQ to be
changed to a CSETQ in the cases where A has an APVAL. This may
be accomplished by defining:

```
MACEXPR(((SETQ(LAMBDA(A B)
     (COND((GET A (QUOTE APVAL))(LIST(QUOTE CSETQ)A B))
          (T(LIST(QUOTE SETQ) A B)))))))
```

(c) MACFEXPR. A macro of this type is defined using the top level
function MACFEXPR, which puts the lambda-expressions on the
property lists of the names under the property *-MACFEXPR.
The lambda expression for a MACFEXPR should have exactly one
lambda variable, which will be bound to the list of arguments
in the form in which *-MACSRCH encounters the MACFEXPR macro.
An example as well as a standard macro included with the
package is the function *. (Note that * is a valid atom name).

```
MACFEXPR(((* (LAMBDA(FORM)
  ((LABEL EXPAND (LAMBDA (CARF FM)
      (COND ((NULL (CDR FM))(CAR FM))
            (T (LIST CARF (CAR FM) (EXPAND CARF (CDR FM))))))))
   (CAR FORM)
   (CDR FORM))))
 ))
```

After short inspection, one can see that *-MACSRCH will expand the form (* SETQ A B C D) to (SETQ A (SETQ B (SETQ C D))), and the form (* PROG2 (PRINT A) (PRINT B)(PRINT C)) to:
(PROG2 (PRINT A) (PROG2 (PRINT B) (PRINT C))).

Two final examples:

```
(* CONS 1 2 3 4)        ------>    (CONS 1 (CONS 2 (CONS 3 4)))
(* anything A)          ------>    A
```

(d) MACRO. The fourth type of macro is the easiest to use.
The top level function is MACRO, which puts the lambda-expression under the property *-MACRO. In this case, *-MACSRCH does not evaluate the lambda-expression but uses it as a template for the expanded form.
For example, if one defines:

```
MACRO(((CAAAAR(LAMBDA(X)(CAR(CAAAR X))))))
```

then (CAAAAR (DOIT (CAR Z))) -------> (CAR (CAAAR (DOIT (CAR Z))))
(CAAAAR MX) -------> (CAR (CAAAR MX))

In many cases, to switch a function from a DEFINE'd function to a macro, all that is necessary is to change "DEFINE" to "MACRO".
Note however that if one defines

```
MACRO(((PRINTQ(LAMBDA(X)(PROG2(PRINT(QUOTE X))(PRINT A))))))
```

that (PRINTQ (CAR Y)) will be expanded to:

```
  (PROG2 (PRINT (QUOTE (CAR Y))) (PRINT (CAR Y)))
```

WARNING: if one defines, say

```
 MACRO(((DOUBLE(LAMBDA(X)(CONS X X))))))
```

then

```
   (DOUBLE (FN VAR))
```

expands to

```
   (CONS (FN VAR) (FN VAR)); i.e, FN is called twice.
```

If DOUBLE were DEFINEd, instead, FN would only be called once.

(e)The use of the atom @ for quote.

There is a feature built into the definition of *-MACSRCH which enables the user, in any DEFINE, to use the atom @ immediately preceding an S-expression instead of the form (QUOTE ...). That is, (SETQ A @ B) is expanded to (SETQ A (QUOTE B)).

The @ must be delimited like any other atom; (SETQ A @B) is an error. This expansion is done before any other expansion, so that (* CONS @ A @ B C) expands correctly to

```
2400.     The @ must be delimited like any other atom;
2410.     (SETQ A @B) is an error. This expansion is
2420.     done before any other expansion, so that
2430.     (* CONS @ A @ B C) expands correctly to
2440.
2450.     (CONS (QUOTE A) (CONS (QUOTE B) C))
2460.
2470.     Associated along with "@" for QUOTE, the atom "¬"
2480.     is the negation of quoting. That is,
2490.     (SETQ A @ (B C)) expands to (SETQ A (QUOTE (B C))),
2500.     but (SETQ A @ (B ¬ C)) ------->
2510.       (SETQ A (LIST (QUOTE B) C))
2520.
2530.     (eq x @ @) ------> (eq x (quote @))
2540.
2550.     (PRINT @(THE VALUE OF X IS ¬ X)) ------->
2560.      (PRINT (LIST (QUOTE THE)(QUOTE VALUE)(QUOTE OF)
2570.            (QUOTE X)(QUOTE IS) X)).
2580.
2590.     This feature is most handy in defining macro functions.
2600.     Below is given the macro CR, a MACEXPR, which takes
2610.     two arguments. The first argument is an atom whose
2620.     name consists solely of A's and D's; the second
2630.     argument is a form. (CR ADDAD X) expands to
2640.     (CADDR (CADR X)). Note that, in the definition of
2650.     the macro CR, the macro * as given above is used.
2660.
2670.
2680.
2690.     MACEXPR ((
2700.        (CR (LAMBDA (ADS FORM)
2710.           (PROG (CRS)
2720.                 (SETQ ADS (EXPLODE ADS))
2730.           LOOP  (COND
2740.                    ((NULL ADS) (GO MKFORM))
2750.                    ((AND
2760.                       (*
2770.                          PROG2
2780.                          (RLIT @ C)
2790.                          (RLIT (CAR ADS))
2800.                          (SETQ ADS (CDR ADS)))
2810.                       (*
2820.                          PROG2
2830.                          (RLIT (CAR ADS))
2840.                          (SETQ ADS (CDR ADS))))
2850.                      (PROG2 (RLIT (CAR ADS)) (SETQ ADS (CDR ADS)))))
2860.                 (RLIT @ B)
2870.                 (SETQ CRS (CONS (MKATOM) CRS))
2880.                 (GO LOOP)
2890.           MKFORM (COND
2900.                    ((NULL CRS) (RETURN FORM)))
2910.                 (SETQ FORM (LIST (CAR CRS) FORM))
2920.                 (SETQ CRS (CDR CRS))
2930.                 (GO MKFORM))))
2940.        ))
2950.
2960.
2970.
2980.
2990.      (f)  The function ¬ as a top level EVAL with macro expansion:
```

The atom ¬ is used as a top level function (as well as uts use in conjunction with @). If one says

¬ expression at the top level, then first macros are expanded in the given expression, and then it is evaluated. Thus at the top level, one can say:

```
¬ (FOR NEW I :=(1 10) XLIST I)
```

and the value returned by evalquote will be (10 9 8 7 6 5 4 3 2 1)

¬ is defined as an FSUBR.

(g) the *-MACPROG feature:

In some cases, one wishes a macro to expand, not into one form, but into many: for example, if one wanted a macro called REPEAT, where (... (REPEAT 5 (CAR X)) ...) would expand to (... (CAR X) (CAR X)(CAR X)(CAR X)(CAR X) ...)

This can be accomplished by having the initial expansion of the macro to be (*-MACPROG ...)
<in this case, (*-MACPROG (CAR X)(CAR X)(CAR X)(CAR X)(CAR X)):
> *-MACSRCH will bring "in line" any form starting with *-MACPROG--

```
MACEXPR(((REPEAT(LAMBDA(N EXPR)
    (CONS @ *-MACPROG
      ((LABEL NT(LAMBDA(M)(COND((ZEROP M)NIL)
                  (T(CONS EXPR(NT(SUB1 M))))))
               N)))))))))
```

Thus (... (REPEAT 2 (FN VAR)) ...) first expands to
(... (*-MACPROG (FN VAR) (FN VAR)) ...) which is then
changed to (... (FN VAR) (FN VAR) ...)

NOTE : IN MACRO EXPANSION, THE INNERMOST MACROS ARE EXPANDED FIRST!!!!

**********(4) STANDARD MACROS **********

(a) CR, as above

(b) *, as above

(c) COMMENT:

COMMENT is a top level FSUBR: it takes an arbitrary number of "arguments" and returns NIL.

COMMENT is also a macro (MACFEXPR) which initially expands to(*-MACPROG) and then to nothing. Essentially, COMMENT deletes itself, leaving not a trace.

(d) FOR **********

FOR is a very powerful and complex macro, which enables the LISP programmer to write AIGCL-like FOR statements and have the initialization and checking of the loop taken

(d) FOR **********

FOR is a very powerful and complex macro, which enables the LISP programmer to write AIGCL-like FOR statements and have the initialization and checking of the loop taken care of.

Basically, a FOR specifies a group of statements which are executed a number of times, with the value of some variable changing each time through the loop. It is possible to nest loops, to have several variables change, rather than just one, and to control the value of the FOR STATEMENT (in LISP, every expression has a value; the FOR does as well).
One may also specify additional terminating conditions, the type of iteration, and the declaration of new variables.
There are four types of iteration:
two ways of stepping through lists
numerical iteration
simple replacement.
(FOR X IN Y DO ...)
would execute the statements represented by '...' once for each element in the list Y, with the variable X set to that element
(FOR X IN @(A B C D) DO (PRINT X)) would print 'A','B','C','D' in that order.

(FOR X ON Y DO ...) again executes the statements, once for each element of Y, but X is first set to Y, then (CDR Y), ...
(FOR X ON @(A B C D) DO(PRINT X)) would print (A B C D), (B C D), (C D), (D) in that order.

(FOR X := (N1 N2 N3) DO ...)
X is set to N1 initially. After each time through the loop, X is set to X plus N3. Before each execution of the loop, X is compared to N2 and the loop is terminated if X-N2 has the same sign as N3. If N1, N2 or N3 are expressions, they are only evaluated once, before the first time through the loop. N3 is optional -- if omitted, 1 is assumed.

(FOR X IS (ADD1 X) ...)
This is an infinite loop. The IS clause does a simple replacement on the variable each time through the loop (at the beginning); NO TESTS ARE MADE__ IT IS ASSUMED THAT the user also has specified a WHILE or UNTIL clause, or a concurant range (described below).

******** CONCURENT RANGES:
(FOR X IN Y AS I :=(1 10) DO ...)
As x steps down the list Y, I is incremented from 1 to 10. The loop terminates when the end of Y is reached, or I is incremented to 11, whichever comes first.

There is no limit set on the number of concurent rages:

```
4200.     ******* NESTED RANGES:
4210.           (FOR X IN Y
4220.             FOR Z IN X
4230.                 .....          )
4240.          If Y is a list of lists, this executes the loop once for
4250.     each element of each element of Y.
4260.
4270.     ********   NEW VARIABLES:
4280.        In order to say (FOR X IN Y ...), X must already be a variable
4290.     available as a PROG or LAMBDA variable.  If it isn't, one can say
4300.        (FOR NEW X IN  ...) and X will be declared local to the FOR.
4310.       this is also useful in enabling the user to use variables
4320.      which might have different meaning elsewhere in a function
4330.      locally to a for statement.
4340.
4350.     ******  ADDITIONAL TERMINATING CONDITIONS :
4360.
4370.          (FOR NEW SEED IN PUMPKIN
4380.            AS NEW FORS :=(BAD WORSE)
4390.              WHILE (NOT(EQUAL FORS HORRIBLE)) DO     .... )
4400.
4410.           The loop will terminate when
4420.              (1)  The end of PUMPKIN is reached
4430.           or (2)  FORS is greater than WORSE
4440.           or (3)  FORS is EQUAL to HORRIBLE
4450.          whichever comes first.
4460.
4470.          (FOR NEW FEATURE IS (ADD1 FEATURE) UNTIL (UNBEARABLE FEATURE) ...)
4480.
4490.              Here is an example of the IS coupled with an UNTIL.
4500.           the loop will terminate when (UNBEARABLE FEATURE) is TRUE.
4510.
4520.     NOTE:    UNTIL differs from WHILE in that UNTIL tests at the end
4530.         of the loop and WHILE tests at the beginning.
4540.
4550.     **********  the IF claues
4560.          (FOR I :=(1 10) IF (NOT(EQUAL I 5)) ...
4570.         will execute the loop 9 times, as I takes on the values
4580.      1,2,3,4,6,7,8,9,10.
4590.
4600.
4610.     ************ SPECIFYING A VALUE TO THE FOR:
4620.
4630.          If one says "DO", the FOR statement returns NIL.
4640.     However, the FOR can be used for "building up" a value, based on
4650.     the value of the last of the statements of the loop. Example:
4660.          (FOR I :=(1 10) PLUS I)     returns the sum of all numbers from
4670.      one to ten.
4680.          (FOR NEW I :=(1 10)
4690.            AS NEW J IS (TIMES I I)
4700.             LIST
4710.               (CONS I J))
4720.       Returns the list of the values of (CONS I J):
4730.         ((1 . 1)(2 . 4)(3 . 9)(4 . 16).....)
4740.
4750.       Notice that other statements can occur between the value indicator
4760.     and the value expression:
4770.
4780.          (FOR NEW WORD IN MESSAGE
4790.            FOR NEW LETTER IN (EXPLODE WORD)
```

```
4800.             LIST
4810.               (PRIN1 LETTER)
4820.               (ENCODE LETTER)))))))
4830.
4840.      returns a list of (ENCODE letter) for every letter
4850.      in every word of MESSAGE.
```

```
4800.           LIST
4810.             (PRIN1 LETTER)
4820.             (ENCODE LETTER))))))))
4830.
4840.      returns a list of (ENCODE letter) for every letter
4850.      in every word of MESSAGE.
4860.
4870.          In general, the FOR clause is terminated by one
4880.     of the special indicators
4890.           DO, LIST, PLUS, TIMES, PROG2, XLIST, AND, OR, MAX, MIN
4900.     or by any function of two arguments;
4910.       followed by a list of PROG statements
4920.       followed by an expression.
4930.
4940.        The statements are executed once each time through the
4950.     loop, and then the final expression is evaluated.
4960.     The value of the FOR IS AS FOLLOWS:
4970.
4980.        If a RETURN statement is executed as one of the
4990.     statements of the FOR, then the loop terminates and the value
5000.     specified in the RETURN is returned.
5010.        Otherwise, if the value indicator is --
5020.
5030.        DO              the FOR returns NIL after the terminating condition is reached
5040.
5050.        XLIST           THE FOR builds up a list, using CONS, of the values of
5060.                        the final expression.  The last value is the first of the
5070.                        list. If the loop is never entered, the FOR returns NIL.
5080.
5090.        PLUS            the FOR accumulates the sum of the values of the final expression
5100.                        if the loop is never entered, the FOR returns 0.
5110.
5120.        TIMES           the FOR accumulates the product of the values of the final
5130.                         expression.  If any of the values are zero, the FOR terminates.
5140.                         If the loop is never entered, the FOR returns 1.
5150.
5160.        MAX             The maximum of the values of the final expression is returned.
5170.                        If the loop is never entered the FOR returns -1.0e77
5180.
5190.        MIN             The minumum of the values of the final expression is returned.
5200.                        If the loop is never entered the FOR returns 1.0e77.
5210.
5220.        PROG2           The last value of the final expression is returned.  If the
5230.                        loop is never entered the FOR returns NIL.
5240.
5250.        AND             If every value of the final expression is non-NIL, the FOR returns
5260.                        T.  If any value is NIL, the FOR terminates and returns NIL.
5270.
5280.        OR              If every value of the final expression is NIL, the FOR returns
5290.                        NIL.  If any value is non-NIL, the FOR terminates and returns
5300.                        that value.
5310.
5320.        LIST            The FOR accumulates a list, using APPEND1,  of the values of
5330.                        the final expression.  The first value is first in the list.
5340.
5350.     any other function of two arguments
5360.                        If the loop is never entered, the FOR returns the atom *FIRST.
5370.                        If the loop is entered only once, the FOR returns the value of the
5380.                           final expression that time through the loop.
5390.                        After the first time through the loop,  the function of two
```

```
                    arguments is applied to the old return value as the first
                    argument and the new value of the final expression as the
                    second argument, to find the new return value.


 any function of two arguments   FIRST   expression
                    If the loop is never entered, the FIRST expression is returned.
                    Each time through the loop, the function of two arguments is
                    applied to the old return value and the new value of the final
                    expression to find the new return value.
                       The initial old return value is the FIRST expression.

      Notes:     XLIST       ==       CONS FIRST NIL
                 MAX         ==       MAX  FIRST -1.e77
                 MIN         ==       MIN  FIRST 1.e77
                 TIMES       ==       TIMES FIRST 1
                 PLUS        ==       PLUS  FIRST 0


**********GENERAL SYNTAX

                    CN list
    (FOR <NEW> var IN list
                    := (n1 n2 <n3>)

      <AS <NEW> var  ... >
      <UNTIL predicate>
      <WHILE predicate>
      <IF predicate>
         ....

                     |  DO
                     |  LIST
                     |  PLUS
                     |  TIMES
                     |  PROG2
                     |  XLIST
                     |  AND
                     |  OR
                     |  MAX
                     |  MIN
                     |  <any function of two args>
          <FIRST  expression>

            <statement><statement> ...    expression          )


**********(5) Useful Lisp functions.**********

GSET        This function is used to set the
            value of a special variable at the
            top level. For example,
               SPECIAL ((AA))
               GSET (AA T)
            then T is placed in the special cell
            associated with AA.

MAPC        MAPC works as described in the
            Weissman LISP Primer. MAPC takes
```

```
            two arguments; a list and a function
            of one argument. MAPC applies the
            function to each element in the list.
            If the list ends (...  . A), then
            MAPC returns A.
```

```
6000.                two arguments; a list and a function
6010.                of one argument. MAPC applies the
6020.                function to each element in the list.
6030.                If the list ends (...  . A), then
6040.                MAPC returns A.
6050.
6060.      GENSYM2   GENSYM2 is like GENSYM except that the
6070.                resultant atom is on the OBLIST, and
6080.                if read back into the system, will
6090.                correspond to its initial value.
6100.
6110.      COMPRESS  COMPRESS takes a list of atoms as its
6120.                argument, and returns the atom whose
6130.                print name corresponds to the concat-
6140.                enation of the names of the atoms in
6150.                its argument.
6160.                 COMPRESS ((A B C D))---> ABCD
6170.
6180.      EXPLODE1  EXPLODE1 is like explode except that
6190.                if the atom being EXPLODEd contains
6200.                digits in its name, these are returned
6210.                as numbers.
6220.                (MAPCAR (EXPLODE (QUOTE A1A)) (FUNCTION NUMBERP))
6230.                  ------> (NIL NIL NIL)
6240.                (MAPCAR (EXPLODE1(QUOTE A1A)) (FUNCTION NUMBERP))
6250.                   -----> (NIL T NIL)
6260.
6270.      SORT      SORT takes two arguments, a list and
6280.                a predicate function of two arguments.
6290.                SORT returns the list, sorted according
6300.                to the functional argument.
6310.                 (SORT @(5 1 2 4)(FUNCTION LESSP))
6320.                   -----> (1 3 4 5)
6330.                 (SORT @(3 1 4 5)(FUNCTION GREATERP))
6340.                   -----> (5 4 3 1)
6350.
6360.      INSERT    INSERT takes three arguments;  the third is an ordering predicate
6370.                the second is a list already ordered by that predicate, and the
6380.                first is an S-expression to be INSERTed into that list in the
6390.                appropriate place.
6400.
6410.      NEQ       NEQ is a semi-predicate of two arguments.
6420.                It returns NIL if the two arguments are
6430.                EQUAL. If the two arguments are not equal
6440.                it returns either the atom GRT or the atom
6450.                LSS depending on the following ordering:
6460.                (1) numbers < atoms < non-atomic S-expresions
6470.                (2) numbers are in numerical order
6480.                (3) atoms are in alphabetic order
6490.                (4) If the CAR's of two S-expressions are
6500.                 not equal, their order determines the
6510.                 order of the S-expressions. If the CAR's
6520.                 are equal, then NEQUAL is applied recursively
6530.                 to the CDR's.
6540.                   (SORT
6550.                    @((1 . X) BA NIL -1 -.5 X (1)
6560.                      OAX (1 2)(A B C) AA 1 A B
6570.                      (A B) (1 A) (1 . A) )
6580.                    (FUNCTION(LAMBDA(X Y)
6590.                     (NOT(EQ(NEQUAL X Y)@ GRT)))))
```

```
6600.                    --------->
6610.                   (-1 -0.5 1 OAX A AA B BA NIL X
6620.                    (1 . A) (1) (1 . X) (1 2) (1 A)
6630.                    (A B) (A B C))
6640.
6650.  LEQ          LEQ takes two arguments; it is similar to NEQ except
6660.               that it returns the atom LSS if the first argument is less
6670.               than the second according to the above ordering, EQL if the
6680.               two arguments are equal, and NIL if the first argument is
6690.               greater than the first.
6700.
6710.  GEQ          like LEQ and NEQ, but returns NIL if the first argument
6720.               is less than the second, EQL if they are equal and GRT if
6730.               the first is greater than the second.
6740.
6750.
6760.  APPREV            APPREV takes two arguments.
6770.                    (APPREV A B) is the same as
6780.                      (APPEND (REVERSE A) B)
6790.
6800.  PUTPROP           Putprop takes three arguments.
6810.                    (PUTPROP NAM VAL IND) puts the value
6820.                    VAL on the property list of NAM under
6830.                    the property IND.
6840.
6850.  DIVIDE           (DIVIDE A B)=(LIST(QUOTIENT A B)(REMAINDER A B))
6860.
6870.  PRINC            PRINC is like PRIN1 except that it
6880.                    accepts arbitrary S-expressions for
6890.                    its one argument. It puts them in the
6900.                    output buffer without ending the line.
6910.
6920.  ASSOC            ASSOC takes three arguments. It is similar
6930.                    to SASSOC in its workings. It searches
6940.                    through the second argument, a list, for
6950.                    an element whose CAR is EQUAL to the first
6960.                    argument. (SASSOC uses EQ instead of
6970.                    EQUAL). 5
6980.                      If such an element is found, the CDR is returned.
6990.                    Otherwise, the third argument is examined.
7000.                    The third argument may either be a function
7010.                    of no arguments or a value. If it is a function,
7020.                    it is evaluated. In any case, the third argument
7030.                    is then returned.
7040.                      (ASSOC @ A ((1 . 2)(A . B)) NIL)---> B
7050.                      (ASSOC @ 1 ((1.0 . 2)(A . B)) NIL)--> 2
7060.                      (ASSOC @ A ((1 . 2)(B A)) @ X)  --> X
7070.                      (ASSOC @ A @((B C)(C D))(FUNCTION
7080.                          (LAMBDA()(PRINT @ HELP)))) ---> HELP, as
7090.                    well as printing HELP.
7100.
7110.  FULLWORDP         FULLWORDP returns T if its argument is a fullcell.
7120.                     (COND((FULLWORDP X)NIL)(T (CAR X))).
7130.
7140.
7150.
7160.  LENGTHEXPLODE          takes as its argument any literal atom or
7170.                         S-expression made only of literal atoms and
7180.                         returns the number of columns it would take to
7190.                         print that S-expression
```

```
7200.
7210.  ABS          ABS takes a number and returns the absolute value of that number.
7220.
7230.  *AND, *OR, *PLUS, *TIMES, *MAX, and *MIN
7240.
```

ABS ABS takes a number and returns the absolute value of that number.

*AND, *OR, *PLUS, *TIMES, *MAX, and *MIN
each of these functions take one argument, a list;where AND takes and indefinate number of arguments, *AND takes a list of the arguments of AND. For example
(SETQ X (*PLUS (QUOTE (1 2 3 4 5)))) will set the value of X to 15.

SUBST15 SUBST as defined in LISP 1.5 manual

MEMQ takes two arguments, an atom and a list. Tests if the first arguement is EQ to any element of the second. Like MEMBER but uses EQ instead of EQUAL.

TTIMER TTIMER has no arguments; it returns the value of a millisecond timer which is decremented as the job is run.

**********(6) Additional features.**********

A. The ARRAY feature.
ARRAYs have been implemented in this system. To create an array, call the function ARRAY with the following arguments:

NAME the atomic name of the array

SIZE a list with one member for each dimension of the array; the dimension may be specified by a number or a dotted pair; if a dotted pair is given, the CAR is considered to be the lower bound and the CDR is the upper bound; if a single number is given, the lower bound is assumed to be 0.

TYPE Arrays can be of the following types:
FIX Fixed point numbers
FLOAT floating point numbers
LOGIC logical numbers
BYTE logical numbers only one byte long
ATOM literal atoms
LIST arbitrary s-expressions

INIT initial values to be stored in the array iin the form of a list of dotted pairs (<the nummber of times to repeat this value> . <initial value>).

ARRAY stores the arrays in BPS and creates two LAP functions; a functions whose name is NAME is created and it, when it is called with the subscripts as arguments returns the value of that position in the array; another function whose name is an asterisk prefixed to NAME is used, wich takes the subscripts as arguments and returns the address where the information is stored.
Example:
ARRAY (A ((1 . 2) 3) LIST NIL)
creates a function A of two arguments; the two arguments must be numbers and (A 1 1) returns the value of the 1,1 cell of A;(A 2 3) returns the value of the 2,3 cell of A. (*A 2 3) returns the address of the 2,3 cell of A.

STOREVAL, STORECHAR, STOREATOM, and STORELIST are used for storing into the respective types of arrays (FIX,LOGIC and FLOAT; BYTE; ATOM; LIST).

With the ARRAY feature, SETQ is defined as a macro which checks if its first argument is an array and if so, expands to the appropriate STORE function.
EXAMPLE:

```
7800.       ARRAY (TMP (5 5) LIST NIL)
7810.       DEFINE(((FILLTMP(LAMBDA()
7820.                   (FOR I :=(1 5)
7830.                     FOR J :=(1 5)
7840.                      DO
7850.                      (SETQ (TMP I J) (TIMES I J)))))))
7860.     is the way to store into an array.
7870.
7880.
7890.
7900.
7910.
7920.      B.  The Timing Feature
7930.     TIMETOT, GETIME,  RETIME:
7940.
7950.          These are functions for localizing those parts of your program which
7960.     take the most time.
7970.        They all take a list of function names as their one argument.
7980.
7990.           TIMETOT redefines the functions given it in such a way that they
8000.     keep track of the amount of time from when they are entered to when
8010.     they are exited;  as well as the number of CONS'es performed between and
8020.     the number of times the functions are called.
8030.           TIMETOT  uses LAP and so must be called before LAP is removed;
8040.     it requires 73 words of Binary Program Space per function timed.
8050.
8060.           GETIME is called at the end of a run, and it prints out a summary
8070.     of the information that TIMETOT directed the functions to accumulate.
8080.
8090.           RETIME reinitializes the counters for the functions given.
8100.
8110.      Example of use:
8120.        DEFINE(((TEST(LAMBDA(X)
8130.           (PROG NIL (F1 X)(F2 X)(F3 X)(F4 X)(F5 X)))))))
8140.     TIMETOT ((TEST F1 F2 F3 F4 F5))
8150.     CLEANUP ()
8160.     TEST((THIS IS A TEST))
8170.     GETIME ((TEST F1 F2 F3 F4 F5))
8180.
8190.
8200.
8210.      C.  Fixdefine
8220.      D.  Cleanup
8230.
```

```
 -- STANFORD HIGH SPEED BATCH MONITOR --
//J778 JOB 'J778,130,01,15,99','MASINTER                    JOB  423
***SERVICE LOG=NO CLASS=Q EXEC=S
```

```
 -- STANFORD HIGH SPEED BATCH MONITOR --
//J778 JOB 'J778,130,01,15,99','MASINTER                    '          JOB  423
***SERVICE LOG=NO CLASS=Q EXEC=5
***PRINT CHAIN=B COPIES=10
// EXEC PGM=WYLIST
//SYSPRINT DD SYSOUT=A
//SYSPUNCH DD SYSOUT=B
//SYSUDUMP DD SYSOUT=A
//SYSIN DD DATA
//
          NORMAL JOB END


 RALPH JOB STATISTICS --    904 CARDS READ --      836 LINES PRINTED --      0 CARDS PUNCHED --   0.02 MINUTES EXECUTION TIME
```

LIST 02/19/72 13:41:43 J778.SYSTEM DOC on SYS05

<MASINTER> TOTAL @ USC

RECORDS FOR MACRO

102721861