MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

Artificial Intelligence
Memo No. 190

March 1970

An Interim LISP User's Guide

John L. White

AN APOLOGY

The substance of this memo is to initiate the naïve LISP user into the intricacies of the system at the Project MAC A.I. Lab. It is composed, at this time, of a Progress Report on the development of the LISP system and a few appendices but as such should be nearly adequate to start out a person who understands the basic ideas of LISP, and has understood a minimal part of the LISP 1.5 Programmers manual or a maximal part of Clark Weissman's LISP 1.5 Primer. At some undetermined time in the future a comprehensive document will be issued, consisting of an elementary introduction to LISP, a self-primer, the core of this document, and numerous reference appendices. The comprehensive guide will then replace A.I. memos numbers 116A, 152, 157, the LISP Progress Report, this memo and all informal notes and communications.

In the meantime, in order to inure the current user to the shock of an information blackout, and in order to give him a glimmer of what it is that he doesn't know about, the following list of Appendix titles if offered:

A: The True Meaning of Top-level Global Variables in MACLISP
B: How to Speak to the LISP Allocator, When Initially Allocating the Size of Storage Areas
C: Syntax for Use with STATUS and SSTATUS
D: All About TRACE and BREAK
E: The EDIT Feature in LISP
F: Using GRIND and INDEX
G: Setting up Displays on the CRT 340
H: Preparations for Compiling
I: Coding in LAP
J: Moby I/O Devices usable in LISP; The Vidissector, the Clock, the Calcomp Plotter, A/D and D/A Converters
K: The PIC-PAC Package for Storing and Using Vidissector Pictures
L: An Annotated Index of Functions, Facilities and Terms

Preliminary versions of some of these appendices are attached to the back of this memo, along with a very temporary Appendix X which attempts to update those changes which are missing or at variance with the main body of this memo or with memo 116A. Except for such variances, memo 116A is still recommended as an annotated index of functions and terms.

CONTENTS

Table of contents i

Notation ii

Introduction v

Refinements and Restrictions of the LISP Interpreter 1

Extended Interpretation of Familiar Functions 5

New Functions Added and Limitations on Familiar Facilities 13

I/O Channels: READ and PRINT 23

The Compiler, LAP, and Auxilliary Aids 34

Future Plans 39

NOTATION

The following items of notation are observed in the writing of this report:

ITEM The double-quote character ["] is used as the standard meta-linguistic word or string quoting device. Alas, in several instances it is used in the more vulgar sense of indicating a non-standard or insecure meaning imputed to a word or phrase. Unfortunately, the double-quote character is itself part of the ASCII alphabet that makes up the LISP input character set (in category (1) of the READ syntax, see page 25), and in order to avoid ambiguity, the examples of s-expressions on the succeeding pages will not utilize it.

ITEM "LISP1.5PM" is an abbreviation for the LISP 1.5 Programmer's Manual, published by the M.I.T. Press.

ITEM "MACLISP" refers to the PDP/6 implementation of the programming language LISP in use at the Artificial Intelligence Group of Project MAC.

ITEM Capital letters of the English alphabet are used as atom constituents in sample s-expressions, and small letters are used as meta-linguistic variables whose range must be deduced from context. Thus "(EVAL s)" could stand for "(EVAL ABC)", or "(EVAL (CONS X (LIST Y)))" and so on. Sometimes it is clear that the range of the variable is restricted to atoms, or to lists, but occasionally the variable may be a list fragment: e.g., "(LIST frag)" could be "(LIST A)", or "(LIST A 21)", or "(LIST X Y (CAR Z))".

ITEM Many sample programs and functions are given in the M-language as used in LISP1.5PM, and in some instances suggestive names are used for variables assumed to be permenantly set to some character object. This is for emphasis, since no reference to an input READ syntax, or to a universal evaluator is necessary for interpretation of the meaning of the program. S-expressions given as examples can only have functional meaning, or computational meaning, when the means of application is specified; that is, they induce functions when paired with a universal evaluator such as EVAL. Since various implementations differ in the action of EVAL, the one described in LISP1.5PM will be considered the common denominator; the one employed by MACLISP, however, will be the arbiter for interpretation in this report. Some examples are best given as s-expressions because of the extended role that F-type functions (FSUBR, FEXPR) play for EVAL, and because of the unity of function type within the M-language.

ITEM Normal input mode for MACLISP is base eight, and hence all numbers appearing in this report are to be understood as represented in base eight. Exceptions: the following constructions indicate base ten - page number references, English-described numbers (as opposed to numeral-described), numbers followed by a decimal-point, numbers having non-octal digits, and the phrase "36-bit" when referring to machine cell capacity.

ITEM There are two print sets accepted for the ASCII alphabet, variously called "old ASCII" and "new ASCII". Generally this report attempts to use or simulate the "old" alphabet of the Model 35 teletype, rather than the "new" alphabet of the Model 37. There are

only a few characters that are really different, a notable example being character number 136 which prints an an up-arrow in the "old" and as a carat in the "new".

ITEM Included in the ASCII character set are non-printing characters obtained by depressing a printing character while holding down the control key; if this is done with "H", for example, the character is called control-H and is denoted by "↑H".

INTRODUCTION

MACLISP is a descendent of the first operational program ever written for the Digital Equipment Corporation PDP/6 computer. It was an interpreter and minimal LISP system, written and debugged in the spring of 1964 by the staff at DEC and members of the Tech Model Railroad Club - an organization which along with the M.I.T. student radio station WTBS has supplied a continuing stream of prospective programmers to the Artificial Intelligence Group. This unlikely feat was accomplished with the aid of a well-equipped PDP/4 for editing, assembling and punching the program on paper tape. At about that time a working PDP/6 was delivered to Project MAC, and the LISP program was an aid in testing the new machine.

Ideas from several other implementations influenced the initial design, notably the CTSS version on the IBM 7094, and the very minimal version for the PDP/1. However, the decision to dispense with the a-list in the implementation, a major factor in the space economy and running speed of MACLISP, came some time later. An improved compiler was written as an adjunct to the system - compilation is done "off-line", and the resulting LAP code loaded into the system when desired. Although the compiler's output is both space-efficient and time-saving, it is the central focus of attention today for improved schemes such as "fast arithmetic", in the hope that many reasonable computations programmed in LISP may run in times comparable to their FORTRAN counterparts.

Indeed, because of the prominence of LISP in Artificial

intelligence applications, there is no other higher-level language used in the A.I. Group, although an impressive array of utility systems contribute to the ease of using both LISP and machine language: TECO, a text editor with display scope, for creating and servicing paper tape, magnetic tape, and magnetic disc files (no punched-card equipment is available); MIDAS, a machine language assembler with superb macro-generating features; DDT, a monitor-like system very helpful in debugging (and extended in the time-sharing version to provide other service functions, such as binary dumping and loading); and ITS, the Incompatible Time-sharing System. The original style of MACLISP was console oriented - i.e., on-line and interactive - with READ input and control commands accepted from an on-line teletype. With little modification, the same structure adapted itself well to the time-sharing framework, namely a job under the ITS system controlling one of the many teletype-like consoles available as remote terminals of the PDP/6 (currently there are four Model 35 teletypes, four General Electric remote keyboards with character-scope displays, one ARDS console, and three telephone data-sets for automatic connection to other Model 35 and Model 37 teletypes). Initially, when the job is started, MACLISP requests and accepts commands for allocating its memory usage - how much core is to be used in total, how much of it is to be used for full-word space, how much for the system push down stack, and so on. From that time on, only a few control characters have an immediate command effect; the main running of the job is controlled by a "top-level" function (see page 10).

Refinements and Restrictions of the LISP Interpreter

There is no explicit a-list for EVAL. All variable bindings occur as SPECIAL values, and EVAL does not search an a-list for variable bindings, but searches the p-list (property list) of the variable to be EVALed, in order to find the address of its SPECIAL value cell. On the whole, this implementation is considerably faster running since atom evaluation is quick and not dependent on the length of the a-list (which increases, of course, with increasing depth of function calling). MACLISP, however, cannot dispense with the binding and unbinding of variables, and a special stack is used to hold the information for unbinding after lambda conversion (also, for unbinding after PROG evaluation). Thus those variables whose values are changed by lambda or prog binding must be restored after the evaluation is completed. The stack of restoration information is called The Special PDL (PDL for push down list) and is a weak equivalent of the a-list in other systems. By means indicated below, a program may obtain a pointer into the stack area which will serve as an a-list to give as an argument to subsequent explicitly called functions, but which will no longer have meaning if returned as a value of the function which created it. In short, such an a-list will be useful at deeper levels in the computation, but not at higher levels. The problem is, quite simply, that this weak a-list is only a linear stack and does not have the tree-like structure necessary in general. Needless to say, there is no need for COMMON variables, since the interpreter has access to the SPECIAL cells, and there are no APVALS, nor functions CSET or

CSETQ. In order to conserve space, a VALUE property pointing to the atom's SPECIAL cell is not placed on its p-list until it is first bound, either through lambda or prog binding or through use of SET.

The interpreter admits three conventions for passing arguments to machine language functions (some systems have many more - the Bolt, Beranek & Newman LISP has five or eight depending on how one counts). The name of each defined function carries on its p-list an indicator of which convention is used, paired with the address of the subroutine code. The three types are called SUBR, FSUBR, and LSUBR. The first two should be familiar enough to persons acquainted with LISP 1.5; the third may be [loosely] viewed as indicating a function generalized along the lines of the function LIST - there are a variable number of arguments and EVAL, when working on a form FF whose car is an LSUBR, obtains the arguments by successive evaluation of cadr[FF], caddr[FF], ... The arguments are placed on a stack, called the Regular PDL, and the number of arguments is itself also passed along to the subroutine. One may [again loosely] view FSUBR's as indicating functions generalized along the lines of the function QUOTE - there is exactly one argument, and when EVAL is working on a form FF whose car is an FSUBR, it simply passes along car[FF] as argument. EXPR's and FEXPR's may be viewed simply as SUBR's and FSUBR's written in LISP code rather than machine code. As in other implementations, if the LAMBDA expression paired with a FEXPR has two lambda variables instead of the expected one, then upon entry to the function the second variable is bound to a representation of the current a-list, which may subsequently be given as an argument to EVAL or APPLY. Indeed, a-list

manipulations are rare among the users of MACLISP and EVAL and APPLY always permit the omission of arguments or lambda variables pertaining to a-lists. Thus one generally writes

(EVAL EXPRESSION)

rather than

(EVAL EXPRESSION CURRENT-A-LIST)

and similarly one usually defines FEXPR's

(LAMBDA (L) (PRINT (CAR L)))

In fact, another important application of LSUBR's is for SUBR type routines which have, say, three arguments, the third of which is almost always given some standard value; thus the subroutine may be called with two arguments only, the third being supplied by default; yet the third may be supplied explicitly in the calling program when some non-standard value is desired. To obtain the L-type argument convention for an EXPR, no new indicator is used (such as one might expect LEXPR), but instead the lambda list is replaced by a non-NIL atom which upon entry is bound to the number of arguments; and in the code, the form

(ARG N)

is used to obtain the n'th argument.

The MACLISP interpreter has been slightly extended in the direction of computed functions. The indicator MACRO is recognized by EVAL as follows: when EVAL'ing FF with car[FF] being atomic and having a MACRO property, the function corresponding to the property is applied to FF yielding FF', and the whole EVAL cycle begins anew on FF'. Functional arguments and computed function descriptions are

discovered by EVAL in the following circumstance:

(1) car[FF] is not a lambda, label, nor funarg expression,
(2) car[FF] is not an atom with some appropriate function indicator on its p-list.

In such a case, FF is replaced by FF' = cons[eval[car[FF]]; cdr[FF]] and again the eval cycle begins anew on FF'. Appendix B of LISP1.5PM shows an implementation which would allow only EXPR's, SUBR's, and lambda, label and funarg expressions to occur in such circumstances. In fact, most systems have not generalized apply to accept an F-type function as argument. A somewhat arbitrary choice has been made for this generalization in MACLISP - the second argument to APPLY is passed along directly as the single argument to the F-form function - and although it has some applications, it has not been used extensively and is still considered open to change.

Extended Interpretation of Familiar Functions

Conditional expression structure is one of the first LISP concepts that one wants to generalize. Rather than try to emulate ALGOL patterns such as "IF", "THEN", and "ELSE", the following were additions made to the interpretation of COND's:

> (1) All COND's will produce a value; if the COND clauses are exhausted with none being selected, then NIL is the value to be returned
> (2) COND "pairs" are extended to COND n-tuples: if the first member of an n-tuple evaluates to non-NIL, then the remainder are evaluated in order and the value of the last one is the return value. One-tuples are permissible, in which case the one value, if non-NIL, is returned.

GO and RETURN may be composed to any depth in the scope of a PROG; evaluation of either one is very much like the appearance of an error during an error-set computation. However, it should be considered a mistake to execute a GO or RETURN which is not explicitly within the scope of a PROG, for such usage cannot be properly compiled. Similarly, it is not possole to "go" to a tag outside the immediately dominating PROG. Computed GO's are permitted; if the argument in a GO is not explicitly an atom, then it is evaluated and an attempt is made to "go" to the result. For example, if X is bound to A, then

```
(GO X)
```

will go to tag X, whereas

```
(GO (EVAL (QUOTE X)))
```

will go to tag A.

Using the LSUBR convention, many familiar functions of two

arguments have been extended to operate on a variable number of arguments, generally by repeated aplication of the function from left to right; APPEND, NCONC, LESSP, GREATERP, MAX, MIN, PLUS, TIMES, QUOTIENT, DIFFERENCE, MAPLIST, MAP, MAPCON, MAPCAR, MAPC, MAPCAN, BOOLE (see page 16 for explanation of BOOLE). The order of arguments to the MAP series of functionals is in conflict with that of LISP 1.5 - the first arg is a function GG of n arguments (n>0) and the remaining n arguments are lists which will be simultaneously mapped into the arguments of GG. The lists need not be of equal length; the process stops when the shortest is exhausted. MAPC, MAPCAR, and MAPCAN are just like MAP, MAPLIST, and MAPCON respectively except that CAR of each successive sublist is taken as argument to the supplied function, rather than the sublist directly.

PROG2 is implemented as an LSUBR with two or more arguments, and whose value is the second argument; evaluation , of course, still follows the regular order. Every lambda expression is also implicitly what in some other systems is called PROGN. Applying

(LAMBDA list e1 e2 . . . en)

will cause the evaluation of e1 to en, in order, returning the value of en as result.

LAST and MEMBER have been extended to provide slightly more useful values:

```
last[l] = [ null[l] v null[cdr[l]] → l
          T → last[cdr[l]]]

member[x; l] = [ null[l] → NIL
                 equal[x; car[l]] → l
                 T → member [x; cdr[l]]]
```

The new interpretation of LAST provides a fast way to find the end of a list rather than the last element of the list. Similarly, if MEMBER is to return a non-NIL value, it will be that tail of the original list whereat the member was found. A function MEMQ is implemented which is merely MEMBER using an EQ test rather than an EQUAL test as displayed above. Similarly, ASSQ is ASSOC with an EQ test (ASSOC uses an EQUAL test).

PRIN1 will print out one full S-expression (not necessarily atomic), and when printing a literal atom whose PNAME contains characters which are not syntactically legal for READ'ing back in, PRIN1 will print a slash before such illegal characters. Thus the atom with PNAME "A.B" will print out as the four characters "A", "/", "." , "B". READ, in turn, recognizes slash as a special character that in effect causes the next character to be treated syntactically as an alphabetic character. For example, the string "A/(B//C " would be read in as the literal atom with PNAME "A(B/C"; PRINC is a new function which will print out one S-expression, without inserting any slashes before READ-illegal characters. PRIN1 is important when one wishes to write S-expressions out on auxilliary memory and read them back in at a later time; PRINC is important when one wants to generate his own output format, or print a message which is stored as the PNAME of some atom.

The function PRINT is defined as

λ[[x]; prog[[]; terpri[]; prin1[x]; princ[space]; return[x]]]

The function TRACE is not built into EVAL, but is encoded in EXPR form. No part of it is normally resident in the system, but it must be read-in from an auxilliary file (in the Time Sharing environment, many such extra packages are stored on a disc pack called the COMMON device). A traced function has the function property on its p-list temporarily replaced by a standard form EXPR or FEXPR which handles the traceing work before actually applying the original function. The facility has been extended to work with every function (yes, even PRINT, COND, SETQ, etc.) except possibly GO and RETURN. A program switch exists to inhibit direct linking from compiled functions, so that even calls from compiled code may be traced (see page 21, STATUS). Provision has also been made for conditional tracing (at each call of the traced function, a predicate will be evaluated to determine whether or not to trace that call), and for conditional break-upon-entry (a BREAK calls a read-eval-print loop similar to the normal top level, but some input, usually $P, is reserved to signal the return from the BREAK and the continuation of computation).

The garbage collector, sometimes called the reclaimer, of MACLISP performs a few more functions than that of more standard implementations. On the simpler side, an internal switch (set by typing ↑D on the job console or by calling SSTATUS with an appropriate argument - see page 21) can cause the garbage collector to print out, on each collection, statistics telling why the collection was

initiated and how much space is available in each storage area. Arrays and compiled programs share the binary program space alloted by the MACLISP allocator, and while one seldom expects to exhaust his memory facilities with "dead" binary programs, some applications create and destroy arrays with alarming rapidity. The garbage collector dynamically handles the assignment and reclamation of space for arrays (calling the time sharing core allocator, when all else fails, to try to extend the job's memory allotment). Occasionally, the living arrays will be relocated and compactified toward the top end of BPORG space, but as yet no facility is available for dynamic relocation of compiled programs (see chapter on future plans, page 39). Occasionally, also, the gabage collector will decide that far too much BPORG space is sitting idle, and it will attempt to return some core memory to the time sharing system, although this feature may be disenabled with SSTATUS.

Although atomic objects read in by READ are placed in the OBLIST to protect them from inadvertent collection and to insure identification of future tokens of the same atom, some applications in natural language occasionally get bogged down with a bloated oblist in which not all atoms are of continuing utility, and the LISP programs themselves are not able to decide which atoms should be REMOB'ed. The concept of a Truly Worthless Atom is defined: a TWA is an atom with a trivial p-list (only a PNAME property) and which is part of no living list structure except the OBLIST. A feature of the garbage collector will reclaim any TWA, but such feature may be turned off or on under program control; because most applications do not need it, it is not

normally on (see page 21 on SSTATUS). Compiled code directly accesses the SPECIAL cells of free (or SPECIAL) variables, and no provision has been made for relocating such addresses when used in binary program space; thus this feature, chosen for speed considerations, is the major drawback to implementation of a compactifying (or free storage relocating) garbage collector, and is the sole reason that an atom which has a VALUE indicator on its p-list cannot be reclaimed as a TWA, even though the atom is no longer bound to any value (see page 2 for more on the VALUE indicator).

MACLISP is essentially a machine language implementation of the universal function EVAL along with some initial a-list (for "environment" since MACLISP has no true a-list) and with many other subroutines useful for reading, manipulating, and writing out S-expressions. The system comes to life through its "top level" function; that is, "top level" may be viewed as the continuous re-application in the top-level environment of some function of no arguments. For systems using EVALQUOTE, the top-level function is very much like

λ[[]; print[apply[read[]; read[]]]]

where READ reads in one S-expression from the current input channel. The normal top-level for MACLISP is equivalent to

λ[[]; errset[prog2[terpri[]; print[eval[read[]]]]; T]]

In the vernacular of programmers, one could say that EVAL is the top-level for MACLISP, rather than EVALQUOTE.

The programmer in MACLISP, however, has the capability to change the form given to ERRSET for evaluation. For example, in order to change to EVALQUOTE, one could replace the PROG2 on the preceeding page with, say,

prog2[terpri[]; print[apply[read[]; read[]]]]

(see discussion of the functions STATUS and SSTATUS on page 21). Three global variables, i.e. those with bindings at the top level, have special relevance to the top-level function:

(1) If the value of BAKGAG is not NIL, then a backtrace is printed out for any error which propogates all the way up to the top level function (errors which have been ERRSET do not pop back beyond the ERRSET in which they occur). The backtrace, essentially an inspection of the system PDL, will print out "fun1 -EVALARGS" to indicate that EVAL has at that point commenced the evaluation of arguments for the function fun1, and "fun1-ENTER" to indicate that function fun1 has been entered but not yet exited. Ocasionally, BAKGAG will print out "fun1-fun2" to indicate that fun2 has been entered by a call from fun1.
(2) If the value of *RSET is not NIL, then the phase of ERRSET which restores the binding of special variables is bypassed (and indeed all variables in MACLISP are special). This applies only to the top-level "ERRSET", but in many cases allows the user to inspect the environment at the time of the error.
(3) ERRLIST, normally set to NIL, is a list of forms to be EVAL'ed upon recovering from an error propogating to top level; this allows the user to supply a little of his own re-initialization for special purposes.

Both *RSET and BAKGAG are also function names such that EVAL'ing (*RSET T) is fully equivalent to EVAL'ing (SETQ *RSET T), and similarly for BAKGAG. Typing ↑G on the job console (or EVAL'ing (IOC G)) has special relevance for the top-level ERRSET - an error is created which no other ERRSET can catch, all I/O channel switches are returned to their initial state (see page 32), the top-level function is restored to the normal one for MACLISP, and the message "QUIT" is

printed on the console. Thus ↑G is an unconditional quit signal, similar to the ↑X quit signal described on page 10.

The function MACDMP provides a means of halting computation and returning control of the console to the monitoring program. Usually the monitor is the extended version of DEC's DDT which is loaded when the user logs in at a console, but occasionally it is ITS directly. In the former case, storage areas not active at the time of MACDMP'ing are zeroed so that a binary dump to auxilliary memory will take less space; in the latter case, it is assumed that the user has relinquished the job, and it is logged out.

New Functions Added and Limitations on Familiar Facilities

All property list [p-list] usage must be in paired form - FLAGS are not allowed. ATTRIB is not implemented (indeed, NCONC could be used to the same purpose), but the function PUTPROP is provided as a means of placing a value and corresponding indicator on an atom's p-list.

putprop[atom; value; indicator]

will insert "indicator" followed by "value" at the beginning of the p-list of "atom", unless there is already an indicator of the same name on the p-list, in which case the old value will be RPLACA'd out with the new. It is likely that the implementation of atomic structures will change, but the functions PUTPROP and GET will always be kept "up to date" (see chapter on future plans). The indicator VALUE, whose paired value points to the atom's SPECIAL value cell, and the indicator ARRAY, are treated specially by the p-list handling routines, and there may soon be a different, and faster, implementation for the three basic and important properties: pname, value, and function. Although GET serves as the standard p-list retrieval function, another function GETL is provided to solve the following problems:

(1) to distinguish between the non-occurrence of an indicator, and the retrieval of a NIL value,
(2) to find which indicator from among a list of indicators is the one which occurrs first (if any from the list occurr at all).

Letting "plist[a]" stand for the p-list of an atom "a", the implementation of GETL is very much like

```
getl[a; l] = [not[atom[a]] → error[];
              T → label[getl];
                  lambda[[x; l];
                      [null[x] → NIL;
                      memq[car[x]; l] → l;
                      T → getl[cdar[x]; l]]]]
                  [plist[a]; l]]
```

DEFPROP is essentially the FSUBR form of PUTPROP, and is generally used at top level to place EXPR, FEXPR, and MACRO properties on atoms; it has been the usual method of defining functions in MACLISP, but DEFUN has been implemented to provide a more clear indication that such a definition is being performed, and to allow smoother experimentation in the structure of p-list and function representation. Some example definitions:

```
(DEFPROP DOUBLE
         (LAMBDA (N) (PLUS N N))
         EXPR)
```

An optional type indicator to DEFUN (from among FEXPR, EXPR, or MACRO) is placed just after "DEFUN", the default option being "EXPR".

```
(DEFUN FEXPR
    FPUTPROP (X)
      (PUTPROP (CAR X) (CADR X) (CADDR X))
      (CAR X))
```

is equivalent to

```
(DEFPROP FPUTPROP
         (LAMBDA (X) (PUTPROP (CAR X) (CADR X) (CADDR X)) (CAR X))
         FEXPR)
```

The pseudo-function ARRAY serves to make array declarations, turning the name of the array into a finite function. An array may have from one to five dimensions [indices], and may be declared to hold list structure or, simply, direct data - the matter of importance is whether or not the garbage collector is to treat the array entries as pointing to living s-expressions. ARRAY is implemented as an

FSUBR, and the order of arguments is slightly different from that in LISP 1.5. Sample form for EVAL:

(ARRAY aname t-or-nil n1 n2 . . . nk)

where 0<k<6, and t-or-nil is the garbage collector switch. The name for the array is taken directly and not evaluated - the other arguments are evaluated. The array setting function, an FSUBR, works through a side-effect in the accessing function. EVAL'ing

(STORE (aname i1 i2 . . . ik) EE)

will store the value of EE in aname[i1; i2; . . . ik]. The MACLISP user has the ability to stretch or shrink the size of an array, even after it has been in use for some time. There are a few applications wherein the proper size of the array is not known until after is has been partially filled. It would be too wasteful to have to declare a new array and transfer the entries by means of LISP code, or to declare initially the array to be of monstrous size. The function RE*ARRAY is called just like ARRAY, and performs a stretching or shrinking in the size declaration (this usually involves a call to ARRAY and a fast, memory block transfer). In order to achieve something like lambda binding of arrays, and the consequent reclamation of their space upon unbinding, one often computes

.nofill

```
(EVAL (CONS
        (QUOTE ARRAY)
        (LIST
            (SETQ A (GENSYM))
            t-or-nil
            n1
            .
            .
            .
            nk)))
```

in order to use A like a functional argument, either with APPLY

(APPLY A (LIST i1 i2 . . . ik))

or, if A has no function indicators on its p-list, directly with EVAL

(A i1 i2 . . . ik)

(see page 4 on computed functions for EVAL) Any array space which becomes cut-off from living list structure, either through repeated application of ARRAY to the same name, or through use of NEWARRAY (which may re-specify size zero to remove an array), or through lambda unbinding, will eventually be reclaimed by the system.

Some functions have not been implemented because their purpose has been fulfilled by other features. For example, BOOLE is very much like a functional - its first argument selects one of the sixteen possible boolean functions of two variables, which is then applied to the remaining arguments as a 36-bit logical bitwise operation - and thus there is no need for LOGOR, LOGAND, or LOGXOR. Similarly the excellent debugging features of the monitor-like system DDT, and the other features of ITS make the function of DUMP unnecessary. In place of TEMPUS-FUGIT, the MACLISP user may read a real time internal clock (accurate to about 5 microseconds), and may read a run-time clock to measure C.P.U. running time used by the job which read it (accuracy depends on changing conditions within ITS, but at best is about 50 microseconds). COPY and CP1 do not exist - one may obtain a copy of an s-expression s, down to atomic level, by

subst[NIL; NIL; s]

and a copy of the top level of a list l by

append[l; NIL]

SUBLIS has been implemented very much as shown for SHARE LISP on page 96 of LISP1.5PM, so that no unnecessary CONS'ing is done; however it uses an EQ test and requires that all the dotted pairs of the first argument be of the form (U . V) where U is an atom (this last restriction allows a clever and speedy implementation). Since there is no a-list, and consequently only SPECIAL values, there is no need for APVALS, nor functions CSET or CSETQ. The function FUNCTION is functionally equivalent to QUOTE, but if the MACLISP user needs a FUNARG binding with an a-list of the limited form described on page 1, he will obtain it using the function *FUNCTION. Other functions have not been implemented at all, and as yet there has been only one incident in which a user desired any of them: CONC, SEARCH, ONEP, PAIR, RECIP, EXPT, LITER, SELECTQ, SPEAK, COUNT, UNCOUNT. None of the functions of the compiler or of LAP are normally resident in the system - see page 34 about their usage.

ERRSET is the MACLISP equivalent of ERRORSET, and since there is no a-list nor any means to trap out on excessive consing, there are no arguments passed along for these two facilities. ERRSET is implemented as an FSUBR and EVAL'ing

(ERRSET EE i)

will return list[eval[EE]] if no errors are encountered during the evaluation of EE; otherwise at the occurance of an error, computation will be interrupted, the state of the system restored to that just before beginning the evaluation of the ERRSET, and an appropriate message printed out if eval[i] is non-NIL. For all errors generated by the system, ERRSET returns the value NIL, but if the function ER

is called during the evaluation of EE, then a harmless error is created (with no corresponding message) and the argument to ERR is returned as the value of the ERRSET. Typing ↑X on the console generates an error with the message "QUIT". Generally, one does not wish to suppress error messages, so an alternate short form is provided:

(ERRSET EE)

evaluates just like

(ERRSET EE T)

In view of the implementation of Input/Ouput for MACLISP (see pages 23,24), it is hardly necessary to mention that none of the functions in Appendix F of the LISP 1.5 Programmer's Manual are needed; and furthermore there is no initial set of character objects, since there is no difficulty in reading in objects with bizarre names using the slash-sign convention for "quoting" characters in the input string (see page 27). Although the $$ convention for reading strings as atoms is not directly implemented, it may be simulated through the device of a READMACRO character (see Example 4, page 30). For really oddball atoms, the string-quoting convention saves the effort of having to "quote" every non-alphabetic character in the name. The functions EXPLODE and EXPLODEC provide the means of breaking up an atom into a list of the characters of its PNAME, using respectively the conventions of PRIN1 and PRINC (see page 7); FLATSIZE and FLATC are equivalent to \[[x];length[explode[x]]] and to \[[x];length[explodec[x]]] respectively. READLIST is the means to apply READ to a list of characters rather than to the string of the

current input channel. MAKNAM also takes as argument a list of characters, and creates an atomic object whose PNAME is composed from those characters; this new object is not INTERN'd.

DELETE, a new function, is provided as a way to splice out an element from a list (in effect using RPLACD); the surgically trimmed list is returned, which will be EQ to the original list unless the element removed was at the first of the list. DELETE is implemented as an LSUBR so that it may be applied with two arguments, or with three arguments - the optional third argument being an upper limit on the number of deletions to be made. The default option for the third argument is "delete all occurrences". DELETE uses an EQUAL test on each element of the list - DELQ is provided to operate just like DELETE except that an EQ test is used. Ordinarily, one writes (SETQ FOO (DELQ X FOO)).

In conjunction with the PROG feature, a fast, machine language DO is provided, whose action is most accurately described by the macro-produced expression used by the compiler in place of the DO expression:

(DO index initval stepfun donep . . .)

expands into

```
(PROG (index)
      (SETQ index initval)
 tag  (COND (donep (RETURN index)))
                .
                .
      (SETQ index stepfun)
      (GO tag))
```

The compilation of a DO actually results in the compilation of its corresponding PROG; using DO with the interpreter however, reduces the

overhead of index manipulations and is much more mnemonic. Note that the fragment indicated above by ". . ." may include prog tags, GO's, RETURN's, etc. Example:

```
(DO I 0 (ADD1 I) (GREATERP I N)
    (COND ((NOT(NUMBERP (ARR I))) (RETURN (QUOTE FOO))))
    (SETQ SUM (PLUS SUM (ARR I))))
```

There are specialized I/O functions to access and utilize the various peripheral devices attached to the PDP6/10: a DEC 340 display scope with character generating features, a light pen input from the display scope, a CalComp plotter with an interpretive subroutine package (which accepts a wide variety of commands including the plotting of a display from the DEC 340), analog-to digital and digital-to-analog converters for use with robotics devices and with a few other random devices, a video signal processor attachable to either a modified III image dissector or an IIT image dissector (an image dissector, or "vidissector", is like a TV camera except that the scan is random-access addressable rather than continuous and sychronized.) There are no card reading or punching facilities, but all the external memory devices available through ITS are available: a paper tape reader and punch, four DEC magnetic "micro" tapes, three IBM model 2311 magnetic disc units, and a Data Products line printer. There is also provided a software simulated vidissector, which can read stored images from auxilliary memory, so that pattern recognition programs may be tested regardless of the physical condition of the two real vidissectors.

The function STATUS has been installed as a centralized means of querying the status of many system variables and conditions; given an appropriate argument, it provides: the limits allocated for the various storage areas in memory, the amount of core occupied by the job, the amount of storage still free in a given area (currently implemented only for "free storage", "full word space", and "binary program space"), the activity of the various I/O channels, the current input channel for READ, the syntactic category of the 200 ASCII characters used for input to READ, switches for various garbage collector features, switch to allow reading of numbers with non-decimal digits (hexadecimal, for example), the switch to inhibit decimal-point print when printout is in base ten, the switch to inhibit direct linking from compiled code, the switch to inhibit address checking on array accessing, and the form of the top level function. The function SSTATUS provides a means of setting all these variables or switches, except for the allocated limits of storage areas. A number of system variables, including some of those mentioned above, are implemented as the special value cell of an atom, and are thus referred to as a "global" variable. There will be an appendix (APPENDIX C)which will explain more of the details of using STATUS and SSTATUS.

COMMENT is a do-nothing function, FSUBR type, which merely returns "COMMENT". DECLARE, also an FSUBR, is similar, except that the compiler takes notice of DECLARE's. BREAK operates as described on page 8 and is called like "(BREAK FOO s)" - if the value of s is non-NIL, then "FOO" is printed out and the breakloop entered.

I/O Channels: READ and PRINT

The major input function for any LISP system is READ, which inputs characters, one at a time, from the current input channel, interprets the characters as being the linear-parenthesized representation of an s-expression (or list), and constructs the s-expression in the free storage area of the system. When a sub-section of the input string is parsed as an atom token (for example, a string of alphabetic characters followed by a space), the function INTERN is called to see whether or not an object of the same PNAME is already in the OBLIST; if so, no new structure is constructed by READ, but instead the atom token is identified with the atomic object already there. Otherwise, a new object has to be created, with PNAME composed of the inputted characters of the atom token, and the resultant placed in the OBLIST. Rather than the card-oriented functions STARTREAD and ADVANCE, MACLISP has the new fucntion READCH which inputs exactly one character from the current input channel and returns the atom whose PNAME is just that One character (called in LISP1.5PM a "character object"); as with READ, the function INTERN is invoked. The new function TYI also inputs exactly one character, but returns the numerical atom whose value is the ASCII value of the inputted character.

There is a unique input channel for READ, READCH, and TYI whose normal, or default, source is the job console. By typing ↑Q on the console, or by calling SSTATUS with an appropriate argument, the

input channel source will be switched to an auxilliary memory file previously selected by the function UREAD (it is, of course, an error to try to change sources when no file has been selected). The argument to UREAD, an FSUBR, specifys the device and two file names, but if no argument is supplied, certain convenient default options will be exercised. As soon as the command is given, READ (or READCH or TYI) begins inputting characters from the selected file, rather than the job console; the job console is re-selected as source whenever ^S is typed on it, or SSTATUS is called with appropriate argument, or when the end-of-file character is encountered in the selected file.

Probably, the most important use of the auxilliary memory as an input source is to augment, massively and rapidly, the store of named LISP functions. Instead of typing many DEFUN's or DEFPROP's at the console, the user simply stores a sequence of such forms on a file, selects the file with UREAD, and gives the ;Q command. The top level function is still running, so under normal circumstances, the forms are read in, EVAL'ed, and printed out (unless all output channels have been disenabled). When all is done, i.e. the end-of-file character is encountered, the job console is re-selected. The E-O-F charater is not actually used by READ or READCH or TYI, but merely serves as a signal to restore the console as input source.

Of course, a selected file may serve as a data source for a program which uses READ. The following sample program reads in a sequence of numbers, terminated by a NIL, and returns their sum.

```
(LAMBDA NIL
    (PROG (SUM X)
        (SETQ SUM 0)
        (UREAD LOTSA NUMBERS)
        (SSTATUS IOC Q)
      A (COND ((NULL (SETQ X (READ)))
               (SSTATUS IOC S)
               (RETURN SUM))
              ((NOT (NUMBERP X))
               (ERR (QUOTE (NON-NUMERIC IN FILE)))))
        (SETQ SUM (PLUS SUM S))
        (GO A)))
```

To illustrate some other features, the example is re-written noting (1) the switch for input channel source is actually the value of the global variable ↑Q, so in this particular case, the IOC (for I/O control) commands may be replaced by a SETQ or lambda binding; more importantly, the status may be pushed and popped by lambda or prog binding; (2) if READ, implemented as an LSUBR, is called with one argument, the E-O-F character not only causes the console to be re-selected as noted above, but also causes READ to return its one argument as a value; if the E-O-F is not encountered while reading the current s-expression from the file, then READ operates normally; thus a read mode is available which will actually read an E-O-F and return some given quantity; (3) The name of the file is made an argument for the sample function and APPLY is used with UREAD (instead of constructing an appropriate form to give to EVAL, a necessary task if APPLY is not capable of applying an F-type function). The revised example does not require a NIL to terminate the numbers on the file, since the E-O-F causes read to return an unlikely quantity.

```
(LAMBDA (FIL)
    (PROG (SUM X ↑Q)
        (SETQ SUM 0)
        (APPLY (QUOTE UREAD) FIL)
        (SETQ ↑Q T)
    A   (COND ((EQ (SETQ X (READ (QUOTE FOOEY-E-O-F)))
                   (QUOTE FOOEY-E-O-F))
               (RETURN SUM))
              ((NOT (NUMBERP X))
               (ERR (QUOTE NON-NUMERIC IN FILE)))))
        (SETQ SUM (PLUS SUM X))
        (GO A)))
```

The syntax used by READ to parse the input stream of characters into an s-expression is almost identical to that described on pages 3 and 24 of LISP 1.5PM; a few extensions, one of them very powerful, are noted below. Except for strings representing numerical objects (on of the more complicated tasks for READ), the syntax is very character oriented and context-independent. Thus the characters meaningful to READ fall into nine categories:

(1) alphabetic, i.e. "A", "B", etc., and extended alphabetic, e.g. "!", "↑", "$", "+", "-", etc.
(2) decimal digits, i.e. "1", "2", "3", . . . "9", "0"
(3) context-dependent number modifiers, i.e.
  (3a) "↑" for fixed point scale factor indicator,
  (3b) "E" for floating point exponent indicator,
  (3c) "←" for a left-shift in base 2 notation,
  (3d) "+" and "-" for algebraic sign,
  (3e) "." for decimal point;
(4) open parens, i.e. "("
(5) dot, i.e. "."
(6) close parens, i.e. ")"
(7) rubout (ASCII value 177)
(8) the literal-character-quoting character, i.e. "/"
(9) macro characters, specifiable under program control.

Except when preceeded by a slash, all other characters encountered by READ are simply ignored and discarded. Not surprisingly, the context-dependent characters of category (3) are also included in other categories as well.

The slash is the standard literal-character-quoting character, which means it is used to incorporate non-alphabetic characters into the print-names of literal atoms, and any ASCII character except rubout which follows slash in the incoming character stream will be at that point treated as if it were of category (1) (see bottom half of page 7 for examples and more details). Actually, any maximal string of composed of characters from categories (1) and (2) will normally be parsed as a literal atom token unless it consists only of digits, in which case, of course, it is parsed as a number atom token. Thus, "1A" is a perfectly good literal atom token for MACLISP, and incidentally, so is "/12" since the first character will be inputted as category (1). There is no limit of 32 characters for atom PNAMEs, as suggested in LISP1.5PM. A feature is provided in which any string composed of letters and digits and preceeded by a "+" or "-" is taken as a number token, using "A", "B", "C", . . . as supra-decimal digits. The feature is not normally turned on, and may be switched with SSTATUS.

When inputting from an auxilliary memory file, the rubout character is ignored, but when inputting from the job console, the rubout character causes the preceeding character to be "rubbed-out", i.e. it is removed from the buffered input string, and immediately printed back again on the console to announce its removal. No special character, or sequence of character is necessary to signal the end of an s-expression since it is determined wholly by syntax. Needless to say, when one s-expression's worth of characters are fuly inputted, typing a rubout will have no effect on them, for the READ'ing will

have been accomplished.

The syntactic catefory of any character may be inspected with STATUS, and may actually be altered with SSTATUS (see page 21). Thus one could re-specify "[" to be in category (4), or "]" to be in (6), instead of in (1). (The unfortunate user could even send himself up the READ stream without a parens by shutting off the category (4) meaning for "("). Syntax information is stored in a 200-word table, along with a PNAME translation; the latter feature finds its use primarily in converting "a" into "A", "b" into "B", etc. so that one need not continually hold down the shift key when using a model 37 teletype. (Model 35 teletypes and GE consoles have only 141 characters on their keyboard - Model 37 teletypes have all 200 characters of the new ASCII alphabet). However, "/a" will still be inputted as "a". (Note that all syntactic and translation properties attributed to characters only apply to the operations of READ. READCH and TYI, explained on page 23, directly input one character.)

The READMACRO character idea is a means of using LISP functions to generate s-expressions at READ time; quite simply, wherever a macro charcter appears in an input stream, it is replaced by the result of calling an associated function. A macro character is declared and its corresponding function specified with SSTATUS, and each such macro is further classified into one of four types (probably best understood from the examples below): (1) elemental, immediate; (2) elemental, delayed; (3) splicing, immediate; and (4) splicing, delayed. The first dimension indicates whether the result of the associated function is to be treated as a single element, or whether

it should be a list which is appended, or spliced, into a forming list (i.e. it is a list fragment). The second dimension indicates the "time" at which the associated function should be called - since READ may be called [recursively] by the associated function, it is important to specify whether the recursively-called READ takes its input as those characters immediately following the occurrance of the macro character, or whether it waits and uses those characters coming in after the first-initiated READ has inputted its share. Functions associated with immediate macros have no arguments, those with delayed macros have one argument, which is bound to the current product of READ.

Example 1. In order to use "'s" in the input stream instead of "(QUOTE s)", the character single-quote is declared as elemental, immediate macro with function

λ[[]; list [QUOTE; read[]]]

Thus when READ'ing "(CONS 1 '(FOO ON YOU))", as soon as the single-quote is encountered, READ is called recursively to obtain (FOO ON YOU) which is then combined in the call to list, and the final result is the same as if

(CONS 1 (QUOTE .(FOO ON YOU)))

were the expression to be read in. This facility with single-quote has been deemed so meritorious that MACLISP contains the definition as part of its initial structure - one would have to remove "'" from the macro character category, again using SSTATUS, if he did not desire its use.

Example 2. Using the same function as in example 1, let "!" be a

splicing, immediate macro. then "(FEVAL 1 !(FOO ON YOU))" is read in as

(FEVAL 1 QUOTE (FOO ON YOU)).

Example 3. Let "<" be a splicing, delayed macro with function

```
λ[[l];
    label[foo;
          λ[[x]; [eq[x; rightanglebracket] → NIL;
                  T → cons[x; foo[read[]]]]]
     ]read[]]
```

The following english description of the action of this function is offered to the the reader who has not already adduced it: a maximal sequence of s-expressions terminated by a ">" is read and a list of them, in the order in which they were read, is returned. With "<" so embellished, the DO example on page 20 could be rewritten

```
(DO I Ø (ADD1 I) (GREATERP I N) <)
(COND ((NOT(NUMBERP (ARR I))) (RETURN (QUOTE FOO))))
(SETQ SUM (PLUS SUM (ARR I)))
>
```

Example 4. Although an elemental, immediate type example has already been given, this one is produced to show how the literal atom string feature described at the top of page 84 of LISP 1.5PM can be simulated in MACLISP. Let "$" be a macro with function

```
λ[[];
    prog[[endch; x; l];
          readch[];
          endch := readch[];
      A   x := readch[];
          [eq[x; endch] → return[readlist[l]]];
          l := append[l; list[slash; x]];
          go[A]]]
```

The syntax for number tokens is similar to that described in LISP1.5PM, but

(1) the input base for integers is not necessarily ten; it is found as the value of the global variable IBASE, which is initially set to eight.

(2) "Q" does not serve as an octal indicator; rather "." serves as a decimal indicator; e.g., the string "60. ", illegal in LISP 1.5, is taken as an integer base ten regardless of the value of IBASE.

(3) "↑" and "←" act like operators on two integer tokens: "a↑b" results in number a multiplied by IBASE b times; "a←b" results in number a multiplied by 2 b times. Assuming IBASE is eight, then, "4096. ", "1←12. ", "10↑3", and "64. ↑2" all denote the same numerical value.

(4) It is permissible to begin a floating-point number token with a decimal point; however one must remember the advice to place spaces around a dot which is intended as a separator for a dotted pair, but which occurrs adjacent to a digit.

Internally, numerical objects are 36-bit quantities with one of two possible indicators - FIXNUM or FLONUM. Thus all record is lost of the form of the token in the input stream which gave rise to the number, and it is possible that a printout of the object will not yield the same string that caused its generation.

For PRINT, PRIN1, PRINC, and TYO output there are four independently enabled channels, each with its own switch in the form of a global variable:

(1) the job console; non-NIL value for ↑W means disenabled
(2) the Data Products line printer; non-NIL for ↑B means enabled
(3) the DEC 340 display scope, buffered for character output by ITS; non-NIL for ↑N means enabled
(4) an auxilliary memory device or remote console, previously selected by the function UWRITE; non-NIL value for ↑R means enabled.

Since all four switches are varibale values, their state may be pushed and popped, along with that of the state of the current input channel (see pages 23, 25), by lambda or prog binding. The variable's names derive from the early implementation of I/O control: typing ⌜Q, ⌜W, ⌜B, ⌜N, or ⌜R respectively set ↑Q, ↑W, ↑B, ↑N , and ↑R to non-NIL, and typing ⌜B, ⌜V, ⌜E, ⌜Y, or ⌜T sets the respective switch to NIL. Although these five switches may be changed with SETQ, the function IOC (and SSTATUS) is provided to perform, by evaluation, the same action that occurs when a control character is typed on the job console. An output channel selected by UWRITE, and possibly written on after enablement, is then closed and filed with the function UFILE - the file structure of ITS does not permit any other access to the output file until it has been properly closed and filed. Optional arguments to UFILE are file names, with certain default options, which will become the two names identifying the file on an auxilliary memory device.

When PRIN1 or PRINC is outputting the characters for a numerical object, the base representation is not necessarily ten, but the value of the global variable BASE, and if the base is ten, then

each integer-output-sting is followed by a decimal point, for proper read-in under the convention (2) described on page 3b. However, this decimal-point-printout can be inhibited by setting the global variable *NOPOINT to non-NIL. No other formatting features yet exist in MACLISP. It is possible to output in base representations larger than ten (and less than thirty-seven); but there is still some ambiguity about this feature.

The Compiler, LAP, and Auxilliary Aids

Many functions, because of their infrequency of use, are not part of the basic MACLISP system, but may be loaded in and used when so desired. The LISP compiler and LAP assembler are two such utility routines. Both are written in LISP code, as EXPR's and FEXPR's, but the compiler is so large that the initial PDP/6 configuration had just about enough core to hold a system with a compiler loaded in and leave some free storage for smooth running. As a result, compiling is usually done only in an augmented LISP system embellished with a compiled version of the compiler. The compiled EXPR's and FEXPR's occupy about 11,000 (4700.) cells and run about ten times faster than their LISP counterparts (for example, with similar amounts of working space, compiling the compiler takes 31. minutes using interpreted code,, but only 3.1 minutes using compiled code). In addition, compiled code requires on the average only about half as many cells as does the LISP code giving rise to it, and does not require the time-consuming marking phase during garbage collection which is necessary to protect the cells holding LISP code. Although the compiler may be applied to functions defined or read into the embellished system, it is generally used as a file transducer - i.e., s-expressions are read in, one at a time to prevent free storage strain, from a selected auxilliary memory file, and those which represent function definitions, such as (DEFUN . . .) or (DEFPROP . . .) , are compiled into corresponding LAP code on an output file. For each function compiled, the generated LAP output is a sequence of

s-expressions, one for each machine instruction or label, terminated by a NIL. This format is used, rather than a single-list format as described in Appendix D of LISP1.5PM, so that LAP may read in the assembly words one at a time and thus avoid undue strain on free storage. Forms in the input file

(DECLARE . . .)

which have no effect when EVAL'ed, signal certain conditions to the compiler, such as special declarations, etc; other forms which are not function definitions or DECLARE's are merely passed from input to output file. This scheme has worked quite well - most debugging is done with interpreted code (which is quite fast anyway), and when a file of functions appears secure, it is compiled for subsequent use. Although one expects a speed-up in general of about a factor of ten, many functions seem to be limited by the speed of machine language SUBR's, such as READ or ASSOC, and some show speed-ups of less than a factor of two.

The first implementations of LAP for MACLISP were entirely in EXPR-FEXPR form, but very shortly an "inner loop" was coded by hand in LAP and appended to the file of EXPR's in order to be bootstrapped in. In fact, so critical is this "inner loop" that when the remainder of LAP is compiled, the observed speed-up is only about a factor of two. Currently, the greater part of this critical loop, comprising about 200 cells, is resident in the system, as a compromise between the undesirable effects of having all 1400 instructions in the system and the undesirably long time required to bootstrap in the critical portions.

As a language, LAP very closely resembles that of MIDAS, the machine-language assembler developed at the Artificial Intelligence laboratory following the style of DEC's MACRO assembler; the major drawbacks are occasioned by (1) the dependence on READ for input, and (2) the difficulty of emulating more than a small subset of MIDAS's extensive macro facilities. The similarities end there, however, for LAP is an in-core assembler and loader, whereas MIDAS is a two-pass, off-line, assemble-only system. To incorporate MIDAS assembled subroutines into MACLISP, a fairly sophisticated relocating and linking loader is required; and occcasionally the extra effort is warranted over the other alternatives: (1) merely using LAP, especially if the subroutines total less than a few thousand instructions; (2) directly assembling the subroutines along with the entire LISP system. Needless to say, for each alternative, there are reasonable applications for which that alternative seems best suited. The assemble and load processing of LAP takes about 14. seconds per 2000-word block of instructions (as measured on an average task - the LAP code for the compiler), whereas the action of MIDAS may often be quicker. On the other hand, LAP needs only about 4000 cells total for smooth operation, whereas MIDAS, with a large hash-coded table, generaly occupies in excess of 40,000 cells.

A sample compilation:

```
(DEFUN POPSYM (X)
       (TESTP (CAR X) FOOVARS)
       (PUTPROP (CAR X) (CDR X) (QUOTE SYM))
       (CAR X))
```

would give rise to the following LAP code (commentary in square brackets is for illustration only, and is not produced by the

compiler):

```
(LAP POPSYM SUBR)
(PUSH P 1)                          [save X on PDL]
(MOVE 2 (SPECIAL FOOVARS))          [get value of special
                                     free variable FOOVARS]
(HLRZ @ 1 1)                        [get (CAR X)]
(CALL 2 (FUNCTION TESTP))           [execute function TESTP]
(MOVEI 3 (QUOTE SYM))               ["SYM" is 3rd arg]
(HRRZ @ 2 @ P)                      [get (CDR X), 2nd arg]
(HLRZ @ 1 @ P)                      [get (CAR X)]
(CALL 3 (FUNCTION PUTPROP))         [perform PUTPROP]
(HLRZ @ 1 @ P)                      [return (CAR X)]
(SUB P (% @ @ 1 1))                 [restore PDL]
(POPJ P)                            [exit]
NIL
```

"CALL" is not a PDP/6 instruction, but causes a trap to a special interpretive routine which either (1) acts as an interface between compiled code and interpreted functions (e.g., if TESTP is an EXPR, then APPLY will have to be called), or (2) smashes the CALL with a direct link to a machine language subroutine (if the code for the SUBR PUTPROP begins on location 672, then the CALL will be replaced by "PUSHJ P,672").

As mentioned already on page 6, the functions of TRACE are encoded in LISP and stored on an auxilliary file. Several other debugging aids are likewise implemented and stored on the COM device, although some have been compiled for greater operating speed:

(1) The whole editing package as used at BOLT, BERANEK, and NEWMAN in their BBN-LISP is available both in LISP and LAP code.

(2) A debugging aid called SPY, available in LISP code; SPY is essentially an evaluation controller with a more sophisticated break-restart facility than BREAK.

(3) A "pretty-print" and indexer package, in LISP code. GRINDEF is used to print out functions in a properly-indented format for easy

viewing; GRIND0 and GRIND1 are file transducers for the same purpose; INDEX scans a file and prepares an index for it including functions defined, free variable functions (or undefined functions) referenced, and a very brief analysis of each defined function.

(4) A display generating package for easier construction of the arrays from which the DEC 340 display scope takes its commands. Available both as a machine-language augmented system, and in pure LISP code for regular systems; there are routines to generate displays of points, lines, curves, connected lists of points, axes, labelling text, and a moveable cursor.

It has been noticed in LAP, in TRACE, and in the display generation packages, that some parts can be encoded in LISP only with great loss of efficiency, while many functions are for all intents and purposes just as efficient when encoded in LISP as when encoded any other way. The interpreter, a very general tool, has its shortcomings; but, for a number of jobs, they are just never noticeable.

N.B. Appendices will appear to explain to the user the details of using the compiler, LAP, and other such aids as above.

Future Plans

For some time there has been expressed a hope of speeding up the running of compiled arithmetic programs. To this end, we are working with the MATHLAB project on a new representation for numbers so that compiled code may handle numbers directly, rather than through the medium of an amorphous garbage-collectable storeage space. It is the consumption of number spaces, and consequent time-consuming garbage collections, which is responsible for the time slowdown of arithmetic operations in LISP. There will be a FIXNUM space and a FLONUM space; numbers will be represented by pointers directly into the appropriate space, without requiring an atom header or any free storage space. However, this in itself is not sufficient for great savings - it will only reduce by some small factor the number of number conses available before a garbage collection is necessary. The innovative feature of the plan lies in the creation of two more linear stack areas, the FIXNUM pdl and the FLONUM pdl. When a compiled function is entered, a small block of the pdl area is taken for holding the partial results of numerical computations, and pointers into the pdl area may be passed along as arguments to other functions just as pointers into the number spaces would be; thus no consing need be done in order to pass along arguments to other functions. In certain cases of compiled code calling EXPR code, additional work may be required at the interface in order to convert numbers in the pdl area into regular LISP numbers (see discussion of "CALL" on page 37).

Certain developments in hardware modifications for the PDP/10

are opening the way for several desirable features in MACLISP. A hardware paging scheme will allow jobs in the ITS system to overlap core areas with other jobs; thus a pure-procedure, or re-entrant, implementation of the basic MACLISP system becomes feasible so that many active LISP jobs may be initiated with each one requiring only as much core as is needed for its own private storage areas. In addition, it will be much more practical to implment a run-time linking relocator for the purpose of loading MIDAS-assembled subroutines into an already running LISP job (a relocating loader could be trivially implemented now, but the linking phase, for global symbolic references, depends on ready access to the job's symbol table; this is somewhat cumbersome in the current implementation of ITS, but with the proposed paging scheme, a job which needs to access its symbol table could attach it simply by adding to its legal page allotment those pages of its symbol table.) We are planning new ways to store binary programs, both those loaded by LAP and those loaded by the yet-to-be-implemented linking relocater, so that the whole of binary program space may be dynamically relocated at will. The next step, then, is to re-work the representation of atoms so that all storage areas may be dynamically expanded and/or relocated; the garbage collector will then be able, as the execution of the job progresses, to re-allocate the amount of core assigned to any given storeage area.

The current representation of atoms has led to the problem discussed at the top of page 10, and this problem must be averted in order to realize the dynamic re-allocator. One way of solving the

problem is the creation of a separate space for the storeage of atoms, but apart from this problem, experience has shown the desirability of an "atom header" space on other grounds. The VALUE, PNAME, and function properties of an atom are generally of paramount importance - the more rapidly they may be retrieved, the better. Our current thinking on this matter calls for a space of atom headers containing, for each atom, immediately accessible pointers to the VALUE, PNAME, and function (if any) properties, with additional bits for garbage collector usage; and for a separate space for PNAME storage. We would expect a speed-up of a factor of two in the interpretation of EXPR code, and a small economy (perhaps 15% to 30%) in the amount of memory occupied by atoms. The proliferation of separate storeage spaces is not so bad as it might seem at first, since one of the ultimate benefits is the dynamic re-allocator - the user would be less conscious than ever of the separation of memory.

The problem referred to above may be circumvented by another application of the paging concept: allocation is fixed, but each space is made to be very, very large. A LISP job would utilize the full eighteen bits of the address space, but only those pages which have been referenced are actually held in memory. The major drawback to this plan is that our hardware modification plans do not provide for a page-on-demand interrupt; but at any rate, this circumspection is at best a postponement of the real problem.

Some discussion is also taking place on the matter of a programmable format specification for number print-out; on the possible application in LISP of the "local - global" distinction found

in the block structure of ALGOL; and on a more versatile I/O channel sturcture (such as that of LISP 1.6 at the Stanford Artificial Intelligence Laboratory.)

APPENDIX A

Preview of Global Variables and Table of Control Characters

NOUUO, NOCHK, NORET, BAKGAG, GCTWA, *NOPOINT, *RSET all set accessible switches, namely their own special value cells. Thus (NOCHK 5) accomplishes the same as (SETQ NOCHK 5).

There are a number of global variables critically pertinent to system operation - a brief catalog follows:

ATOM	VALUE OR MEANING
BASE	Radix to be used when printing out fixed point numbers. Used by PRINT.
BPORG BPEND	When the system is initially loaded, all the memory from BPORG up to the top of the core allotment is available as a kind of amorphous storage (not part of free storage or full word space. Subroutines loaded by LAP are generally stored in the lower end of the space and Arrays are dynamically stored at the higher end. BPORG is adjusted by LAP to indicate the lowest currently available cell of this space, and BPEND is updated by the array handler and garbage collector to indicate the highest currently available cell.
CHRCT	Number of character spaces remaining in the current output line. PRINT enforces a maximum output line length of LINEL (q.v.), and when CHRCT reaches zero, a carriage return is emitted and CHRCT reset to LINEL.
DISLIST	One means of displaying information on the 340 display scope is by the creation of "display arrays" - each such array is a

sequence of instructions to the 340 (see Manual for usage). Dislist is a list at the special array cells of those displays that one want active. In time sharing LISP the display is not a actually carried out unless the ↑F switch is on (q.v.).

ERRLIST When at non-ERRSET error is detected, control passes to an initialization routine, just before re-entering the top level main loop. Average other things mapc [EVAL; ERRLIST] is performed, thus allowing the user to add his own two cents to the error receiving process.

IBASE Radix to be used by READ when converting a number-like string on input. May be any value from 1 to 36., except that any number-like string having non-decimal digit must begin with either +" or -".

EXAMPLES:

If IBASE is 8, then "47" is read as 39.

If IBASE is 12., then "14" is read as 16.

If IBASE is 16., then "+1A" is read as 27.

LINEL The line length for PRINT to use. After LINEL characters have been outputted, a carriage-return line-feed is automatically inserted. When the job is started up, LINEL is set to 72. for jobs controlling teletype, and to 45. for those controlling the datanets.

NIL NIL

OBLIST A list of lists: each interior list (of atoms) is called a bucket. Every atom read in by READ on READCH, or explicitly INTERN'd is put on the OBLIST. If there is already an atom of the same pname on the OBLIST, then no new entry is made to OBLIST, but instead INTERN returns a pointer to the already-existing atom.

TTY When job is started up, set to NIL if at teletype, to 0 if at GE

Datanet.

Here "x" and "y" are used as meta-characters. Typing ⌈x - control x - On a job console, or evaluating (10C x) or (SSTATUS 10C x) sets the x switch to non-NIL. The x switch is merely the value of the atom ↑x - up-arrow x -, except for G and X. Each switch, except for A,G, and X has a corresponding control character to turn it off (set to NIL), denoted by y.

X	Meaning of Switch	Y
A	Available to user, not used by system.	(none)
B	Enable line printer to receive PRINT output.	E
D	Turn on output of garbage collector statistics (one set produced at each collection)	C
F	Enable displaying from the display arrays on DISLIST.	Y
G	Unconditional quit, not caught by ERREST.	(none)
N	In time sharing, turn on the DIS devices to receive PRINT output. (Physically, the DIS device is the same as the 340 display, and one cannot use them simultaneously).	Y
P	Seize the Calcomp plotter facility.	U
Q	Let READ receive its input from the currently selected input file, rather than the job console.	S
R	Enable the currently open output file to receive PRINT output.	T
W	Disenable the job console from receiving PRINT output.	V
X	Quit, acts very much like an internal error.	(none)

APPENDIX B

Some Allocator Goodies

When a LISP job is started up under time sharing, the system first interrogates the user as to the allocation of memory for the various storage areas used. The system types:

```
LISP 105
ALLOC?
```

If the user then types "N" for "no" certain standard options are taken; if he types "Y" for "yes" the system then prints out, line by line, the standard options, pausing at the end of each line for the user's response.

```
CORE   = 22
FXS    = 400
FLS    =  400
REGPOL = 777
SPECP  = 777
FXDL   = 7
FXDL   = 7
```

By typing a space or carriage return, the user accepts the standard option; by typing a number, he specifies that allocation (a string of digits is a number base eight; a string of digits followed by a decimal point is a number base ten): by typing an alt-mode all

remaining standard options are taken and the allocation phase completed; by typing a control-G the allocation phase is re-started.

In the current system, CORE cannot be specified to be less than 20 blocks (16K words). Regardless of the specification for FXS (FIXNUM space) remaining after allocation is divided as follows:

31/32 to free storage and 1/32 added to the FXS allocation. This is partly motivated by the fact that PNAME strings are stored in FXS. FLS is FLONUM space, regular and special PDL's are obvious, and the two number PDL's will be relevant only for functions compiled by the soon-to-be seen fast arithmetic compiler.

Additional core may be grabbed by the system when more binary program space is needed.

APPENDIX C - STATUS and SSTATUS

The FSUBR's STATUS and SSTATUS are implemented to aid in querying the state of the LISP system, and in setting some of its conditions. The first item in the arglist is always an atom which tells what kind of query, or command, is wanted; but some of the other items may be EVAL'ed, just as with the FSUBR "ARRAY". A dispatching is done on the first five characters of the PNAME of the first item, so that if atoms like CORE and NOUUO are REMOB'ed, the functions of STATUS and SSTATUS may still be properly performed. In fact, the main purpose of the function EXCISE is to reclaim from freestorage all those atoms with SUBR properties whose functions have been subsumed by the STATUS series. In the tables below, the small letters x, y, n, m, as well as "dev" and "usr" are used as metavariables ranging over LISP atoms. "frag" is used for representing a list fragment.

arglist for STATUS	meaning, or value
(CHTRAN n)	The character translation value in the nth entry of the READ translation table.
(CORE)	Number of blocks of core occupied by the job.
(CRUNIT)	A 2-list of the most recently referenced device and sname. Updated by UREAD, UWRITE, and SSTATUS
(DATE)	A 3-list with year, month, and day number. E.g., on Mar 20, 1970, this would be (70. 3. 20.)
(DAYTIME)	A 3-list with hours, minutes, and seconds of the day.
(FREE y)	Amount of space available in the y space. Currently y can be BPS for binary program space, FS for freestorage, FXS for fixnum sapce, FLS for flonums.
(GCTWA)	The GCTWA feature (see page 9 of Progress Report) consists of two switches - one forcing TWA removal on every garbage collection, and one forcing removal only on the next occurring collection. 0 neither switch is on 1 "next" switch is set, "every" is off 10, 11 "every" switch is set.
(IOC x)	If x is a character with a control action listed at the end of appendix A, the this gives the state of the x switch. E.g., (STATUS IOC B) tells whether the line printer is selected.

(MACRO x)	NIL if x is not a READMACRO character (see Progress Report page 27); otherwise a number describing the type of macro. 0 normal 1 splicing 10 delayed 11 delayed and splicing
(RUNTIME)	same as calling the function RUNTIME
(SYNTAX n)	The bitwise decomposition of this number tells the syntactic categories of the nth entry in the READ syntax table for characters.
(TIME)	same as calling function TIME
(TOPLEVEL)	NIL if standard; otherwise the form used in the top level function (see Progress Report page 10)
(TTY)	NIL if job console is a teletype, 0 if a GE datanet.
(UREAD)	NIL if no file currently open by UREAD; otherwise a list describing the open file. E.g., (FNAME1 FNAME2 dev usr)
(UWRITE)	NIL if no device open by UWRITE for writing; otherwise a 2-list describing the currently open device in format similar to CRUNIT
(+)	T if the super-decimal digit feature is enabled, NIL if not. This feature allows non-decimal digits to be used when reading in numbers: any atom beginning with "+" or"-", except those single-character atoms themselves, will be interpreted as a number base IBASE. "A" is used as the tenth digit, "B" the eleventh, etc.
(x)	If x is among NOUUO NORET *NOPOINT NOCHK *RSET BAKGAG, returns the value of the associated switch.

arglist for SSTATUS	meaning, or action
(CHTRAN n m)	Sets nth character translation to m.
(CORE n)	Requests the time-sharing system to set the job core allotment to n blocks
(CRUNIT dev)	Updates the "current I/O unit" to be device dev
(CRUNIT dev usr)	As above, but also changes sname to usr.
(FREE BPS n)	Will insure that there are at least n words available in binary program space. Will increase job core allotment if necessary

(IOC x)	same as EVAL'ing "(IOC x)"
(MACRO x y)	Sets x to be a READMACRO character of regular type with associated function to be the value of y. E.g., (SSTATUS MACRO /' (QUOTE (LAMBDA NIL (LIST (QUOTE QUOTE) (READ)))))
(MACRO x y S)	splicing type (see Progress Report page 27)
(MACRO x y D)	delayed type
(MACRO x y S D)	splicing and delayed type
(SYNTAX n m)	Sets nth entry in READ syntax table to m.
(TOPLEVEL x)	Sets the toplevel form to the value of x. (see Progress Report page 10)
(UREAD frag)	same as EVAL'ing "(UREAD frag)"
(UWRITE frag)	same as EVAL'ing "(UWRITE frag)"
(+ y)	Sets the state of the super-decimal digit feature to the value of y. see above under STATUS
(x y)	If x is among NOUUO NORET *NOPOINT NOCHK *RSET BAKGAG, sets the state of the x switch to the value of y.

APPENDIX D

Some About Trace and Break

The FSUBR BREAK is used much like a break point in DDT. EVAL'ing

(BREAK ident predicate)

will do nothing if the value of "predicate" is NIL, otherwise will print out "ident" for identification and then enter a READ-EVAL-PRINT loop. Thus one may inspect variable values and initiate certain remedial actions. Exiting from the break, which returns NIL is done by typing the atom "$ P" (alt-mode "P" space). An alternate form is available:

(BREAK ident predicate retval)

in which the value of "retval" is returned, rather than NIL. By using the EDIT feature, it is very convenient to insert and delete such "breakpoints" in EXPR and FEXPR code.

TRACE is a FEXPR generally found on the File COM:TRACE LISP, used to trace the flow of control of a program. A function which has been set for tracing will regurgitate a little note each time the function is entered or exited. The basic format of the message is, for entry,

(n ENTER FOO arglist)

where n is the recursion depth at this call to the function FOO, and arglist is a list of the arguments for this call [unless foo is a FEXPR or FSUBR, in which case arglistis directly the one argument - see Progress Report page 2]. The base format open exiting is

(n EXIT foo value)

where value is the returned value of function Foo.

The user, when setting up a trace, may request other values to be printed out along with these messages; for example, if (CDR L) and (PLUS BRORG LOC) were requested, a sample message might look like

(6 ENTER LAPEVAL (1) // (0 1) 35001)

when L = (1 0 1), BPORG = 34777, LOC = 2 and the argument to LAPEVAL is 1. Similarly upon exiting one might see:

(6 EXIT LAPEVAL 1 // (0 1) 35001)

In addition, the user may request (1) that only one, or perhaps neither, of "arglist" and "value" be printed out, (2) that calls to the trace-set function be"traced" only when a given predicate is true, and (3) that a conditional break point be placed as the first item of execution of the function.

The syntax of a trace request is very simple - evaluate "(TRACE r1 r2 rn)" where each ri is either an atom foo (meaning set up a trace for function foo with standard options) or a list interpreted as follows:

(foo c) Trace foo, showing both "arglist" and "value" if c = BOTH; showing neither if C = NIL; showing only "arglist" if c = ARG; showing only "value" if c = VALUE

(foo c s1 s2...sn) As above, but also show the values of s1, s2,...sn.

(foo COND pred e) As above for "(foo c)" but trace a call to foo only if "pred" evaluates to non-NIL when foo is entered. The atom ARGLIST, where it occurs in pred, will have as value the arglist discussed above.

(foo BREAK pred c) - As above for "(foo c)", but insert a breakpoint with identification foo and conditional predicate pred, just after entry to foo.

Combinations of these forms are permitted, such as (foo COND p1 BREAK p2 c s1 s2) but one must not forget to include c as part of each individual trace request. The standard option mentioned above is just "(foo BOTH)".

A function set for tracing actually has its function property altered - SUBR's an EXPR's are turned into a new EXPR; FSUBR's and FEXPRS are turned into a new FEXPR. This new function handles the administrative details of tracing (and incidentally may introduce quite an overhead in time), and applys the original function to the arglist. One may reset functions to their original pre-trace definitions by evaluating

(UNTRACE foo1 foo2 ... foon)

EVAL'ing "(TRACE)" will yield a list at all functions currently being traced and EVAL'ing "(UNTRACE)" will untrace all such functions.

EXAMPLES:

1) To trace BUGLE whenever its first argument is greater than 30, and to trace all calls to HORN. Let us suppose that BUGLE is defined as (DEFUN BUGLE (N X) (PROG2 (SLEEP N) (HORN X))) then -

(TRACE HORN (BUGLE COND((GREATERP N 30) BOTH))

will do the job, as well as

(TRACE HORN (BUGLE COND (GREATERP (CAR ARGLIST) 30) BOTH))

2) To trace DEFPROP, showing only the arglist, and breaking whenever a property is about to be defined for BLAND:

(TRACE (DEFPROP BREAK (EQ (CAR ARGLIST) 'BLAND) ARG))

3) To trace LAP, showing only the amount of binary program space

(TRACE (LAP NIL (DIFFERENCE BPEND BPORG)))

One of the functions on the TRACE file is "?" EVAL'ing "(?)" will cause a printout of a short note about TRACE with a few example uses. Since this note takes up valuable space, it may be removed by EVAL'ing "(??)". Eval'ing "(REMTRACE)" will remove all the functions of TRACE.

APPENDIX F

Using Index and Grind

PART 1 Introduction

INDEX and INDEX1 are functions that **permit** the LISP programmer to more easily debug or interpret long, complicated LISP programs. Both functions operate on some file indicated by the user producing new file with information for each function defined by a DEFPROP or DEFUN:

1. type (EXPR, FEXPR, MACRO)
2. arguments
3. free variables appearing
4. free variables modified by SETQ
5. functions in the file used in the definition
6. functions in the file that use the function defined
7. undefined functions or free variables used as function names

Additionally, notes are made of any functions defined more than once and of any function already defined in the system.

To use these functions, one need only perform the following incantations:

1. Read in some version of GRINDEF:

   (UREAD E GRIND COM) { or, (UREAD GRIND LISP COM) }

   ⌐Q [means type Q with CONTROL key held down]

2. Read in index file:

   (UREAD INDEX LISP COM)

   ⌐Q

3. Attack file:

(INDEX1 filename1 filename2 device user)

(UFILE filename 3 filename 4)

The commentary will then appear in a file bearing the names filename3 and filename 4 as selected by the user.

It should be realized that the functions involved are still in development, and any bugs should be reported to Patrick Winston.

WARNING: the programs assume all fixed point numbers are decimal.

Part II Details

Three free variables give the user some control over what the program does, namely ALPHA, GRIND, and GOBBLE. Normally all come set to T. If ALPHA is set to NIL, the information for the functions appears in the same order as do the functions in the file under investigation. If ALPHA is set to T, the order is alphabetical.

If GRIND is set to T, function definitions are added and are interleaved with function commentary. If GRIND is set to NIL function definitions are omitted.

If GOBBLE is set to T, GRIND is set to T automatically and all commentary appears inside the scope of COMMENTs. The idea is to produce an indexed file that LISP can read and enjoy as if it were the original file. Material in the original file that is not the form of a function definition also appears in the indexed version according to the following rules:

1. Anything appearing after the first function definition in the

original file appears at the end of the indexed file.

2. Anything appearing before the first function definition in the original file appears at the beginning of the indexed file. This prevents misplacement of any loader code that may be present.

With GOBBLE set to T, it is believed that all material in the original file will be retained, even multiple definitions bearing the same name.

INDEX1 is designed to parallel file conventions of GRIND1 returns the same sort of thing as its value. That is:

(INDEX1 fn1 fn2 dev user)

returns

(UFILE fn 1 fn 2 dev user)

INDEX differs from INDEX1 only in that the usually desired file naming is done automatically according to the following conventions:

1. If GOBBLE is set to T, the indexed version has the same names as the original file and replaces it.

2. Otherwise, if the original file has the names fn1 fn2, then the indexed file will have the names fn1 INDEX. Naturally care should be taken to avoid indexing a file with the names fn1 INDEX, for the original file will be lost in this case and the indexed file will not contain all of its material.

EVAL'ing "(REMGRIND)" will remove those functions read in from the GRIND FILE; "(REMINDEX)" will remove those read in from the INDEX FILE.

APPENDIX I

USING LAP

This section of this memo replaces entiley A.I memo 152, PDP-6 LAP.

INTRODUCTION

Lap is a LISP FEXPR (or FSUBR when compiled)which is executed primarily for its side effect -- namely assembling a symbolic listing into core as a machine language subroutine. As such, it is about the most convenient and rapid way for a LISP user to add machine language primitives to the LISP system, expecially if the functions in question are in a developmental stage and are reasonably small (e.g. 1-500 instructions). Also, the LISP compiler currently gives its result as a file of LAP code, which may then be loaded into core by LAP.

Virtually, any function definition, whether by DEFPROP, LABEL, or LAP, is an extension of LISP's primitives; and as in any actual programming language, the side-effects and global interactions are often of primary importance. Because of this, and because of the inherently broader range of machine instructions and data formats, a function quite easily described and written in PDP-6 machine language may accomplish what is only most painfully and artificially written in LISP, One must, then, consider the total amount of code in each language to accomplish a given task, the amount of commentary necessary to clarify the intent of the task given the program (in this sense, LISP code rates very high -- a major benefit of the confines of Lisp is that a good program serves as its own comment,

and usually needs no further elucidation), and other considerations of programming convenience.

Experience has shown that many such subroutines may be assembled by a small system, i.e. one such as the current LAP, without conditional assembly, macro, or sophisticated literal generation features. These latter three features are the major differences in language between LAP and MIDAS; the major operational differences are (1) LAP is one-pass and MIDAS is two, (2) LAP uses the LISP READ function while MIDAS is more efficient, and (3) LAP assembles directly into the binary program space of the LISP system using it while MIDAS files its assembly on a peripheral device (which must then be loaded by STINK or the ITS version of DDT). Thus one must consider the scope of his task in relation to the language desired and the operational ease preferred.

Unfortunately neither LAP nor the system reported in A.I. memo No. 127 solves the problem of loading and running arbitrary binary programs jointly with LISP, Something like a runtime primitive STINK is needed for LISP, and such may have to wait for further development in the multiprogramming capabilities of the PDP-6 systems.

FORMAT OF LAP USAGE

A call to LAP is even a little more non-standard than indicated in the introduction in that not all the arguments are included in the S-expression which commences assembly -- LAP repeatedly calls READ, operating on the S-expressions read-in (from the current input device and file), until a NIL is encountered, at which time assembly is terminated. Only after successful

termination of assembly is BPORG updated and the correct flag (SUBR, FSUBR or LSUBR) inserted on the property list of the atoms which name the newly assembled functions. Thus a call to LAP would look like the sequence.

```
(LAP  FOO  SUBR)
(DEFSYM  A  1)
(HLRZ  A, 0 (A))
(POPJ P)
NIL
```

Instead of the following written in a hypothetical style after 7090 LAP

```
(LAP   ((FOO SUBR 2)
        (HLRZ A, 0 (A))
        (POPJ  P)
        ((HLRZ . 554←27.)
         (POPJ . 263←27.)
         (P . 14)
         (A . 1)))
```

The most serious drawback to the latter style is the strain placed on free storage, since the entire expression would have to be in core before evaluation could begin.

Hence evaluation of (LAP name indicator) or (LAP name indicator address update) begins a LAP assembly for a function with name "name" of type "indicator" (such as SUBR, FSUBR, or LSUBR) and with entry point the first location assembled into; if the second form is used, assembly

begins in the core location "address" instead of BPORG. Ordinarily at assembly termination, BPORG is set to the address following the last one assembled into by LAP, but if "update" is NIL, BPORG is undisturbed.

LAP acts on the quantities it reads as follows:

QUANITITY	ACTION
NIL	Terminate assembly and return. Literal generated constants are assembled into core, symbol definitions from DEFSYM are flushed, and worthless atoms are removed from the oblist. A common error is to forget that carriage return and E-O-F are not atom break characters; NIL should be followed by a space.
atom	Assign "atom" an assembly symbol value equal to the address of the current assembly location; no additional assembly takes place. Thus one uses atoms for symbolic location tags and under certain conditions these names are entered in DDT's symbol table (see below).
(DEFSYM atom sexp . . . atom sexp)	Assign "atom " an assembly symbol value equal to (EVAL sexp); no additional assembly takes place, and these names are not entered into DDT's symbol table.
(ENTRY name type) (ENTRY name)	Sets up "name " as a function of type "type" (default-same as call to LAP)and with entry point the current assembly location; no further assembly takes place.
(COMMENT list-fragment)	By a neat technique, no unnecessary atoms remain on the oblist after assembly; however, during assembly, there must be enough full word space to hold print names

for all the atoms and to hold the numerical values of a few LISP numbers.

(SYMBOLS t-or-nil) NIL turns off and non-NIL turns on the LAP feature which passes along symbolic location names to the job symbol table; currently all symbols so entered are treated as global, but at some time in the future this may be modified to permit flexible duplication of symbols in several programs. If the SYMBOLS pseudo-up appears anywhere in an assembly, then the names of functions thus defined will be transmitted to DDT. NOTE WELL: Although LISP atoms may be composed from upwards of 80 characters, those used as tags which are entered in the symbol table should include only legal MIDAS characters, and only the first six characters of the atom's PNAME are relevant to this feature.

(BLOCK n) A block of n words is assembled, each containing a zero.

(ASCII sexp) The s-expression sexp is EXPLODEC'd and the resulting list of 7-bit characters is assembled, 5/word, in successive words.

(SIXBIT sexp) Same as for ASCII, except that the 6-bit form of the character is used, 6/word.

(x)

(x ; list-gragment) x (which is not one of the above items) is evaluated by LAPEVAL and the numerical result stored in the current assembly location, which is then advanced

by one. For the meaning of LAPEVAL, see the next section on assembly constituents. "list-fragment" is ignored and may serve as commentary (see note above for Comment).

(x y)

(x y ; list-fragment) Same as immediately above, but (LSH (LAPEVAL y) 23.) is added into the stored result.

(x y z ; list fragment) Same as immediately above, but (BOOLE 1 (LAPEVEL z) 777777) is added into the stored result. Forward reference symbols may appear only in the z field; that is, if a symbol is used before it is defined, it must be used only in the address part of the instruction.

(x y z w)

(x y z w ; list fragment) Same as immediately above, but the numerical value of (LAPEVAL w), treated as a 36-bit quantity, is swapped left-half for right and then added into the stored result.

LAP initially checks whether or not the atom @ is a member of the list forming the assembly word, and if so sets the indirect bit (bit 13) and deletes the @ from the indicated assembly; thus an @ does not count as one of x, y, z, or w.

One notices that there is a strong similarity between LAP format and MIDAS format, an essential difference being that LAP processes assembly quantities "in orde let-to-right, to determine which is the AC field, which the address field, and which the index field. One must remember that the LISP read routine imposes a certain dissimilarity in text for the two assemblers, since "space", "comma", "left paren", and "right paren" are the only break characters for atom names.

Hence spaces are necessary on both sides of a semi-colon or at-sign when they are used as described above, and the AC field may not be ommitted in instructions like (JRST 0 ADDRESS). The index field need not be enclosed in parentheses as in MIDAS, but in general there is no harm in doing so (see "anyother list" under assembly constituents).

NORMAL AND ERROR RETURNS

Normally, after terminating assembly, LAP returns a list containing the current value of BPORG, and the names of the subroutines just assembled (there may be more than one entry for the routine assembled, the principle entry is declared in the call to LAP and others may be declared by means of the pseudo-operation ENTRY). If, after assembly, some referenced symbols remain undefined, the message "UNDEF SYMBOLS", followed by the offending atoms, will be printed out. If there were any mutliple-defined symbols, "AMBIG SYMBOLS" is printed along with a list of the offenders. One particular disaster caught by LAP is indicated by the message "not enough core".

Since LAP uses so many free variables (and for several other reasons), one should allow a call to LAP to exit by itself rather than stopping it with ^G or some other ruse.

ASSEMBLY CONSTITUENTS

Each of the parts of an assembly word (x, y, z, or w) is evaluated by LAPEVAL, in the context of the assembly. The assembly quantities whose CAR is among DEFSYM, EVAL, COMMENT, and SYMBOLS, may be termed pseudo-ops in that they do not give rise to an assembly word but merely

give directions to the assembler. @ and ; are treated specially by LAP and are not considered to be assembly constituents.

QUANTITY	VALUE
number	Fixed-point numbers always evaluate to themselves. Floating-point numbers in an address field may produce Random Results.
NIL	Same as (QUOTE NIL).
*	The address of the current assembly location. Same as in MIDAS.
atom	Except for @, ;, *, and NIL, all atoms evaluate to their assembly symbol value; i.e. (GET (QUOTE atom) (QUOTE SYM))
QUOTE sexp)	(MAKNUM (QUOTE sexp) (QUOTE FIXNUM). For example, (MOVEI 1, (QUOTE (SMALL LIST))) assembles into an instruction which moves a pointer to the list (SMALL LIST) into accumulator 1.
(SPECIAL atom)	Provides a pointer to the value cell of "atom". thus (MOVE 1, (SPECIAL FOO)) moves the value of FOO into 1 instead of a pointer to FOO, as would happen if (QUOTE FOO) were used. In addition (MOVE 1, (SPECIAL BAR)) (MOVEM 1, (SPECIAL FOO)) accomplishes in a SUBR what (SETQ FOO BAR) does in an EXPR.
(FUNCTION atom)	Essentially the same as (QUOTE atom), but is used to

QUANTITY	VALUE
	emphasize that "atom" is used as a function name (see section on UUO instructions).
(ASCII sexp)	Provides a 36-bit ascii representation of the first five characters from (EXPLODEC sexp).
($IXBIT sexp)	As above, except sixbit representation.
(% x y z w)	Literal generation feature, like [x y, z (w)] in MIDAS. Assembles (x y z w) as described above and provides the address thereof. Similarly, the forms (% x), (% x y), (% x y z) may be used. A literal constant is restricted to the z-field (or address field) of a LAP instruction, but may appear nested to any finite depth. Example: (MOVE 1, (%1.0)) moves a machine floating point number into 1, whereas (MOVEI 1, (QUOTE 1.0)) moves a LISP number.

SYMBOL DEFINITIONS

LAP is initialized with a few basic symbol definitions needed by code generated from the compiler, such as P = 14, and proper addresses for NUMVAL, SPECBIND, UNBIND, FLOAT, etc. Not all PDP-6 op-codes are pre-defined, but if one of the missing ones is used in LAP code, then LAP will obtain a correct value for it from DDT. Temporary symbol definitions may be made by the user with the pseudo-op DEFSYM (see section on "Format of LAP usage).

If there remain any undefined, referenced symbols of the end of a LAP assembly, then the DDT symboltable is interrogated to try to find a definition.

Thus one can write LAP code as if it were being assembled together with the LISP system. When passing along entries to the DOT symbol table, or when looking up a symbol in it, any character of the symbol not a legal MIDAS syllable constituent is converted to the character " ."

Four trap instructions are provided to help link up compiled code with other functions, possibly in EXPR forms: CALL, CALLF, JCALL, JCALLF. The first two simulate a transfer like PUSHJ P, FOO; while the latter two simulate a transfer like JRST FOO. The accumulator field of an instruction with op-code in this sense tells which of the three argument conventions has been followed; In (CALL n (FUNCTION f)) if $0 \leq n \leq 5$, then the EXPR-SUBR convention is used, with the n arguments located in accumulators 1 to n; If n = 17, then the FEXPR-FSUBR convention is used, with the one argument located in accumulator 1; If n = 16 then the LSUBR convention is used, which assumes that accumulator 6 contains the negative of the number of arguments, which are stacked up on the regular PDL last on top, and the return address is on the PDL before all the arguments. Under favorable conditions, when CALL'ing a SUBR, LSUBR, or FSUBR, no further interfacing need be done and the CALL instruction is actually replaced in core with a PUSHJ or JRST. The ops CALLF and JCALLF, however, are never replaced. Similarly, if NOUUO is non-NIL, the instruction-modify phase is inhibited.

AVAILABILITY

The COM device holds files E LAP and C LAP. E LAP is an EXPR version of the functions necessary to make LAP work, and C LAP is a compiled version of the same. If one plans to load in a file with calls to LAP on it, then he should first load in one of these two LAP files. In the future, this might be done

Automatically, but right now the user must explicitly take care of loading in needed auxilliary functions.

Evaling "(REMLAP)" will remove a$lmuch of LAP as possible, and the garbage collector will be able to reclaim its space.

AN EXAMPLE

```
(LAP DONTFOOP SUBR)
        (MOVEI 2 (QUOTE PNAME))
(ENTRY DONTFOO)
        (PUSH P 1)
        (CALL 2 (FUNCTION GET))
        (HLRZ 1 0 1 ; THIS COULD ALSO BE (HLRZ 1 @ 1))
        (MOVE 1 0 (1) ; GETS FIRST WORD)
        (CAMN 1 (% ASCII FOO))
        (JRST 0 POPAJ)
        (POP P 1)
        (JCALL 1 (FUNCTION PRINT))
        NIL
```

NUMERICAL ROUTINES ENCODED IN LAP

On one of the tapes of Jon L. White, there is a file LAP routines for the functions SIN, COS, ATAN [arc tangent], SQRT, LOG [Natural log], and EXP [base e]. Also there is a file @ IAS which is a general matrix inverter and simultaneous linear equation solver. Calling

(IAS A NEW N M)

performs gaussian row reduction on the first N rows of the array A (and in fact operates on only the first M columns); so that if M≤N then the N + 1 st through the Mth columns of the output array contain the solutions to the implicit M-N+1 systems of NxN simultaneous linear equations, while the first N columns contain the inverse matrix of $\begin{bmatrix} A_{11} & \cdots \\ \vdots & \ddots \\ & A_{nn} \end{bmatrix}$ If NEW is 'T' then a new array of size NxM is declared and the answers are stored in it leaving the input array A alone; if NEW is 'NIL' then the output array is stored directly over the input array and no new array declarations are done.

Currently, maximization of pivotal elements is not done; thus IAS will give wrong answers on certain numerically ill-conditioned matrices even though they be non-singular. It is possible to remedy this problem, at some expense, if necessary. IAS also uses a portion of binary program space for temporary storage and may give an error message if not enough space is available.

APPENDIX J

MOBY I/O

NVSET

Used to set certain programmable conditions for the new video processor The arguments are in order:

Filter - three bits (0 - 7) to designate the presence or absence of the color filters over the lens.

confidence - should be 0,1,2, or 3. Determines the speed and accuracy of the video processor, 0 being the slowest but most accurate.

resolution - that number of equally-spaced lines over the field of view - maximum of 40000. (Actually, the video pocessor always dissects the scene onto a 20000 x20000 grid. This merely provides a scale factor for the arguments to NVID and NVFIX). For many reasons 1000 to 10000 is an extraordinarily good range for this argument.

dim - should be in the range 0 - 17 inclusive. Selects one of sixteen dim cut off levels. See a hacker for more elucidation.

xyz - Zero means the video processor receives its input from the new ITT vidisctor (TVC); non-zero means the signal comes from the old ITT vidisector (TVB).

NIL may be used for any of the above arguments, in which case the specified condition is not changed. Initially all conditions are 0 except resolution, which is 20000.

NVID

Reads the new video processor and returns a floating-point number (with an information content of ten bits) which is an inverse linear measure of the light intensity at the selected vidisector point. (The manual switch located on the video processor should be in the "LIN" position when NVID is used. Use NVFIX when it is in the "LOG" position). The two arguments are respectively abscissa and ordinate values for the image dissector, and must be fixed-point. Returns -1.0 for the dim or dark cutoff condition.

NVFIX

As above for NVID, but is essentially a logrithmic measure of intensity, scaled between 1 and 1777. Returns a fixed point number. (NVFIX and NVID will differ at most in the three least significant bits depending on the state of the "LIN-LOG" switch. See NVID) . Return 0 for the dim or dark cutoff condition. NVFIX can also read in block mode: nvfix [arrayname; n] will assume, for $0 \leq i < n$, that arrayname [2i] has an x-value and arrayname [2i+1] has a y-value , and will replace the latter by NVFIX [x;y]. LISP numbers are used both coming and going, but generally small numbers will suffice.

One can write a small LAP routine to fill the first n words of some array for input to NVFIX, if the points to be read are related

in some reasonable fashion. In each word the left-half holds the x -value, the right-half the y-value. and the return value is in the right-half.

NVSET is used to set the conditions for NVFIX regardless of which mode is used. An error occurs if the first argument to NVFIX is neither an array nor a number.

MPX To use the A/D and D/A converters, open up the facility with a

(MPX ad da)

Where ad specifies the mode of operations for the A/D and da for the D/A converter. Legal ranges for ad and da are:

NIL - leave status alone

0 - close out device

1 - open device in normal (image) mode

2 - open device in fast (ascii) mode

See the ITS reference manual for more information about this device.

IMPX Reads the input multiplexor, A/D, channel specified by its argument (a number $\leq$ 77).

OMPX First argument is a channel number, and second is a value to be outputted on the D/A output multiplexor.

PLOT This is just the Plot function of Mike Speciner's interpretative plotter. Will require some tutoring before the user will know what he is doing.

PLOTLIST If first argument is a list constructed like a DISLIST, then it will be plotted on the calcomp plotter just as it would be displayed on the CRT 340. If optional second argument is given, then a point will not be plotted as a dot, but rather as the first character of the PNAME of the second argument. Example; to have points plot out as asterisks:

```
(PLOTIST DISLIST (QUOTE *))
```

PLOTTEXT PRINC's its argument at wherever the plotter head finds itself; sample use:

```
(PLOTTEXT (QUOTE Y-AXIS))
```

LPEN Has no arguments. Reads the 340 light pen scanner and returns (count . (x . y)) where count is the number of times the light pen was seen (since the last call to LPEN or since the program's beginning) and x and y are the average abscissa and ordinate values of the light pen when seen.

APPENDIX K

PIC - PAC STUFF

In an effort to utilize taped vidisector scenes, several functions for performing the necessary I/O have been added to LISP. (See PICARRY, READPIC and WRITEPIC). There are obvious advantages for the debugging programmer to having standard, well-described scenes available, as it were, through a simulated vidisctor. Before using the routines, however, one must become aware of the image conventions of PicPac. Images (or scenes) are considered to be rectangular sub-portions of a u init square, and hence image co-ordinates are floating-point numbers between 0.0 and 1.0. This facilitates the mapping of an image space onto various I/O devices. Needless to say, some discretized approximation to the image is what is actually stored on tape, so that the co-ordinates mentioned in READPIC really refer to the nearest lattice point in the image space recorded on tape. Once an array has been read in, however, there is no further use of image space co-ordinates except for the description produced by DESCR. Reference to the array is done as usual on ordinary LISP arrays. The PicPac system will be maintained by Larry Krakauer, Room 819, 545 Technology Square.

PICARRAY Declares an image array for use with PicPac. Its use is exactly the same as the function ARRAY: its arguments are respectively:

the array name

NIL

the x dimension (or number of rows)

the y dimension (or number of columns)

The array elements are accessed as usual - (arrayname n m) evaluating to the n,mth entry in the array.

READPIC Reads into the array specified by the first argument (which must have been declared by PICARRAY), receiving data from the device and file selected by the most recent UREAD. The second and third arguments specify lower-left x- and y- co-ordinates respectively; the fourth argument is a delta d such that adjacent entries in the array are filled by incrementing the image co-ordinates by an amount d. (See PicPac for an elucidation of image space co-ordinates). An alternative form is to specify separate deltas for the x and y directions: (READPIC array lowx lowy d) or (READIC array lowx lowy dx dy). The coordinates of the upper-right point of the image area read in are given by upx = lowx + xdim*dx and upy = lowy + ydim*dy, where xdim and ydim are the x-and-y dimensions respectively of "array". All arguments except "array" are assumed to be floating, in accord with the PicPac convention; however fixed-point numbers may be used; it will be assumed that they refer to a 10000 by 10000 grid and they will be floated accordingly. If a section of the requested input area, i.e. the rectangle from (lowx,lowy) to (upx,upy), is not recorded on the input file, then READPIC prints an error comment and returns NIL; otherwise it returns the name "array".

WRITEPIC FSUBR

(WRITEPIC array) writes out on the currently open output device the entire array specified by the argument (in binary image mode). In general, WRITEPIC is preceded by a UWRITE and followed by a UFILE.

DESCR Part of the PicPac package If there is an argument, its value is assumed to be an array name and DESCR (for "describe") produces a list of ten numbers, associated with the array: (xdim ydim lowx lowy dx dy hash vd light data) where the last three numbers give information about the vidiscetor used, the lighting, and the mode of the data. Numbers three through six are in floating point, and "hash" is irrelevant. xdim and ydim are the x and y dimension (see PICARRAY). If there is no argument given, the array described is the one which the most recent call to READPIC read from. The array stored on tape which serves as the data source for the call to READPIC will in general have a description different from that of the array in core. (DESCR NIL) is NIL so that (DESCR (READPIC arr lowx lowy delta)) produces either a description or NIL depending on whether or not the read was successful. (See READPIC).

DESCRX Same as DESCR except that numbers three through six are converted to fixed point, assuming a grid of 10000 by 10000.

APPENDIX X

An attempt to update A.I. memo 116A with respect to items not well-discussed elsewhere.

Two quick little goodies: (1) accumulator 13 is not used by LISP anywhere, and may be ulitized to the programmer's pleasure; (2) when debugging, one often finds himself crawling around in DDT to inspect the ruins after a catastrophe, and it is an annoyance to see DDT type at you 34735 when you know that this is a pointer to some list - therefore, if the currently open cell in DDT points to an s-expression, typing "P.$X" will cause DDT to enter a special part of LISP which will print out this quantity in the usual LISP style. [Actually "P." is a symbol in the symbol table equivalent to "PUSHJ P, PSYM" , so symbols must be loaded for this to work.]

The remainder of this appendix is in the mosaic style of A.I. memo 116A, and there are no apologies for its terseness.

ASSOC Uses EQUAL

ASSQ Uses EQ

Compiler

One may load a LISP compiler by typing at DDT "COMPLR H". One may still use the previous method of compiling a file, but a more convenient may now exists. When the compiler is ready it will type out "LISP COMPILER 77". If the user then types "(MAKLAP)" he may then type a task specification exactly as with MIDAS, TJ6 and COPY, namely <target file><left arrow><source file><carriage return> to specify that the source file is to be compiled and the resulting LAP code

stored away on the target file. Example:

(MAKLAP)

FOO COMP ←FOO BAR (CR)

or (MAKLAP)

FOO BAR (CR)

In the latter case, the target file will be FOO LAP. The same options exist as in MIDAS for default on specifying device and sname.

Current I/O Device

LISP has bookkeeping room for one input file, opened by UREAD, and one ourput file, opened by UWRITE. There is an internal register herein called current device name, that remembers which auxilliary storage device contains the newly opened file, and this register may be changed by optional arguments to UREAD, UWRITE and UKILL. An open read file is closed by reading an E-O-F character, and an open write file is closed by executing a UFILE (q.v.)

DISAD No longer keeps the counters DISCH or DISLP; but has been extended to work well with nearly all the ASCII characters, including carriage return, line feed, and lower case.

Display The time-sharing system provides a means for "printing" on the CRT 390 scope display.The N switch will send PRINT output to it. If one builds up his own display arrays, either with DISAD, or by some display generating routines, then each array A to be part of the picture must be put on DISLAST as follows:

(SETQ DISLIST (CONS (GET (QUOTE A) (QUOTE ARRAY))

DISLIST))

typing ↑F starts the display running from DISLIST.

EXCISE An EXPR which REMOB's many system atoms which the average user finds of little value. Thus about 500-1000 words of free storage may be gained. Almost all the functions so lost may be duplicated with STATUS or SSTATUS. Atoms remob'ed are REMOB RANDOM GCTWA NOUUO NOCHK RE*ARRAY VALRET CRUNIT ↑D FIXP FLOATP EXAMINE DEPOSIT TIME NORET SLEEP RUNTINE LISTEN LPEN DISAD DISINI ↑F and all functions mentioned in appendices J and K.

FIXP A SUBR, NON-NIL only for fixed-point numbers.

FLOATP A SUBR, NON-NIL only for Floating-point numbers.

IOG has been changed to a more useful form. Essentially operates as if it were defined

```
(DEFUN IOG FEXPR (L)
        (PROG  ( ↑Q ↑R ↑W ↑B ↑N)
        (COND ((CAR L) (EVAL (LIST (QUOTE IOC)  (CAR L)))))
        (EVAL CADR L))))
```

MACDMP In time sharing, passes control to DDT rather than the non-time-sharing "MCDMP". If job is disowned, then a logout is performed. Has argument which is given as a VALRET string to DDT. E.G.

(MACDMP (QUOTE TECOJ/↑X/.LISP$J))

MAX An LSUBR which returns the maximum(numerical)of its arguments, using contagious floating arithmetic. Thus (MAX is 4.0 7) is 10.0

MIN As for MAX, but returns minimum.

Numbers LISP numbers; stored in special areas, are represented simply by a pointer into that area. Thus there is a FIXNUM space, and a FLONUM space, as well as Binary program space, and free storage space (in which lists are stored).

RUNTIME For time-sharing LISP, returns the number of microseconds of CPU time used by the current job, as a fixed-point LISP number, accurate to about 50 microseconds.

SLEEP (SLEEP n) causes the program to stop temporarily and take a nap for n thirtieths of a second. SLEEP is a SUBR.

TIME For time-sharing LISP, returns the time counter from the TS system, as a fixed point LISP number. Currently, this is the number of thirtieths of a second that the system has been running. Note that in TS LISP there is no settime. (Using STATUS, the user may read a real time clock to obtain the time of day. See Appendix C under "DAYTIME")

UKILL An FSUBR (UKILL n) or (UKILL UTn) still flaps tape n. But any other syntax on the arglist will be interpreted as a file specification (like those with UREAD which it tries to delete. e.g. (UKILL LOST FILE DSK PHW) will delete some worthless file on the DSK device with sname PHW.

VALRET PRINC's its one argument as a valret string to DDT. Logs out if job is disowned.