STANFORD ARTIFICIAL INTELLIGENCE REPORT
MEMO AI-90

May, 1969

STANDARD LISP*

by

Anthony C. Hearn**

ABSTRACT: A uniform subset of LISP 1.5 capable of assembly under a wide range of existing compilers and interpreters is described.

*Research sponsored in part by the Air Force Office of Scientific Research, Office of Aerospace Research, U.S. Air Force, under AFOSR Contract No. F44620-68-C-0075, and in part by the Advanced Research Projects Agency of the Secretary of Defense (SD-183).

**Alfred P. Sloan Foundation Fellow.

SECTION 1. INTRODUCTION

When it was first formulated in 1960, (1) the programming language LISP was a truly machine independent language. However, even the earliest computer implementation encountered problems in input-output control and the handling of free variables which were not considered in the original paper. Successive implementations of LISP on more sophisticated machines have solved such problems by independent methods and introduced extensions of the language peculiar to those machines. Consequently, a LISP user now faces considerable difficulty in moving a program from one machine to another and is often involved in weeks of debugging in the process. As a possible solution to this problem, this paper is an attempt to provide a uniform subset of LISP 1.5 and its variants as it exists today. The version of LISP described, which we call Standard LISP, is sufficiently restricted in form so that programs written in it can run under any LISP system upwardly compatible with LISP 1.5 as described in the LISP 1.5 Programmer's Manual (2). As function names vary from system to system and input-output control is different, some modification of the code is of course necessary before function definitions can be successfully compiled in any given system. However, this modification is performed automatically by a preprocessor, which is custom built for a particular system. This preprocessor is a LISP program which is loaded before any Standard LISP programs are run, and could be built automatically into a system if only Standard LISP programs are run. Parts of this preprocessor are similar for all systems, but some of its peculiar to a given implementation. Standard LISP preprocessors have been written for SHARE LISP for the IBM 7090 series machines, Stanford LISP/360 for IBM System 360 machines, Stanford AI LISP 1.6 for the

PDP-6 and PDP-10, BBN LISP for the SDS 940 and Texas LISP for the CDC 6600. For convenience in exposition we shall refer to the first four systems as SHARE LISP, LISP/360, PDP LISP and BBN LISP respectively.

In Section 2 of this paper, the structure of Standard LISP programs is described. Standard LISP conforms as closely as possible to LISP 1.5 as defined in the LISP 1.5 Programmer's Manual, and the necessary deviations and extensions are described in detail. In Section 3, the structure of the Standard LISP Preprocessor is described, and the preprocessor for LISP/360 is given in Appendix A as an example. The translation of general LISP programs into Standard LISP is also discussed in Section 3. A listing of all functions defined in Standard LISP is given in Appendix B, and a function for reading Standard LISP programs is given finally in Appendix C.

The naming of new functions in a language and their definition is of course a very subjective matter, but little justification will be offered for their choice. The author would however appreciate hearing from anyone with criticisms or suggestions for improvements in the formulation.

SECTION 2. STRUCTURE OF STANDARD LISP PROGRAMS

2.1 Preliminary

In order to achieve the greatest possible compatibility with existing LISP systems, Standard LISP is based as closely as possible on the language described in the LISP 1.5 Programmer's Manual. However, there are six main areas where Standard LISP makes significant departures from the description in the Manual in order to offer maximum flexibility in programming. These areas are as follows:

(i) Handling of free variables and constants
(ii) Functional arguments
(iii) Character reading and printing
(iv) External file management
(v) Function and MACRO definitions
(vi) Array handling

Each of these modifications will be described in detail in subsequent parts of this Section. A number of additional limitations forced on Standard LISP programmers because of deficiencies in the design of various systems are also given in this Section.

Several functions given in the LISP 1.5 Programmer's Manual have been redefined in Standard LISP for maximum compatibility with other systems and several additional functions of proven utility have been included. These functions are defined in detail in Appendix B and are therefore not discussed in this Section.

2.2 Free Variables

One area in which the greatest differences between various LISP implementations can be seen is in the handling of free variables. Most systems allow the use of a special cell (usually under an APVAL or VALUE property) for storing those free variables which are global to all functions or are constant in the system. In systems with compilers there is in addition a mechanism for storing and retrieving free variables in compiled functions on a push-down stack. However, communication between a free variable in a compiled and an interpreted function is not possible in general in such a system, as the mechanism for handling each is different. Two systems (PDP LISP and BBN LISP) solve this problem by making all interpreter variables SPECIAL and using a push-down-stack rather than an ALIST for storing their bindings. Standard LISP, however, cannot assume that such communication is possible and therefore imposes certain limitations in the use of free variables on the user. In order to provide a partial solution, however, three classes of free variables are recognized, each of which is handled differently by the preprocessing stage of an assembly. The three classes are:

(i) _Constant variables_ or _constants_ are variables which are global to all functions and whose value remains fixed during the life of a given system.

(ii) _Global variables_ are variables which are global to all functions in that they do not appear in the variable list of any LAMBDA or PROG expression.

(iii) _Extended variables_ are variables which are bound in a LAMBDA or PROG variable list of some function, but free in another.

Constant variables are declared by the function CONSTANT during the preprocessing stage. CONSTANT takes a list of pairs of constants and their values as its arguments, and is defined in Standard LISP as

```
constant[u] = deflist[u;CONSTANT]
```

The preprocessor replaces all such constants by the list

```
(QUOTE<value>)
```

on assembly. This mechanism provides a convenient method for compensating for system differences in the handling of atomic symbols containing arbitrary characters. For example, the atom 'AN ATOM' would be written as $$$AN ATOM$ in SHARE LISP and LISP/360, and as AN/ ATOM in PDP LISP, so such a string cannot be introduced directly into a standard LISP program. However, the programmer can use a free variable ANATOM for example to represent this string, and the appropriate value for the atom declared during the setup stage. Any necessary character value objects, such as COMMA and LPAR, for example, should also be declared using CONSTANT. An example of the call of CONSTANT for LISP/360 would be:

```
CONSTANT(((COMMA $$$,$) (LPAR $$$($) (ANATOM $$$AN ATOM$)))
```

Global variables differ from extended variables in that only one fixed cell is necessary to store the value pointer for the variable, and therefore no push-down-stack/ ALIST conflict arises. Thus it is possible to communicate between such variables in interpreted and compiled functions provided that references to such variables in interpreted code are changed to references to the particular cell where the value pointer is stored. In most systems, any such variables which appear in compiled functions must be declared SPECIAL before compilation. The SPECIAL declaration initializes such variables to NIL and sets up the necessary cell for storing the value pointer. Standard LISP introduces two functions GTS and PTS which allow the programmer to get or change the value of these special cells directly. In SHARE LISP for example these functions are defined as follows:

```
gts[u] = cdar[prop ]u; SPECIAL ;λ[NIL;error[cons[u;(NOT SPECIAL)]]]]]
pts[u;v] = rplacd[car[prop[u;SPECIAL;λ[NIL;list[put[u;SPECIAL;list[NIL]]]]]];v]
```

in other systems, equivalent definitions are always possible. If a given function definition is to be interpreted and not compiled, then the preprocessor changes every reference to global variables in the definition to calls to GTS and PTS. No action is necessary by the preprocessor if the function is compiled.

Extended variables are the only class of free variables which impose limitations on the user. In all systems known to the author such variables require SPECIAL declaration before compilation of functions using them. However, it is not in general possible to mix interpreted and compiled functions using these variables, and so functions containing such variables should either be all compiled or all interpreted.

All references to global and extended variables are made directly to the variable in Standard LISP unless the user wishes to make an explicit call using GTS. Similarly all changes should be made using SETQ, or PTS for explicit references. The functions CSET, CSETQ and SET are consequently not defined or available in Standard LISP.

2.3 The Free Variables ALIST and OBLIST

Many systems allow the user to reference the ALIST and OBLIST directly. However, the structure and referencing (and even existence) of these lists vary so much from system to system that no direct reference can be made in Standard LISP. The interpreter functions EVAL and APPLY are however still available as *EVAL and *APPLY. *EVAL takes a single form as argument and returns its evaluated value. *APPLY takes two arguments, a function and a list of arguments for that function, respectively and returns the value of applying the function to the argument list. Thus in SHARE LISP these two functions are defined as:

```
*eval[u] = eval[u;alist[]]

*apply[u;v] = apply[u;v;alist[]]
```

where alist [] is defined by

```
      LAP (((ALIST SUBR ϕ)(CLA $ALIST)(TRA 1 4)) NIL)
```

2.4 Functional Arguments

Incorporating functional arguments in LISP systems poses many design problems which are still not completely resolved. However, most systems recognize the difference between a call to a functional argument which is essentially a quoted function definition, and one in which the form of the functional argument changes during the evaluation, as given in Saunders' famous example (3) for instance. Standard LISP uses the PDP LISP technique for distinguishing between these two types of calls, viz., a quoted definition is referred to by FUNCTION and a modifiable form by *FUNCTION. The uses of QUOTE to define functional arguments is not allowed. The preprocessor can of course modify these forms to whatever particular calling method is used in the relevant system. Free variables in functional arguments which are not constants require SPECIAL or COMMON declarations in most compiler systems. These declarations are made in a user-defined initialization stage of the preprocessor as described in Section 3.

2.5 Character Reading and Printing

In its purest formulation, the constituent characters of a LISP atom have no inherent meaning. However many applications of LISP require the inspection and manipulation of these characters and the ability to create new atoms from a list of characters. It is also often necessary to read and write individual characters and exercise some control over the format of output.

Standard LISP introduces the following functions for this purpose. All can be readily defined in most systems.

readch[ ]	Reads and returns one character from the input buffer.
princ[u]	Adds the character string u to the output buffer. Returns u.
explode[u]	Returns a list of the constituent character atoms comprising the literal atom u. u may be an integer, but not a floating point number, as various systems use different print representations for these.
compress[u]	Creates an atom (literal atom or number) from the list of characters u, adds it to the OBLIST if one exists, and returns the atom.
liter[u]	A predicate function which is true if u is one of the character atoms A through Z and false otherwise.
digit[u]	A predicate function which is true if u is one of the character atoms 0 through 9.
otll[u]	If u = NIL then this function returns the current length of the output buffer line, otherwise it sets the buffer length to u.
pos[ ]	Returns the number of characters presently in the output buffer.
spaces[n]	Adds n spaces to the output buffer and returns NIL.

In order to achieve compatibility with all systems it is necessary to make a special distinction between character atoms as such and ordinary atoms. A character atom is one returned by READCH or in the list returned by EXPLODE. The functions LITER, and DIGIT should only be used with character atoms as arguments and COMPRESS can only take a list of character atoms. Character atoms do not possess property lists in the usual sense and bear no relation to the ordinary atom of the same name.

Thus the character atom A is not equal in any sense to the atom A or the character atom 1 to the LISP number 1. If particular character atoms are needed in a program, they should be included as constants (e.g., ONE) in the main program and appropriately defined in the preprocessor. Alternatively, they can be created using EXPLODE.

2.6 File Management

The ability of most modern operating systems to handle varying input and output devices for data was not foreseen in the original definition of LISP. Standard LISP therefore includes the following functions to handle such file management. It is assumed that in each system there is a standard input and output device initially offered to the user.

open [u; INPUT]	Declares the file u as an input file.
open[u;OUTPUT]	Declares the file u as an output file.
rds[u]	Declares that all input now comes from the file u. If u is NIL, then the standard input device is selected. If u is not NIL, then u must have previously been declared through an OPEN statement as an input file.
wrs[u]	Declares that all output now goes to the file u. If u is NIL, then the standard output device is selected. If u is not NIL, then u must have been previously declared through an OPEN statement as an output file.
close [u]	Closes the file u. If u is an output file, then the necessary end-of-file marks are written.

The form of the argument u for these functions will vary from system to system. In some, it will be an atom specifying the name of a file in some unique manner. In others it will be a list giving details of device, filename, project/programmer numbers, etc. However, the differences should not affect the design of programs or their calling sequences.

2.7 User-defined Functions and Macros

Most LISP systems provide the user with a function DEFINE for introducing function definitions into the system. However, the action of DEFINE varies markedly from system to system. In SHARE LISP-like systems, the definition is added to the property list of the function name with an indicator EXPR. Other systems use CAR of the function name as a pointer to the definition and in some cases compile the function directly into the system. A Standard LISP function DEFINE is also available for introducing user-defined functions into the system, but as its action varies from system to system its definition must be included in the system preprocessor. The Standard LISP DEFINE normally includes a call to the system compiler, but this may be varied as required. In addition, DEFINE calls the preprocessor translator which modifies the function definition before incorporation in the system as we shall explain in detail in Section 3.

Special forms introduced as FEXPRs and FSUBRs may also be defined in Standard LISP. As the preprocessor also modifies the definitions of these forms, a special function DEFEXPR, whose argument has the same form as DEFINE, is provided for introducing them.

The advantages of FEXPRs and FSUBRs are not universally recognized among LISP system designers, and the current tendency is to incorporate a macro defining facility in order to reduce execution time in compiled code. In line with this trend, Standard LISP also includes a function MACRO similar in form to that described by

Weissman (4). As with DEFINE and DEFEXPR, the argument list is modified by the preprocessor and the code is compiled if required. In systems with no macro handling facility the preprocessor can be used to expand any macros found in a function definition at DEFINE time as shown in Appendix A. On the other hand, if the system has no FEXPR or FSUBR facility, such forms can be handled with macros or again by the preprocessor. Consequently, all macros and special forms must be in the system before defining any function which uses them.

Because function definitions are stored differently from system to system, Standard LISP includes a function GETD for retrieving a function definition when required. With interpreted code, GETD returns a pointer to the S-expression definition of the function. If the function has been compiled, GETD still returns a non-NIL value so that a test for an existing function definition can be made, but it is not possible to interpret this value in Standard LISP.

2.8 Array Handling

Many LISP systems recently developed allow for the definition of arrays in which the elements may be numbers as well as pointers to S-expressions. In order to maintain our downwards compatibility with SHARE LISP, however, Standard LISP restricts arrays to the LIST array described in the LISP 1.5 Programmer's Manual. Consequently the word LIST in the SHARE LISP array declaration is redundant, and is omitted in Standard LISP. Thus ARRAY is a function of one argument which is a list of arrays to be declared. Each item is a list containing the name of the array and its dimensions, an example being

```
    array [((ALPHA (7 10)) ((BETA (3 4 5))]
```

After ARRAY has been executed, the arrays declared exist and their elements are all set to NIL. Indices always range from 0 to n-1.

On the other hand, Standard LISP provides two distinct functions for setting array elements and getting their values. SETEL is a function of two arguments, the first a list consisting of the array name and the relevant coordinates, and the second the value of the element. Similarly, GETEL is a function of one argument which returns the value of an element. For example, to set the (3,4) element of the array ALPHA to A, we write

```
     setel [(ALPHA 3 4) ; A]
```

and to get the value of the (0,1,2) element of BETA we write

```
     getel [(BETA 0 1 2)]
```

2.9 Standard LISP Program Restrictions

In order to achieve compatibility with as many systems as possible, the following additional restrictions and features must be borne in mind when writing programs in Standard LISP.

(i) The programmer is talking to EVALQUOTE rather than EVAL. Moreover, there are no OVERLORD cards in the SHARE LISP sense, so that a special reading function may be required in some systems. A suitable function to do this is given in Appendix C.

(ii) Literal atoms are limited to 24 characters. Decimal integers are restricted to 7 digits ($\pm 2^{23}$), floating point numbers to 8 digits plus an optional signed or unsigned exponent less in magnitude than 37. Numbers relating to other bases are not allowed explicitly and must be included as constants.

(iii) Functions may not have more than five arguments and be compiled in PDP LISP.

(iv) A GO statement may only occur at the top level of a PROG or as one of the value parts in a top level COND statement within a PROG (otherwise the function will not compile in the 7090 and LISP/360 systems).

(v) The atom NIL represents falsity; F is not available for this purpose. Furthermore, all S-expressions other than NIL are considered to be true in predicate tests.

(vi) A COND form must terminate with a pair (T <form>) unless the COND occurs within a PROG form.

(vii) Free variables should not have the same name as a function in the program (otherwise some compilers will fail).

(viii) No atom may have more than 24 characters.

(ix) Because the Standard LISP function DEFINE includes a call to the preprocessor, all FEXPRS and MACROS must be in the system before defining any function which uses them.

(x) LABEL is not defined in Standard LISP.

SECTION 3. THE STANDARD LISP PREPROCESSOR

3.1 Preliminary

The Standard LISP preprocessor is a LISP program which is responsible for modifying any functions or MACROS defined in Standard LISP so that they conform with the particular properties of the system in which they are being assembled. The preprocessor is written directly in the particular LISP language under consideration and we cannot therefore describe in detail its form for each system. However, the general characteristics remain the same and we shall describe them here.

The preprocessor divides naturally into three parts as follows:

(i) The translator, which modifies the S-expression definitions of functions before they are compiled or interpreted.

(ii) Definition of functions in Standard LISP not implemented in the particular system.

(iii) A user supplied initialization section where free variables are declared.

3.2 The Translator

The Standard LISP translator is a recursive function TRANS which modifies the code of any function or macro introduced into the system. TRANS takes two arguments, the first being the expression to be modified and the second NIL if the function is to be compiled and T if not. The code for the LISP/360 translator is given in Appendix A. This code is intended only as a guide for writing preprocessors for other systems, but in most cases it will not require revision except as indicated below.

In order to use the function TRANS, it is necessary to define in the preprocessor the functions DEFINE, DEFEXPR and MACRO. These functions call an auxiliary function DEF1 which performs four separate actions on each definition considered.

(i) It checks each function name for a flag LOSE. If the flag is found, it does not introduce the function into the system. In this way any functions not needed in a particular implementation of a large system may be excluded without modifying the main program.

(ii) If the function has no such flag, DEF1 now checks for the existence of a functional property indicator. If it finds one, a message

(***** <function name> REDEFINED)

is printed. This is very useful in checking for conflicts between LISP system functions and Standard LISP functions. If a conflict is found, the relevant function can be renamed using the NEWNAM mechanism described below.

(iii) It now applies the function TRANS to the function name and its definition. The explicit action of TRANS is described below.

(iv) The function is now compiled into the system, using in most systems the equivalent of the function COM1 of the SHARE LISP compiler, or interpreted if the function has the flag NOCOMP on its property list or a global variable NOCMP is true. <u>This code may need modification in some systems</u>. In LISP/360, COM1 takes three arguments, the first the function name, the second the definition of the function if it is an EXPR (or NIL) and the third the definition if it is an FEXPR (or NIL).

The necessary flags for functions which are not to be included in an assembly or are not to be compiled are added by the functions LOSE and NOCOMP respectively. Each of these functions takes a list of function names as argument. Their definitions are given in Appendix A, and their use is obvious.

As an alternative to flagging those functions which are not to be compiled, it may sometimes prove useful to interpret a whole batch of functions. In this case, the global variable NOCMP can be turned on and off by the function NOCOM also defined in Appendix A.

After all functions have been considered in a given DEFINE or MACRO statement, a list of those functions which have been compiled into the system is returned (excluding those 'lost').

3.3 Modifications to Code Performed by TRANS

Besides replacing any constants declared by CONSTANT with the corresponding quoted values as described in Section 2.2, and expanding any FEXPRs or macros in systems without facilities for handling them, TRANS performs two other types of code modification on any definition given to it. These modifications are declared in advance by the functions NEWNAM and NEWFORM.

NEWNAM, like CONSTANT, takes a list of pairs of atoms which it gives to DEFLIST with the indicator NEWNAM. If TRANS meets any atoms with this property in a functional position, it replaces them by the corresponding value. This mechanism is useful for two purposes:

(i) To rename those functions in a Standard LISP program whose names conflict with LISP system functions. In some cases, however, it will be necessary to define the renamed Standard LISP function in terms of the function with the old name in the LISP system. This can be done using the NEWFORM mechanism described below, or by renaming the functions with the NEWNAM mechanism and then defining them without applying the translator to their definition. An example of this is the function

EXPLODE in the LISP/360 Preprocessor in Appendix A.

(ii) To rename those functions in a Standard LISP program whose definitions coincide with system functions of a different name. Examples of this may be found in the call of NEWNAM in the LISP/360 Preprocessor.

NEWFORM takes a list of pairs of atoms and lambda functional definitions which it gives to DEFLIST with the indicator NEWFORM. When TRANS meets an atom in a functional position with such a property it modifies its translated arguments according to the prescription given by the lambda expression. This mechanism again has two uses:

(i) To redefine Standard LISP functions in terms of LISP system functions whose definitions (and maybe even name) coincide, but the arguments are in a different order. For example, PDP LISP contains a function PUTPROP which is the same as the Standard LISP function PUT except that the second and third arguments are in the opposite order. So in assembling Standard LISP programs in this system, NEWFORM has an entry

```
(PUT  (LAMBDA  (U  V  W)  (PUTPROP  U  W  V)))
```

We note, however, that this translation will not be correct if a form depends upon the order of its arguments for its effect. For example, the Standard LISP form

```
(PUT  U  (SETQ  V  (QUOTE NO))  V)
```

would not translate correctly into PDP LISP using the NEWFORM entry above.

(ii) To 'open compile' functions with a short definition in the particular LISP implementation. Examples of this are shown in the LISP/360 preprocessor.

If the second argument of TRANS is true, indicating that the function is not to be compiled, then the translator checks for occurrences of special variables appearing alone or as the first argument of SETQ. In the former cases, it replaces the code by a call to PTS and in the latter a call to GTS. In some systems, however, it may be easier to declare such variables as COMMON and use CSETQ to set their values. A check for common variables is therefore included in the definition of TRANS for illustrative purposes.

3.4 Translator Associated Functions

In addition to the functions described above, references are made to the functions SUBLIS, PAIR and DEFLIST defined in Standard LISP. If these functions are not present in a given system, their definitions must be included with the translator, as with SUBLIS in the LISP/360 Translator in Appendix A.

3.5 Standard LISP Functions Not in System

No LISP system yet written contains all Standard LISP functions exactly as we have defined them, and it will therefore be necessary to include such definitions in the preprocessor, either as a function definition compiled directly into the system without using TRANS, or by using the NEWNAM and NEWFORM mechanisms described earlier. Examples of this are shown in Appendix A.

3.6 User-supplied Preprocessor Section

The final section of the preprocessor initializes a user's free variables and must be supplied by the user for a given program. This section will contain the table of constants which the programmer uses as a call to CONSTANT, and also any necessary declaration of other free variables which the system requires. These latter declarations cannot form part of the Standard LISP program, as the declaration mechanism varies from system to system. However, Standard LISP does allow use of a function SPECIAL to declare any free variables in function definitions generated by a user's program so that these functions may be compiled at the time of definition if required.

3.7 Program Translator

Because Standard LISP provides only a subset of the functions available in any given LISP system, it is not possible to automate completely the translation of any given LISP program into Standard LISP. However, it is usually only the system functions which deal with machine dependent properties which cannot be translated so that most programs can be converted fairly readily.

A program translator from PDP LISP to Standard LISP for example has been designed on similar lines to the Standard LISP preprocessor described in this Section. Apart from using reversed entries in the NEWNAM and NEWFORM tables, the program translator checks for three particular features in the PDP LISP program being converted.

(i) Any quoted expressions are checked for characters which are neither letters or digits. If any are found, the quoted expression is replaced by a generated symbol.

(ii) Any forms containing functions with extended properties are converted by explicit routines designed for these functions. For example, a COND form can have more than one consequent in each term in PDP LISP, so that each COND expression must be checked and changed when necessary.

(iii) If a function is found which is not translatable, the form in which it occurs is replaced by a generated symbol.

Whenever replacement by a generated symbol occurs, the programmer is informed of this on his console. In addition, the generated symbol and its equivalent are placed on a table which is printed at the end of the translation. This table is searched for the prior existence of a non-translatable form before a new symbol is generated. In case (i) above, the generated symbol and its equivalent become CONSTANT entries in the Standard LISP preprocessor. In case (iii), the symbols can be replaced (using an editing program) by whatever Standard LISP form the programmer decides upon.

Program translators will be constructed for other systems as the need arises.

CONCLUSION

The formulation of LISP presented in this paper offers the user a language which may be run on most machines with a minimum of fuss. It has been successfully used to run one very large LISP program[5] in five different LISP systems without difficulty. We hope that this attempt at standardization will also have some influence on the design of new LISP systems and make easier the description of LISP algorithms and programs in the literature.

REFERENCES

1. John McCarthy, Comm of the ACM, 3, 184 (1960).

2. John McCarthy, Paul W. Abrahams, Daniel J. Edwards, Timothy P. Hart, Michael I. Levin, LISP 1.5 Programmer's Manual, MIT Press, 1965.

3. Robert Saunders, LISP - On the Programming System, in The Programming Language LISP: Its Operation and Applications, edited by E. C. Berkeley and D. G. Bobrow, MIT Press, (1964).

4. Clark Weissman, LISP 1.5 Primer, Dickenson, 1967.

5. Anthony C. Hearn, REDUCE - A User Oriented Interactive System for Algebraic Simplification, in Interactive Systems for Experimental Applied Mathematics, (Academic Press, New York, 1968).

APPENDIX A

```
*** T H E   S T A N D A R D   L I S P   P R E P R O C E S S O R ***
***                     F O R   L I S P / 3 6 0                    ***
```

```
DEFLIST (((COMMENT (LAMBDA (U A) NIL))) FEXPR)

COMMENT (STANDARD LISP TRANSLATOR)

SPECIAL ((NOCMP))

(LAMBDA (U) (COMPILE (DEFINE U))) ((

(DEFINE (LAMBDA (U)
    (DEF1 U (QUOTE EXPR))))

(DEFEXPR (LAMBDA (U)
    (DEF1 U (QUOTE FEXPR))))

(MACRO (LAMBDA (U)
    (DEF1 U (QUOTE MACRO))))

(DEF1 (LAMBDA (U V)
    (PROG (W X Y Z)
    A  (COND ((NULL U) (RETURN Y)))
       (SETQ X (CAAR U))
       (COND ((FLAGP X (QUOTE LOSE)) (GO B))
             ((GETD X)
                (PRINT (LIST (QUOTE *****) X (QUOTE REDEFINED)))))
       (SETQ Y (NCONC Y (LIST X)))
       (SETQ W (OR NOCMP (FLAGP X (QUOTE NOCOMP))))
       (SETQ X (LIST (TRANS X NIL) (TRANS (CADAR U) NOCOMP)))
       (COND ((OR W (EQ V (QUOTE MACRO))) (DEFLIST (LIST X) V))
             (T (COM1 (CAR X)
                      (AND (EQ V (QUOTE EXPR)) (CADR X))
                      (AND (EQ V (QUOTE FEXPR)) (CADR X))))))
    B  (SETQ U (CDR U))
       (GO A))))

(TRANS (LAMBDA (U V)
    (COND ((NULL U) NIL)
          ((ATOM U) (COND ((NUMBERP U) U)
                          ((AND V (GET U (QUOTE SPECIAL)))
                             (LIST (QUOTE GTS) (LIST (QUOTE QUOTE) U)))
                          (T ((LAMBDA (X)
                             (COND (X (LIST (QUOTE QUOTE) X))
                                   (T ((LAMBDA (Y)
                                        (COND (Y Y) (T U)))
                                       (GET U (QUOTE NEWNAM))))))
                             (GET U (QUOTE CONSTANT))))))
          ((ATOM (CAR U))
             (COND ((EQ (CAR U) (QUOTE QUOTE)) U)
                   ((AND V (EQ (CAR U) (QUOTE SETQ)))
                      (APPEND (COND ((GET (CADR U) (QUOTE SPECIAL))
                                     (LIST (QUOTE PTS)
                                       (LIST (QUOTE QUOTE) (CADR U))))
```

```
                                        ((FLAGP (CADR U) (QUOTE COMMON))
                                         (LIST (QUOTE CSETQ) (CADR U)))
                                        (T (LIST (QUOTE SETQ) (CADR U))))
                                  (TRANS (CADDR U) T)))
                        (T (PROG (X)
                          (RETURN (COND
                             ((SETQ X (GET (CAR U) (QUOTE NEWFORM)))
                              (SUBLIS (PAIR (CADR X) (MAPTR (CDR U) V))
                                      (CADDR X)))
                             ((SETQ X (GET (CAR U) (QUOTE NEWNAM)))
                              (CONS X (MAPTR (CDR U) V)))
                             ((SETQ X (GET (CAR U) (QUOTE MACRO)))
                              (TRANS (APPLY X (CDR U) NIL)))
                             (T (CONS (CAR U) (MAPTR (CDR U) V)))))))))
                (T (MAPTR U V)))))

(MAPTR (LAMBDA (U V)
    (COND ((ATOM U) (TRANS U V))
          (T (CONS (TRANS (CAR U) V) (MAPTR (CDR U) V))))))

(CONSTANT (LAMBDA (U)
    (DEFLIST U (QUOTE CONSTANT))))

(LOSE (LAMBDA (U)
    (FLAG U (QUOTE LOSE))))

(NEWFORM (LAMBDA (U)
    (DEFLIST U (QUOTE NEWFORM))))

(NEWNAM (LAMBDA (U)
    (DEFLIST U (QUOTE NEWNAM))))

(NOCOMP (LAMBDA (U)
    (FLAG U (QUOTE NOCOMP))))

(NOCOM (LAMBDA (U)
    (SETQ NOCMP U)))
))

UNSPECIAL ((NOCMP))

(LAMBDA (U) (COMPILE (DEFLIST U (QUOTE EXPR)))) ((

(GETD (LAMBDA (U)
    (OR (GET U (QUOTE EXPR))
        (GET U (QUOTE FEXPR))
        (GET U (QUOTE SUBR))
        (GET U (QUOTE FSUBR))
        (GET U (QUOTE MACRO)))))

(SUBLIS (LAMBDA (U V)
    (COND ((NULL U) V)
          (T ((LAMBDA (X)
               (COND (X (CDR X))
                     ((ATOM V) V)
                     (T (CONS (SUBLIS U (CAR V)) (SUBLIS U (CDR V))))))
             (SASSOC V U NIL))))))
))
```

```
CONSTANT ((
     (BLANK $$$ $)
     (COMMA $$$,$)
     (DOLLAR $$/$/)
     (LPAR $$$($)
     (RPAR $$$)$)
     (STAR $$$*$)
))

NEWNAM ((
     (DIGIT DIGP)
     (EXPLODE *EXPLODE)
     (LITER LETP)
     (OPEN *OPEN)
     (OTLL *OTLL)
     (PRINC PRIN1)
     (RDS *RDS)
     (SPACES XTAB)
     (WRS *WRS)
     (*FUNCTION FUNCTION)
))

NEWFORM ((
     (*APPLY (LAMBDA (U V) (APPLY U V ALIST)))
     (CAAAAR (LAMBDA (U) (CAAR (CAAR U))))
     (CAAADR (LAMBDA (U) (CAAR (CADR U))))
     (CAADAR (LAMBDA (U) (CAAR (CDAR U))))
     (CAADDR (LAMBDA (U) (CAAR (CDDR U))))
     (CADAAR (LAMBDA (U) (CADR (CAAR U))))
     (CADADR (LAMBDA (U) (CADR (CADR U))))
     (CADDAR (LAMBDA (U) (CADR (CDAR U))))
     (CADDDR (LAMBDA (U) (CADR (CDDR U))))
     (CDAAAR (LAMBDA (U) (CDAR (CAAR U))))
     (CDAADR (LAMBDA (U) (CDAR (CADR U))))
     (CDADAR (LAMBDA (U) (CDAR (CDAR U))))
     (CDADDR (LAMBDA (U) (CDAR (CDDR U))))
     (CDDAAR (LAMBDA (U) (CDDR (CAAR U))))
     (CDDADR (LAMBDA (U) (CDDR (CADR U))))
     (CDDDAR (LAMBDA (U) (CDDR (CDAR U))))
     (CDDDDR (LAMBDA (U) (CDDR (CDDR U))))
     (DIVIDE (LAMBDA (U V) (CONS (QUOTIENT U V) (REMAINDER U V))))
     (ERRORSET (LAMBDA (U V) (LIST U)))
     (PUT (LAMBDA (U V W) (PROG2 (DEFLIST (LIST (LIST U W)) V) W)))
     (GENSYM (LAMBDA NIL (GENSYM1 (QUOTE $$$   G$))))
     (ONEP (LAMBDA (N) (EQUAL N 1)))
     (READCH (LAMBDA NIL (READCH NIL)))
));

COMMENT (STANDARD LISP FUNCTIONS DEFINED IN TERMS OF SYSTEM
         FUNCTIONS OF THE SAME NAME)

COMMENT (THE FOLLOWING LIST IS USED BY EXPLODN1 DEFINED BELOW)

DEFLIST (((NASL (((0 . $$$0$) (1 . $$$1$) (2 . $$$2$) (3 . $$$3$)
                  (4 . $$$4$) (5 . $$$5$) (6 . $$$6$) (7 . $$$7$)
                  (8 . $$$8$) (9 . $$$9$))))) SPECIAL)
```

```
(LAMBDA (U) (COMPILE (DEFLIST U (QUOTE EXPR)))) ((

(*EXPLODE (LAMBDA (U)
    (COND ((NUMBERP U) (EXPLODN U))
          (T (EXPLODE U)))))

(EXPLODN (LAMBDA (U)
    (COND ((ZEROP U) (LIST (QUOTE $$$0$)))
          ((NOT (FIXP U)) (ERROR (LIST (QUOTE EXPLODN) U)))
          ((MINUSP U) (CONS (QUOTE $$$-$) (EXPLODN (MINUS U))))
          (T (EXPLODN1 U)))))

(EXPLODN1 (LAMBDA (U)
    (PROG (X Y Z)
    A  (COND ((ZEROP U) (RETURN Z)))
       (SETQ X (REMAINDER U 10))
       (SETQ Y NASL)
    B  (COND ((EQUAL X (CAAR Y)) (GO C)))
       (SETQ Y (CDR Y))
       (GO B)
    C  (SETQ Z (CONS (CDAR Y) Z))
       (SETQ U (QUOTIENT U 10))
       (GO A))))

(*OPEN (LAMBDA (U V)
    (PROG2 (OPEN U (QUOTE ((LRECL . 80) (BLKSIZE . 80))) V)
           U)))

(*RDS (LAMBDA (U)
    (COND ((NULL U) (RDS (QUOTE LISPIN)))
          (T (RDS U)))))

(*WRS (LAMBDA (U)
    (COND ((NULL U) (WRS (QUOTE LISPOUT)))
          (T (PROG NIL (OTLL 72) (ASA NIL) (WRS U))))))
))

UNSPECIAL ((NASL))

COMMENT (STANDARD LISP FUNCTIONS NOT DEFINED IN LISP 360)

DEFINE ((

(COPY (LAMBDA (U)
    (COND ((ATOM U) U)
          (T (CONS (COPY (CAR U)) (COPY (CDR U)))))))

(COMPRESS (LAMBDA (U)
    (PROG2 (COND ((DIGIT (CAR U)) (MAPCAR U (FUNCTION RNUMB)))
                 (T (MAPCAR U (FUNCTION RLIT))))
           (MKATOM))))

(GTS (LAMBDA (U)
    ((LAMBDA (X)
       (COND ((NULL X) (ERROR (LIST (QUOTE GTS) U)))
             (T (CAR X))))
    (GET U (QUOTE SPECIAL)))))
(PTS (LAMBDA (U V)
    (CAR ((LAMBDA (X) (COND ((NULL X) (PUT U (QUOTE SPECIAL) (LIST V)))
                            (T (RPLACA X V))))
         (GET U (QUOTE SPECIAL))))))

(MAP (LAMBDA (U PI)
    (PROG NIL
    A  (COND ((NULL U) (RETURN NIL)))
       (PI U)
       (SETQ U (CDR U))
       (GO A))))

(MAPCON (LAMBDA (U PI)
    (COND ((NULL U) NIL)
          (T (NCONC (PI U) (MAPCON (CDR U) PI))))))

(REVERSE (LAMBDA (U)
    (PROG (V)
    A  (COND ((NULL U) (RETURN V)))
       (SETQ V (CONS (CAR U) V))
       (SETQ U (CDR U))
       (GO A))))

(SUBST (LAMBDA (U V W)
    (COND ((EQUAL V W) U)
          ((ATOM W) W)
          (T (CONS (SUBST U V (CAR W)) (SUBST U V (CDR W)))))))

(*EVAL (LAMBDA (U)
    (EVAL U ALIST)))
))

DEFEXPR ((

(PROGN (LAMBDA (U A)
    (PROG NIL
    A  (COND ((NULL (CDR U)) (RETURN (EVAL (CAR U) NIL))))
       (EVAL (CAR U) NIL)
       (SETQ U (CDR U))
       (GO A))))
))


COMMENT (THE FUNCTIONS POS AND *OTLL AND THE ARRAY FUNCTIONS ARE
         DEFINED IN LAP AND NOT INCLUDED IN THIS LISTING)


COMMENT (E N D   O F   L I S P   3 6 0   P R E P R O C E S S O R)
```

APPENDIX B

THE FOLLOWING FUNCTIONS DEFINED IN THE LISP 1.5 PROGRAMMER'S MANUAL ARE AVAILABLE IN STANDARD LISP

ADD1	LENGTH	PRIN1
AND	LESSP	PRINT
APPEND	LIST	PROG
ATOM	LITER	PROG2
CAR...CDDDDR	LOGAND	QUOTE
COND	LOGOR	QUOTIENT
CONS	LOGXOR	READ
COPY	MAP	RECIP
DEFLIST	MAPCON	REMAINDER
DIFFERENCE	MAPLIST	REMFLAG
DIGIT	MAX	REMPROP
EQ	MEMBER	RETURN
EQUAL	MIN	REVERSE
ERROR	MINUS	RPLACA
EXPT	MINUSP	RPLACD
FIXP	NCONC	SASSOC
FLAG	NOT	SUB1
FLOATP	NULL	SUBLIS
GET	NUMBERP	SUBST
GO	ONEP	TERPRI
GREATERP	OR	TIMES
LEFTSHIFT	PAIR	ZEROP
	PLUS	

The following functions defined in Standard LISP have the same names but different properties from the LISP 1.5 Programmer's Manual definitions. These changes are necessary in order to provide for maximum compatibility with other system definitions.

array [u]	initializes arrays. Same definition as for SHARE LISP except that 'LIST' is omitted.
define [u]	introduces function definitions into system. Returns a list of function names introduced.
divide [u;v]	returns cons [quotient [u;v] ; remainder [u;v]] where u and v are integers.
errorset [u;v]	if an error occurs during the evaluation of u, NIL is returned. The error message is

	printed if and only if v is T. If no error occurs, list [u] is returned. In systems which lack this facility, errorset [u;v] may be replaced by list [u], but no error recovery is then possible except at the top level.
function[u]	defines a quoted functional argument
gensym []	creates a new atom (e.g., GØ123), adds it to the OBLIST if one exists and returns atom.
setq [u;v]	sets the PROG or free variable u to the value of v. Returns value of v.
special [u]	initializes the list of u of free variables for compilation. Returns u.

The following additional functions are defined in Standard LISP:

close [filename]	closes filename by writing appropriate end-of-file marks, etc. Returns filename.
compress [u]	creates an atom (literal atom or number) from list of characters u, adds it to the OBLIST if one exists, and returns the atom.
defexpr [u]	introduces special form definitions (as FEXPRs or FSUBRs) into system. Returns a list of function names introduced.
delete [u;v]	deletes the first top level occurrence of the S-expression u from the list v.
explode [u]	returns a list of the constituent characters of atom u. Must also work for integers. e.g. explode [123] = (1 2 3).

fix [u]	returns integer part of number u.
flagp [u;v]	returns T if atom u has flag v on its property list, otherwise NIL.
float [u]	Returns floating point equivalent of number u.
getd [u]	returns pointer to definition of u, if u is a function or macro, and NIL otherwise.
getel [[u;m_1,..,m_n]]	returns (m_1,...,m_n) element of array named u.
gts [u]	returns ('gets') the value of the special atom u.
macro [u]	introduces macro definitions into system. Returns a list of macro names introduced.
mapcar [u;fn]	if null [u] then NIL else cons [fn[car[u]]; mapcar [cdr [u];fn]].
open [filename; stat]	opens a file named filename on some external device. stat = INPUT or OUTPUT. Returns filename.
princ [u]	adds character string u to output buffer. Returns u.
pos []	returns number of characters presently in output buffer. Initialized to zero at every call of TERPRI.
progn [u_1,...,u_n]	evaluates forms u_1,...,u_n and returns value of u_n.
pts [u;v]	u is declared special if not already. PTS then gives u the special value v and returns v.

put [u; ind; prop]	puts prop on property list of u with indicator ind. Returns prop.
rds [filename]	selects filename as input device. All input now comes from filename. Returns filename.
setel [[u;m_1,...,m_n]; v]	sets (m_1,...,m_n) element of array named u to v. Returns v.
spaces [n]	adds n spaces to output buffer. Returns n.
wrs [filename]	selects filename as output device. All output now goes to filename. Returns filename.
*apply [u;v]	applies function u to list of arguments v, and returns result.
*eval [u]	evaluates expression u (using current ALIST if one exists) and returns value.
*function [u]	defines a functional argument whose form may change during evaluation.

APPENDIX C

The following function may be used to assemble a Standard LISP program in systems with OVERLORD directives or EVAL listen loops. It is assumed that the atom STOP occurs at the end of the program.

```
(SREAD (LAMBDA NIL
 (PROG (X Y)
  A     (TERPRI)
        (COND
            ((NULL (SETQ X (ERRORSET (READ) T))) (GO ERR))
            ((EQ (SETQ X (CAR X)) (QUOTE STOP)) (GO B))
            ((OR (NULL (SETQ Y (ERRORSET (READ) T)))
                 (EQ (SETQ Y (CAR Y)) (QUOTE STOP ))) (GO ERR)))
        (PRINC (QUOTE $$$FUNCTION... $))
        (PRINT X)
        (TERPRI)
        (TERPRI)
        (PRINC (QUOTE $$$ARGUMENTS... $))
        (PRINT Y)
        (TERPRI)
        (TERPRI)
        (SETQ Y (ERRORSET (*APPLY X Y) T))
        (COND ((NULL Y) (GO A)))
        (TERPRI)
        (PRINC (QUOTE $$$VALUE... $))
        (PRINT (CAR Y))
        (GO A)
ERR     (TERPRI)
        (PRINC (QUOTE $$$READ ERROR$))
 B      (RETURN (QUOTE $$$*$)))))
```