LISP 1.5 AND ITS IMPLEMENTATION ON THE IBM SYSTEM/360 AT RPI

by

Jonathan K. Millen

and

Jack P. Gelb

Rensselaer Polytechnic Institute

Troy, New York

LISP 1.5 AND ITS IMPLEMENTATION
ON THE IBM SYSTEM/360 AT RPI

by

Jonathan K. Millen

and

Jack P. Gelb

First edition published January, 1969

Second edition published January, 1970

Additional copies of this manual may be purchased by contacting:
Student Union Bookstore
Rensselaer Polytechnic Institute
Troy, New York 12181

January 8, 1970
RPI Computer Laboratory
Pub. # PAD-10

PREFACE TO THE SECOND EDITION

And, thus, is the torch called LISP passed from hand to hand to hand to ... ! This new edition of the LISP manual attempts to correct those errors, present in the original version, which were gleefully (perhaps too gleefully) pointed out (with large fingers) to the authors. Also worth noting are the increased scope of the PROGram feature through the addition of SETQ and modifications to COND; the improved error-checking and error messages for EXPLODE, IMPLODE, DEFLIST, PRINT, and PRIN1; and the availability, under OS, of several debugging aids and auxiliary programs. Most of these improvements are the work of Jonathan Millen, to whom this author remains indebted.

J. P. G.
January 8, 1970

PREFACE

The original LISP system was the brain-child of Professor John McCarthy. It was completed in 1960 at M.I.T. Programmed first for the IBM 7090, LISP has spawned versions for the IBM 7094, PDP-1, PDP-6, and IBM System/360.

The R.P.I. LISP interpreter for the System/360/50 was written in 1965 by William Lehrman for the degree of Master of Electrical Engineering. The garbage collector, arithmetic features, and print program were written by Jonathan Millen. The compiler was written by Jack Gelb and Jonathan Millen, the authors of several lesser functions and this manual. Mel Sabel wrote the self-relocation routine and the I/O modules to allow LISP to operate through teletypes.

The authors gratefully acknowledge the encouragement and advice of Professors Dean Arden and Jack Hollingsworth of R.P.I.

J. K. M.
J. P. G.
January 10, 1969

TABLE OF CONTENTS

1. INTRODUCTION

This manual has a twofold purpose: to provide an elementary introduction to the LISP 1.5 language, and to serve as a specification manual for the implementation of LISP on the R.P.I. System/360/50. For a more extensive explanation of LISP 1.5, and information on other implementations, see references (1, 2).

At R.P.I., LISP is currently available as a self-relocating program under the Disk Operating System (DOS). LISP programs may be run in batch (background) mode through card input, or in foreground via the the teletypes. In background, LISP is catalogued in a similar manner to FORTRAN, COBOL, and the assembler, and programs are executed following standard DOS conventions. In foreground, LISP is invoked through the CONVERSE teletype monitoring program (in-house). An Operating System (OS) version of LISP is also available.

The language LISP has proved extremely valuable in fields in which computer processing of list-structured data is required. Some areas in which LISP is particularly useful include symbol manipulation, tree searching, graph theory, automata theory (including formal languages and automata simulation), artificial intelligence, and natural language processing. Aside from computer programming, the LISP metalanguage has also been used as a formal language.

For programmers not familiar with list processing techniques, LISP is not an easy language to grasp. It bears little resemblance to the more common programming languages such as FORTRAN and COBOL. The three main features of LISP which, together, distinguish it from these other languages are:

(1) the use of lists and atoms as data

(2) the interpretative nature of the processor

(3) recursive definition of functions.

2. LIST STRUCTURES

2.1 A list structure is an atom or list.[1]

2.2 Atoms

2.2.1 An atom is one of a set of objects in one-to-one correspondence with the set of character strings over a certain alphabet.

The alphabet used (here) consists of all the characters for which there are keys on a keypunch, with the exception of the blank, comma, period, left parenthesis, and right parenthesis.

2.2.2 The string corresponding to a given atom is called its print name.

2.2.3 At present, print names must be 72 characters or less.

2.3 Lists

2.3.1 A list is a finite sequence (n-tuple) of list structures.

2.3.2 The empty sequence (0-tuple) is, in particular, a list. It also happens to be an atom, with the print name 'NIL'.

2.4 S-expressions

2.4.1 An S-expression is a string of characters (written or punched by a programmer) to represent a list structure.

2.4.2 The S-expression representing an atom is its print name.

2.4.3 A list can be represented by: a left parenthesis; followed by the representations of its elements, in order and separated by blanks; followed by a right parenthesis.

2.4.4 Other S-expressions representing the same list structure can be obtained from the one just described by adding or deleting blanks which are adjacent to other blanks or to parentheses.

2.4.5 Blanks and commas are interchangeable.

---

[1] Dotted pairs are allowable list structures, but are not suggested for beginners. See 4.4.5, 9.3, and reference (1).

2.4.6 It is important to remember that placement of parentheses is critical. (A B), for example, is a list of two atoms, while ((A B)) has only one element, namely, (A B). ((A)(B)) has two lists as elements. In LISP,

P A R E N T H E S E S A R E

N E V E R O P T I O N A L.

3. PROCESSING OF LISP PROGRAMS

3.1 A LISP program is a sequence of list structures.

3.2 The LISP processor is a machine language program which computes the value of a list structure, according to definitions and rules enunciated in the remainder of this manual.

3.3 The LISP processor is an interpreter. This means that when each expression in the program is read, it is evaluated before the next one is read.

The processor prints the value of each list structure as it is found.

3.4 Notation

If s is a list structure, we shall, in this manual, write [s] to signify the value of s. (This notation is not recognized by the LISP processor).

4. FUNCTIONS -- PART I

4.1 Functions

4.1.1 Functions are implemented in LISP by lists, and also by machine language subroutines. Either type has zero or more arguments, and returns a value. Some functions also have an effect, such as printing, punching, or internal changes in lists or in the values of atoms.

4.1.2 Many functions are built-in; others must be defined by the programmer. The latter are defined by lists in a manner to be described below. They may later be transformed into machine-language subroutines by the COMPILE function.

4.1.3 An atom is referred to as a function if its print name is the name of a function.

4.1.4 The atom OBLIST is a constant whose value is the object list, which includes all built-in functions. See Appendix I.10.

4.2 Function calls

4.2.1 A function call is a list $(f,a_1,\ldots,a_n)$ where f is a function.

4.2.2 If $(f,a_1,\ldots,a_n)$ is a function call,* $[(f,a_1,\ldots,a_n)]$ is the value returned by the function f when entered with arguments $[a_1],\ldots,[a_n]$.

4.2.3 The LISP processor always evaluates the arguments in a function call in left-to-right order, before entering the function.

4.2.4 The list structures in a LISP program are mostly function calls. Even the definition of functions is accomplished through a function call, on the special function DEFINE.

4.3 The QUOTE operator.

4.3.1 [(QUOTE s)] = s.

4.3.2 The purpose of the QUOTE operator is to avoid the requirement of evaluation of arguments in a function call.

---

* where f is (the name of) a machine language subroutine,

The QUOTE operator is the usual means of introducing data into a LISP program.

QUOTE is used like quotation marks in English. Just as 'LISP' is a four letter word, while LISP is not a four letter word, but rather a programming language, [(QUOTE OBLIST)] is an atom and [OBLIST] is a list of over a hundred elements.

4.4 Three important built-in functions

In the definitions below, assume that [u] = $(a_1,\ldots,a_n)$. Each function is defined by stating its value in a typical function call. (This manner of definition is a convenience used in this manual, but is not recognized by the LISP processor).

4.4.1 [(CAR u)] = a_1

4.4.2 [(CDR u)] = $(a_2,\ldots,a_n)$

4.4.3 If [v] = a_0, then [(CONS v u)] = $(a_o,a_1,\ldots,a_n)$.

4.4.4 Examples

[(CAR (QUOTE (A N D)))] = A

[(CAR (QUOTE ((A B))))] = (A B)

[(CAR (QUOTE A))] is not defined.

[(CDR (QUOTE (A N D)))] = (N D)

[(CDR (QUOTE ((A B))))] = () = NIL

[(CDR (QUOTE ()))] is not defined.

[(CONS (QUOTE A) (QUOTE (N D)))] = (A N D)

[(CONS (QUOTE (A)) (QUOTE (B)))] = ((A) B)

4.4.5 [(CONS (QUOTE A) (QUOTE B))] = (A . B).
This is called a dotted pair, and, although not a list structure as we have defined it, can be used as a data structure, and as an element of a list or another dotted pair.

4.4.6 Frequently a programmer wishes to take a sequence of CAR's and CDR's, such as

(CAR(CDR(CDR(CAR(CDR u))))).

In the R.P.I. LISP system, the programmer may simply write

(CADDADR u)

instead of the long composition; where each 'A' stands for a CAR and each 'D' for a CDR. The system will automatically define the new function. Any combination of up to seven CAR's and CDR's can be abbreviated in this manner.

4.4.7 The names "CAR' and 'CDR' arose from IBM 7090 nomenclature. They stand for "Contents of Address part of Register" and "Contents of Decrement part of Register", respectively.

'CONS' is short for 'Construct.'

5. DEFINE and function definitions

5.1 A <u>function</u> <u>definition</u> is a list of the form:

$$(f\ (\text{LAMBDA}\ (x_1,\ldots,x_n)\ s))$$

where f is a non-numeric atom whose print name is the desired name of the function;

$x_1,\ldots,x_n$ are atoms, called the dummy variables of f;

s is a list structure (usually involving $x_1,\ldots,x_n$) such that it is desired to have

$$[(f,\ x_1,\ldots,x_n)] = [s].$$

5.2 A function definition may not be placed by itself in a LISP program. A definition is brought into effect by a call on the function DEFINE.

5.3 If $d_1,\ldots,d_n$ are function definitions of functions $f_1,\ldots,f_n$, then

$$[(\text{DEFINE}\ (\text{QUOTE}\ (d_1,\ldots,d_n)))] = (f_1,\ldots,f_n).$$

Furthermore, the act of evaluating this function call makes the definitions of the functions $f_1,\ldots,f_n$ known to the system.

5.4 Example

```
[(DEFINE (QUOTE (
   (F (LAMBDA (X Y) (CONS (CAR X) (CDR Y) ) ))
 )))] = (F)

[(F (QUOTE (A B)) (QUOTE (C D E)) )] = (A D E)

[(F (QUOTE (I'M RIGHT)) (QUOTE (HE'S WRONG)) )]
     = (I'M WRONG)
```

5.5 LAMBDA-expressions

5.5.1 A <u>LAMBDA-expression</u> is a list of the form

$$(\text{LAMBDA } (x_1,\ldots,x_n)\ s),$$

where $x_1,\ldots,x_n$ are non-numeric atoms.

5.5.2 A LAMBDA-expression is a function, and may be the first element of a function call.

5.5.3 $[((\text{LAMBDA}(x_1,\ldots,x_n)\ s),a_1,\ldots,a_n)] = [s]$ under the conditions $[x_1] = [a_1],\ldots,[x_n] = [a_n]$.

5.5.4 If a function f was defined by $(f(\text{LAMBDA}(x_1,\ldots,x_n)s))$, then

$$[(f,a_1,\ldots,a_n)] = [((\text{LAMBDA}(x_1,\ldots,x_n)s),a_1,\ldots,a_n)].$$

5.5.5 The binding of each value $[a_i]$ to the corresponding x_i in 5.5.3 is recorded by the processor on an internal list called the <u>association list</u> <u>(a-list)</u>. Such a value takes precedence over any constant value that x_i may have.

6. FUNCTIONS -- PART II

6.1 Boolean expressions

6.1.1 A Boolean expression is a list structure whose value is T (representing truth) of NIL (representing falsity).

6.1.2 [T] = T and [NIL] = NIL. Thus, T and NIL are Boolean expressions.

6.2 Predicates

6.2.1 A predicate is a function which returns a value which is always either T or NIL.

6.2.2 If f is a predicate, the function call $(f,a_1,\ldots,a_n)$ is a Boolean expression.

6.3 Conditional expressions.

6.3.1 A conditional expression is a list structure of the form

$$(\text{COND } (b_1, s_1),\ldots,(b_n, s_n)).$$

6.3.2 Let $b_1,\ldots,b_n$ be Boolean expressions. Then

$$[(\text{COND}(b_1,s_1),\ldots,(b_n,s_n))]$$
$$= \text{if } [b_1] \text{ then } [s_1], \text{ else}$$
$$\text{if } [b_2] \text{ then } [s_2], \text{ else}$$
$$\vdots$$
$$\text{if } [b_n] \text{ then } [s_n].$$

6.3.3 The Boolean expressions are evaluated only until the first true one is hit. At least one is required to be true; to ensure this, it is customary to let b_n = T. Only one of the s_i is evaluated, the one with the first true b_i.

6.4 Some important built-in predicates

6.4.1 [(ATOM u)] = T if [u] is an atom, otherwise NIL.

6.4.2 [(EQUAL u v)] = T if [u] = [v], otherwise NIL.

6.4.3 If [u] or [v] is a non-numeric atom,
[(EQ u v)] = T if [u] = [v], NIL if [u] ≠ [v].

EQ is undefined when [u] and [v] are lists or numbers. Its advantage over EQUAL is in speed.

6.4.4 [(NULL u)] = T if [u] = NIL, NIL otherwise.

i.e., [(NULL u)] = [(EQ u NIL)].

6.4.5 These may be combined with AND, OR, NOT, etc. See Appendix I.3.

6.5 Example

```
[(DEFINE (QUOTE (
   (F (LAMBDA (X Y)
     (COND
       ((EQ (CAR X) (CAR Y))(QUOTE WHOOPEE))
       (T (QUOTE (NO MATCH)))
   )))
 )))] = (F)

[(F (QUOTE (HELP ME))(QUOTE(HIT ME)))] = (NO MATCH)
[(F (QUOTE (ZIP CODE))(QUOTE(ZIP ME UP)))] = WHOOPEE
```

7. THE TECHNIQUE OF RECURSIVE FUNCTION DEFINITION

One naturally wishes to take as much advantage as possible of built-in functions. For example, to construct $(x_2,\ldots,x_n,x_1)$ from [X] = $(x_1,\ldots,x_n)$, it suffices to take the value of

(APPEND (CDR X) (LIST (CAR X))).

We could, then, define a function ROTATELEFT this way:

```
[(DEFINE (QUOTE (
   (ROTATELEFT (LAMBDA (X)
     (APPEND (CDR X) (LIST (CAR X)) )
   ))
 )))] = (ROTATELEFT)

[(ROTATELEFT (QUOTE (X1 X2 X3)))] = (X2 X3 X1).
```

But any interesting program is bound to require a construction or test which cannot be fabricated out of existing functions. Suppose, for example, we wanted a function ROTATERIGHT which would construct from $(x_1,\ldots,x_n)$ the list $(x_n,x_1,\ldots,x_{n-1})$. Such a function requires a recursive definition. (How does one know? Familiarity with the built-in functions).

Here is a recursive definition of ROTATERIGHT:

```
(ROTATERIGHT (LAMBDA (X)
  (COND
    ((NULL (CDR X)) X)
    (T (CONS
         (CAR (ROTATERIGHT (CDR X)))
         (CONS (CAR X) (CDR (ROTATERIGHT (CDR X))))
    ))
)))
```

Rather than try to explain why it works, let us show how it was constructed. It has a simple form, which may be summarized thus:

if shortest case, then easy answer,

else reduce the problem to a shorter case.

The 'else' clause in the example was based on the observation that is not hard to obtain $(x_n,x_1,\ldots,x_{n-1})$

from x_1 (i. e., [(CAR X)]), and $(x_n,x_2,\ldots,x_{n-1})$ (i. e., [(ROTATERIGHT (CDR X))].). This is done by CONSing x_n (that is, [(CAR(ROTATERIGHT(CDR X)))]) to the result of CONSing x_1 onto $(x_2,\ldots,x_{n-1})$.

It would seem that the shortest case is not when [(CDR X)] is NIL but rather when [X] itself is NIL. But if the former test is omitted, then if [X] = (A), say, the 'else' case involves [(CAR(ROTATERIGHT NIL))], which would be [(CAR NIL)], and certainly not desirable (as well as undefined).

Still, it would not hurt to include the case [X] =NIL anyway, since the definition above will not handle it. Thus, there may be more than one 'shortest case', or cases which, for one reason or another (such as efficiency) are accorded individual clauses. There may even be several ways of reducing the problem to a shorter case, depending on the form of the argument.

Including the case [X] = NIL makes the definition:

```
(ROTATERIGHT (LAMBDA (X)
  (COND
    ((NULL X) NIL)
    ((NULL (CDR X)) X)
    (T (CONS
         (CAR (ROTATERIGHT (CDR X)))
         (CONS (CAR X) (CDR (ROTATERIGHT (CDR X))))
    ))
)))
```

The efficiency-minded reader will want to know whether he can avoid evaluating (ROTATERIGHT (CDR X)) twice. Yes: one way is to define another function:

```
(PUTSECOND (LAMBDA (Z Y)
  (CONS (CAR Y) (CONS Z (CDR Y)))  ))
(ROTATERIGHT (LAMBDA (X)
  (COND
    ((NULL X) NIL)
    ((NULL (CDR X)) X)
    (T (PUTSECOND (CAR X) (ROTATERIGHT (CDR X)) ))
)))
```

The other way is to abbreviate the above by making use of a LAMBDA-expression (see 5.5).

Instead of calling PUTSECOND by name, its LAMBDA-expression is used in its place:

```
(ROTATERIGHT(LAMBDA(X)
  (COND
    ((NULL X) NIL)
    ((NULL (CDR X)) X)
    (T ((LAMBDA (Z Y)(CONS(CAR Y)(CONS Z(CDR Y))))
        (CAR X) (ROTATERIGHT (CDR X)) ))
)))
```

8. ARITHMETIC

8.1 Notation

8.1.1 Numbers in the LISP system are atoms, and are treated as constants.

8.1.2 Numbers are 'self-defining'; that is, the value of a number is itself, hence numbers need not be quoted.

8.1.3 Presently, only integers and integer arithmetic are supported by LISP. Positive integers are represented by strings of digits preceded optionally by a plus (+) sign.

Negative integers are represented by strings of digits immediately preceded by a minus (-) sign.

8.1.4 The range of LISP numbers follows the rules for System/360 integers. That is, any integer i must satisfy:

$$-2^{31} \leqslant i \leqslant 2^{31}-1.$$

8.1.5 All arithmetic is performed modulo 2^{31}.

8.2 The LISP arithmetic functions do not check whether their arguments are numbers. When there is a question, the programmer should use the predicate NUMBERP to test.

NUMBERP takes one argument. If the value of the argument is a number, the value of the function is T. Otherwise, the value is NIL.

8.3 Details of the arithmetic functions are given in Appendix I.2.

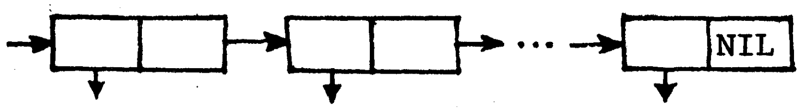
8.4 Examples

```
[(NUMBERP 24)] = T
[(NUMBERP (QUOTE A))] = NIL
[(NUMBERP (QUOTE 24))] = T
[(PLUS 2 4 6 8 10)] = 30
[(PLUSL (LIST 2 4 6 8 10))]= 30
[(QUOTIENT 5 3)] = 1
[(QUOTIENT(TIMES 3 5) 5)] = 3
```

9. IMPLEMENTATION

9.1 Lists

The implementation of a list is as a 'linked list' or chain of double words, pictured below.



The first word of each double word points to the implementation of the corresponding element. The second word points to the remainder of the list (which is empty, or NIL, at the end).

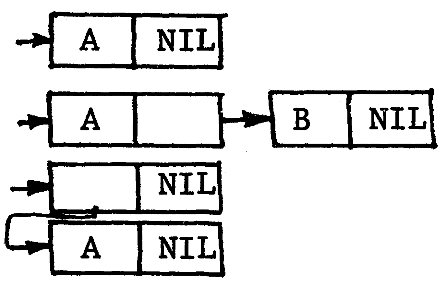
Atoms are also linked lists; see 9.2. In diagrams such as the above, pointers to atoms are often indicated by the print name of the atom.

Examples

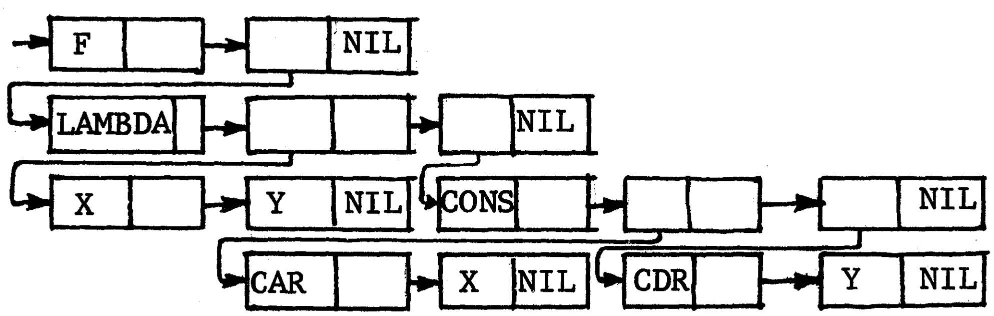
(A)

(A B)

((A))



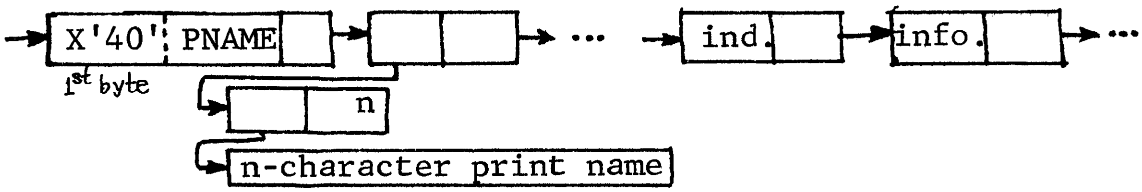
(F (LAMBDA (X Y) (CONS (CAR X) (CDR Y))))



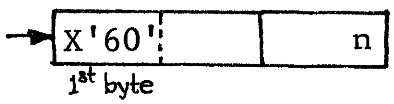
9.2 Atoms and property lists

9.2.1 The implementation of an atom is a linked list, called a <u>property list</u>.

9.2.2 The property list of a non-numeric atom has the form:



9.2.3 The property list of a number has the form:



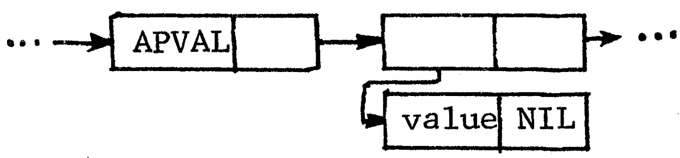
(n fixed-point)

9.2.4 The system indicators are: PNAME, EXPR, FEXPR, SUBR, FSUBR, and APVAL.

The information under EXPR or FEXPR is a pointer to the implementation of a LAMBDA-expression.

The information under SUBR or FSUBR is a pointer to the entry of a machine language subroutine.

The information under APVAL is a constant value, depressed one level as shown:



9.2.5 The basic property-list modification functions are DEFLIST, REMPROP, and GET. Programmers may use DEFLIST to add indicator-information pairs, REMPROP to remove them, and GET to retrieve information under a given indicator.

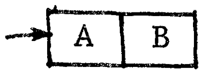
DEFLIST changes the information under an existing indicator, or, if the indicator is not present, inserts the indicator-information pair as the third and fourth elements of the property list.

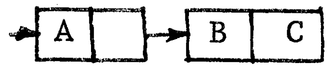
REMPROP removes only the first occurence of an indicator. Note that its value is the information formerly carried under the indicator.

9.3 Dotted pairs

9.3.1 The implementation of a dotted pair is a double word whose first word points to the implementation of the first element, and whose second word points to the implementation of the second element.

9.3.2 Examples

(A . B) 

(A . (B . C)) 

(A B) = (A . (B . NIL)) 

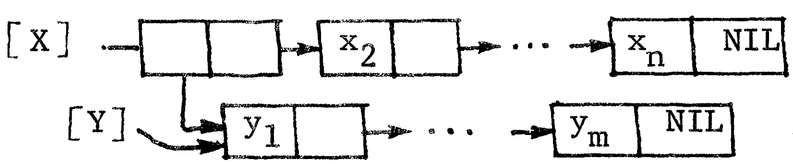
9.4 List-changing functions

The functions RPLACA, RPLACD, NCONC, and EFFACE manipulate list structures in core. Their effects are diagrammed below.

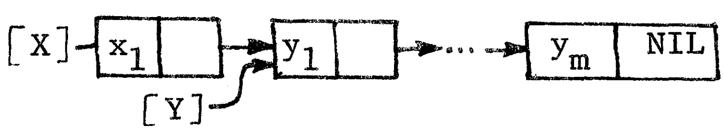
Let [X] = 

and [Y] = 

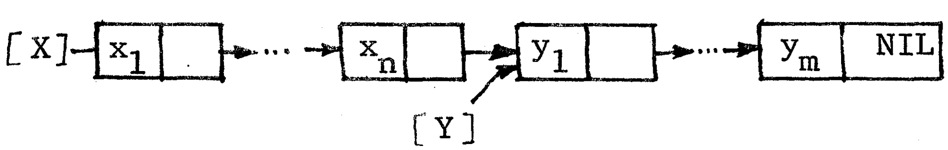
The effect of (RPLACA X Y) is:



The effect of (RPLACD X Y) is:

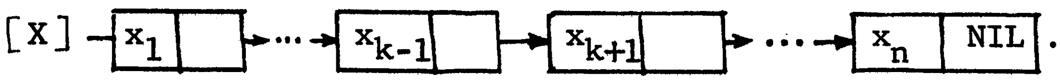


The effect of (NCONC X Y) is:



Let [Z] = x_k (some element of [X]).

Then the effect of (EFFACE Z X) is:



9.5 FEXPR's

If a LAMBDA-expression is placed on the property list of an atom f under the indicator FEXPR, the evaluation of a function call with first element f differs from that in the case of an ordinary EXPR.

9.5.1 The effect of evaluating

```
(DEFLIST (QUOTE (
  (f (LAMBDA (v) s))
)) (QUOTE FEXPR))
```

where f and v are non-numeric atoms,

is that subsequently

$[(f,a_1,\ldots,a_n)] = [s]$ under the condition that

$[v] = (a_1,\ldots,a_n)$.

9.5.2 Motivation for defining FEXPR's:

(1) The programmer chooses which of $a_1,\ldots,a_n$ to evaluate;

(2) Such functions do not have a fixed number of arguments, but can handle any number.

9.5.3 Example

CSETQ is actually an FEXPR. The LAMBDA-expression for CSETQ is:

```
(LAMBDA (X) (CSET (CAR X) (EVAL (CADR X)) )).
```

10. SYSTEM SIZE LIMITATIONS

10.1 The size of the LISP processor is about 40,000 bytes (not counting FSL or BPS), about a third of which is the compiler.

10.2 The Free Storage List (FSL) is a chain of 'available' double words from which all list structures are built -- including property lists, and lists which have been read in, as well as those constructed by the programmer with CONS (or APPEND, or LIST, etc.). Ultimately, all lists are created through calls to CONS, which detaches one double word from the FSL each time.

The function SIZE can be used to check the number of double words remaining in the FSL at any time.

10.3 The Garbage Collector

Eventually, the FSL may be emptied. If this happens, the next call on CONS initiates a garbage collection. A garbage collection has two phases: the marking phase, and the linear sweep.

The marking phase traces through all list structures currently in use. (Those not in use include top-level lists previously evaluated and their values, plus all lists created in the process.) Each item is marked by turning on the high-order bit of the first word of the double word.

The linear sweep makes one pass through the FSL area, turning off the marking bit on the marked items, and chaining the rest into a new FSL. If everything was marked, i. e., nothing is available for inclusion into the new FSL, a message is printed and the run terminated.

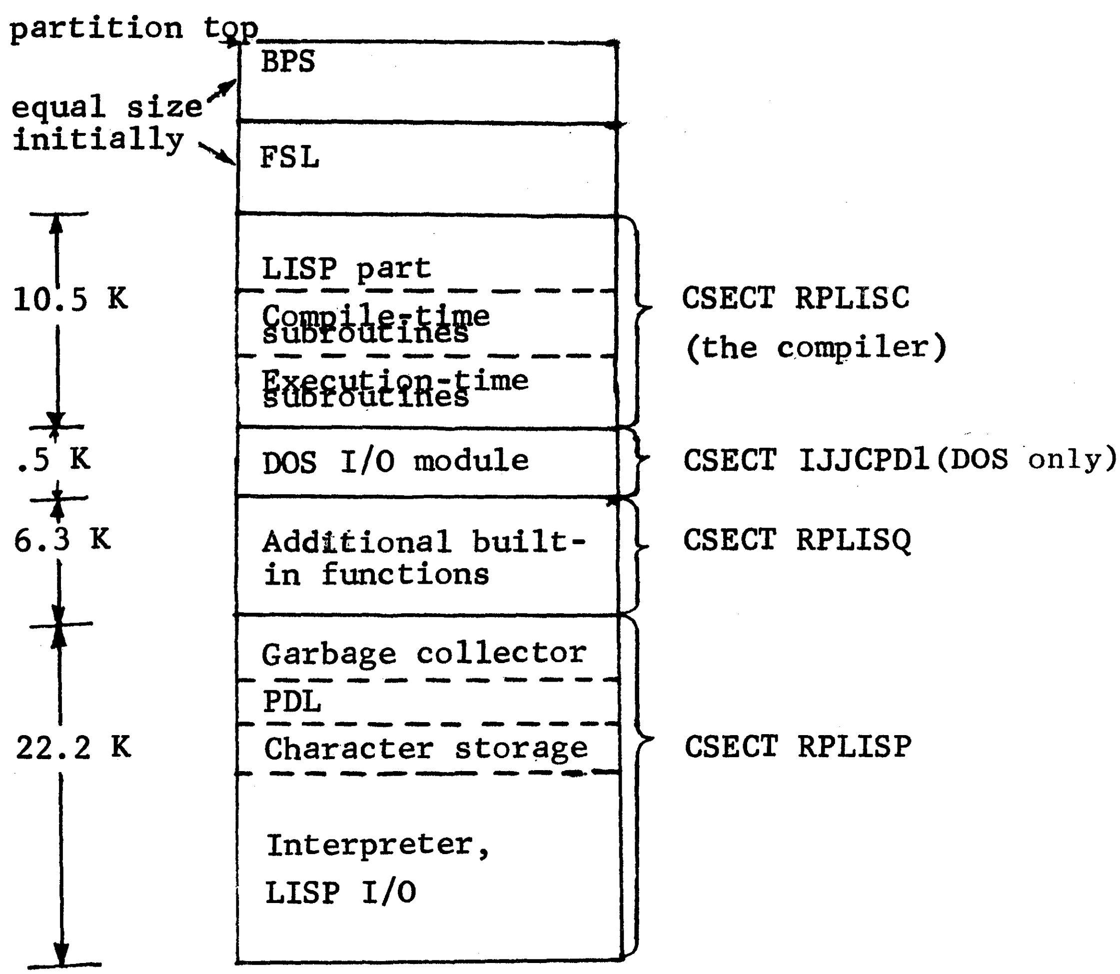
Most of the time taken by a garbage collection is in the marking phase. Garbage collection times range from less than a second to several seconds.

10.4 The push-down list (PDL) is not a list, but a fixed area used as a push-down stack for processing recursion. Its size is currently 1300 words. The size of the push-down stack limits depth of recursion to about 100 for EXPR's and about 1000 for SUBR's (hence, compilation pays in depth of recursion as well as speed).

Exhausting the PDL is an unrecoverable error. If it happens while executing a function which is not debugged, it may be due to a divergent loop in the function.

10.5 The storage area for print names of atoms holds 5000 characters.

10.6 Core map of the LISP system



11. LISP Input-Output

11.1 Normally, the programmer does not have to perform any input or output operations. Data is read in as QUOTEd expressions in the LISP program, and the only output is the final value of a list structure, which is printed automatically.

However, if additional or formatted printing is desired, list structures are to be read under program control, or cards are to be punched, the functions PRINT, PRIN1, TERPRI, EJECT, READ, PUNCH1 and TERPCH are available.

11.2 READ has no arguments. Its value is one list structure read from SYSIPT under DOS, or the SYSIN data set under OS (i.e., the same place as the program), just as the LISP processor would do it. Note that only one list structure is read -- in particular, only one atom (isolated) per line.

11.3 PRINT has one argument, which it prints just as the processor would do it. Its value is its argument.

11.4 EJECT spaces the printer to the top of the next page. It cannot be used in DOS foreground.

11.5 The remaining functions make use of a 100-character common buffer.

11.5.1 TERPRI prints the contents of the buffer and empties it.

11.5.2 TERPCH punches the first 72 characters in the buffer and empties it (producing one card).

11.5.3 PRIN1 and PUNCH1 add the print name of their atomic argument to the buffer, starting at the first unused location. PUNCH1 may not be used in DOS foreground(see 13.).

An overflow (more than 100 characters for PRIN1 in background, 72 for PRIN1 in foreground, 72 for PUNCH1) causes an automatic call on TERPRI or TERPCH, respectively. This is done in such a way that print names of atoms are never split unless they are longer than one line (or card, respectively).

12. The LISP COMPILER

12.1 Introduction

In addition to processing functions interpretively, the RPI LISP system provides for the translation of defined functions directly into machine code. This facility is made available through a built-in package of LISP functions and machine-language subroutines known collectively as the LISP Compiler.

The effect of the LISP Compiler is to transform the LAMBDA-expression of a function into an equivalent machine language subroutine. This compiled code is constructed in an area of core (called Binary Program Space, or BPS) above the free storage list. Functional expressions (FEXPR's) are not presently compilable; however, it is expected that a future OS version P will permit such operations. LISP subroutines (SUBR's) and functional subroutines (FSUBR's, such as LIST, PLUS, etc.) may not be compiled since they already exist in machine language form. Functions which contain references to PROG or to FEXPR's other than CSETQ are not presently compilable.

The compiler is invoked by evaluating the function COMPILE with a single argument whose value is the list of functions to be compiled. The value of COMPILE is the value of its argument.

12.2 Compiler functions

12.2.1 COMPILE

The action of COMPILE is to produce the equivalent machine code of the LAMBDA-expression under the EXPR indicator on the property list of each function in the argument list, and replace the EXPR indicator with a SUBR indicator. The value under the new indicator is the address, in BPS, of the first instruction of the compiled function. Standard LISP linkage conventions (roughly the same as DOS linkage conventions) for subroutines are observed.

To perform the actual compilation, COMPILE invokes several LISP functions to handle special cases and housekeeping procedures. These functions, in turn, call several machine language routines to produce the actual code. Since LISP functions are, in general, recursive, both the compiler and the compiled code are reentrant.

12.2.2 DECK and NODECK

A future version of the LISP Compiler is expected to produce object decks at the user's option, and accept object decks as input. Toward this end, the DECK and NODECK control functions are included in the compiler package. Execution of the NODECK function turns on a bit in the compiler. Subsequently-compiled functions will demonstrate faster execution times than comparable functions previously compiled, owing to a resultant decrease in the use of built-in bookkeeping procedures. Execution of the DECK function turns off the control bit. Hence, the subsequent compilation of functions will produce code with 'normal' execution speed. The value of both DECK and NODECK is NIL.

Note that the execution of DECK or NODECK after a function has been compiled has no effect on the compiled code of that function.

The compiler operates in DECK mode unless NODECK IS specified.

12.2.3 THECOMPILER

THECOMPILER is an atom in the LISP system whose value is a list of the LISP functions contained in the compiler package. The execution of the list structure (COMPILE THECOMPILER) results in the compilation of the LISP part of the compiler. Subsequent use of the COMPILE function will utilize the compiled compiler.

12.2.4 ADDBPS

At initialization time of the LISP system, BPS and FSL are of approximately equal size. The ADDBPS function permits the user to modify the size of both of these core areas during program execution. The single argument of ADDBPS is a positive (or negative) integer. The effect of ADDBPS is to add (subtract) that number of double words to (from) the lower end of BPS. The additional storage locations are taken from (added to) the top of the free storage list. The value of ADDBPS is the new size, in double words, of binary program space.

ADDBPS should only be invoked when the prior contents of BPS and the FSL are no longer needed, since the function involves a garbage collection and repositioning of system pointers. It is usually called immediately after system initialization.

12.2.5 CLEARBPS

The CLEARBPS function is called with no arguments. Its effect is to set the pointer to the next free area in BPS equal to the beginning of BPS. Hence, previously compiled functions may be destroyed by future compilations. The value of CLEARBPS is the size, in double words, of binary program space.

CLEARBPS is primarily invoked when previously compiled functions are no longer needed and the full BPS is required for future functions.

12.2.6 EXCISE

The EXCISE function provides for the removal of part or all of the LISP Compiler from the system, and the conversion of the space it occupied into additional free storage. EXCISE takes one argument which has either the value T or the value NIL. With an argument of NIL, EXCISE removes that part of the compiler which performs the actual compilation. BPS is not changed, and execution time routines residing in the compiler remain. With an argument of T, EXCISE removes the entire compiler and destroys BPS. In either case, the vacated core positions are added to the FSL, and further compilation is impossible.

The value of EXCISE is its argument.

12.3 Efficiency

Compiled functions average the same size as their equivalent expressions, but run approximately eight times faster. Each compiled function is implicitly addressed, permitting the maximum size of a compilable LISP function to be limited only by the size of BPS. At present, no code optimization is performed, although it is expected that future versions will provide for it.

Compiling the compiler in DOS background took 35 seconds on our System/360/50.

12.4 Error messages

12.4.1 COMPILE

<u>Message</u>: OUT OF BINARY PROGRAM SPACE. RUN TERMINATED.

<u>Reason</u>: The space remaining in BPS is not sufficient to compile the function.

<u>Action</u>: Remaining input cards will be read but not evaluated. The job is terminated.

12.4.2 ADDBPS

<u>Message</u>: SPACE REQUESTED NOT AVAILABLE.

<u>Reason</u>: An attempt has been made to decrease BPS below zero or increase it above the maximum available core size.

<u>Action</u>: BPS is not changed, and processing continues normally.

13. OPERATIONAL PROCEDURES

13.1 Deck setup for DOS background

13.1.1 List structures are evaluated merely by placing them in the program.. They are evaluated in order of appearance. The order is unimportant, except that a function defined by the programmer must be named before the function is evaluated by name.

13.1.2 The deck submitted consists of two control cards, the list structures to evaluate (i. e., the LISP program), and a /* and, finally, a /& card, as illustrated.

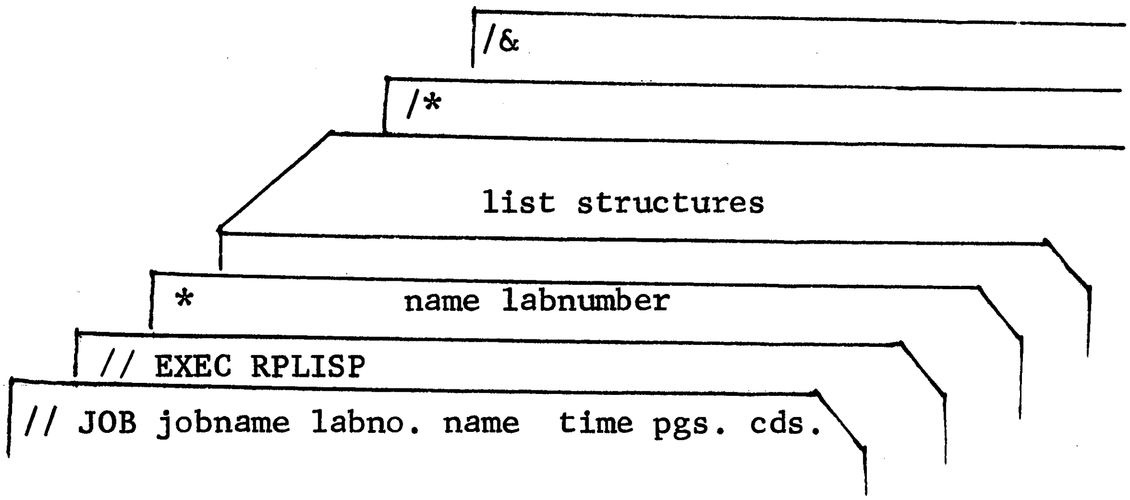
13.1.3 The format of the program cards containing the list structures is not critical. Spaces may be inserted for readability wherever desired, and list structures may extend over any number of cards.

Blank cards may appear only inside a list structure. Blank cards otherwise inserted will result in an error.

13.1.4 Only the first 72 columns of a card may be used for list structures; the remaining columns are ignored, and hence may used for sequencing or identification purposes.

LISP treats column 72 of each card as adjacent to column 1 of the following card.

13.1.5 Any card with an asterisk in column 1, with the exception of the LISP option cards (see 13.2), is a comment card, which is printed but otherwise ignored. The computer laboratory requires that each LISP program contain a comment card with the programmer's name and lab number on it, and that this card be placed at the beginning of the program.



Background Deck Setup

13.2 Running under DOS foreground - teletype

LISP may be uses in foreground via the CONVERSE teletype monitor, an RPI in-house program. System control cards are not necessary; instead, CONVERSE will request control information from the user. The use and order of list structures comprising a LISP program are the same as for background (see 13.1); LISP treats teletype input as card images.

13.3 LISP option cards

All LISP option cards contain an asterisk (*) in column 1 (so that they will not be evaluated), followed by a key word starting in column 2. They may appear anywhere in the program, and affect the system operation from that point on.

13.3.1 NOLABEL

Under normal operation, the LISP interpreter will print the line 'THE VALUE OF THE ABOVE LIST STRUCTURE IS' beneath every evaluated list structure. When the system is operating via a teletype, the printing of this line is time-consuming; in background, it may spoil the appearance of the output. Hence, the NOLABEL option suppresses the printing of the value message.

13.3.2 NODLM

The character period (.) has special meaning in the LISP system. Commas (,) are treated as blanks, and are not printed in output unless specifically requested. The NODLM option permits periods and commas to be treated as any other character acceptable in print names of atoms.

13.3.3 NOLIST and LIST

NOLIST suppresses the printing of the input text. LIST restores the printing of the input text.

13.3.4 SUPERLISP

SUPERLISP has an effect in background only. It provides for the inclusion of the entire partition in FSL and BPS, instead of the normal 128K CPU core. It makes available approximately one million bytes of bulk core storage. While desirable for programs requiring large list structures and heavy use of recursion, it increases processing time by as much as a factor of four.

13.3.5 COMPLAIN and QUIET

COMPLAIN will cause the system to print a message every time a garbage collection occurs, indicating the number of available free storage items. QUIET suppresses the printing of the message.

13.3.6 NOAUTOCR

NOAUTOCR suppresses the automatic recognition of functions of the form CDADR (see 4.4.6). DEFCR may then be used to define functions of this type if desired.

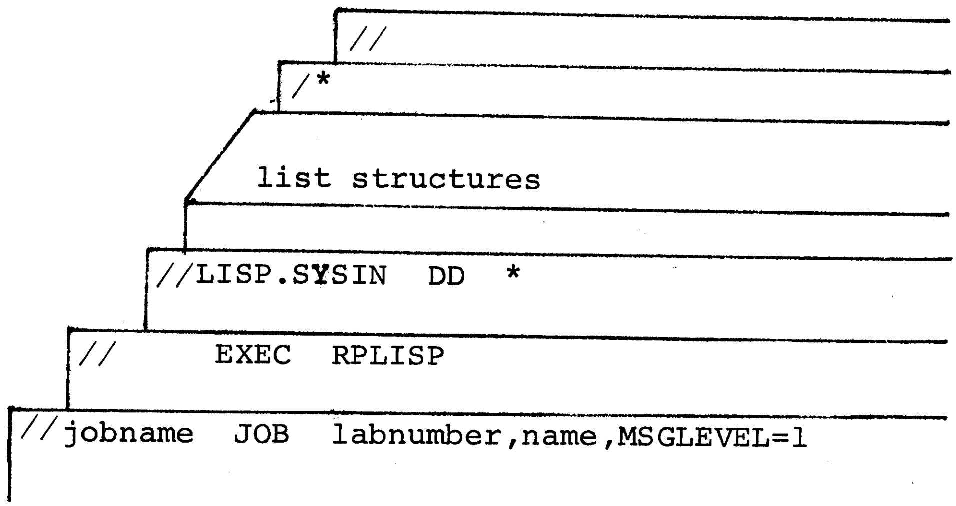
13.3.7 STOP

STOP should be the last statement in a LISP program entered via the teletype. It indicates program termination, and is handled similarly to a /* card in background.

13.3.8 Standard options (assumed initially) are LIST and QUIET.

13.4 Deck Setup for OS

The deck setup for OS is the same as that for DOS background (13.1), except for the control cards described in 13.1.2 . Refer to the illustration below for the OS control cards and deck setup.



<u>OS DECK SETUP</u>

If the function TERPCH is used in the program, to produce punched output, the EXEC card must be changed to read:

```
//      EXEC   RPLISP,PARM.LISP=PUNCH
```

APPENDIX I

Built-in LISP functions and constants

This appendix contains descriptions of all functions and constants defined in the LISP system. Below is a summary of their classification in this appendix.

Functions are defined in this appendix by stating their value in a function call of the form (f, al,...,an).

I.1 GENERAL FUNCTIONS

Function	Arguments	Value	Effect
APPEND	[a1]= (u1,...,un) [a2]= (v1,...,vn)	(u1,...,un,v1,...,vn)	
ATOM	[a1]	T if [a1] is an atom NIL otherwise	
CAR	[a1]= (u1,...,un)	u1	
CDR	[a1]= (u1,...,un)	(u2,...,un)	
COND	a1,...,an, where ai = (bi,fi)	[fi], where [b1],...,[bi-1] are NIL and [bi] = T	
CONS	[a1] [a2]= (v1,...,vn)	([a1],v1,...,vn)	Uses up one double word from FSL
CR	[a1] [a2] is a list of 0's and 4's	Takes CAR's and CDR's of [a1] in the order in which they are indicated in [a2] by 0's and 4's, respectively.	
EQ	[a1] or [a2] atomic and non-numeric	T if [a1] = [a2] NIL otherwise (see 6.4.3)	
EQUAL	[a1], [a2]	T if [a1] = [a2] NIL if [a1] ≠ [a2]	
EVAL	[a1]	[[a1]]	
EXPLODE	[a1] atomic	a list of atoms whose print names are the individual characters in the print name of a1	
GENSYM	none	a new atom with a print name of the form Xnnnnn, n a digit	
IMPLODE	[a1] = (u1,...,un) ui atomic	an atom whose print name is the concatenation of those of u1,...,un	
LABEL	a1 atomic a2	(a1)	[a1]= a2 recorded on a-list

GENERAL FUNCTIONS (cont'd.)

Function	Arguments	Value	Effect
LENGTH	[a1] = (u1,...,un)	n	
LIST	[a1],...,[an]	([a1],...,[an])	Uses up n double words from FSL
MAPCAR	[a1] a function [a2] = (v1,...,vn)	([([a1] v1)],..., [([a1] vn)])	
MEMBER	[a1] [a2] = (v1,...,vn)	T if [a1] = vi, some i NIL otherwise	
NULL	[a1]	T if [a1] = NIL NIL otherwise	
PROG2	[a1], [a2]	[a2]	(evaluates a1 and a2 in that order)
QUOTE	a1	a1	
SASSOC	[a1] atomic [a2] = (v1,...,vn)	the first element vi of [a2] whose CAR is [a1], or NIL if none.	

I.2 ARITHMETIC FUNCTIONS

Function	Arguments	Value	Effect
ADD1	[a1]	[a1]+ 1	
DIFFERENCE	[a1], [a2]	[a1] - [a2]	
EXPT	[a1], [a2]	$([a1])^{[a2]}$	
GREATERP	[a1], [a2]	T if [a1]>[a2] NIL otherwise	
LESSP	[a1], [a2]	T if [a1]<[a2] NIL otherwise	
MAX	[a1],...,[an]	max([a1],...,[an])	
MAXL	[a1]= (u1,...,un)	max(u1,...,un)	
MIN	[a1],...,[an]	min([a1],...,[an])	
MINL	[a1] = (u1,...,un)	min(u1,...,un)	
MINUS	[a1]	-[a1]	
MINUSP	[a1]	T if [a1] is negative NIL otherwise	
NUMBERP	[a1]	T if [a1] is a number NIL otherwise	
ONEP	[a1]	T if [a1]= 1 NIL otherwise	
PLUS	[a1],...,[an]	[a1]+ ... +[an]	
PLUSL	a1 = (u1 ,...,un)	u1 + ... + un	
QUOTIENT	[a1], [a2]	greatest integer in $\frac{[a1]}{[a2]}$	
SUB1	[a1]	[a1]- 1	
TIMES	[a1],...,[an]	[a1]* ... *[an]	
TIMESL	[a1]= (u1,...,un)	u1 * ... * un	
ZEROP	[a1]	T if [a1]= 0 NIL otherwise	

I.3 LOGICAL FUNCTIONS

Function	Arguments	Value	Effect
AND	a1 ,..., an	T if [ai] = T, for all i, NIL otherwise	Evaluates the ai only to the first NIL
ANDL	[a1] = (u1,...,un)	T if ui = T, for all i, NIL otherwise	
NOT	[a1]	T if [a1] = NIL NIL otherwise	
OR	a1,...,an	T if [ai] = T, some i NIL otherwise	Evaluates to first T
ORL	[a1] = (u1,...,un)	T if ui = T, some i NIL otherwise	

I.4 SYSTEM FUNCTIONS

Function	Arguments	Value	Effect
ADDBPS	[a1] an integer	New size of BPS	(see 12.2.3)
EXCISE	[a1]	[a1]	(see 12.2.6)
RECLAIM	none	NIL	Forces garbage collection
SIZE	BPS/FSL/ALL COMPILER/PARTITION or [a1], a1 not any of the above	Size, in double words**	
TIME	none	Interval time elapsed (in seconds) since last called; first call produces value zero.	

** BPS — Unused Binary Program Space

BPS	Unused Binary Program Space
FSL	Free Storage List (remaining for use)
COMPILER	All parts of the compiler (excluding BPS)
PARTITION	The foreground or background partition in which the program is being run
ALL	Area above start of compiler
[a1]	Double words in the implementation of [a1]

Any or all of BPS, FSL, COMPILER, PARTITION, ALL may appear as arguments, or the argument may be a single list structure, but the keywords and list structures may not be mixed.

I.5 PROPERTY-LIST FUNCTIONS

Function	Arguments	Value	Effect
CSET	[a1] atomic, [a2]	([a1])	[[a1]]= [a2]
CSETQ	a1 , a2	(a1)	[a1]= a2
DEFCR	[a1]= CDAAR, e.g.	[a1]	[([a1] u)] = [(CDR(CAR (CAR u)))], etc. see 4.4.6, 13.3.6
DEFINE	[a1]= ((u1,v1),..., (un,vn))	(u1,...,un)	[(ui,a1,...,an)] = [(vi,a1,...,an)] see 5.
DEFLIST	[a1]= ((u1,v1), ...,(un,vn)) [a2]atomic	(u1,...,un)	[(GET(QUOTE ui)a2)] = vi (see 5.)
GET	[a1]= (u1,,...,un) (usually a property list) [a2]atomic (an indicator)	if ui =[a2] for some i, then ui+1, else NIL (the value under the indicator)	
REMPROP	[a1] atomic [a2]atomic	[(GET a1 a2)]	[(GET a1 a2)] = NIL

I.6 LIST-CHANGING FUNCTIONS

Function	Arguments	Value	Effect
EFFACE	[a1] [a2]= (v1,...,vn)	if vi =[a1] then (v1,...,vi-1,vi+1,...,vn) else[a2]	(see 9.4)
NCONC	[a1]= (u1,...,un) [a2]= (v1,...,vm)	(u1,...,un,v1,...,vm)	"
RPLACA	[a1]= (u1,...,un) [a2]	([a2],u2,...,un)	"
RPLACD	[a1]= (u1,...,un) [a2]= (v1,...,vm)	(u1,v1,...,vm)	"

I.7 I/O FUNCTIONS

Function	Arguments	Value	Effect
EJECT	none	NIL	spaces printer to top of next page
PRINT	[a1]	[a1]	prints [a1]
PRIN1	[a1] atomic	[a1]	adds the print name of [a1] to buffer
PUNCH1	[a1] atomic	[a1]	adds the print name of [a1] to buffer
READ	none	a list structure read in	reads the next complete list structure from SYSIPT (or from teletype, in fg)
TERPCH	none	NIL	punches the contents of the buffer on one card
TERPRI	none	NIL	prints the contents of the buffer on one line

I.8 PROGRAM FEATURE FUNCTIONS

Function	Arguments	Value	Effect
GO	a1 atomic	(see Appendix II)	
PROG	a1,...,an		"
SETQ	a1 atomic, a2	[a2]	Binds [a2] to a1 on a-list

I.9 COMPILER FUNCTIONS

Function	Arguments	Value	Effect
CLEARBPS	none	Size, in double words, of BPS	
COMPILE	[a1]= (v1,...,vn)	(v1,...,vn)	See 12.2.1
DECK	none	NIL	See 12.2.2
NODECK	none	NIL	See 12.2.2

I.10 CONSTANTS

Constant	Value
T	T
NIL	NIL
BLANK	
COMMA	,
PERIOD	.
LPAR	(
RPAR	)
OBLIST	The list of atoms in the system, including all built-in functions and constants, followed by those supplied (since the start of a run) by the programmer.
THECOMPILER	A list of the functions which make up the LISP Compiler.

APPENDIX II

The LISP program feature

II.1 The LISP program feature helps the user to write programs containing many interdependent variables. It mimics conventional programming languages by executing list structures for the sake of their effects rather than their values.

II.2 A PROG-expression is a list of the form:

(PROG ($v_1,\ldots,v_n$) $s_1,\ldots,s_m$)

where $v_1,\ldots,v_n$ are program variables (perhaps none),

and $s_1,\ldots,s_m$ are statements or labels.

A <u>statement</u> is a function call.

A <u>label</u> is a non-numeric atom.

A <u>program variable</u> is a non-numeric atom.

II.3 As soon as the statement (RETURN s) is executed, evaluation of the PROG-expression is completed and it returns a value of [s].

II.4 To branch to a location in the program designated by a label a, execute (GO a).

II.5 Program variables are atoms whose values are set by SETQ during execution of the PROG. SETQ takes two arguments; it assigns its unevaluated first argument the value of its second argument. Unlike CSETQ, the value assigned by SETQ is recorded on the <u>a-list</u> and hence disappears at the conclusion of the PROG; thus, LAMBDA-variables may be SETQd, too. The value of SETQ is the value of its second argument.

II.6 COND statements appearing within a PROG need not contain at least one true condition. If no true condition is encountered, control passes to the next sequential list structure following the COND. If a true condition <u>is</u> found, and the corresponding statement is not a GO or RETURN, the statement is evaluated and <u>then</u> control is passed.

II.7 Example:

```
(DEFINE (QUOTE (
(FACTORIAL (LAMBDA (N) (PROG (V W)
     (SETQ V N)
     (SETQ W 1)
 LOOP (COND ((ZEROP V)(RETURN W)) )
     (SETQ W (TIMES W V))
     (SETQ V (SUB1 V))
     (GO LOOP)
)))   )))
```

APPENDIX III

Differences between System/360 LISP 1.5 and 7090 LISP 1.5

III.1 Functions which differ (in number of arguments, or value, or effect) from the 7090 function of the same name:

COMPILE	GENSYM
CSET	PROG
CSETQ	REMPROP
EFFACE	RETURN
EVAL	SASSOC

III.2 Functions not in 7090 LISP 1.5 :

ADDBPS	MINL
ANDL	ORL
CLEARBPS	NODECK
CR	PLUSL
DECK	PUNCH1
DEFCR	TERPCH
IMPLODE	TIME
MAPCAR	TIMESL
MAXL	

III.3 There are many functions in 7090 LISP not in System/360 LISP.

III.4 Print names of atoms are not restricted to 7090 'atomic symbols'.

APPENDIX IV

LPCP: LISP Parenthesis Counting Program

A program to aid the user in matching parentheses in LISP programs has been written, and is available, under OS, in the LISP disk library.

It produces a listing of the LISP deck with a number or letter below each parenthesis, so that the first (left) parenthesis is numbered 1 and the symbols under matching parentheses are the same. The count proceeds: 1,...,9,A,B,... .

Your LISP program may be checked by LPCP, and then turned over to the LISP system for processing, by using the deck setup shown below:

```
//jobname   JOB   labnumber,name,MSGLEVEL=1

//JOBLIB   DD   DSNAME=LISP,DISP=SHR

//COUNT   EXEC   PGM=LPCP

//SYSOUT   DD   SYSOUT=A

//SYSUT    DD   DSNAME=&LISPIN,DISP=(,PASS),UNIT=2311,

//       SPACE=(TRK,(50,10),RLSE)

//SYSIN    DD   *

            (LISP deck)

/*

//GO      EXEC   RPLISP

//SYSIN    DD   DSNAME=&LISPIN,DISP=(OLD,DELETE)

//
```

APPENDIX V

System Messages

1. ***GARBAGE COLLECTION. FSL REMAINING -
   See 10.3, 13.3.5

2. THE VALUE OF THE ABOVE LIST STRUCTURE IS -
   See 13.3.1

3. (MSG TOO LONG) RETURN =
   See a LISP system programmer.

4. E R R O R UNSAVE ENTERED MORE THAN SAVE. RETURN =
   See 13.1.3

5. E R R O R PUSH DOWN LIST FULL. RETURN =
   See 10.4

6. E R R O R NON-BLANK CHARACTER AFTER MATCHING RIGHT PARENTHESIS
   See 2.4

7. E R R O R IMPROPER PARENTHESIS COUNT
   See 2.4

8. E R R O R ITEM NOT UNSAVED ON RESET =
   Seea LISP system programmer.

9. E R R O R UNBOUND ATOM
   See 5.5.5

10. E R R O R UNDEFINED FUNCTION
    See 4., 5.

11. E N D O F L I S P R U N
    Sign-off message.

12. E R R O R CHARACTER STORAGE SPACE EXHAUSTED RUN TERMINATED
    See 10.5

13. E R R O R ILLEGAL PERIOD
    See 13.3.2, 4.4.4

14. E R R O R ATOMIC SYMBOL EXCEEDING 72 CHARACTERS
    See 2.2.3

15. S T A R T O F L I S P R U N
    Sign-on message.

16. E R R O R NO PRINT NAME FOUND FOR ATOM
    See a LISP system programmer.

17. E R R O R NO TRUE CONDITION IN CONDITIONAL EXPRESSION
See 6.3.3

18. E R R O R ARGUMENT LIST FOR 'LAMBDA' SHORTER THAN VARIABLE LIST
See 5.

19. E R R O R EXCESS RIGHT PARENTHESIS.
See 2.4

20. E R R O R VARIABLE LIST FOR 'LAMBDA' SHORTER THAN ARGUMENT LIST
See 5.

21. E R R O R NO FSL RECLAIMABLE. RUN TERMINATED.
See 10.3

22. E R R O R NUMBER USED AS FUNCTION
See 5.1

23. E R R O R CDR OF NIL
See 4.4.4

24. E R R O R CAR OF AN ATOM ATTEMPTED
See 4.4.4

25. SPACE REQUESTED NOT AVAILABLE
See 12.4.2

26. UNPRINTABLE.
A function has been called with an argument for which it is undefined.

27. PRIN1 HAS BEEN ENTERED WITH AN IMPROPER ARGUMENT.
See 11.4.3

28. E R R O R SYSTEM ATOMS MAY NOT BE REDEFINED*
See appendices I and VI for a list of system atoms.

29. E R R O R OUT OF BINARY PROGRAM SPACE. RUN TERMINATED
See 12.4.1

30. E R R O R EXPLODE HAS BEEN ENTERED WITH AN IMPROPER ARGUMENT
Argument for EXPLODE is not atomic.

31. E R R O R IMPLODE HAS BEEN ENTERED WITH AN IMPROPER ARGUMENT
Argument for IMPLODE is not a list of atoms.

Note: a value printed after a recoverable error is whatever caused the error(unless NIL or UNPRINTABLE).

*Function definitions following the improper one in the same call on DEFINE are not recorded.

APPENDIX VI

Some LISP functions

VI.1 A few simple examples

```
(DEFINE (QUOTE (

(REMAINDER (LAMBDA (M N)
  (DIFFERENCE M (TIMES N (QUOTIENT M N))) ))

(LASTELEMENT (LAMBDA (L)
  (COND
    ((NULL (CDR L)) (CAR L))
    (T (LASTELEMENT (CDR L)))
)))

(SELECT (LAMBDA (MASK L)
  (COND
    ((NULL L) NIL)
    ((ONEP(CAR MASK))(CONS(CAR L)(SELECT(CDR MASK)(CDR L))))
    (T(SELECT (CDR MASK) (CDR L)))
)))

(REVERSE (LAMBDA (L REVL)
  (COND
    ((NULL L) REVL)
    (T (REVERSE (CDR L)(CONS(CAR L)REVL)))
)))

(MERGE (LAMBDA ( X Y)
  (COND
    ((NULL X) Y)
    ((NULL Y) X)
    ((LESSP(CAR X)(CAR Y))
     (CONS(CAR X)(MERGE (CDR X) Y)) )
    (T (CONS (CAR Y) (MERGE X (CDR Y))) )
)))

)))
        THE VALUE OF THE ABOVE LIST STRUCTURE IS -
(REMAINDER LASTELEMENT SELECT REVERSE MERGE)
```

The five functions above are independent. Some examples of what each is supposed to do are given below.

[(REMAINDER 8 2)] = 0, [(REMAINDER 11 3)] = 2

[(LASTELEMENT(QUOTE(A B C)))] = C,

[(LASTELEMENT(QUOTE((A))))] = (A)

[(SELECT(QUOTE(1 0 1 1 0 0 1))(QUOTE(A B C D E F G)))]
    = (A C D G)

[(REVERSE(QUOTE(A (B C) D)) NIL)] = (D (B C) A)

[(MERGE(QUOTE(2 5 7 8))(QUOTE(3 6 7 9 10)))]
    = (2 3 5 6 7 7 8 9 10)

VI .2 Some simple applications

On the following pages are listings and illustrations of some simple applications of LISP. The functions shown have been stored as teletype files.

Several applications which are more extensive (and thus too long to reproduce here) are available at RPI, including: symbolic differentiation, reduction of finite state automata, production of syntactic monoids, and polynomial manipulations.

VI.2.1 DIVIDE finds the quotient of two integers to any desired number of places (the third argument -1), expressed as a list. PUTPOINT turns the result of DIVIDE into an atom that looks like a fixed point number.

```
    $FILE 'DIVID'
LOADED
05: $EXECUTEP
01: (DEFINE(QUOTE(
02: (DIVIDE(LAMBDA(N D P)
03:   ((LAMBDA(Q)
04:     (COND
05:       ((ZEROP P)NIL)
06:       ((ZEROP Q)(CONS 0(DIVIDE(TIMES 10 N)D(SUB1 P))))
07:       (T(CONS Q(DIVIDE
08:         (TIMES 10(DIFFERENCE N(TIMES Q D))) D (SUB1 P))))
09:     ))(QUOTIENT N D)) ))
10: (PUTPOINT(LAMBDA(X)
11:   (IMPLODE(CONS(CAR X)(CONS PERIOD(CDR X)))) ))
12: )))

         THE VALUE OF THE ABOVE LIST STRUCTURE IS -

(DIVIDE PUTPOINT)

12: $EDITSTART
01: (DIVIDE 23 13 10)

         THE VALUE OF THE ABOVE LIST STRUCTURE IS -

(1 7 6 9 2 3 0 7 6 9)

02: (PUTPOINT(QUOTE(1 7 6 9 2 3 0 7 6 9)))

         THE VALUE OF THE ABOVE LIST STRUCTURE IS -

1.769230769

03: (PUTPOINT(DIVIDE 37900 491 63))

         THE VALUE OF THE ABOVE LIST STRUCTURE IS -

77.18940936863543788187372708757637474541751527494908350305498981

04:
```

VI.2.2 CONCLUSION finds the conclusion of a syllogism. Its two arguments are the major and minor premise, respectively.

```
5: $FILE 'SYLL1'
OADED
5: $EXECUTEP
1: (DEFINE (QUOTE (
2: (PARTAFTER (LAMBDA (WORD PHRASE)
3: (COND
4:   ((NULL PHRASE) NIL)
5:   ((EQ (CAR PHRASE) WORD)(CDR PHRASE))
5:   (T (PARTAFTER WORD (CDR PHRASE))) ) ) )
': (CLASS (LAMBDA(PREMISE) (PARTAFTER (QUOTE IS) PREMISE)))
 : (SUBCLASS (LAMBDA(PREMISE)(CAR (PARTAFTER(QUOTE EVERY) PREMISE))))
 : (INDIVIDUAL (LAMBDA(PREMISE) (CAR PREMISE)))
 : (INDIVIDUALSCLASS (LAMBDA(PREMISE)(CAR(CDR(CLASS PREMISE)))))
 : (CONCLUSION (LAMBDA (PREMISE1 PREMISE2)
 : (COND
 :   ((EQ (INDIVIDUALSCLASS PREMISE2)(SUBCLASS PREMISE1))
1:    (CONS(INDIVIDUAL PREMISE2)(CONS(QUOTE IS)(CLASS PREMISE1))))
5:   (T (QUOTE (NO CONCLUSION))))))))

         THE VALUE OF THE ABOVE LIST STRUCTURE IS -

PARTAFTER CLASS SUBCLASS INDIVIDUAL INDIVIDUALSCLASS CONCLUSION)

5: (CONCLUSION (QUOTE(EVERY MAN IS MORTAL))
5:             (QUOTE(SOCRATES IS A MAN)))

         THE VALUE OF THE ABOVE LIST STRUCTURE IS -

SOCRATES IS MORTAL)

5: $EDITSTART
1: (CONCLUSION(QUOTE(EVERY SEAGULL IS A BIRD))
2:            (QUOTE(THIS IS A BIRD)))

         THE VALUE OF THE ABOVE LIST STRUCTURE IS -

NO CONCLUSION)

5: (CONCLUSION (QUOTE(EVERY RPI/PROGRAM IS DEBUGGED AND RUNNING))
1:             (QUOTE((THE LISP SYSTEM) IS AN RPI/PROGRAM)))

         THE VALUE OF THE ABOVE LIST STRUCTURE IS -

THE LISP SYSTEM) IS DEBUGGED AND RUNNING)

 :
```

VI.2.3 POLISH converts a character string (expressed as a list of atoms) representing an algebraic expression in FORTRAN notation into the Lukaszewicz ('Polish') postfix form.

```
   $FILE 'POLSH'
LOADED
27: $EXECUTEP
01: (CSETQ OPERPREC (QUOTE((+ 1)(- 1)(* 2)(/ 2)(** 3))))

         THE VALUE OF THE ABOVE LIST STRUCTURE IS -

(OPERPREC)

02: (CSETQ OPERATORS (QUOTE(+ - * / **)))

         THE VALUE OF THE ABOVE LIST STRUCTURE IS -

(OPERATORS)

03: (DEFINE(QUOTE(
04:    (PRECEDENCE(LAMBDA(OPERATOR PRECLIST)
05:       (COND
06:          ((NULL PRECLIST)(PROG2(PRINT(QUOTE(ILLEGAL OPERATOR IN
07:             INPUT STRING)))4))
08:          ((EQ OPERATOR (CAAR PRECLIST))(CADAR PRECLIST))
09:          (T(PRECEDENCE OPERATOR (CDR PRECLIST))) )))
10:    (POLISH1(LAMBDA(X OPSTACK)
11:       (COND
12:          ((NULL X) OPSTACK)
13:          ((ATOM X) X)
14:          ((NOT(ATOM(CAR X)))(APPEND(POLISH(CAR X))(POLISH1 (CDR X)
15:             OPSTACK)))
16:          ((NOT(MEMBER(CAR X)OPERATORS))(CONS(CAR X)(POLISH1(CDR X)
17:             OPSTACK)))
18:          (T((LAMBDA(G)
19:             (COND
20:                ((NULL OPSTACK)(POLISH1(CDR X) G))
21:                ((NOT(GREATERP (PRECEDENCE (CAR X) OPERPREC)
22:                           (PRECEDENCE (CAR OPSTACK) OPERPREC)))
23:                   (CONS(CAR OPSTACK)(POLISH1 X (CDR OPSTACK))))
24:                (T(POLISH1 (CDR X) G))))(CONS(CAR X)OPSTACK)))
25:       )))
26:    (POLISH(LAMBDA(X)(POLISH1 X NIL)))
27: )))

         THE VALUE OF THE ABOVE LIST STRUCTURE IS -

(PRECEDENCE POLISH1 POLISH)

27:
```

```
   $EDITCONT
28: (POLISH (QUOTE (A + B)))

         THE VALUE OF THE ABOVE LIST STRUCTURE IS -

(A B +)

29: (POLISH (QUOTE (A + B - C + D - E)))

         THE VALUE OF THE ABOVE LIST STRUCTURE IS -

(A B + C - D + E -)

30: (POLISH (QUOTE (A / B + (C ** (E - (F * G))))))

         THE VALUE OF THE ABOVE LIST STRUCTURE IS -

(A B / C E F G * - ** +)

31:
```

VI.2.4 CANONICAL finds a canonical form for logical expressions in LISP prefix notation. The result is dependent on an ordering of the variables in the expression, which is specified by its second argument. OUTFORM, EXPAND, and their subfunctions CAL and DIST turn the canonical form into the conventional 'sum-of-products'.

```
   $FILE ´BOOLE´
LOADED
38: $EXECUTEP
(DEFINE(QUOTE(

(CANONICAL(LAMBDA(X V)
  (COND
    ((NULL V)(EVAL X))
    (T(LIST
      (PROG2(CSET(CAR V)T)(CANONICAL X(CDR V)))
      (PROG2(CSET(CAR V)NIL)(CANONICAL X(CDR V)))
    ))
)))
(EXPAND(LAMBDA(X V)
  (COND
    ((NULL V)X)
    (T(APPEND
      (DIST(CAR V)(EXPAND(CAR X)(CDR V)))
      (DIST(LIST(QUOTE NOT)(CAR V))(EXPAND(CADR X)(CDR V)))
    ))
)))
(DIST(LAMBDA(X Y)
  (COND
    ((NULL Y)NIL)
    ((EQ Y T)(LIST(LIST X)))
    (T((LAMBDA(DXCY)
      (COND
        ((EQ(CAR Y)T)(CONS(LIST X)DXCY))
        ((NULL(CAR Y))DXCY)
        (T(CONS(CONS X(CAR Y))DXCY))))
     (DIST X(CDR Y))) )
)))
(OUTFORM(LAMBDA(X)
  (CONS(QUOTE OR)(CAL X))))
(CAL(LAMBDA(X)
  (COND
    ((NULL X)NIL)
    (T(CONS(CONS(QUOTE AND)(CAR X))(CAL(CDR X))))
)))

)))

          THE VALUE OF THE ABOVE LIST STRUCTURE IS -

(CANONICAL EXPAND DIST OUTFORM CAL)

38:
```

```
    $EDITCONT
45: (CSETQ VARLIST (QUOTE(A B C)))

         THE VALUE OF THE ABOVE LIST STRUCTURE IS -

(VARLIST)

46: (CANONICAL (QUOTE(AND A(OR B(NOT C)))) VARLIST)

         THE VALUE OF THE ABOVE LIST STRUCTURE IS -

(((T T) (NIL T)) ((NIL NIL) (NIL NIL)))

47: (DEFINE(QUOTE(
48: (LOGICALLYEQUIVALENT(LAMBDA(X Y)
49:   (EQUAL(CANONICAL X VARLIST)(CANONICAL Y VARLIST))
50: )) )))

         THE VALUE OF THE ABOVE LIST STRUCTURE IS -

(LOGICALLYEQUIVALENT)

51: (LOGICALLYEQUIVALENT
52:   (QUOTE(AND A(OR B (NOT C))) )
53:   (QUOTE(OR (AND A B)(AND A(NOT B)(NOT C))) )
54: )

         THE VALUE OF THE ABOVE LIST STRUCTURE IS -

T

55: (OUTFORM(EXPAND(CANONICAL
56:   (QUOTE(AND A(OR B(NOT C))) )VARLIST)VARLIST) )

         THE VALUE OF THE ABOVE LIST STRUCTURE IS -

(OR (AND A B C) (AND A B (NOT C)) (AND A (NOT B) (NOT C)))

57:
```

VI.2.5 ATURING simulates and traces the operation of a Turing machine (specified by a transition table) on any given tape. It prints the current state, the non-blank part of the tape, and indicates the position of the head, at each step. The example shows a Turing machine for the Euclidean Algorithm.

```
    $FILE  'TUR5'
LOADED
15: $EXECUTEP
01: (DEFINE(QUOTE(
02:     (ATURING (LAMBDA(A B C D) (TURING (PRIN1 A) B (PRL C)
03:              (PRR D)  )))
04:     (TURING (LAMBDA (P TM L R) (SUBTURING (MATCH P R TM) )))
05:     (MATCH (LAMBDA (P R TM)
06:        (COND
07:           ((NULL TM)(QUOTE (*)))
08:           ((EQUAL (LIST P (CAR R))(CAAR TM))(CDAR TM))
09:           (T (MATCH P R (CDR TM)))  )))
10:     (SUBTURING (LAMBDA (W)
11:        (COND
12:           ((EQ (CAR W)(QUOTE *))(QUOTE HALTED))
13:           (T (TURING (PRIN1 (CADDR W)) TM
14:              (PRL
15:                 (COND
16:                    ((EQ (CADR W)(QUOTE R))(CONS (CAR W) L))
17:                    (T (COND
18:                          ((NULL L)(QUOTE (0)))
19:                          (T (CDR L)) ) ) ) )
20:              (PRR
21:                 (COND
22:                    ((EQ (CADR W)(QUOTE L))(CONS
23:                       (COND
24:                          ((NULL L)(QUOTE 0))
25:                          (T (CAR L)) )
26:                       (CONS (CAR W)(CDR R)) ))
27:                    (T (COND
28:                          ((NULL (CDR R))(QUOTE (0)))
29:                          (T (CDR R)) ) ) ) ) ))) ))
29: $FILE 'TUR6'
LOADED
29: * TUR6 CONTAINS THE PRINTING FUNCTIONS PRK, PRR, AND PRA.
29: $EXECUTE

          THE VALUE OF THE ABOVE LIST STRUCTURE IS -

(ATURING TURING MATCH SUBTURING PRL PRR PRA)

29:
```

```
    (CSETQ TMM (QUOTE (
29: ((A Z)Z L A)((A 1)1 L A)((A 0)0 L A)((A 2)1 R B)((A X)X R E)
29: ((B Z)Z R B)((B 1)1 R B)((B 0)0 R B)((B Y)Y L C)
29: ((C 0)0 L C)((C 1)0 L D)((D 1)1 L A)((D Z)1 L F)
29: ((E 1)2 R E)((E Z)Z L A)
29: ((F 1)1 L F)((F 2)2 R G)((F X)X R H)
29: ((G 1)Z L A)((H 1)0 R   I) )))

            THE VALUE OF THE ABOVE LIST STRUCTURE IS -

(TMM)

29: (ATURING(QUOTE E)TMM(QUOTE(X))(QUOTE(1 1 Z 1 Y)))
E X*1*1 Z 1 Y
E X 2*1*Z 1 Y
E X 2 2*Z*1 Y
A X 2*2*Z 1 Y
B X 2 1*Z*1 Y
B X 2 1 Z*1*Y
B X 2 1 Z 1*Y*
C X 2 1 Z*1*Y
D X 2 1*Z*0 Y
F X 2*1*1 0 Y
F X*2*1 1 0 Y
G X 2*1*1 0 Y
A X*2*Z 1 0 Y
B X 1*Z*1 0 Y
B X 1 Z*1*0 Y
B X 1 Z 1*0*Y
B X 1 Z 1 0*Y*
C X 1 Z 1*0*Y
C X 1 Z*1*0 Y
D X 1*Z*0 0 Y
F X*1*1 0 0 Y
F*X*1 1 0 0 Y
H X*1*1 0 0 Y
I X 0*1*0 0 Y

            THE VALUE OF THE ABOVE LIST STRUCTURE IS -

HALTED

29:
```

VI.3 Utilities

VI.3.1 TRACE

If arg1 is a function, the effect of evaluating (TRACE arg1) is to produce trace printing whenever the function is entered subsequently. The function is modified so that it prints the values of the arguments in each function call, and also the value that it returns. Each message is indented a number of columns equal to the depth of recursion at the time.

Any number of functions can be in the trace condition at one time, and any function can be returned to normal by the function UNTRACE.

Below are some illustrations of the use of TRACE and UNTRACE.

```
    (DEFINE(QUOTE(
04: (ROTATERIGHT(LAMBDA(X)
05:   (COND
06:     ((NULL(CDR X))X)
07:     (T(PUTSECOND(CAR X)(ROTATERIGHT(CDR X))))
08: )))
09: (PUTSECOND(LAMBDA(Z Y)
10:   (CONS(CAR Y)(CONS Z(CDR Y)))
11: ))
12: )))

         THE VALUE OF THE ABOVE LIST STRUCTURE IS -

(ROTATERIGHT PUTSECOND)

13: (TRACE(QUOTE ROTATERIGHT))

         THE VALUE OF THE ABOVE LIST STRUCTURE IS -

(ROTATERIGHT)

14: (ROTATERIGHT(QUOTE(A B C)))
(ROTATERIGHT ENTERED WITH)
(A B C)
  (ROTATERIGHT ENTERED WITH)
  (B C)
   (ROTATERIGHT ENTERED WITH)
   (C)
   VALUE
   (C)
  VALUE
  (C B)
VALUE
(C A B)

         THE VALUE OF THE ABOVE LIST STRUCTURE IS -

(C A B)

15:
```

```
    (TRACE(QUOTE PUTSECOND))

         THE VALUE OF THE ABOVE LIST STRUCTURE IS -

(PUTSECOND)

70: (ROTATERIGHT(QUOTE(A B C)))
(ROTATERIGHT ENTERED WITH)
(A B C)
  (ROTATERIGHT ENTERED WITH)
  (B C)
   (ROTATERIGHT ENTERED WITH)
   (C)
   VALUE
   (C)
   (PUTSECOND ENTERED WITH)
   B
   (C)
   VALUE
   (C B)
  VALUE
  (C B)
  (PUTSECOND ENTERED WITH)
  A
  (C B)
  VALUE
  (C A B)
VALUE
(C A B)

         THE VALUE OF THE ABOVE LIST STRUCTURE IS -

(C A B)

71: (UNTRACE(QUOTE ROTATERIGHT))

         THE VALUE OF THE ABOVE LIST STRUCTURE IS -

(ROTATERIGHT)

72: (ROTATERIGHT(QUOTE(A B C)))
(PUTSECOND ENTERED WITH)
B
(C)
VALUE
(C B)
(PUTSECOND ENTERED WITH)
A
(C B)
VALUE
(C A B)

         THE VALUE OF THE ABOVE LIST STRUCTURE IS -

(C A B)

73:
```

Only functions defined by the programmer can be traced by the function TRACE. A function in the trace condition cannot be compiled, because it refers to the FSUBR PROG.

Below are the definitions of TRACE and UNTRACE. Some functions called by TRACE refer to a constant RECLEV, which is set to zero before the first evaluation of TRACE.

```
(TRACE(LAMBDA(F)
  ((LAMBDA(G)
    (DEFINE(LIST
      (LIST F(LIST(QUOTE LAMBDA)(CADR G)
        (LIST(QUOTE PROG)NIL
          (LIST(QUOTE PRIND)
               (LIST(QUOTE QUOTE)(CONS F(QUOTE
                       (ENTERED WITH)))))
          (LIST(QUOTE PRA)(CONS(QUOTE LIST)(CADR G)))
          (LIST(QUOTE PRV)(CADDR G)) )) ))) )
  (GET F(QUOTE EXPR)) )))

(PRA(LAMBDA(L)(COND((NULL L)(CSETQ RECLEV(ADD1 RECLEV)))
                   (T(PROG2(PRIND(CAR L))(PRA(CDR L))))
)))

(PRV(LAMBDA(X)
  (PROG2(CSETQ RECLEV(SUB1 RECLEV))
  (PROG2(PRIND(QUOTE VALUE))
          (PRIND X)))
))

(PRIND(LAMBDA(X)(PROG2(INDENT RECLEV)(PRINT X))))

(INDENT(LAMBDA(N)
  (COND((ZEROP N)NIL)(T(PROG2(PRIN1 BLANK)(INDENT(SUB1 N)))))
))

(UNTRACE(LAMBDA(F)((LAMBDA(G)
  (DEFINE(LIST(LIST F(LIST(QUOTE LAMBDA)(CADR G)
    (CADR(CADDDDR(CADDR G))) )))))(GET F(QUOTE EXPR))
)))
```

VI.3.2 PRETTYPRINT

The effect of evaluating (PRETTYPRINT s c) is to print the list structure s in a readable format, with column c as the left margin. PRETTYPRINT is intended for use with function definitions (which may be obtained even after definition by the use of GET).

Below is a sample of PRETTYPRINT output.

```
    (PRETTYPRINT(LIST(QUOTE PRETTYPRINT)(GET(QUOTE PRETTYPRINT)
02:     (QUOTE EXPR))) 0)

(PRETTYPRINT
  (LAMBDA
    (U LEFT)
    (COND
      ((OR (ATOM U) (LESSP (WIDTH U) (DIFFERENCE 70 LEFT)))
        (PROG2 (INDENT (SUB1 LEFT)) (PRINT U))
      )
      ((EQ (CARP (CAR U)) (QUOTE LAMBDA))
        (LIST
          (INDENT (SUB1 LEFT))
          (PRINT LPAR)
          (PRETTYPRINT (CAR U) (ADD1 LEFT))
          (PPL (CDR U) LEFT)
        )
      )
      (T
        (LIST
          (INDENT LEFT)
          (PRIN1 LPAR)
          (PRINT (CAR U))
          (PPL (CDR U) LEFT)
        )
      )
    )
  )
)

          THE VALUE OF THE ABOVE LIST STRUCTURE IS -

(NIL (PRETTYPRINT))

03:
```

VI.3.3 LAYOUT

The LAYOUT function is used to print or punch a sequence of assembly-language address constants which, when assembled, result in the implementation of the list structure which was its argument. LAYOUT is used by the LISP system programmers to add built-in EXPRS. (Because of its restricted application, no examples are given).

APPENDIX VII

RPLISS

To improve readability, LISP programs are usually written with blanks freely interspersed among list structures and line indentations to indicate list levels. However, in the case of large, frequently used programs which are debugged to the programmer's satisfaction, the associated punched card deck may be unmanageably large and may take a long time to input. In order to facilitate efficient deck handling, the separate program RPLISS is available under DOS.

RPLISS accepts, as input, a standard LISP program, and produces a punched card deck as output. The output deck is equivalent to the input deck, with all unnecessary blanks removed. Comment cards (those with an asterisk in column 1) in the input deck remain unchanged, and hence may be used to separate the output list structures.

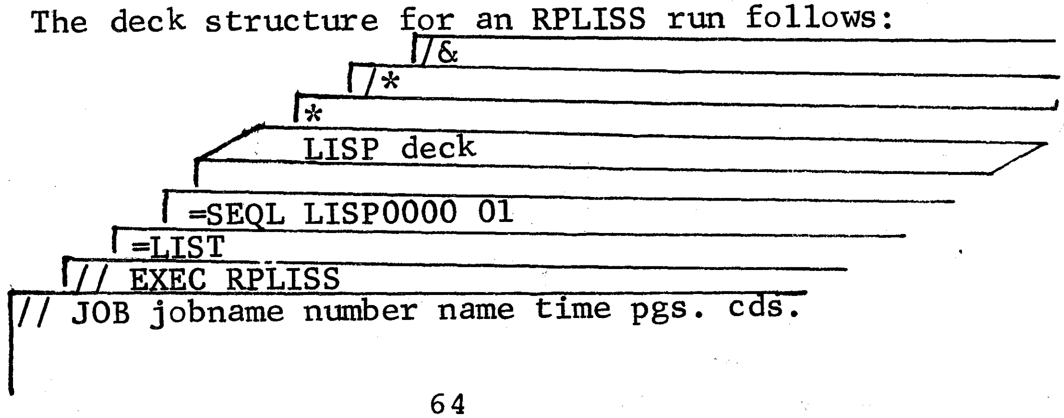
RPLISS utilizes two optional control cards which govern the output. Both cards must contain an equal sign (=, 6-8 punch) in column 1. The control information begins in column 2:

1) =LIST
   This card will cause RPLISS to produce a listing of the output deck on the device corresponding to SYSLST.

2) =SEQL sequencestart increment
   This card causes the identification field (columns 73-80) of the output deck to be sequenced. The 'sequencestart' field should contain a 4-character identification name beginning in column 7, immediately followed by the 4-digit sequence number of the first output card. Columns 16 and 17 should contain the 2-digit sequencing increment.

The output deck is produced on the device corresponding to SYSPCH.

The deck structure for an RPLISS run follows:



APPENDIX VII

Index to functions in the RPI system

Function	Indicator	Classification	Text section
ADDBPS	SUBR	System(I.4)	12.2.4
ADD1	SUBR	Arithmetic(I.2)	
AND	FSUBR	Logical(I.3)	
ANDL	SUBR	Logical(I.3)	
APPEND	SUBR	General(I.1)	
ATOM	SUBR	General(I.1)	6.4.1
CAR	SUBR	General(I.1)	4.4.1
CDR	SUBR	General(I.1)	4.4.2
CLEARBPS	SUBR	Compiler(I.9)	12.2.5
COMPILE	EXPR	Compiler(I.9)	12.2.1
COND	FSUBR	General(I.1)	6.3,II.6
CONS	SUBR	General(I.1)	
CR	SUBR	General(I.1)	
CSET	EXPR	Property-list(I.5)	9.2.7
CSETQ	FEXPR	"	"
DECK	SUBR	Compiler(I.9)	12.2.2
DEFCR	SUBR	General(I.1)	4.4.6, 13.3.6
DEFINE	EXPR	Property-list(I.5)	5.
DEFLIST	SUBR	"	9.2.5
DIFFERENCE	SUBR	Arithmetic(I.2)	
EFFACE	SUBR	List-modifying(I.6)	9.4
EJECT	SUBR	I/O(I.7)	11.4
EQ	SUBR	General(I.1)	6.4.3
EQUAL	SUBR	"	6.4.2
EVAL	SUBR	"	
EXCISE	SUBR	System(I.4)	12.2.6
EXPLODE	SUBR	General(I.1)	
EXPT	SUBR	Arithmetic(I.2)	
GENSYM	SUBR	General(I.1)	
GET	SUBR	Property-list(I.5)	9.2.5
GO	SUBR	Program feature(I.8)	II.4
GREATERP	SUBR	Arithmetic(I.2)	
IMPLODE	SUBR	General(I.1)	
LABEL	FSUBR	General(I.1)	
LENGTH	SUBR	"	
LESSP	SUBR	Arithmetic(I.2)	
LIST	FSUBR	General(I.1)	

The RPLISS program may also be used under OS. The same effects and internal control cards appear; the deck structure is below:

```
//jobname   JOB   labnumber,name,MSGLEVEL=1

//JOBLIB   DD   DSNAME=LISP,DISP=SHR

//stepname   EXEC   PGM=RPLISS

//SYSPRINT   DD   SYSOUT=A

//SYSPUNCH   DD   SYSOUT=B

//SYSIN      DD   *

         (LISP deck)

/*

//
```

APPENDIX VIII

LISP Debugging Hints

Whenever the LISP processor discovers an error, a recovery process is initiated. In most cases, this procedure results in the printing of an error message (cf. Appendix V), the aborting of attempts to evaluate the current list structure, and the continued processing of the remaining list structures. Some errors are so severe as to necessitate job termination; when this occurs under OS, a user completion code is printed (corresponding to an error message number in Appendix V) and, if a SYSUDUMP card is present, a dump will appear. (Since a LISP dump is extremely long, it is advised that a LISP system programmer be acquainted with the problem before the dump is requested.) If an error message is, indeed, printed, the best procedure is to locate the message in Appendix V and examine the accompanying reference section.

If no error message is printed and 'END OF LISP RUN' appears, but function calls are printed without evaluation, a parenthesis miscount is indicated; that is, there is at least one missing right parenthesis, or extra left parentheses. If your functions do not contain too complicated list structures, you may wish to check matching parentheses by hand. LPCP (Appendix IV) may be used to allow the computer to check for you.

The appearance of system completion code 106 advises that your //JOBLIB card should be removed. A system completion code of 322 means your program ran out of time. If you feel that the time allotted was sufficient, then your program is most likely looping indefinitely.Under OS, whenever your job ends with a non-zero system completion code, not all of your output may be printed. To guarantee that you get all your output, include the following control card just before your //LISP.SYSIN :

```
//LISP.SYSPRINT  DD  DCB=(BLKSIZE=121,BUFNO=1)
```

Trqcing your program logic is an effective way of discovering problem spots. The LISP TRACE function (as described in Appendix VI) can be used for this purpose. Under OS, this function can automatically be added to your deck by replacing the //LISP.SYSIN DD * card with the two cards:

```
//LISP.SYSIN  DD  DSNAME=LISP(TRACE),DISP=SHR

//            DD  *,DCB=BLKSIZE=80
```

Function	Indicator	Classification	Text section
MAPCAR	SUBR	General(I.1)	
MAX	FSUBR	Arithmetic(I.2)	
MAXL	SUBR	"	
MEMBER	SUBR	General(I.1)	
MIN	FSUBR	Arithmetic(I.2)	
MINL	SUBR	"	
MINUS	SUBR	"	
MINUSP	SUBR	"	
NCONC	SUBR	List-modifying(I.6)	9.4
NODECK	SUBR	Compiler(I.9)	12.2.2
NOT	SUBR	Logical(I.3)	
NULL	SUBR	General(I.1)	6.4.4
NUMBERP	SUBR	Arithmetic(I.2)	8.2
ONEP	SUBR	Arithmetic(I.2)	
OR	FSUBR	Logical(I.3)	
ORL	SUBR	"	
PLUS	FSUBR	Arithmetic(I.2)	
PLUSL	SUBR	"	
PRINT	SUBR	I/O(I.7)	11.3
PRIN1	SUBR	"	11.5.3
PROG	FSUBR	Program feature(I.8)	II.2
PROG2	SUBR	General(I.1)	
PUNCH1	SUBR	I/O(I.7)	11.5.3
QUOTE	FSUBR	Gernral(I.1)	4.3
QUOTIENT	SUBR	Arithmetic(I.2)	
READ	SUBR	I/O(I.7)	11.2
RECLAIM	SUBR	System(I.4)	
REMPROP	SUBR	Property-list(I.5)	9.2.5
RPLACA	SUBR	List-modifying(I.6)	9.4
RPLACD	SUBR	"	"
SASSOC	SUBR	General(I.1)	
SETQ	FSUBR	Program feature(I.8)	II.5
SIZE	FSUBR	System(I.4)	
SUB1	SUBR	Arithmetic(I.2)	
TERPCH	SUBR	I/O(I.7)	11.5.2
TERPRI	SUBR	"	11.5.1
TIME	SUBR	System(I.1)	
TIMES	FSUBR	Arithmetic(I.2)	
TIMESL	SUBR	"	
ZEROP	SUBR	Arithmetic(I.2)	

REFERENCES

(1) McCarthy, J., et al. LISP 1.5 Programmer's Manual. The MIT Press, Cambridge, Massachusetts, 1965.

(2) Berkeley, E. C. and Bobrow, D. G., editors. The Programming language LISP: Its Operation and Applications. The MIT Press, Cambridge, Massachusetts, 1966.

(3) McCarthy, J. 'Recursive functions of Symbolic expressions and their computation by machine'. Communications of the ACM, April, 1960.

(4) Rosen, Saul, editor. Programming Systems and Languages, McGraw-Hill Book Company, New York, 1967