THE PROGRAMMING LANGUAGE LISP:

Its Operation and Applications

Information International, Inc.

THE PROGRAMMING LANGUAGE LISP:

Its Operation and Applications

EDMUND C. BERKELEY *and* DANIEL G. BOBROW
editors
and fifteen authors

This research was sponsored by the Advanced Research Projects Agency, Department of Defense, Washington, D. C., under Contract SD 162. This technical report was prepared under said Contract.

Published by
Information International, Inc.
200 Sixth St.
Cambridge, Mass. 02142

March, 1964

First printing, March, 1964

Printed in the United States of America

Authors

Authors

Paul W. Abrahams

Edmund C. Berkeley

Fischer Black

Daniel G. Bobrow

Thomas G. Evans

Mark Finkelstein

Edward Fredkin

Elaine Gord

William Henneman

Timothy P. Hart

Michael I. Levin

Lionello A. Lombardi

Malcolm Pivar

Bertram Raphael

Robert A. Saunders

Editors

Edmund C. Berkeley

Daniel G. Bobrow

Acknowledgements

Work reported here by authors Edmund C. Berkeley, L. Peter Deutsch, Mark Finkelstein, Edward Fredkin, Elaine Gord, William Henneman, Malcolm Pivar, and Robert A. Saunders was supported entirely or largely by Information International Inc. under contract SD-162 with the Advanced Research Projects Agency, Department of Defense.

Work reported here by authors Daniel G. Bobrow, Timothy P. Hart, Michael I. Levin, L. A. Lombardi, and Bertram Raphael was mainly or partially supported by Project MAC, an M.I.T. research program sponsored by the Advanced Research Projects Agency, under Office of Naval Research Contract Nonr-4102(01). Some of the work of Bobrow and Hart was also supported by Information International under its contract; and some of the work by Berkeley and Deutsch was also supported by Project MAC under its contract.

The articles and papers (or portions thereof) by authors Paul W. Abrahams, Fischer Black, and T. G. Evans were received as welcome contributions to this report, and were not supported by the foregoing contracts.

All the assistance which has been contributed to the substance and content of this report has been most helpful and is gratefully acknowledged.

Grateful acknowledgement should also be made to the persons who typed or drew the final photo offset master copy for this report: Ann Baker, Diana Cormier, Michelle Ingersoll, Gayle Johnson, and Lorraine Simkins.

Edmund C. Berkeley

Daniel G. Bobrow

Editors

Preface

One of the tasks under Contract SD-162 issued by the Advanced Research Projects Agency to Information International, Inc., was to help make the programming language LISP more understood, more available, and more useful for programmers and mathematicians. The language LISP (short for a "LISt Processing" language) is a remarkable and powerful language, because not only does it govern the operation of a computer, but also it is a mathematical language with great flexibility and power for expressing processes in mathematics, logic, and symbol manipulation in general.

As a part of the present task, this collection of contributions from various authors has been prepared and published.

The section of this report "Acknowledgments" states the connections of the authors with various projects and activities, the original sources of the contributions, and the support which enabled the contributions to be written. The authors include many persons who have worked under contracts with the Advanced Research Projects Agency other than the contract with Information International Inc.

Part I of this report contains articles and papers written primarily for persons with no prior knowledge of LISP or only a little. If anybody desires to learn LISP, however, Part I of this report is not sufficient, and he should have at hand a copy of: "LISP 1.5 Programmer's Manual", by John McCarthy and others; published by The MIT Press, Cambridge 39, Mass.; date, August 17, 1962; cost, $3.00.

Part II of this report contains articles and papers written primarily for persons with a substantial prior knowledge of LISP.

Copies of this report are available from the Defense Documentation Center (DDC) and the Office of Technical Services (OTS).

We should like to express our thanks and appreciation to all those persons who contributed to this report, and enabled it to become, in our opinion, a step along the road towards more versatile and more powerful operation and control of computers.

Also, we should like to express our great appreciation to the Advanced Research Projects Agency for its support of this work, for otherwise much of what is here would never have come into existence.

In spite of many efforts to avoid errors, no editors or authors can be sure that all errors have been eliminated. Any corrections, comments, or suggestions sent to the editors will be very welcome.

Information International Inc.

Edmund C. Berkeley

Daniel G. Bobrow

Distribution

	Addressee	Number of Copies
1.	Advanced Research Projects Agency Washington, D. C. Att'n Dr. J. C. R. Licklider	50
2.	Carnegie Institute of Technology Computation Center, Pittsburgh, Pa. Att'n Dr. Alan Perlis	30
3.	Information International, Inc. Cambridge, Mass. Att'n Mr. Edward Fredkin	150
4.	Mass. Inst. of Technology, Project MAC and Computation Center, Cambridge, Mass. Att'n Mr. Richard Mills	100
5.	Stanford Research Institute Palo Alto, Calif. Att'n Mr. John H. Wensley	20
6.	Stanford University, Computer Science Division Stanford, Calif. Att'n Prof. John McCarthy	60
7.	System Development Corporation Santa Monica, Calif. Att'n Mr. Arthur M. Rosenberg	50
8.	University of California at Berkeley Berkeley, Calif. Att'n Prof. Edward Feigenbaum	20
9.	University of California at Los Angeles Los Angeles, Calif. Att'n Mr. C. A. Irvine	20
10.	Defense Documentation Center (DDC) Cameron Station, Va.	500

Table of Contents

PART III — Appendices

LISP — A Simple Introduction

Edmund C. Berkeley

Information International, Inc.

TABLE OF CONTENTS

I. INTRODUCTION

II. CONDITIONS, PREDICATES, AND NUMERICAL FUNCTIONS

III. DEFINING AND USING NEW TERMS AND EXPRESSIONS

IV. ATOMIC EXPRESSIONS

V. RECURSIVE DEFINITIONS

VI. THE PROGRAM FEATURE

VII. FUNCTIONS OF LISTS

VIII. CONCLUDING REMARKS

APPENDIX

I. INTRODUCTION

1. What is LISP?

Among the new languages for instructing computers is a remarkable one called LISP. The name comes from the first three letters of LIST and the first letter of PROCESSING. Not only is LISP a language for instructing computers but it is also a formal mathematical language, in the same way as elementary algebra when rigorously defined and used is a formal mathematical language.

The LISP language and its implementation on the IBM 7090 computer were worked out by a group including John McCarthy, Stephen B. Russell, Daniel J. Edwards, Paul W. Abrahams, Timothy P. Hart, Michael I. Levin, Marvin L. Minsky, and others.

LISP is designed primarily for processing data consisting of lists of symbols. It has been used for symbolic calculations in differential and integral calculus, electrical circuit theory, mathematical logic, game playing, and other fields of intelligent handling of symbols.

Much authoritative information about LISP can be found in: "LISP 1.5 Programmer's Manual" by John McCarthy, Paul W. Abrahams, Daniel J. Edwards, Timothy P. Hart, and Michael I. Levin, published by The M.I.T. Press, Mass. Inst. of Technology, Cambridge 39, Mass., dated August 17, 1962, 105 pp. Most people approaching LISP find, however, that this manual is rather difficult to comprehend on a first reading, because in many cases the ideas and terms are presented from an advanced point of view assuming a good deal of already known information on the part of the reader.

The purpose of the present article is to make a bridge between the ideas and terms of ordinary English and elementary mathematics, and the ideas and terms known and used by LISP programmers.

This bridge begins in Section 3 below, but before we start on the bridge, we need to make some remarks about the way the LISP system operates inside a computer. These remarks are in Section 2 — which may be skipped on a first reading by someone who is willing to take "the LISP interpreter" (the program on the computer which interprets LISP expressions) on faith for the time being.

2. 7090 LISP, and PDP-1 LISP

a. Two Forms of LISP

We shall discuss two distinct forms of LISP. One is the original and full form of LISP for operation on the very powerful (and expensive) IBM 7090 computer made by International Business Machines Corporation. This form of LISP, because of the speed and capacity of the computer, enables many interesting and important investigations to be made.

The usual way in which LISP expressions are given to the 7090 computer to be evaluated is through punch cards prepared ahead of time by the human programmer. Results produced by the computer are usually put on magnetic tape, which is then printed off-line. But sometimes with a time-sharing facility, as for example at Project MAC at Mass. Inst. of Technology, the 7090 computer is directly accessible to the human being by means of an electrically controlled typewriter or teletype station. In the case of direct time-sharing access, the computer takes control of the electric typewriter or teletype, and operates it.

The second form of LISP is a somewhat simplified and modified form of LISP which was worked out by L. Peter Deutsch for operation on the PDP-1 computer made by Digital Equipment Corp., Maynard, Mass. This form of LISP uses for the basic functions about 1500 registers, and for working storage from about 500 to about 14000 registers (the latter in a four-core PDP-1) as may be chosen. In comparison with 7090 LISP, PDP-1 LISP is significantly limited. But it is useful because it is flexible, permits much investigation, and the correction of preliminary expressions. In PDP-1, beyond the basic functions, only those functions chosen may be included in the system when placed for use on the computer; this choice is not possible in 7090 LISP. Additional technical information about PDP-1 LISP is given in a later appendix in this book.

The usual way in which LISP expressions are given to the PDP-1 computer to be evaluated is through typing on the keys of the computer-associated typewriter, or through punched paper tape read in by the photoelectric tape reader. Results produced by the computer are usually given to the human being either by characters typed automatically by the typewriter or punched automatically in paper tape, which may be printed in an off-line Flexowriter.

b. Functions and Properties Available

The functions and properties available for use in 7090 LISP are stated in the LISP 1.5 Manual. Additional functions such as

LAST (in the sense "the last of" a list) are also available from program libraries in the computer centers where the LISP system has been implemented on the 7090 computer.

The functions and properties available for use in PDP-1 LISP are many fewer and are stated in the appendix on PDP-1 LISP.

Additional functions for both forms of LISP may be readily defined and read in to the LISP system when desired.

c. Differences between 7090 LISP and PDP-1 LISP

The differences between the two forms of LISP are like the differences in idiom or dialect between one part of a country and another part of the country. We will consider the form (or idiom) of LISP for the 7090 and the form (or idiom) of LISP for the PDP-1.

d. The LISP Interpreter

First of all, when any calculation whatever is given to any appropriately programmed computer, the machine:

— takes in signals or characters expressing the calculation which is intended;
— performs internal calculating operations; and
— puts out the result of the calculation.

In the case of a computer programmed with a LISP system, we say that the machine (the programmed computer):

— takes in an expression written in LISP;
— applies the LISP interpreter and evaluates the expression; and
— puts out the value of the expression.

For example, in multiplying 3486 by 6598 on an ordinary Friden desk calculator, the information put in consists of:

(1) pressing the keys 3, 4, 8, 6 (in columns 4, 3, 2, 1, respectively, but in any arbitrary sequence) on the main keyboard;
(2) pressing the keys 6, 5, 9, 8 (in that particular sequence) on the multiplier keyboard; and, finally,
(3) pressing the key "MULT", which causes the machine to start computing.

In other words, the desk calculator takes in an expression equivalent to 3486 x 6598, evaluates it, and gives out the value

23000628 in the dials of the result register.

The LISP interpreter is an elaborate function inside the computer which essentially operates on two arguments given to it. The first argument is the name of a function. The second argument is a list of one or more arguments to which that function is to be applied. Of course, the 7090 LISP interpreter can respond properly to several hundred function names; it is not restricted like the ordinary desk calculator to responding to just one of the four arithmetical functions, addition, subtraction, multiplication, and division.

e. The Internal 7090 Form, the External 7090 Form, and the PDP-1 Form

The 7090 LISP interpreter handles expressions in two styles or idioms or forms. One of these is called the internal form. The internal form is the way in which expressions are handled inside the 7090 computer almost all the time. Also, it is the usual standard form of LISP expressions inside a computer regardless of what kind of computer it may be. The other of these forms is called the external form, or the form on the top level; this is the form in which expressions go into the 7090 computer. It saves some trouble and time for human beings, to have this second style or form for use in going into the 7090 computer.

The PDP-1 LISP interpreter handles expressions only in one form. This form is almost the same as the internal form of expressions for the 7090 LISP interpreter, but not quite.

In the following explanation of LISP, we shall nearly all the time be dealing with expressions in the internal form for the 7090 LISP interpreter, and the only form for the PDP-1 LISP interpreter. From time to time we shall discuss the relation between the internal form of expressions for the 7090 LISP interpreter and the external form of such expressions.

3. A Very Simple Example of LISP

Let us now consider a very simple example of LISP.

Suppose that we take a list consisting for example of A, B, C, D, E in that order. Suppose we choose the problem:

Select the first element in that list.

The following expression:

```
(CAR (QUOTE (A B C D E)))
```

if evaluated by the LISP interpreter (the programmed computer which interprets the LISP expression) would give the answer:

A

Since the first one of the list "A, B, C, D, E" is "A", the interpreter has operated correctly.

The way in which this expression would usually be put into the 7090 LISP interpreter would make use of the external form (or idiom — see Section 2) and would be:

CAR ((A B C D E))

The way in which this expression would be put into the PDP-1 LISP interpreter would be the same as the first expression:

(CAR (QUOTE (A B C D E)))

together with one more pressing of the typewriter space bar following the last parenthesis.

4. Meaning of CAR

In order to understand what has happened, let us take a look at the various parts of the example. What does CAR mean? The word CAR is a LISP expression which is a function name. Its meaning is "the first of". CAR applied to any list of elements produces the first element in the list. The word "CAR" is derived from the initial letters of three words in the phrase "Contents of Address part of Register", and this phrase has to do with the organization of the computer registers to hold lists.

5. Meaning of QUOTE

What does QUOTE mean? The expression QUOTE tells the computer that what follows is to be treated as itself, not as the name for something else. This meaning is like the meaning in ordinary English when we use quotation marks and say:

"Paris" has five letters.

and mean:

The word Paris has five letters.

We do not say:

Paris has five letters.

because Paris is a city and it makes no sense to say that a city has five letters; what a city has is people, streets, buildings, etc.

In English one of the standard uses of quotation marks is to produce a name for an expression, instead of designating what the expression usually refers to. This is the use of QUOTE in LISP.

6. Use of Parentheses

The expression:

```
(CAR (QUOTE (A B C D E)))
```

uses six parentheses. They are very important. They designate scope or extent of expressions — i.e., where they begin and where they finish. Parentheses have to be very precisely positioned. In order to understand them, we shall first number them in associated pairs:

```
(CAR (QUOTE (A B C D E)))
1    2      3         321
```

The first left parenthesis No. 1 tells the LISP interpreter that this is the start of a calculation. The final right parenthesis No. 1 tells the interpreter that this is the finish of an expression. The interpreter evaluates the expression and produces the answer.

The first parenthesis No. 1 marks the beginning of the scope of CAR, the extent of the expression to which CAR applies. The second parenthesis No. 1 marks the end of the scope of CAR.

The first parenthesis No. 2 marks the beginning of the scope of QUOTE, and the second parenthesis No. 2 marks the end of the scope of QUOTE.

The first parenthesis No. 3 marks the beginning of a list, and the second parenthesis No. 3 marks the end of the list.

Always, all parentheses in the expressions of LISP language occur in pairs of left and right parentheses; generally, each pair of parentheses marks scope, the extent to which an expression applies. The parentheses in LISP are never optional as they are sometimes in mathematics: they are required parts of expressions.

7. Separation of Expressions

The expression:

```
(CAR (QUOTE (A B C D E)))
```

contains within it seven separate expressions CAR, QUOTE, A, B, C, D, and E. It is important that each expression be separated, marked off, or delimited from another expression.

Four characters are used in LISP for separating expressions: left parenthesis, right parenthesis, comma, and space. In LISP the comma is completely interchangeable with the space; also one or more spaces are treated exactly as a single space.

In 7090 LISP a left parenthesis or a right parenthesis delimits an expression, whether or not any spaces are between the expression and the parenthesis. Thus:

```
(CAR(                    ( CAR(
(CAR (                   ( CAR (
```

are all freely interchangeable, but the style (CAR (is the preferred form. In cases where parentheses do not occur, as for example between elements in the list A B C D E, the spaces delimit the expressions.

In PDP-1 LISP, of the four expressions involving CAR just above, three are acceptable, but only:

```
(CAR (
```

is properly delimited. In PDP-1 LISP, the spaces are important and have to be precisely positioned. To talk about the spaces here, we shall number them:

```
(CAR (QUOTE (A B C D E)))
    1      2  3 3 3 3    4
```

The first space (No. 1) tells the interpreter that it has reached the end of the expression CAR. If the first three characters of an expression were CAR and the fourth character were any character except a space (or a parenthesis or a comma) say X, then the computer would treat CARX as either part or all of another expression different from CAR, and not to be confused with CAR.

The second space, No. 2, terminates the expression QUOTE,

in the same way as the first space No. 1 terminates the expression CAR.

The next four spaces all numbered 3 and separating A B C D E tell the machine that the expressions separated by them are all part of the same list. In ordinary English, we might write "A, B, C, D, E" writing commas — but the LISP interpreter regularly takes in and puts out lists using spaces without commas.

The last space numbered 4 tells the interpreter in PDP-1 LISP to proceed to the calculation, and to put out the result. This space is not necessary in 7090 LISP.

8. The Function CDR

We have illustrated CAR as a function in LISP. What are some other functions?

One of them is CDR. pronounced "could-er", which means "the rest of". CDR of the list A B C D E is the list B C D E. To express this in a way which is acceptable to the LISP interpreter, the following is used:

```
(CDR (QUOTE (A B C D E)))
```

and the LISP interpreter gives as result: (B C D E). The two parentheses around (B C D E) cause the LISP interpreter to treat this expression as a list of four elements. The first parenthesis marks the beginning of the list, and the last parenthesis marks the end of the list.

This raises the question: Is A the same as (A)? The answer is "No". A is an element, and (A) is a list containing one element A.

The abbreviation CDR comes from the phrase "Contents of Decrement part of Register."

9. The Function CONS

Another function in LISP is CONS, pronounced with a soft s, which refers to "the construct of" when applied to an element and a list, or two lists. CONS of the element A and the list B C D E is the list A B C D E.

If the LISP interpreter evaluated:

```
(CONS (QUOTE A) (QUOTE (B C D E)))
```

then the result would be:

(A B C D E)

The function CONS puts back together what CAR and CDR take apart. For example, suppose we have the list A, B, C, D. CAR gives the element A. CDR gives the list B,C,D. CONS applied to these two portions gives the list A, B, C, D.

This is correctly written for the LISP interpreter as:

(CONS (CAR (QUOTE (A B C D))) (CDR (QUOTE (A B C D))))

and the result is:

(A B C D)

To put the first element of the list A B C D together with the rest of the list E F G H, we use the same functional expression with one change in the last argument:

(CONS (CAR (QUOTE (A B C D))) (CDR (QUOTE (E F G H))))

and the result is:

(A F G H)

In some cases, CONS will be used to operate on two lists, as in:

```
(CONS (QUOTE (A B)) (QUOTE ((C D) (E F))))
0     1      2   21 1      23   3 3   3210
```

The result in this case is:

((A B) (C D) (E F))

which is a list of three lists, each of which is a list of two elements. Notice how pairs of parentheses with the same number delimit the scope of expressions.

II. CONDITIONS, PREDICATES, AND NUMERICAL FUNCTIONS

If CAR, CDR, and CONS were the only three functions in LISP, then of course the language would be quite uninteresting, and hardly anything of importance could be accomplished. The first step towards a more interesting language and more important results is the treatment by LISP of what are called <u>conditional expressions</u>.

1. Conditional Expressions

In LISP language a conditional expression has the following kind of pattern:

> If statement P is true, take expression E. Otherwise,
> if statement Q is true, take expression F. Otherwise,
> if statement R is true, take expression G.

The truth values of the statements are examined in sequence by the computer, until the first true statement is found. Then the expression paired with this true statement is taken as the value of the entire conditional expression. If none of the statements are true, the value of the entire conditional expression is undefined.

The conditional expression above would be correctly written for the LISP interpreter as follows, under the assumption that each of P, Q, R, E, F, and G is an expression long enough to need around it parentheses marking its scope:

```
(COND ((P) (E)) ((Q) (F)) ((R) (G)))
```

In the following, the parentheses are numbered to indicate the scope of expressions. A left parenthesis is paired with the first right parenthesis following it, bearing the same number.

```
(COND ((P) (E)) ((Q) (F)) ((R) (G)))
0     12 2 2 21 12 2 2 21 12 2 2 210
```

The parentheses numbered 1 mark the start and finish of certain pairs for COND to pay attention to; the parentheses numbered 2 mark off the constituents of each pair, the statement and its corresponding expression.

The result of giving this LISP expression to the computer depends on which of the statements P, Q, R is true. The result is: E if P is true; F if P is false and Q is true; G if both P and Q are false and R is true; and undefined, if all of P, Q, and R are false.

But this expression will not operate in the computer unless the computer has a way of determining whether the statements marked P, Q, R are true or false. And at this point, we have not told the computer anything about statements P, Q, or R, and the computer has no way of computing the result of the expression.

In order to illustrate how the computer can compute the result of an expression like this, we may put in known truth values in place of the unknown truth values of the statements P, Q, R. LISP recognizes two truth values, T for "true" and NIL for "false". These are so basic to the operation of LISP that LISP provides that QUOTE does not have to precede them. T is treated the same as (QUOTE T) and NIL is treated the same as (QUOTE NIL). (Note: This is because T has the constant value T and NIL the constant value NIL.)

To put in known truth values, in place of the conditions in COND, is ordinarily not useful except for illustration, with one important exception; in the case of the last condition occurring in the whole conditional expression (the condition which refers to all the remaining cases), T is regularly used. (Note: There is an exception to this "regular rule"; the exception is explained below under the topic "The Program Feature".)

For example, evaluating the expression:

```
(COND (NIL (QUOTE D))
0     1    2       21
       (T (QUOTE E)))
       1  2       210
```

will give the result E. The conditional expression:

```
(COND (T (QUOTE D))
0     1  2       21
       (NIL (QUOTE E)))
       1    2       210
```

has the value D. The conditional expression:

```
(COND (T (QUOTE D))
       (T (QUOTE E)))
```

also has the value D.

If we give the computer:

```
(COND (NIL (QUOTE D))
      (NIL (QUOTE E)))
```

then the computer will report an error and print out as a "diagnostic" for the error, a symbol which stands for "illegal COND". It is illegal because no condition is true.

2. Absolute Value

A specific, familiar example of a conditional expression in elementary mathematics is the ordinary definition of the absolute value of a number. The absolute value of a number X is a number which may be defined in precise English (resembling the LISP conditional expression) as:

> If X is negative, take minus X. Otherwise, take X.

Let's write this in LISP. In LISP we have available a function GREATERP, which is used in the expression (GREATERP X Y), standing for "X is greater than Y". It is true if X is greater than Y, and false if X is not greater than Y. In LISP we also have available the function (MINUS X) standing for "the negative of X".

So the absolute value of X could be computed in LISP from:

```
(COND ((GREATERP 0 X) (MINUS X))
0     12              2 2       21
      (T X))
      1   10
```

This expression as it stands would however not be accepted by the computer, because LISP would not "know" what number X stood for, and so it could not compute the result. In order for this expression to be accepted by the computer, X would have to be replaced by a specific number, such as 36, or (MINUS 13), or 999975, etc.

Another point: in the case of a number we do not have to use QUOTE, because a number is accepted as itself. For example, we do not have to write (MINUS (QUOTE 36)); this we can write simply as (MINUS 36). The computer will however accept both (MINUS 36) and (MINUS (QUOTE 36)).

3. Predicates

In order to make use of COND in LISP, there have to be expressions which can be true or false, and these are called "predicates". We shall consider some examples:

> EQ, NULL, NUMBERP, and GREATERP

These are available in PDP-1 LISP and in 7090 LISP. In fact, we have already used the predicate GREATERP.

4. The Predicate EQ

EQ is a predicate which accepts two atomic symbols (we shall define this term a little later) such as "B" or "CONS" or "367", and compares them character by character. If the two atomic symbols are precisely the same character by character, then EQ is true, otherwise not. For example, it is true that "the atomic symbol A equals the atomic symbol A". So:

```
(EQ (QUOTE A) (QUOTE A))
```

gives as the result T standing for the truth value true.

It is false that "the atomic symbol A equals the atomic symbol B". So:

```
(EQ (QUOTE A) (QUOTE B))
```

gives as the result NIL, standing for the truth value false.

In 7090 LISP, EQ is not defined for numbers; another predicate EQUAL is used instead. This is because in 7090 LISP numbers are not stored as characters. In PDP-1 LISP, however, EQ is defined for numbers; and, for example, the expression

```
(EQ 7 7)
```

is accepted and has the value T.

5. The Predicate NULL and the Empty List

The predicate NULL accepts a single argument X, which may be any LISP expression. If this argument X when evaluated is equal to NIL, then (NULL X) is true, and has the value T. If the argument X when evaluated is not equal to NIL, then (NULL X) is false, and has the value NIL.

By convention, there is only one empty list in LISP; this is a list (or the list) containing no elements. It is denoted by ( ) or by NIL; these representations are equivalent. Thus, if L is a list, then (NULL L) is true if L is empty, and is false otherwise.

For example,:

```
(CDR (QUOTE (A)))
```

is ( ), or NIL, the empty list. Therefore:

```
(NULL (CDR (QUOTE (A))))
```

is true, and has the value T.

Since NIL is also used as the truth value for "false", if P is a statement which can be true or false, then (NULL P) has the value T if P is true, and the value NIL if P is false. So, in this case (NULL P) is equivalent to NOT-P.

For example, it is true that "the atomic symbol A is not equal to the atomic symbol B". So:

```
(NULL (EQ (QUOTE A) (QUOTE B)))
```

has the value T.

In 7090 LISP, there is a separate function NOT equivalent to NULL, and F is used as well as NIL to denote the truth value false.

6. The Predicate NUMBERP

This predicate stands for "is a number". "36 is a number" becomes:

```
(NUMBERP 36)
```

Put into the computer, it will give as a result: T.

7. The Predicate GREATERP

As mentioned before, this predicate stands for "is greater than". "36 is greater than 34" becomes "the truth value of 36 being greater than 34";

```
(GREATERP 36 34)
```

will give as the result T.

8. Numerical Functions

Predicates are functions that have for values either true (T) or false (NIL). There exist in LISP many kinds of functions besides predicates, including functions which may have numbers, letters, lists, etc., for their values.

We shall consider first some of the functions of numbers.

Among the numerical functions defined in LISP are PLUS, MINUS, TIMES, and QUOTIENT. PLUS takes any number of arguments. MINUS takes only one argument (producing the negative of a number), so that subtraction has to be performed using both PLUS and MINUS. TIMES takes any number of arguments. QUOTIENT takes two arguments, the first one being the dividend and the second one being the divisor.

For example:

Example of LISP Expression	Result in Decimal (7090)	Result in Octal (PDP-1)
(PLUS 2 3 5)	10	12
(MINUS 5)	-5	777772
(TIMES 2 3 6)	36	44
(QUOTIENT 6 3	2	2
(QUOTIENT 334 333)	1	1
(QUOTIENT 333 334)	0	0

III. DEFINING AND USING NEW TERMS AND EXPRESSIONS

One of the features of LISP is the power to define new terms and expressions, as may be desired, and then to make use of them. This is like the power in mathematics when solving a problem to say "Let x equal" or "Let F(x) be a function such that...."

1. CSET

The expression CSET in LISP establishes (or "sets") a name which will have a given constant value. (The letter C in CSET refers to the first letter of "Constant".) The constant value may be determined as a number, or a list, or the result of evaluating another expression. Then, whenever you may make use of the name, the computer will substitute the value. You make use of CSET with the LISP idiom:

```
(CSET (QUOTE .....)  (QUOTE .....))
                1              2
```

where Blank 1 is filled with the name you have chosen, and Blank 2 is filled with the expression. If Blank 2 is filled with a number, the parentheses and the QUOTE may be dropped.

For example, if the LISP interpreter evaluates:

```
(CSET (QUOTE DAN) 314)
```

the name DAN is given a permanent value 314, and you may use DAN at any time, meaning the number 314. For example you can use DAN in an expression like the following:

```
(PLUS DAN DAN DAN)
```

The interpreter will produce:

```
942 (in decimal, with the IBM 7090), or
1144 (in octal, with the PDP-1)
```

To give to the computer the instruction to set the name DAN at 314, you would instruct the IBM 7090 LISP system at the top level:

```
CSET (DAN 314)
```

but to instruct the PDP-1 LISP system you would use:

```
(CSET (QUOTE DAN) 314)
```

If you wish to look up the expression at which DAN has been set, you may give the LISP interpreter:

DAN

and the interpreter will give the value:

314

2. CSETQ

The expression CSETQ in LISP, like the expression CSET, establishes or "sets" a name which will have as its value a given constant, or the result of evaluating another expression. You make use of CSETQ with the LISP idiom:

```
(CSETQ ..... (QUOTE .....))
        1           2
```

where Blank 1 is filled with the name you have chosen, and Blank 2 is filled with the expression.

For example, you may designate the meaning of JILL by putting in:

(CSETQ JILL (CAR (QUOTE (D E F G))))

If the LISP interpreter receives this expression, it sets the value of JILL to the value of the second expression.

If in PDP-1 LISP you wished to verify what JILL stands for, you would put in:

JILL

and the PDP-1 would respond by typing out:

D

which is correct, since D is the first one of the list D E F G. On the 7090, you could have the interpreter find the value of JILL by having it evaluate:

(PRINT JILL)

CSET and CSETQ are identical except that with CSET you must write QUOTE in front of the first argument, and with CSETQ you must not.

3. DEFINE and LAMBDA

DEFINE and LAMBDA are two of the expressions in 7090 LISP which enable us to define new functions, give them names, and make use of them.

For example, suppose you have PLUS and you would like to define DOUBLE. The DOUBLE of m of course is m plus m; the DOUBLE of y is y plus y; the double of 213 is 213 plus 213 or 426. What we want to accomplish is to lay down a rule like this: For any x, the double of x is x plus x.

We can establish this sort of definition in LISP and make use of it. In 7090 LISP, we put in the expression:

```
DEFINE(((DOUBLE (LAMBDA (X) (PLUS X X)))))
```

The computer will respond:

```
(DOUBLE)
```

telling us that this function is now available to us.

To DOUBLE the number three, for example, if the LISP interpreter is given:

```
(DOUBLE 3)
```

it will give the value:

```
6
```

DEFINE is short for DEFINE THE EXPRESSION. LAMBDA is to some extent equivalent to the English phrase "FOR ANY". Thus the LISP statement:

```
DEFINE(((DOUBLE (LAMBDA (X) (PLUS X X)))))
```

is in effect something like the statement

```
DEFINE THE EXPRESSION:  DOUBLE (FOR ANY X) (PLUS X X)
```

What is the meaning of all this? The general form of the LISP idiom which we are using is:

```
DEFINE((( ..... (LAMBDA (.....) (.....)))))
            1            2      3
```

The word "lambda" (the Greek name for the letter L) is a sign used by the symbolic logician Alonzo Church who in the 1940's pointed out the need for naming a function in mathematics (such as $y = x^2$ (a certain parabola) or $z = u^2$ (the same parabola)) independently of the algebraic letters being used to talk about it).

Blank 1 above is filled with any name that we wish to use for a function being defined. Blank 3 is filled with a defining expression, often a conditional expression. Blank 2 is filled with a list of variables which we may call the "lambda list".

The order of the variables in the lambda list is the precise order in which values for those variables have to be mentioned when the defined expression is used. Most of the time these variables are specified or limited or bound in the defining expression that occupies Blank 3. Some of the time however the variables are not specified or limited or bound currently in the defining expression that fills Blank 3, but instead at some other point in another definition. Such variables are called free variables or parameters. However, before the computer will compute a result, any free variable must be bound to a value.

DEFINE in 7090 LISP has a further useful property. Suppose you have more than one definition — say three — that you wish to express; you can then put all of them inside the DEFINE, writing as follows:

```
            DEFINE((
1st def.:   (..... (LAMBDA (.....) .....))
2nd def.:   (..... (LAMBDA (.....) .....))
3rd def.:   (..... (LAMBDA (.....) .....))
            ))
```

The outer left parenthesis after DEFINE indicates that the argument of DEFINE will follow. The inner left parenthesis after DEFINE indicates that the argument is a list. The two right parentheses at the end close the two expressions. When only one expression is being defined, the list is a list of one member, consisting of the single expression being defined.

4. The Expression DEX

In LISP on the PDP-1, LAMBDA is used in just the same way but DEFINE is not used. What is used instead of DEFINE is the expression DEX, and the idiom is:

```
(DEX ..... (LAMBDA (.....) .....))
      1              2      3
```

Blanks 1, 2, and 3 are filled in just the same way.

In PDP-1 LISP, if you wish to define three expressions, you need to write the DEX idiom completely three times.

5. The Function SQUARE

"The square of any number X is X times X." In 7090 LISP, we put this definition into the computer as follows:

```
DEFINE(((SQUARE (LAMBDA (X) (TIMES X X)))  ))
     012       3       4 4 4         432  10
```

The computer responds:

(SQUARE)

indicating that the function SQUARE is available.

To test the operation of this definition, we put into the 7090 LISP interpreter (using the 7090 external form of the LISP interpreter):

SQUARE (3)

and the computer responds: 9

The internal form for this instruction for calculation is:

(SQUARE 3)

In PDP-1 LISP only there must be in addition a space at the end, which means "go" or "finished".

This is the form for commanding the PDP-1 LISP interpreter to calculate. Since the PDP-1 LISP interpreter regularly uses octal numbers, the computer responds with 11, which is nine in octal. The 7090 gives the result 9, in decimal.

6. The Function CUBE

Similarly, "the cube of any number X is X times X times X".

```
DEFINE(((CUBE (LAMBDA (X) (TIMES X X X)))))
     012     3       4 4 4          43210
```

7. The Function TRIPLE

```
DEFINE(((TRIPLE (LAMBDA (X) (PLUS X X X)))))
```

From this point on we shall assume DEFINE((.....)) or (DEX .....) and simply write the internal part of the defining expression.

8. The Function SMALLER

"The smaller of two numbers X and Y is Y if X is greater than Y, otherwise X." In precise English:

The smaller of two numbers X and Y is computed from:

> If X is greater than Y, take Y. Otherwise, take X.

In LISP we write for the defining expression:

```
(SMALLER (LAMBDA (X Y) (COND
     ((GREATERP X Y) Y)
     (T X))))
```

```
Testing: SMALLER (11 15)        (for the 7090 — external form)
or:      (SMALLER 11 15)        (for the PDP-1 and the 7090
                                   internal form)
```

and the computer responds: 11

9. The Predicate ZEROP

Let us now express in LISP the condition "x is equal to zero" where x is a number. Often in calculations, this is a useful condition. We give the predicate "is equal to zero" the name ZEROP. (Note: Often in LISP, the letter P attached to the end of a word or name signifies that it designates a predicate.) We translate evaluation of the condition "x is equal to 0" into the task of computing the truth value of the predicate "is equal to zero". This is computed from:

> If x is equal to 0, take true, Else take false.

This is written in PDP-1 LISP as follows:

```
(ZEROP (LAMBDA (X) (COND
            ((EQ X 0) T)
            (T NIL))))
```

We give this to the PDP-1 within a DEX statement. The computer accepts it and types back:

ZEROP

In PDP-1 LISP, in the definition above, we can use the predicate EQ for relating X and 0 because numbers are treated as atomic symbols. In 7090 LISP we would need to replace the predicate EQ by the predicate EQUAL (see definition below), since the predicate EQ will not operate correctly for numbers. (Note: In 7090 LISP the defining of ZEROP is not necessary since it is already available in the LISP system with a subroutine expressed in machine language.)

To test this defined predicate, we may try various numbers:

Expression	Result
(ZEROP 0)	T
(ZEROP 7)	NIL
(ZEROP 1234)	NIL
(ZEROP 777776)	NIL

In PDP-1 LISP, minus zero is not zero and is not equal to zero. If in PDP-1 LISP we type in: (ZEROP 777777), the result is: NIL

10. Availability of Expressions for Definitions

For 7090 LISP, it is stated that all the expressions appearing in the "LISP 1.5 Programmer's Manual" are available in the LISP system, and can be used by anyone who wishes.

For the basic PDP-1 LISP system as of March, 1964, Table 1 is a complete list of expressions available. (Many of these expressions are not defined in this paper because they are outside of the province of the present explanation; however, many of them are defined in a later appendix in this volume.)

This list is named the OBLIST. When OBLIST is typed, and the space bar pressed, the PDP-1 will immediately print out the entire list of expressions; the printing is in rows, with a single space between the end of one name and the beginning of the next name. The word "oblist" comes from "object" and "list".

During the course of any computation using LISP the computer adds to the OBLIST any other terms that may be defined or used in the computation.

Table 1

THE LIST OF EXPRESSIONS AVAILABLE IN BASIC PDP-1
LISP: — THE "OBLIST"

APVAL	LAMBDA	QUOTE
ATOM	LIST	QUOTIENT
CAR	LOC	READ
CDR	LOGAND	RETURN
COND	LOGOR	RPLACA
CONS	MINUS	RPLACD
EQ	NIL	SASSOC
EVAL	NULL	SETQ
EXPR	NUMBERP	STOP
FEXPR	OBLIST	SUBR
FSUBR	PLUS	T
GENSYM	PRINT	TERPRI
GO	PRIN1 (the last char-	TIMES
GREATERP	acter is one)	XEQ
	PROG	

If you wish to define a new expression, then in order to avoid confusion, you should regularly use as a name one which differs from every one of the expressions in the existing OBLIST. There are exceptions to this rule: (1) if you wish to use for the same name, a definition different from the one in the OBLIST, you can "write over" the definition, producing a new one to be used, and your definition will govern; (2) if you remove an expression from the OBLIST entirely (which is possible, though the method for doing so is not explained here), you may use the released name in any way you wish.

11. Alternative Definitions

Many functions of course can be defined in alternative ways. There are other ways of defining ZEROP, for example. In English:

> x is equal to 0 if and only if x is not greater than 0 and x is not less than 0.

This may be expressed in more precise English, in a LISP style as follows:

x is equal to 0:

> If x is not greater than 0, and
> if x is not less than 0, take true. Else take false.

Now the expression GREATERP is in the OBLIST; so it can be used. The expression AND has a definition (see Appendix 4) by means of which it can be put into the OBLIST, and used. But an expression LESSP is not in the OBLIST (for the PDP-1). The easiest thing to do is to replace the English "x is not less than 0" by the English "0 is not greater than x", and then translate into LISP:

```
(ZEROP (LAMBDA (X) (COND
0      1       2 2 2
          ((AND (NOT (GREATERP X 0))
          34    5    6           65
                (NOT (GREATERP 0 X))) T)
                5    6           654  3
          (T NIL))))
          3     3210
```

ZEROP can also be expressed in another way in LISP-style English, avoiding AND:

> If 0 is not greater than x, then
>
> > If x is not greater than 0, then true. Else False. Else
>
> False.

Translating into LISP:

```
(ZEROP (LAMBDA (X) (COND
0      1       2 2 2
          ((NULL (GREATERP 0 X))
          34     5           54
                (COND ((NULL (GREATERP X 0)) T)
                4     56     7           76  5
                      (T NIL)))
                      5     543
          (T NIL))))
          3     3210
```

This definition will give results no different from any other definition of ZEROP.

IV. ATOMIC EXPRESSIONS

For a long time we have avoided discussing what are called atomic expressions or atomic symbols or atoms. But we have used them, both individually and as elements of lists. Examples are:

- single letters such as: A, B, C, D, E, F, ...
- numbers such as: 5, 314, 777772, ...
- names of expressions in LISP such as CAR, CDR, PLUS, ...

We now state a definition: In 7090 LISP an atomic symbol (or atomic expression or atom) is (1) a string of numerals and capital letters of not more than 30 characters, in which the first character is a letter, or (2) a number, consisting of a string of numerals, optionally starting with + or -, and with or without a period (which indicates the "point" in the scale of notation). Since 7090 LISP regularly uses decimal notation for numbers, the digits 8 and 9 will be accepted as numerals. Neither spaces nor parentheses may be used inside an atomic symbol. In PDP-1 LISP an atomic symbol is (1) a string (of indefinite length) of lower case letters, upper case letters, numerals, or other characters (except: space, tab, comma, carriage return, period, left parenthesis, right parenthesis, over-bar, vertical bar) or (2) an integer consisting of a string of numerals. Since PDP-1 LISP uses the octal system, the digits 8 and 9 will not be accepted as numerals. The two characters over-bar and vertical bar (which are completely equivalent) allow construction of special atomic symbols using forbidden characters; when one of these is used, the next character is put into the symbol name irrespective of whether it is forbidden or not.

The marker which precedes an atomic symbol is either a parenthesis or a space. The marker which ends an atomic symbol is either a space or a parenthesis. If by mistake we seek to make an atomic symbol containing one or more spaces or parentheses, the LISP program inside the computer will recognize the expression as two or more atomic symbols.

These symbols are called atomic because they are treated in LISP as wholes and within LISP are not split into individual characters.

For example, A, C, R, CAR, CDR have no relation to each other in LISP because no part of LISP can regularly either observe or report that these five expressions have some characters in common.

The predicate EQ reports on the equality of two atomic symbols; it is not defined for expressions that are not atomic.

The predicate ATOM is true if its argument is an atomic symbol; it is false if its argument is not an atomic symbol.

V. RECURSIVE DEFINITIONS

A most important feature of LISP is the ability to make use of recursive definitions of functions. These are definitions which first define an idea in one or more special starting or finishing cases, and then define the idea in the general case in terms of a preceding or adjacent case.

For example, in arithmetic a geometric series can be defined making use of a recursive definition:

> (i) A geometric series is a series with a first term equal to a, and with any term equal to the preceding term multiplied by a constant r.

Also, an arithmetic series can be defined recursively:

> (ii) An arithmetic series is a series with a first term equal to a, and with any term equal to the preceding term plus a constant difference d.

1. The Predicate EQUAL

An example of a recursive definition is LISP is the definition of the predicate EQUAL. The definition makes use of EQ for a special case and then makes use of EQUAL to define the general case in terms of the preceding case.

Two expressions in LISP are EQUAL, if they are made up of equal atomic symbols, in precisely the same structure. More precisely, the expression X in LISP is EQUAL to another expression Y in LISP if and only if:

> (i) X is an atomic symbol and Y is an atomic symbol, and X EQ Y; else,
> (ii) if CAR X is EQUAL to CAR Y, and CDR X is EQUAL to CDR Y.

How does the definition operate? Let's take an example. Suppose X is (A B C) and Y is (A B C). Here are the steps:

First Application of the Definition, to (A B C). X is (A B C) and Y is (A B C). But condition (i) is not true, because (A B C) is not an atomic symbol. Go to condition (ii). Calculate CAR X

and CAR Y, CDR X and CDR Y. CAR X is A; CAR Y is A. CDR X is the list (B C) and CDR Y is the list (B C).

Second Application of the Definition, to A. The new X is the old CAR X, which is A; the new Y is the old CAR Y, which is A. These are atomic. Apply (i). The two atoms A satisfy EQ; and so condition (i) is met.

Third Application of the Definition, to (B C). (B C) is CDR of the original X and CDR of the original Y. These are not atomic. So condition (i) is not met. Go to condition (ii). Calculate CAR X, CAR Y, CDR X, and CDR Y, where X and Y are now equal to the list (B C). These are B and (C).

Fourth Application of the Definition, to B. Condition (i) is true for the two atoms B.

Fifth Application of the Definition, to (C). (C) is CDR of (B C), and is not atomic. So condition (i) is not met. Go to condition (ii). Calculate CAR X, CAR Y, CDR X, and CDR Y, where X and Y are now equal to the list (C). The results are C and NIL.

Sixth Application of the Definition, to C. Condition (i) is true for the two atoms C.

Seventh Application of the Definition, to NIL. The end of every list is the expression NIL, which is called the terminator. Condition (i) is true since CDR (C) is NIL, in both cases, and (EQ NIL NIL) is true.

2. Its Definition in LISP

Now how do we express this definition in LISP?

There are several steps which we have to take, including:

(1) recognition of the occurrence of the name of the function in its own definition;
(2) translation of the relations in the definition into relations recognized in LISP.
(3) adjustment of the definition to the nature of the conditional expression in LISP.

Supoose we try to translate straightforwardly into LISP:

```
(EQUAL (LAMBDA (X Y) (COND
0      1       2   2 2
     ((AND (ATOM X) (ATOM Y) (EQ X Y)) T)
     34    5      5 5      5 5      54  3
     ((AND (EQUAL (CAR X) (CAR Y)) (EQUAL (CDR X)
     34    5      6     6 6     65 5      6     6
                                      (CDR Y))) T)
                                      6     654  3
     (T NIL) )))
     3     3 210
```

Is there anything wrong with this defining expression? Let's refer again to the meaning of the conditional expression in LISP. Applying this meaning, we can see that as soon as the predicate

```
(AND (ATOM X) (ATOM Y) (EQ X Y))
```

fails, the conditional expression will be referred to

```
(AND (EQUAL (CAR X) (CAR Y)) .....
```

So the case where only one of X and Y is atomic will not be covered, and the definition will not function properly.

A corrected statement of the defining expression is as follows:

```
(EQUAL (LAMBDA (X Y) (COND
0      1       2   2 2
     ((ATOM X) (EQ X Y))
     34      4 4      43
     ((ATOM Y) NIL)
     34      4    3
     ((EQUAL (CAR X) (CAR Y)) (EQUAL (CDR X) (CDR Y)))
     34      5     5 5     54 4      5     5 5     543
     (T NIL))))
     3     3210
```

3. The Function REMAINDER

In elementary school we all learned what the remainder is when we divide one number by another. If we divide 26 by 7, for example, the 7 goes in three times, and since 7 threes are 21, we subtract 21 from 26, and the remainder is 5. Now how do we express remainder in English that can be translated into LISP?

Stated in precise English, and using a recursive definition (without making use of division), the remainder when the dividend Y is divided by the divisor X, is determined as follows:

(i) If Y equals X, the remainder is 0; otherwise,
(ii) If Y is less than X, the remainder is Y; otherwise,
(iii) The remainder is the same as the remainder we would get if the result of Y minus X were divided by the divisor X.

For example, consider the following series of operations:

In Decimal (7090)	Equivalent in Octal (PDP-1)
26	32
7	7
19	23
7	7
12	14
7	7
5	5

To say this in LISP we again make use of the defining expression:

```
(..... (LAMBDA (.....) .....))
```

The expression "y equals x" is translated into (EQUAL Y X). The expression "y is less than x" is translated into (GREATERP X Y). Since there is no condition for the third case, we use T. The expression "y minus x" is translated into (PLUS Y (MINUS X)).

The entire definition is therefore translated into:

```
(REM (LAMBDA (Y X) (COND
0    1       2   2 2
     ((EQUAL Y X) 0)
     34         4  3
     ((GREATERP X Y) Y)
     34             4  3
     (T (REM (PLUS Y (MINUS X)) X) ))))
     3  4    5       6       65  4 3210
```

If the interpreter evaluates this expression, it responds REM.

Testing with examples, we get the following:

Expression (decimal, 7090)	Value	Expression (octal, PDP-1)	Value
(REM 2 2)	0	(REM 2 2)	0
(REM 5 7)	5	(REM 5 7)	5
(REM 12 7)	5	(REM 14 7)	5
(REM 26 7	5	(REM 32 7)	5

4. The Function GREATEST COMMON DIVISOR

The greatest common divisor (G.C.D.) of two whole numbers is the largest number which will exactly divide both of them. There is an arithmetical process (or algorithm) given by Euclid which quickly finishes and which quickly finds the greatest common divisor. The process is often learned in school and is illustrated in the following arithmetic, which finds the greatest common divisor of 28 and 91 (in decimal):

```
4) | 28 | 91 | (3
   | 28 | 84 |
   |  0 |  7 |
```

(In English, condensed) 28 divided into 91 goes 3 times with 7 over; 7 divided into 28 goes 4 times exactly.) This yields 7 as the G.C.D.

Since the PDP-1 computer sometimes does not have built-in multiplication and division, suppose we change the arithmetical process to successive subtraction, and find the last remainder which is not zero:

```
 28   91
  7   28
 21   63
  7   28
 14   35
  7   28
  7    7
  7
  0
```

Stated in precise English:

> The greatest common divisor of X and Y is found from:
>
> > If X is greater than Y, take the G.C.D. of Y and X. Else
> > If the remainder of Y divided by X is zero, take X. Else
> > Take the G.C.D. of X and the remainder of Y divided by X.

This is translated into LISP as follows:

```
(GCD (LAMBDA (X Y) (COND
0    1       2   2 2
```

```
((GREATERP X Y) (GCD Y X))
34            4 4       43
((ZEROP (REM Y X)) X)
34      5       54  3
(T (GCD (REM Y X) X)))))
3  4    5       5  43210
```

If this is put into the computer, it responds by putting out:

GCD

Testing examples of GCD, we have:

Expression in Decimal	Value	Expression in Octal	Value
(GCD 7 7)	7	(GCD 7 7)	7
(GCD 19 7)	1	(GCD 23 7)	1
(GCD 28 7)	7	(GCD 34 7)	7
(GCD 28 35)	7	(GCD 34 43)	7
(GCD 30 40)	10	(GCD 36 50)	12

VI. THE PROGRAM FEATURE

There is another feature in LISP which allows a still wider variety of useful instructions to the computer. This is the feature named the "program feature", which is called by the LISP expression PROG, pronounced "prōg" rhyming with "rogue".

In using the program feature, we make use of auxiliary variables that keep track during successive cycles or loops of any temporary results within a program.

1. The Function QUOTIENT

For a simple illustration, let us use a PROG for the process of computing a quotient, supposing that we have a computer without built-in division. To see the process clearly, let us take as an example 22 (in decimal) divided by 7. The answer is 3, of course, since 7 divides into 22 three times with remainder 1. We could employ a process using successive subtraction, and in this case it would be:

1st cycle:	22 minus 7 gives 15	count 1
2nd cycle:	15 minus 7 gives 8	count 1 more; total count is 2
3rd cycle:	8 minus 7 gives 1	total count 3

4th cycle attempted: 1 minus 7, result negative;
stop; take 3 as the quotient

We can organize this process with two auxiliary variables, which we can call U and V: U starts with the dividend 22 and in this case is successively 22, 15, 8. Y starts with 0, and in this case is successively 0, 1, 2, 3, finishing with the desired quotient 3.

In English, the directions written out would be about as follows:

Set U at the dividend (in this case, 22).
Set V at 0.
Mark this point in the process with a label K.
If the divisor (in this case, 7) is greater than U,
stop and give as the result of the computation
(i.e., the quotient) the value of V.
Otherwise,
Set U at the old U minus 7.
Set V at the old V plus 1.
Go back to K and repeat.

In LISP, the successive directions become the following:

```
(SETQ U 22)
(SETQ V 0)
K
(COND ((GREATERP 7 U) (RETURN V)))
(SETQ U (PLUS U (MINUS 7)))
(SETQ V (PLUS 1 V))
(GO K)
```

Now let us generalize so that the process applies to any number N as dividend and any divisor D; let us also put in the rest of the idiom of LISP so that the command will actually be accepted as a defining condition.

```
(QUOTIENT (LAMBDA (N D) (PROG (U V)
0         1       2   2 2     3   3
     (SETQ U N)
     3        3
     (SETQ V D)
     3        3
```

```
K       (COND ((GREATERP D U) (RETURN V)))
        3     45            5 5        543
        (SETQ U (PLUS U (MINUS D)))
        3       4       5       543
        (SETQ U (PLUS 1 V))
        3       4        43
        (GO K))))
        3    3210
```

Then we can test the operation of this definition, and have the interpreter evaluate:

```
(QUOTIENT 22 7)
```

The value of this expression is 3.

Notice that when using the program feature it is not necessary to make the COND complete with T for the last case, the final ELSE or OTHERWISE.

The function QUOTIENT can also be defined recursively. Using precise English we write:

To find the quotient of N divided by D:

If D is greater than N, take 0. Else,
If D is equal to N, take 1. Else,
If D is less than N, add 1 to the quotient
of N-D divided by N.

This translates into the following defining condition for QUOTIENT:

```
(QUOTIENT (LAMBDA (X Y) (COND
0         1       2   2 2
     ((GREATERP X Y) 0)
     34            4  3
     ((EQUAL X Y) 1)
     34      4  3
     ((GREATERP Y X) (PLUS 1 (QUOTIENT (PLUS Y (MINUS X))
     34            4 4       5         6       7       76
                                                   X))))))
                                                    543210
```

In general the program feature can be replaced by recursion, and vice versa. But use of the program feature often saves a good deal of computer time and storage space.

VII. FUNCTIONS OF LISTS

We have paid some attention to functions of numbers, and we have illustrated recursion and the program feature using numbers. But LISP is more useful for dealing with functions of lists than for dealing with functions of numbers. So let us now look carefully at some more functions of lists.

The list functions which we have so far defined are CAR, CDR, CONS, COND, and EQUAL.

COND is a list function because it operates on a list, each element of which is a list of two elements. There is also the additional requirement that the first element of each of the doublets on which COND operates needs to be a predicate, which has a truth value of either true or false.

In regard to EQUAL, it should be noted that one list is only equal to another list provided the elements of the list are equal, and the order of each list of elements is the same. This means that a list is not a class, because one class is equal to another class if the members are the same regardless of any order in which the members may be presented.

1. The Function APPEND

A useful list function is APPEND, which appends two lists, fastening them together into one list. APPEND of the list A B C and the list D E F G is the list A B C D E F G.

If the LISP interpreter were given the expression:

```
(APPEND (QUOTE (A B C)) (QUOTE (D E F G)))
```

it would respond:

```
(A B C D E F G)
```

If just one element is to be attached to a list, and the element is changed into a list by putting parentheses around it, and then APPEND is applied:

```
(APPEND (QUOTE (A)) (QUOTE (B C D E)))
```

the result is the list: (A B C D E).

To produce the same effect with CONS, parentheses are not

put around the element. Thus the value of the expression:

```
(CONS (QUOTE A) (QUOTE (B C D E)))
```

is the same list: (A B C D E).

The LISP definition of APPEND makes use of recursion:

```
(APPEND (LAMBDA (X Y) (COND
0       1       2   2 2
     ((NULL X) Y)
     34      4  3
     (T (CONS (CAR X) (APPEND (CDR X) Y))))))
     3  4     5     5 5       6     6  543210
```

2. The Function LENGTH defined with the Program Feature

The number of elements in a list is called its "length". Here is a LISP expression for determining the length of a list using the program feature:

```
(LENGTH (LAMBDA (L) (PROG (U V)
     (SETQ V 0)
     (SETQ U L)
     K
     (COND ((NULL U) (RETURN V)))
     (SETQ U (CDR U))
     (SETQ V (PLUS 1 V))
     (GO K))))
```

For example, (LENGTH (QUOTE (2 3 4 6 7))) will be 5.

3. The Function LENGTH, defined Recursively

If we want to define LENGTH recursively, the expression is even simpler and easier:

To find the LENGTH of a list M:

> If the list is null, take 0. Else,
> Add 1 to the length of CDR M.

The LISP defining expression is:

```
(LENGTH (LAMBDA (M) (COND
0       1       2 2 2
     ((NULL M) 0)
     34      4  3
```

```
(T (PLUS 1 (LENGTH (CDR M))))))
3  4       5       6     6543210
```

4. The Predicate MEMBER

An element is a member of a list if it can be found among the elements of the list.

In precise English:

> The truth of "element A is a MEMBER of the list X" is found from:
>
> If X is empty, take NIL. Else,
> If A is equal to CAR X, take true. Else
> Take the truth of "A is a MEMBER of CDR X."

In LISP, we express this as:

```
(MEMBER (LAMBDA (A X) (COND
0       1       2   2 2
     ((NULL X) NIL)
     34      4    3
     ((EQ A (CAR X)) T)
     34     5     54  3
     (T (MEMBER A (CDR X))))))
     3  4         5     543210
```

For examples, we can put the following operations on the PDP-1 computer:

Expression	Explanation	Value
(CSETQ JILL (QUOTE (A B C D)))	This names the list (A B C D) as JILL.	JILL
JILL	Verifying meaning of "JILL."	(A B C D)
(MEMBER (QUOTE A) JILL)	Computing the truth of "A is a member of JILL."	T
(MEMBER (QUOTE K) JILL)	Computing the truth of "K is a member of JILL."	NIL

5. The Function LAST

Another list function is LAST. The LAST element of a list is found from the operation:

> If the list is empty, take NIL. Else
> If CDR of the list is empty, take CAR of the list. Otherwise, take LAST of CDR of the list.

In LISP:

```
(LAST (LAMBDA (L) (COND
0     1       2 2 2
     ((NULL L) NIL)
     34      4    3
     ((NULL (CDR L)) (CAR L))
     34     5     54 4     43
     (T (LAST (CDR L))))))
     3  4     5     543210
```

6. The Function UNION

Another function of lists is UNION. The union of two lists X and Y is a list which contains every element which is in one list or the other or both. But the order in which the elements is presented is first, all elements which are in the first list X and not in the second list Y, and second, all elements in the second list Y whether or not they are in list X.

In precise English:

> The UNION of two lists X and Y is:
>
> > If X is null, take Y. Else,
> > If CAR X is a member of Y, take the UNION of
> > CDR X and Y. Else,
> > Take CONS of CAR X and the UNION of CDR X and Y.

In LISP:

```
(UNION (LAMBDA (X Y) (COND
0      1       2   2 2
      ((NULL X) Y)
      34      4  3
      ((MEMBER (CAR X) Y) (UNION (CDR X) Y))
      34       5     5  4 4      5     5  43
      (T (CONS (CAR X) (UNION (CDR X) Y))))))
      3  4     5     5 5      6     6  543210
```

For an example, if the LISP interpreter receives the following expressions, the values will be as shown:

Expression	Value
(UNION (QUOTE (E B C D A F)) (QUOTE (A B C D)))	(E F A B C D)
(CSETQ JAIN (QUOTE (A B C D)))	JAIN (on PDP-1)
(CSETQ JASS (QUOTE (D A B C)))	JASS (on PDP-1)
(UNION JASS JAIN)	(A B C D)
(UNION JAIN JASS)	(D A B C)

7. The Function DIFFLIST

The function DIFFLIST operates on an element A and a list X, and removes from list X any element which is equal to A.

In precise English:

The result of DIFFLIST on element A and list X equals:

If X is empty, then NIL. Else,
If A is equal to CAR X, take the result of
DIFFLIST on A and CDR X. Else,
Take the CONS of CAR X and the result of
DIFFLIST on A and CDR X.

In LISP:

```
(DIFFLIST (LAMBDA (A X) (COND
0         1       2   2 2
     ((NULL X) NIL)
     34      4    3
     ((EQUAL A (CAR X)) (DIFFLIST A (CDR X)))
     34        5     54 4           5     543
     (T (CONS (CAR X) (DIFFLIST A (CDR X)))))))
     3  4     5     5 5           6     6543210
```

Example:

Expression	Value
(DIFFLIST 7 (QUOTE (11 13 7 16)))	(11 13 16)

8. The Function SUBST

This function substitutes a given element for another element if the latter is found in a list. For example we may have the list (2 4 6 10), and we wish to put 7 in place of 6.

We direct the PDP-1 to do this, by typing:

```
(SUBST 7 6 (QUOTE (2 4 6 10)))
0         1      2        210
```

and the result is: (2 4 7 10)

The definition of SUBST that must be placed previously in the computer is:

```
(SUBST (LAMBDA (X Y Z) (COND
0      1       2     2 2
     ((EQUAL Y Z) X)
     34         4  3
     ((ATOM Z) Z)
     34       4  3
     (T (CONS (SUBST X Y (CAR Z)) (SUBST X Y (CDR Z
     3  4     5          6     65 5          6
                                            )))))))
                                            6543210
```

The meaning of this defining condition is:

If Y is equal to Z, take X. Else,
If Z is an atom, take Z. Else,
Take the result of applying CONS to the
SUBST of X, Y, and CAR of Z, and the
SUBST of X, Y, and CDR of Z.

9. The Function ASSOC

The list function ASSOC is given an element and a list consisting of pairs of elements. The function looks through the list, and finds the "associated" pair which has the given element as its first member. For example, suppose J13 is the name of a list: ((1 3) (2 6) (3 11) (4 14)) and suppose the given element is 3. Then ASSOC will find and report (3 11) as the associated pair of 3 in the list J13.

First, J13 is established as a name:

```
(CSETQ J13 (QUOTE ((1 3) (2 6) (3 11) (4 14))))
0          1      23                        3210
```

The defining condition for ASSOC is:

```
(ASSOC (LAMBDA (X Y) (COND
0      1       2   2 2
     ((EQUAL (CAR (CAR Y)) X) (CAR Y))
     34      5    6     65  4 4     43
     (T (ASSOC X (CDR Y))))))
     3  4        5     543210
```

The associated pair of 3 in the list J13 is the value of the expression:

(ASSOC 3 J13)

The result from the computer is: (3 11)

10. The Function MINIMUM

Another list function we can define is the minimum of all the elements of a list (if the elements are numbers). In order to define this function we make use of the function of two numbers, SMALLER, defined earlier.

To define minimum, we use recursion and precise English:

> The MINIMUM of the list X is computed from:
>
> If X is empty, take NIL. Else,
> If CDR X is empty, take CAR of X. Else,
> Take the MINIMUM of the SMALLER of CAR of X
> and the MINIMUM of CDR of X.

In LISP this becomes:

```
(MINIMUM (LAMBDA (X) (COND
0        1       2 2 2
     ((NULL X) NIL)
     34      4    3
     ((NULL (CDR X)) (CAR X))
     34     5     54 4     43
     (T (SMALLER (CAR X) (MINIMUM (CDR X)))))))
     3  4        5     5 5        6     6543210
```

As a test:

(MINIMUM (QUOTE (11 10 15 7 3 6))) gives 3.

In 7090 LISP the function minimum is available as MIN.

11. The Function SEQUENCE

Another function of lists that we can define is the operation "to sequence" meaning "to put the (numerical) elements of a list into sequence". What we have to do is to find successive minimums and put them into a new list in the proper order.

To visualize the process and see what operations need to be

done in what order, let us take a sample illustration and notice precisely the pattern of the operations performed.

Suppose we take the list (11 10 7 15 3) and see what needs to be done to produce the list (3 7 10 11 15). The successive operations are shown in the following table:

(1) Cycle	(2) List Being Worked On	(3) Minimum Found	(4) List Being Assembled
0	—	—	NIL
1	(11 7 10 15 3)	3	(3)
2	(11 7 10 15)	7	(3 7)
3	(11 10 15)	10	(3 7 10)
4	(11 15)	11	(3 7 10 11)
5	(15)	15	(3 7 10 11 15)
6	NIL	—	—

Let us call Column (2) the variable U; it starts with the original list, and each successive entry equals the preceding entry less the element removed. Let us call Column (3) V; it is the minimum of the entry in column (2), an element and not a list. Let us call Column (4) W; it is the result of "appending" the entry in Column (3) changed into a list of one element to the previous entry in Column (4).

Since we now understand the process, we can express it in LISP language using the program feature:

```
(SEQUENCE (LAMBDA (L) (PROG (U V W)
0         1       2 2 2    3     3
     (SETQ U L)
     3        3
     (SETQ V (MINIMUM L))
     3       4         43
     (SETQ W NIL)
     3          3
K    (COND ((NULL U) (RETURN W)))
     3     45      5 5        543
     (SETQ V (MINIMUM U))
     3       4         43
     (SETQ U (DIFFLIST V U))
     3       4            43
     (SETQ W (APPEND W (LIST V)))
     3       4         5      543
     (GO K))))
     3    3210
```

(Note: The LISP expression LIST in the next-to-the-last line changes the element V into a list V which can be operated upon by APPEND.)

To test this we can give the computer an example:

```
(SEQUENCE (QUOTE (11 10 7 15 3)))
```

and it will give back as a result:

```
(3 7 10 11 15)
```

12. The Function MAPLIST

Suppose we have one list of elements and we want to make a second list of elements, each of which is the result of a given operation on an element of the first list. An example might be that we have the list 1 2 3 4, and we wish to produce the list of triples, 3 6 9 12 (in decimal), which is 3 6 11 14 in octal.

The list function which we use in this case is MAPLIST.

In order to use MAPLIST to perform this task, first we define a function which triples the first element of a list; we might call this function TRIPCAR:

```
(TRIPCAR (LAMBDA (X) (TRIPLE (CAR X))))
0        1       2 2 2      3     3210
```

We then define MAPLIST with the following recursive definition:

```
(MAPLIST (LAMBDA (X A) (COND
0        1       2   2 2
     ((NULL X) NIL)
     34      4    3
     (T (CONS (A X) (MAPLIST (CDR X) A))))))
     3  4     5   5 5        6     6  543210
```

Here is the meaning of it:

```
If the list X is empty, take NIL.
Otherwise, apply CONS to the result of applying
   A to X and the result of applying MAPLIST to
   CDR of X and A.
```

Notice that A is a free variable which has to have a function name for its value when used in a computation.

Then we can give the LISP interpreter the expression:

(MAPLIST (QUOTE (1 2 3 4)) (QUOTE TRIPCAR))

and the interpreter will give back the value:

(3 6 11 14)

If we wished, we could supply MAPLIST with the function CDR. Thus if the interpreter evaluates:

(MAPLIST (QUOTE (1 2 3 4)) (QUOTE CDR))

the computed result will be:

((2 3 4) (3 4) (4) NIL)

VIII. CONCLUDING REMARKS

1. Generality and Power

There are several comments which it seems to me should be made about LISP from the point of view of a person approaching it for the first time. The first comment is that LISP greatly enlarges our conception of the nature of mathematical objects. In prior centuries men became accustomed to noticing as interesting mathematical objects: numbers; the points and lines of geometry; the magnitudes and directions of forces; sequences of numbers, usually infinite and usually with fairly simple rules for the construction of successive terms; and much more.

Now with the advent of LISP our horizons in mathematics are considerably extended. With LISP we take into mathematics finite sequences of a great variety of structures (lists), and also a mathematical grasp on the processes of effectively computing with them (recursion, the program feature, etc.). This new expansion of mathematical nature, of man's view of mathematical objects, is exciting.

Second, LISP is not only a mathematical language but also a language for instructing computers. So instead of humanly verifying a symbolic mathematical calculation, if it is expressed in LISP, we can put it on a computer and have the computer verify it.

Third, there appears to be no barrier to putting into LISP any kind of logical or mathematical manipulation that may be desired. It seems to be a truly general language, with closely

linked computing power.

2. Computation

It may be worthwhile to point out that what we are invariably doing with any LISP expression is giving the computer something to do, some computation to make, some command to be executed. Sometimes the computation is in terms of numbers and the answer may be a number. Sometimes the computation is in terms of lists and the answer will be a list or a number. Sometimes the computation is in terms of expressions, and the answer will be a number or a list or an expression.

If all the information has been given to the computer, and the instructions are complete and correct, the answer will be an applicable one. If not all the data has been given to the computer, the result may be a number if the missing data are not important for the determination of the number. Otherwise, the answer (or result of the computer's operation) is likely to be a signal of any one of many kinds, that the answer cannot be computed for a stated reason. Or perhaps the computer will go into a loop and keep doing something over and over and over without ever finishing. In such cases, the programs and expressions given to the computer must be corrected.

3. Incompleteness of this Explanation

Finally, it must be emphasized that the present explanation is very far from complete. There is much more to LISP than is presented here; and the first place where more information about LISP may be obtained is the "LISP 1.5 Programmer's Manual" cited in the first section to this article.

It is hoped however, that this explanation will be of help to people who are setting out to understand LISP.

APPENDIX

TEST AT PROJECT MAC

In order to test the relation of operations with LISP on the PDP-1 computer to operations with LISP on the 7090 computer, a visit was made to Project MAC at Mass. Inst. of Technology on Feb. 7, 1964. A LISP system working under the Compatible Time-Sharing system was used to define **SMALLER**, **MINIMUM**, **DIFFLIST**, and **SEQUENCE**, and to try two tests of sequencing numbers.

Following is a copy of the actual run on the computer at one of the time sharing stations. The total computer time used for editing and computing was 1 and ½ minutes.

Input

```
PRINTF LISP DATA
WAIT,
00010 DEFINE ((
00020 (SMALLER (LAMBDA (X Y) CØND
00030     ((GREATERP Y X) X)
00040     (T Y))))
00050 (MINM (LAMBDA (L) (CØND
00060     ((NULL L) NIL)
00070     ((NULL (CDR L)) (CAR L))
00080     (T (SMALLER (CAR L) (MINM (CDR L)))))))
00090 (DIFFLIST (LAMBDA (A X) (CØND
00100     ((NULL X) NIL)
00110     ((EQUAL A (CAR X))
00120          (DIFFLIST A (CDR X)))
00130     (T (CØNS (CAR X) (DIFFLIST A (CDR X)))))))
00140 (SEQUENCE (LAMBDA (L) (PRØG ( U V W)
00150     (SETQ U L) (SETQ V (MINM L)) (SETQ W NIL)
00160     A (CØND ((NULL U) (RETURN W)))
00170     (SETQ V (MINM U))
00180     (SETQ U (DIFFLIST V U))
00190     (SETQ W (APPEND W (LIST V)))
00200     (GØ A))))
00210  ))
00220  SEQUENCE
00230     ((11 10 7 15 3))
00240 (LAMBDA () (SEQUENCE (QUØTE (33 12 54 7 68 2))))
00250     ()
00260 STØP))
READY.
```

Output

```
RESUME LISPF
WAIT,
 VALUE
 (SMALLER MINM DIFFLIST SEQUENCE)
 VALUE
 (3 7 10 11 15)
 VALUE
 (2 7 12 33 54 68)
READY.
```

Comments

The numbers in sequence identifying the successive lines of the input are not taken in ("seen") by the LISP system. LISP DATA is the name of the input file.

The expression "RESUME LISPF" starts the LISP system evaluating. The three values in the output correspond with the three expressions to be evaluated: first, the command DEFINE, on lines 10 to 210; second, the command SEQUENCE, on lines 220 and 230; and third, the command on lines 240 and 250, which shows an alternative method available in 7090 LISP for giving the calculation SEQUENCE to the computer. The expression READY indicates that the current operations are finished and that the Compatible Time-Sharing System is awaiting another command.

LISP – On the Programming System

Robert A. Saunders

Information International, Inc.

List Structure

LISP is a programming system which is very useful for manipulating symbols and complex data structures. The programmer unfamiliar with systems of this type will find that LISP is considerably different from any system he has used before. This paper is intended to be used in conjunction with the LISP 1.5 Manual (1) to get one started, as easily as may be, in LISP.

LISP is an acronym for List Processor. A list is one type of S-expression, which latter is the basic building block of LISP. We quote from the LISP Manual: "An S-expression is either an atomic symbol or it is composed of these elements in the following order: a left parenthesis, an S-expression, a dot, an S-expression, and a right parenthesis." An atomic symbol is a symbol composed of an indefinite number of letters or digits (in the IBM 7090 LISP system, not more than 30) and beginning with a letter. Atoms are usually considered indivisible, but we will mention here that atoms have a structure called a "property list" which contains information such as its BCD (binary-coded decimal) representation and other data. Property lists will be

discussed in detail later.

Given these definitions, Figure 1 illustrates some S-expressions, along with a graphic-symbolic representation of them. The rectangles in Figure 1 denote computer registers, or cells of storage, in which the parts of expressions are stored. Each line and arrow denotes a "pointer", that is to say, an address stored in one register that "points" to another register. The type of binary tree structure shown in Figure 1 can be as complex as required.

"List notation" is a device for representing more complicated data structures than can conveniently be represented by a binary tree. List notation is an abbreviation for dot notation. A list can be defined as any number (including zero) of S-expressions (where by S-expressions we shall mean lists as well as the dot notation structures introduced previously), separated by spaces (or commas — space and comma being interchangeable), and the whole surrounded by parentheses. Examples are shown in Figure 2. The list (B C D), for example, is an abbreviation for (B . (C . (D . NIL))) where NIL is a special atom marking the end of a list. The structure ( ) is always equivalent to NIL. The reader should practice going from list to dot notation and back again to become familiar with the idea.

List Structure Operators

"List structure" is a general term applied to anything that can be written in LISP, using dot notation, list notation, or both. List structure is manipulated by LISP functions, which are themselves written as S-expressions. A function call is denoted by a list whose first element is the function designator and whose succeeding elements (if any) are the arguments of the function. Thus

(CONS A B)

is a call for the function CONS to be applied to two arguments, A and B.

(EQ (CAR J) (CDR Q))

is a call for the function EQ to be applied to the results of applying the function CAR to J and the function CDR to Q.

Some atoms, although written as if they were ordinary functions,

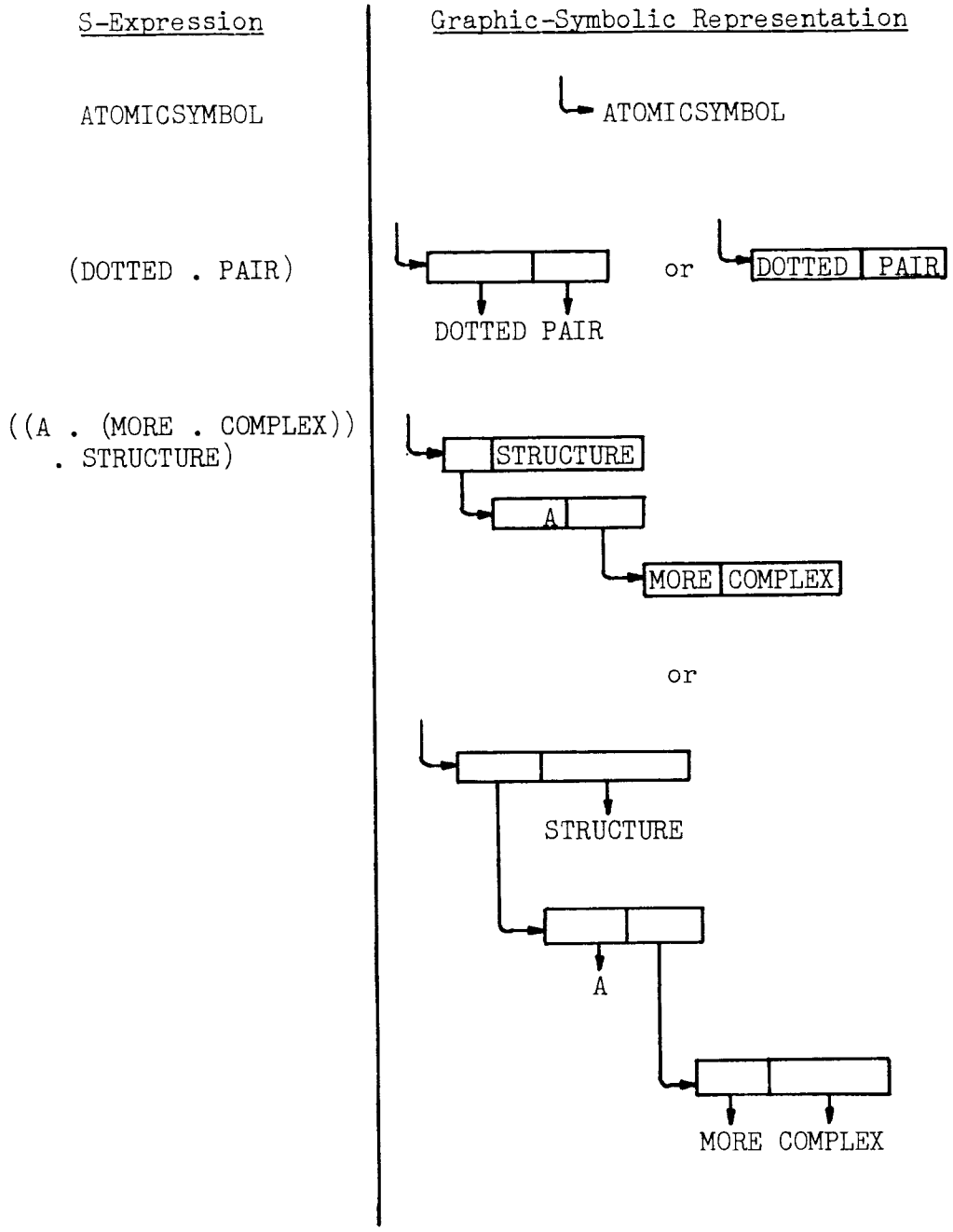


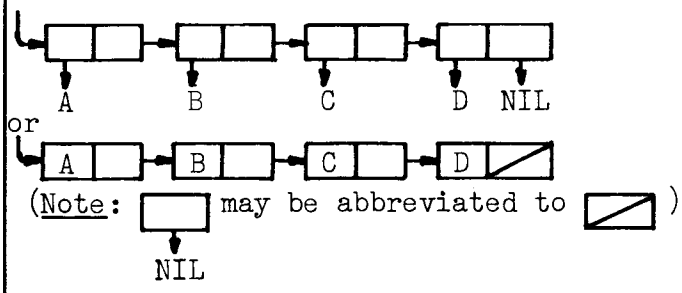
Figure 1. Some Sample S-Expressions, and their Graphic-Symbolic Representation

S-Expression

Graphic-Symbolic Representation

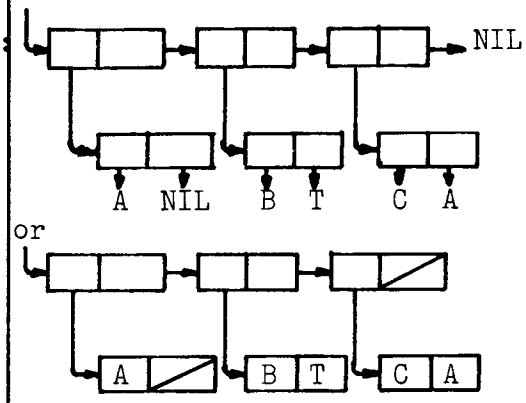
1. A list of atoms:

   (A B C D)



2. A list of dotted pairs:

   ((A . NIL) (B . T)
   (C . A))

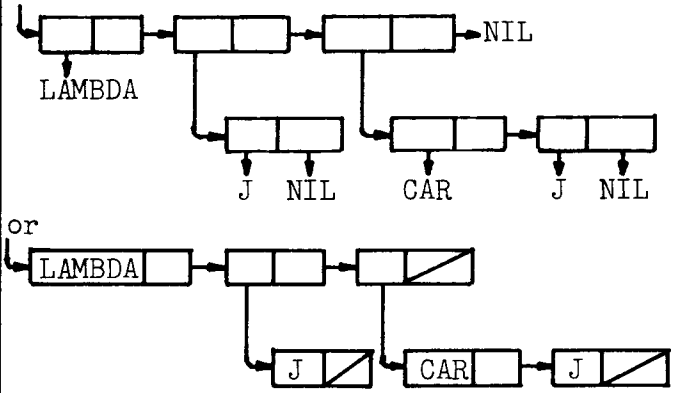


3. A list of various elements:

   (LAMBDA,(J),(CAR,J))

   (Note: Comma is interchangeable with space.)

   Here (J) and (CAR J) are lists within a list.



4. A list of no elements:

   ()

Figure 2. Some Sample S-Expressions in List Notation, and their Graphic-Symbolic Representation

have special significance to the system, and give special treatment to their arguments. Representative of these are QUOTE, LAMBDA, and LABEL. These are called special forms and will be discussed below at appropriate places.

LISP is so arranged that recursive functions are admissible — i.e. functions that use themselves in the evaluation process. This is a very powerful feature, and recursive coding in LISP is a frequent occurrence. The LISP 1.5 Manual gives details and examples.

Some LISP Functions

A host of functions are defined in LISP operating systems for manipulating list structure. We shall discuss the more basic of these here, referring the reader to the manual for a treatment of the more involved ones.

CAR is a function of one argument (which is an S-expression, of course; there are no other kinds of arguments) and gives as its value the left-hand S-expression of the dotted pair. To make the call of the function CAR work with the argument to which it applies, it is necessary to use the special form QUOTE. QUOTE is used to signify that an expression stands for itself rather than for something to be interpreted further; thus it serves to isolate a program from its data. This also will be discussed further below. Here are some examples of CAR:

```
(CAR (QUOTE (A . B))) → A
(CAR (QUOTE ((A . B) . (C . D)) )) → (A . B)
(CAR (QUOTE (A B C))) →
        (CAR (QUOTE (A . (B . (C . NIL))) )) → A
(CAR (QUOTE ((A) (B)) )) →
        (CAR (QUOTE ((A . NIL) . ((B . NIL) . NIL)) )) →
                (A . NIL) → (A)
```

It will be seen from the above that CAR of a dotted pair is the left member, and CAR of a list is the first member. CAR is an acronym for "contents of address of register," which refers to the way list structure is organized in storage in the 7090 LISP system.

CDR is a function of one argument which gives the right hand half of a dotted pair. To see what it does to a list, let's work out some examples:

(CDR (QUOTE (A . B))) → B
(CDR (QUOTE (A B C))) →
    (CDR (QUOTE (A . (B . (C . NIL))))) →
        (B . (C . NIL)) → (B C)
(CDR (QUOTE (J))) →
    (CDR (QUOTE (J . NIL))) → NIL

We see that CDR of a list is ALL OF THE LIST EXCEPT THE FIRST ELEMENT. In brief, it is the rest of the list. CAR of a list of one element is the element, and CDR of such a list is NIL. There will be considerable occasion to use this fact.

CONS: So far we have taken list structure apart; now let's put some together. CONS of two arguments constructs (hence the name) a dotted pair of those arguments, in order. Thus:

(CONS (QUOTE A) (QUOTE B)) → (A . B)
(CONS (QUOTE A) (QUOTE (J K L))) →
    (A . (J K L)) → (A J K L)
(CONS (QUOTE A) (QUOTE NIL)) → (A . NIL) → (A)

The examples illustrate a very important use of CONS, i.e., tacking an element onto the front of a list.

We need now some functions for testing list structure. A predicate is a function whose only allowed values are the Boolean values T and F, for truth and falsity respectively. In all versions of LISP, the actual Boolean value for falsity is NIL, and the atom F is recognized as equivalent to (QUOTE NIL). The actual Boolean value for truth is either T or *T*, depending on the version of LISP you use; but T is recognized as being equivalent to (QUOTE T) or (QUOTE *T*), whichever is appropriate. Here are some predicates:

EQ is a predicate for testing whether two atoms are the same, and is thus a function of two arguments. Its value is undefined (i.e., unpredictably T or NIL) when applied to S-expressions other than atoms, although we can say for certain that EQ of two different S-expressions is NIL.

(EQ (QUOTE A) (QUOTE B)) → NIL
(EQ (QUOTE A) (CAR (QUOTE (A B C)))) → T or *T*
(EQ (CDR (QUOTE (A))) (QUOTE NIL)) → T or *T*

NULL is a predicate for testing for NIL. It is a function of one

argument, and returns truth if that argument is NIL.

```
      (NULL (QUOTE NIL)) → T or *T*
∴.    (NULL F) → T or *T*
      (NULL (QUOTE A)) → NIL
      (NULL (CDR (QUOTE (A)) )) → T or *T*
```

NULL can be regarded as a special case of EQ. The following are equivalent:

```
(NULL (CDR (QUOTE (A)))) and
(EQ (CDR (QUOTE (A))) (QUOTE NIL))
```

NULL is particularly useful for testing for the end of a list.

ATOM is a predicate of one argument which determines whether that argument is an atomic symbol.

```
(ATOM (QUOTE ATOMICSYMBOL)) → T or *T*
(ATOM (QUOTE (A B C))) → NIL
(ATOM (QUOTE NIL)) → T or *T*
(ATOM (CAR (QUOTE (A)))) → T or *T*
```

The basic device for using predicates is a function called COND (for "conditional"). COND is a special form which takes an indefinite number of arguments. Each argument is a list of two elements. COND proceeds from argument to argument, evaluating the first element of each; and the value of the COND is the second element of the first argument whose first element is true.

```
(COND ((QUOTE NIL) (QUOTE A)) ((QUOTE *T*)
        (QUOTE B)) ) → B
(COND ((NULL (CDR (QUOTE (A)))) (QUOTE (DONE)))
        (T (QUOTE FOO)) ) → (DONE)
```

The first element of each argument is ordinarily a predicate, although this is not necessarily so; any function may be used, and it will be considered true if its value is anything except NIL. If the COND runs out of arguments, its value is undefined: an error diagnostic will occur.

Variables and Bindings

Ordinarily we want to write a program that will handle a variety of inputs, and this is true of LISP. In the examples so far, we have seen functions applied to specific arguments to get specific results. Now we must provide for arbitrary arguments. This is done by means of bound variables.

RULE: AN ATOM NEVER STANDS FOR ITSELF UNLESS IT IS PART OF A QUOTED EXPRESSION.

Consider the atoms in the expression

```
(CONS J (QUOTE (A B C)))                    (1)
```

CONS stands for the function which does the CONS-ing; A, B, and C are part of a quoted expression, so they stand for themselves. Now J doesn't stand for itself. What does it stand for?

To answer this question, we need to dig deeper. In LISP, functions are evaluated by an interpreter. The interpreter contains an association list, or A-list, of bindings of variables. The A-list is a sort of symbol table which contains variables and what they stand for. It is a list of dotted pairs, and if at the time of evaluating (1) the A-list were

```
((Q . (J K)) (J . (NIL)) .... )
```

the value of (1) would be ((NIL) A B C). In this example, J is a bound variable which is bound on the A-list. The interpreter looks at an atom's property list before looking for the atom on the A-list, since variables can be, and functions usually are, bound on property lists. A variable or an atom that is a function may have several different bindings at one time. The interpreter searches first the property list of the atom and then the A-list, stopping when it first finds a binding. An atom may be bound as a function or as a variable or both. Now property lists distinguish between functional and variable bindings, but the A-list has no such distinctions. If a binding of a variable, e.g. J, is required, it will search first the property list of J for a variable binding (called an APVAL); if it is unsuccessful, it will then search the A-list for a binding of J. In the example above, J is bound to (NIL).

The principal way of putting variable bindings on the A-list is by

use of LAMBDA. LAMBDA is used as if it were a function of two arguments, which are (1) a list of the variables to be bound and (2) the expression which is to be evaluated while that binding is in force. The entire expression containing the LAMBDA is a functional expression, and takes as arguments expressions following it in the usual way. Thus the expression

$$\underbrace{((\text{LAMBDA (J) (CONS J (QUOTE (A B C)))})}_{\text{functional expression}} \underbrace{(\text{QUOTE (NIL)})}_{\text{argument}}) \quad (4)$$

performs the following:

1. Evaluates (QUOTE (NIL)), yielding (NIL);
2. Pairs this with J, the variable to be bound, and puts (J . (NIL)) at the head of the A-list;
3. Evaluates the arguments of CONS. J is bound on the A-list to (NIL), so (NIL) is the first argument. (QUOTE (A B C)) is a quoted expression, whose value is (A B C);
4. Evaluates CONS, and finds on its property list that it is a machine language function;
5. Passes the arguments to CONS, which returns ((NIL) A B C);
6. Strikes the binding of J from the A-list.

In this example, one variable was bound by the LAMBDA. Any reasonable number of variables can be bound at one time; all that is necessary is that the number of arguments supplied be the same as the number of variables to be bound.

Since the A-list is searched from the top down, the same variable may be bound several times, and in each case the most recent binding will be used.

The use of QUOTE should now be quite clear: you quote something that you do not want evaluated, but want to stand for itself.

Free Variables

When the LISP interpreter evaluates a variable, it doesn't necessarily confine itself to that part of the A-list attached by the present binding; it searches the entire A-list until it either finds a binding, or

runs out of A-list, which results in an error call. A variable used but not bound within the scope of the current function is said to be a free variable.

LABEL

We have seen that variables are usually bound on the A-list, and functions are usually bound on property lists. The A-list is useful for short-term bindings, as these are erased when you are through with them, while the property lists are altered only upon specific direction to the system. A function name required only for a short time, say within a recursion, can be bound on the A-list through use of LABEL. LABEL is written as a function of two arguments, where the first is the atom to be bound and the second is a functional expression. For example, consider:

```
(LAMBDA (J) (COND ((P1 J) (F1 J)) (T ((LABEL FN
        (LAMBDA (J) (COND ((NULL J) NIL) (T (CONS
        (CONS (CAR J) NIL) (FN (CDR J)))) ))) (CAR J))) ))
```

This binds FN so that the appearance of FN within the functional expression is treated as a reference to the entire functional expression.

LABEL is little used; it is usually more convenient to use a separate function bound on a property list.

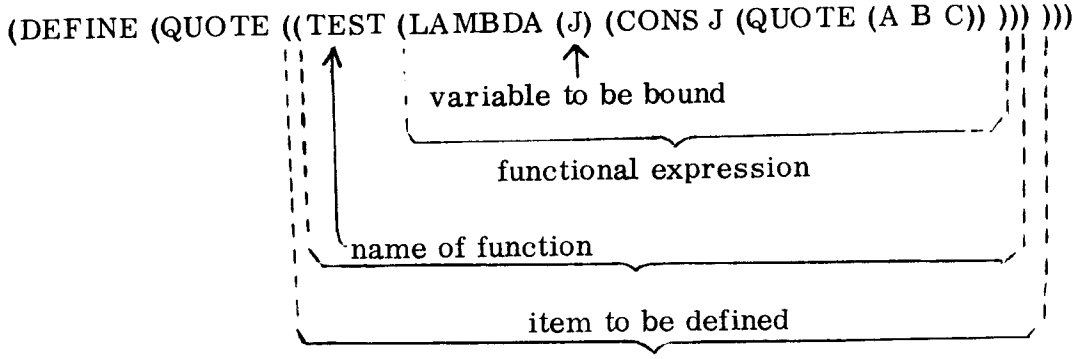
Defining Functions

We want to be able to put functional expressions on the property lists of atoms so that they are available for long term use in the system. A couple of points must be covered first.

So far we have discussed functions whose operation makes no change (at least no "visible" change) in the state of the LISP system except to return a value when called. There are also functions, called "pseudo-functions" in the LISP 1.5 Manual, which have various important side-effects resulting from their operation. Indeed, the side-effect may be the prime purpose in using a function; the resulting value may be trivial, such as a supplied argument or NIL. These include, as obvious examples, input-output functions such as PRINT. There are also functions which make a permanent change in memory (permanent in the

sense that you have to try explicitly to make such change go away), such as RPLACA, RPLACD, and DEFINE.

The pseudo-function DEFINE takes one argument, which is a list of "items". Each "item" is a list of two elements, the first of which is an atom, and the second is a functional expression to be associated with it. Thus we might write:



Functions defined with DEFINE have the S-expression on their property list preceded by the indicator EXPR. Other S-expressions, with other indicators such as FEXPR, can be put on property lists with other defining functions.

The interpreter, when it sees an atom in the position of a function, will search the atom's property list to see if it carries a functional expression. The functional expression is applied to the arguments of the function.

CSET

Variable bindings which are to be used over a long period of time are most conveniently bound on property lists. The function CSET takes two arguments: the value of the first argument is an atom upon whose property list is created a binding to the value of CSET's second argument. Such a binding is called an APVAL An example:

(CSET (QUOTE JJ) (QUOTE (TTTT)))

Having done this, we could use JJ as in the following example:

Use	Result
(CONS JJ (QUOTE (A B C)))	((TTTT) A B C)

Since the interpreter searches property lists before the A-list, a LAMBDA binding of a variable with an APVAL won't "take." Thus T, F, and NIL, for example, can never be used as variables bound by a LAMBDA. In 7090 LISP, we have as part of the system the result of (CSET (QUOTE T) (QUOTE *T*)).

EVAL

The notation we have been using for functions is that used by EVAL, which is the major part of the interpreter. In addition to being called implicitly in the process of function evaluation, EVAL is available explicitly to the programmer. The programmer may call upon it explicitly as a function of two arguments: the expression to be evaluated (called a form), and the A-list to be used. Thus:

(EVAL (QUOTE A) (QUOTE ((A . (Q R D))))) → (Q R D)
(EVAL (QUOTE (CAR P)) (QUOTE ((P . (Q R S))))) → Q
(EVAL (QUOTE (CDR (QUOTE (J K L)))) (QUOTE NIL)) →
(K L)

Notice that we have two levels of evaluation here. In the last example, the interpreter evaluates the arguments of EVAL, so the arguments passed to EVAL are (CDR (QUOTE (J K L))) and NIL. The value of (CDR (QUOTE (J K L))) is then determined precisely as if it had been seen by the interpreter.

Functions and Special Forms

There are two rather different types of atoms bound on property lists which take the position of functions in expressions. We have seen several examples of each. These can be categorized as follows:

1. Functions, such as CONS, which take a fixed number of arguments, all of which are evaluated before being passed to the function.

2. Special forms such as LAMBDA, QUOTE, and COND, which either (a) take a variable number of arguments (COND), or (b) require that their arguments not be evaluated (LAMBDA, QUOTE, COND), or (c) require access to the present A-list (LAMBDA).

Functions of the first class are called EXPR's or SUBR's, and functions of the second class are called FEXPR's or FSUBR's. FEXPR's and FSUBR's are always written as if they were functions of precisely two arguments. When the interpreter encounters an FEXPR or FSUBR, the list of unevaluated arguments (i.e., CDR of the expression headed by the FEXPR or FSUBR) is given to the FEXPR or FSUBR as its first argument, and the current A-list is given as the second argument. Since COND is a special form,

(COND ((ATOM A) A) (T (FN (CAR A))))

is interpreted by giving as arguments to COND:

1. (((ATOM A) A) (T (FN (CAR A))))

2. The current A-list.

Now COND can't be supplied the value of its explicit arguments (clearly ((ATOM A) A) is gibberish if evaluated — for we wind up applying *T* or NIL as a function to A). But we do need to do some evaluating. We must evaluate (ATOM A), and

1. If it is true, evaluate A and return it as answer;

2. If it is not true, we must evaluate T, which is found to be true, and upon so finding it, we must evaluate (FN (CAR A)) and return its value as answer.

These evaluations are performed by calling upon EVAL. The first argument of EVAL is the expression to be evaluated (called a form) and the second is the A-list to be used. Thus, (although no LISP system does it quite this way, for reasons which will become clear presently) we can as an illustration write COND as an FEXPR which calls another function, EVCON, written as an EXPR, to do the work:

```
COND:   (LAMBDA (L A) (EVCON L A)); FEXPR

EVCON:  (LAMBDA (L A) (COND ((NULL L) (ERROR
                (QUOTE A3)))
        ((EVAL (CAR (CAR L)) A) (EVAL (CAR (CDR
                (CAR L))) A))
        (T (EVCON (CDR L) A)) )); EXPR
```

Let's trace this through, supposing that A is bound to the atom Q.

Expression as seen by EVAL:

(COND ((ATOM A) A) (T (FN (CAR A))));

Arguments of COND, and hence arguments of EVCON:

(((ATOM A) A) (T (FN (CAR A)))); ((A . Q))

Since L isn't NIL, we call EVAL of:

(ATOM A); ((A . Q))

EVAL finds that Q is indeed atomic and returns T. So we call EVAL:

A; ((A . Q))

EVAL promptly returns Q, and we are done.

Since EVCON contains a COND, we cannot in fact write the function in exactly this way; in substance, however, EVCON and COND are coded in machine language in the interpreter to act in this way. (Thus they are actually SUBR and FSUBR rather than EXPR and FEXPR.)

EXPR's and SUBR's

The indicators EXPR and SUBR are used on the property lists of atoms to indicate functional bindings of two different types. The indicator EXPR denotes a functional binding written as an S-expression, and it is this type that is created by DEFINE. Most of the functions in the LISP system, however, are written in machine language rather than as S-expressions. Following the SUBR on the property list is a special cell containing information that the interpreter requires in order to link to the machine language subroutine. An entirely analogous relation exists between FEXPR's and FSUBR's: following FEXPR there is an S-expression, and following FSUBR there is a subroutine linkage control word.

Functional Arguments

Certain functions conveniently take other functions as arguments. The most common example is MAPLIST, whose definition is:

```
(LAMBDA (L FN) (COND ((NULL L) NIL)
(T (CONS (FN L) (MAPLIST (CDR L) FN))) ))
```

FN is in the position of a function (it is CAR of a list) and is a bound variable. Let's use MAPLIST in an example to see how it works. Suppose we have a list, say (A B C), and we want to CONS an X onto each element to get ((A . X) (B . X) (C . X)). We could write:

```
((LAMBDA (J) (MAPLIST J (QUOTE (LAMBDA (K)
        (CONS (CAR K) (QUOTE X)) )))) (QUOTE (A B C)))
```

Now let's trace through this. The arguments of MAPLIST are:

```
(A B C); (LAMBDA (K) (CONS (CAR K) (QUOTE X)))
```

which are bound to L and FN respectively. MAPLIST is evaluated in the usual way; the interpreter finds the binding of all functions except FN on their property lists, while FN is bound on the A-list.

The use of QUOTE with a functional argument leads to a problem which is best illustrated with an example. Consider the function TESTR defined as:

```
(LAMBDA (L FN) (COND ((NULL L) NIL) ((P1 L) (FN))
        (T (TESTR (CAR L) (QUOTE (LAMBDA NIL
                (TESTR (CDR L) FN))) )) ))
```

Assume P1 is some predicate, which we will choose to be true or false as suits our convenience. Let's apply this to ((A) (B) (C)); (LAMBDA (X) X) and see what happens.

```
L:  ((A) (B) (C))
FN: (LAMBDA (X) X)
```

L isn't NIL, and for the sake of argument we assume P1 of ((A) (B) (C)) to be false; so, we take the T leg of the conditional, enter the first TESTR, and get:

```
L:  (A)
FN: (TESTR (CDR L) FN)
```

and now we are in trouble.

What the programmer intended to have upon interpreting FN was

CDR of ((A) (B) (C)), i.e. ((B) (C)); and what he will get is CDR of (A), which is NIL. The wrong binding of L will have been used. What is needed here is a device to preserve the A-list as it was upon entering TESTR for the first time, and to make this A-list available at the right time. The interpreter contains a monument to this problem called the FUNARG device. We can straighten things out by defining TESTR as:

```
(LAMBDA (L FN) (COND ((NULL L) NIL) ((P1 L) (FN L))
      (T (TESTR (CAR L) (FUNCTION (LAMBDA NIL
            (TESTR (CDR L) FN))) )) ))
```

The only change is the substitution of FUNCTION for QUOTE. Proceeding as before, we enter the first TESTR and evaluate its arguments. As before, (CAR L) is (A), but now we run into (FUNCTION (TESTR (CDR L) FN)). The interpreter recognizes FUNCTION, replaces it by the special atom FUNARG, and attaches the old A-list to give

```
FN: (FUNARG (TESTR (CDR L) FN) ((L . ((A) (B) (C)))
      (FN . (LAMBDA (X) X))))
```

The interpreter, upon interpreting FN, now sees the FUNARG, and retrieves the function (TESTR (CDR L) FN), and the old A-list; and now we have the correct L and will get what the programmer expected.

Admittedly, this is a complicated example, but the point is simple enough: when quoting a function, use FUNCTION instead of QUOTE and you will get what you want. The problem arises with free variables in functional arguments; in TESTR, both L and FN are essentially used free.

Property Lists

The property lists used in the IBM 7090 LISP System will be discussed since they are representative of property lists in general and are reasonably simple. Generally, properties are of two types. One word (one element) properties, called flags, indicate information simply by their presence or absence. TRACE (see below) is representative of this type.

Most properties consist of two elements. The first element is an indicator, such as EXPR; the second is a pointer to associated information, such as a functional expression in the case of EXPR.

Here is a list of the properties used in the 7090 LISP system

TRACE — Flag; causes interpreter to print out function called, its arguments, and its value.

EXPR — Indicator; marks functional definition.

FEXPR — Indicator; marks special form functional definition.

SUBR — Indicator; used in case of EXPR's translated to machine language subroutines. Marks pointer to a word of non-list structure carrying address of subroutine and number of arguments it expects.

FSUBR — Similar to SUBR, except doesn't evaluate arguments. List of unevaluated arguments and current A-list supplied as arguments.

APVAL — Indicator; acronym for APPLY VALUE. Indicates a binding of the atom in which it appears to the S-expression pointed to.

PNAME — Indicator; acronym for PRINT NAME. Pointer points to a list of pointers to words containing the BCD representation of the character strings in the atom's name.

SPECIAL — Indicator; pointer is to a cell (the SPECIAL CELL) which contains certain variable bindings used in compiled code. See Compilation for a more detailed discussion.

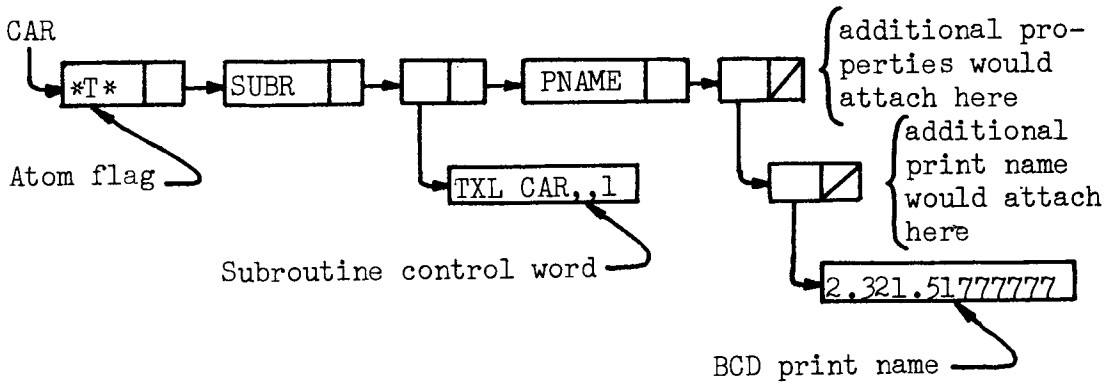
SYM — Indicator; short for symbol value. Pointer points to a non-list word containing a number. SYM's are used by LAP (LISP Assembly Program) for storing the values of machine instruction such as CLA, STO, etc.

COMMON — Flag; used by the compiler to denote variables whose values are to be found by the interpreter.

Other indicators may be originated and used by the programmer. Various functions are available for manipulating property lists; property

lists are fine places on which to hang miscellaneous information about atoms.

Here is a typical property list.



Numbers in LISP

So far we have confined our attention to symbol manipulation. LISP also has facilities for numerical work. An atom starting with a digit is interpreted as a number. Numerical atoms always stand for themselves, and so need not be QUOTE'd. There are a set of functions for doing arithmetic of which PLUS is representative:

(PLUS 3 4) → 7
((LAMBDA (J) (PLUS J 10)) 34) → 44

Other functions are available, but vary from version to version, so the reader is referred to the manual for his particular version of LISP for a discussion of specific numerical functions.

Arithmetic in LISP is not very fast, as the system has to resort to various tricks to turn a number into an entity that is referenceable by a list structure pointer.

PROG

No discussion of LISP is complete without a mention of PROG. PROG is a special form which permits writing LISP manipulations in a style similar to ALGOL and FORTRAN. Let's illustrate with an example; LAST will be defined as a function to find the last element of a list:

```
(DEFINE (QUOTE (
(LAST (LAMBDA (J) (PROG (K)
        (SETQ K J)
A       (COND ((NULL K) (RETURN NIL)) ((NULL (CDR K)
              (RETURN (CAR K))) )
        (SETQ K (CDR K))
        (GO A) ))) )))
```

PROG is written as if it were a function of an indefinite number of arguments. The first "argument" is a list of program variables, which are bound on the A-list to NIL when the PROG is entered. Succeeding arguments, if atomic, are statement labels, and otherwise are statements. SETQ is a special form which re-binds its first argument (which it effectively quotes) to the value of its second argument. COND is essentially the same as when used outside PROG; if it runs out of arguments without finding a true predicate, however, control passes to the next statement instead of giving an error diagnostic. GO is a special form for transferring control. If its argument is atomic, control passes to the statement bearing that label; otherwise the argument is evaluated and control passes to the label which is the same as the resulting value. RETURN causes the PROG to be quitted, returning as result the value of the argument of RETURN. If a PROG runs out of statements, i.e. "drops out the bottom," it returns NIL.

PROG works by first binding the program variables, then scanning its arguments for statement labels which are put on a list of labels called a GOLIST. Statement labels can be the same as bound variables; since they are kept on different lists, however, no confusion will arise. The PROG is executed by EVAL-ing each statement as it is encountered and throwing away the value.

SETQ isn't restricted to rebinding program variables; it can bind any variable on the A-list. In this example, K isn't really necessary; J, originally bound by the LAMBDA, could be used as well. It is sometimes useful to have a PROG re-bind a variable in a higher function; all that is required is that the variable to be bound be somewhere on the A-list.

There is nothing that can be done using PROG that can't be done without it, and vice versa; whether one uses PROG or not is a matter of taste and convenience. Some purists will contend that using PROG is somehow tinged with immorality, and not "pure" LISP; this is silly, of course, but they do have a point: one should learn to write LISP

both with and without PROG, or one may fall into the habit of using PROG for everything. The last example could be written without PROG, but recursively, as follows:

```
(DEFINE (QUOTE (
(LAST (LAMBDA (J) (COND ((NULL J) NIL) ((NULL
        (CDR J)) (CAR J))
        (T (LAST (CDR J))) ))) )))
```

Input-Output and EVALQUOTE

Input and output in LISP are handled by built-in pseudo-functions READ and PRINT, which read and print one S-expression, respectively. All the examples we have discussed so far are in the format used by EVAL. In PDP-1 LISP, EVAL reads one S-expression, evaluates it (starting with a null A-list), and prints out the result. The usual LISP job consists of defining some functions, and then trying them on some special cases. It turns out that you have to QUOTE practically everything: the argument of DEFINE must be quoted, and the test case arguments must be quoted also. This is enough of a nuisance so that in most LISP systems, input is done through a function called EVALQUOTE which does the quoting for you. EVALQUOTE reads "pairs", consisting of a function name (or functional expression), followed by a list of arguments. Thus, each pair consists of exactly two S-expressions. Some examples:

EVAL	EVALQUOTE
(CSET (QUOTE JJ) (QUOTE (A B C)))	CSET (JJ (A B C))
(CONS (QUOTE F) (QUOTE (G)))	CONS (F (G))
((LAMBDA (J) (CAR J)) (QUOTE (A . B)))	(LAMBDA (J) (CAR J)) ((A . B))
(DEFINE (QUOTE ((FN (LAMBDA NIL NIL)))))	DEFINE (((FN (LAMBDA NIL NIL))))
(FN)	FN NIL

EVALQUOTE breaks down the argument list into arguments, quotes each, and passes function and arguments off to the interpreter to be evaluated.

The LISP READ program consists of two basic parts. There is a machine language routine to convert character strings into atoms. Its output is the atoms read, with special atoms for parentheses and period. A recursive subroutine CONS's these into list structure. When a character string is read, it must be compared with the character representations of all atoms seen so far, to determine whether this string is a new atom or a reference to one seen before. Therefore there must be a means of rapid access to all the atoms in the system. There exists, therefore, a list called the object list, or OBLIST, of all atoms. To speed things up, this is usually organized as a list of a hundred or so sub-lists, where the atoms are distributed among the sublists by a computation upon their BCD representations. This computation is chosen so as to give as uniform and random a distribution of atoms among the various sub-lists as possible. Some versions of LISP have the atom OBLIST APVAL'ed to point to the object list, so it can be accessed by EVAL.

Compilation

Interpreters are generally slow; LISP's is no exception. The slowness is primarily a result of searching the A-list for variable bindings, and searching property lists for functional bindings. The situation can be alleviated to a considerable degree by compiling machine code to evaluate functions. As a bonus, one obtains more storage space, for the compiled code generally takes less storage than the S-expression, which can be thrown away after it is compiled. Nothing is free, however, and the price of compilation is several difficulties:

1. Free variables require special treatment. In interpreted code, as we have seen, all variable bindings are accessible because the interpreter can search the entire A-list if necessary. In compiled code, the bindings of the present function are stored in its private area of the pushdown list, and no other part of the pushdown list is accessible; so one can't get at variables bound outside the current function.

There are various artifices to alleviate this situation. One is to establish a special cell for each variable used free and to post the present binding in it each time it is re-bound. The old binding is saved on the pushdown list and is restored when necessary. This method is reasonably fast, but the special cells aren't generally available to the interpreter. Another way is to post variable bindings on the A-list and

use EVAL to look them up. This, although ultimately general, throws away most of the speed advantage of compiled code. In the 7090, variables declared COMMON are treated in this way. COMMON variables are required to get at APVAL's.

2. There are problems with functional arguments. Again, we don't have access to the A-list, so we can't use the FUNARG device to keep bindings straight. In 7090 LISP, functional arguments must be COMMON, so the burden is thrown onto the interpreter, which can use the FUNARG device. In Q-32 and M-460 LISP, which are the only other systems containing compilers as of this writing, the problem is more or less ignored; one still writes FUNCTION, and functional arguments must be declared SPECIAL. Except in pathological cases like TESTR, a little care is all that is needed to avoid looking at incorrect bindings. The proposed LISP for the CDC 6600 has a device in which SPECIAL variables are stored on the pushdown list and referenced indirectly through the special cells; and all bindings of each special variable are chained on the pushdown list. Some code is used to exclude that part of the pushdown list which would follow the equivalent of a FUNARG by tracing down the chained bindings.

3. Compiled calls to FEXPR's or FSUBR's can lead to trouble, since most of the usefulness of compilation is evaluating arguments of functions, and FEXPR's and FSUBR's get theirs evaluated whether you want it or not. As long as the compiler is dealing with FEXPR's and FSUBR's defined in the system, however, there is no serious problem, for the compiler is designed to recognize these and give them appropriate treatment. Hart has proposed a method of abolishing user-defined FEXPR's and FSUBR's by use of a system of macros, and this system is incorporated in those versions of LISP which are compiler-based. (2, 3, 4)

Because of the advantages of compiled code, we can expect to see more and better compiler-based LISP systems in the future. These will probably take input in languages similar to ALGOL.

Conclusion

This paper seeks to ease the average programmer's entrance into LISP programming. The reader is advised to refer to the LISP 1.5 Manual and Hart's LISP Exercises (in this volume) to broaden his understanding of the language. The best teacher is often experience;

and access to a computer should be impetus enough for the reader to try his hand at writing LISP.

References

1. McCarthy, John, et al., LISP 1.5 Programmers' Manual. The M.I.T. Press, Cambridge, Mass., 1962.
2. Hart, T. P. and T. G. Evans, "The M-460 LISP 1.5 System". Air Force Cambridge Research Laboratory memorandum. (Reprinted in this volume.)
3. Hart, T. P., "MACRO Definitions for LISP". Artificial Intelligence Project, Research Laboratory of Electronics and M.I.T. Computation Center Memo 57, Oct., 1963.
4. Saunders, R., "The LISP System for the Q-32 Computer." Information International, Inc. memorandum, Cambridge, Mass., Apr., 1964. (Reprinted in this volume.)

LISP−240 Exercises with Solutions

Timothy P. Hart and Michael I. Levin

Project MAC
Massachusetts Institute of Technology

The following exercises are carefully graded to mesh with the sections in Chapter I, "The LISP Language," in the LISP 1.5 Programmer's Manual. Each exercise should be worked immediately after reading the manual section indicated. Instructions, such as "Read Section 1.1", refer to this manual.

The exercises are more finely subdivided than the manual sections. If you don't understand an entire manual section, try the exercises — they may teach you enough to allow you to proceed on through the manual section.

A. S-expressions. Read Section 1.1.

Which of the following are S-expressions?

1. (X . Y)
2. ((Z)
3. CHI
4. (BOSTON . NEW . YORK)
5. ((SPINACH . BUTTER) . STEAK)

b. Elementary Functions. Read Section 1.2.

Write the S-expression which is the value of each of the following expressions. (Some of them are undefined!)

6. car[(A . B)]
7. cdr[(A . B)]
8. car[((FOO . CROCK) . GLITCH)]
9. cdr[((FOO . CROCK) . GLITCH)]
10. car[X]
11. cons[POOT;TOOP]
12. cons[X;(Y . Z)]
13. eq[X;X]
14. eq[Y;Z]
15. eq[(Q . R);(Q . R)]
16. atom[X]
17. atom[(TOOT . TOOT)]
18. atom[ISTHISATRIVIALEXERCISE]

C. Function Composition.

The notation of function composition is extremely useful: car[cdr[(A . (B . C))]] means that S-expression which is the car part of the cdr part of (A . (B . C)), namely B. The general rule for evaluating composed functions is to evaluate the innermost ones first, then the next outer layer, and so on, until the entire expression has been evaluated.

Example:

cons[cdr[((A . B) . C)];car[((A . B) . C)]] = cons[C;(A . B)] = (C . (A . B))

Evaluate the following:

19. car[cons[A;B]]
20. cons[car[(A . B)];cdr[(A . B)]]
21. car[cdr[(A . (B . C))]]
22. eq[car[(A . B)];cdr[(C . B)]]
23. eq[cdr[(A . B)];cdr[(C . B)]]
24. eq[A;car[cons[car[(A . B)];C]]]
25. atom[cons[A;B]]
26. atom[cdr[car[((A . C) . B)]]]
27. car[cdr[car[((A . (B . C)) . D)]]]

D. Variables.

The notion of a variable is very important: a small letter appearing in an expression in places where we expect to find an S-expression is to stand for the same (unspecified) S-expression in every place where it occurs in that S-expression. Using this notation we can give the following definitions:

car[(x . y)] = x
cdr[(x . y)] = y
cons[x;y] = (x . y)

(Notice that these definitions don't allow us to find an S-expression equivalent to car[ATOM]; this is why we say that such an expression is undefined.)

Which of the following identities are always true?

28. car[cons[x;y]] = x
29. cons[car[cons[x;y]];y] = (x . y)
30. atom[cons[u;v]] = T
31. cons[A;cons[x;Y]] = (A . (x . Y))
32. car[car[cdr[(A . ((B . C) . D))]]] = C
33. atom[cdr[cons[x;Y]]] = T
34. eq[X;car[cons[X;cons[u;v]]]] = T

E. List Notation. Read Section 1.3.

We will use list notation almost exclusively from now on; so do not try to avoid learning it. The second set of examples in the manual on page 4 are key prototypes for remembering what the basic functions do to S-expressions in list notation. Remember that:

car gets the first element of a list.
cdr eliminates the first element of a list by moving the leading left parenthesis past the initial S-expression.
cons inserts its first argument at the head of the list which is its second argument.
cdr of a list with only one element is the atomic symbol NIL.

Translate the following S-expressions into dot notation:

35. A
36. ((PLOOP) FLOP TOP)
37. ((X GLITCH) (Y CROCK)))
38. (((X)))
39. (SNAP (CRACKLE (POP)))

Which one of the following S-expressions cannot be expressed in list notation? Translate the rest into list notation.

40. ((A . NIL) . ((B . NIL) . NIL))
41. (A . (B . (C . NIL)))
42. (NIL . NIL)
43. ((A . (B . NIL)) . ((C . NIL) . NIL))
44. ((X . NIL) . ((NIL . Y) . NIL))

For the following, write the S-expression which is the value of each expression. Use list notation for your answer whenever possible. Some of them may be undefined.

45. car[(A B C)]
46. cdr[(A B C)]
47. car[(AB CD)]
48. car[(A)]
49. cdr[(A)]
50. car[A]
51. cdr[NIL]
52. cons[A;(B)]
53. cons[A;(B C)]
54. cons[A;B]
55. cons[A;NIL]
56. cons[NIL;A]
57. cons[(A);(B C)]
58. cons[(A B);(C D)]
59. cons[(A B);C]
60. cons[(A . B);((C . D)(E . F))]
61. cons[(A B);NIL]
62. car[(((A)))]
63. cons[eq[A;A];(F T F)]
64. atom[NIL]

F. Multiple car-cdr Functions.

Write multiple car-cdr LISP functions which will find the "A" in each of the following (denote the S-expressions by "x"):

65. (A)
66. (C A T)
67. (B . A)
68. (S T A Y)
69. (1 2 3 A)
70. ((A . B) (C . D))
71. ((B . A) (C . D))
72. (((C)) ((A)))
73. ((X . Y) (A . B))
74. ((((A))))

G. The Functions list and null.

We will use the predicate null to test for the atomic symbol NIL. null[NIL] = T, null[x] = F, if x is any other S-expression than NIL. The expression "( )" is identical to NIL, that is, eq[( );NIL], so null [( )] = T and atom[( )] = T, since null [NIL] = T and atom[NIL] = T. Notice this consistency: cdr[(A)] = ( ) = NIL.

The function list is used to create lists. It is a shorthand notation as the following identities illustrate:

$list[\ ] = (\ ) = NIL$
$list[x] = (x) = cons[x;NIL]$
$list[x_1;x_2] = (x_1\ x_2) = cons[x_1;\ cons[x_2;\ NIL]]$
.
.
.
$list[x_1;x_2;\ldots;x_n] = (x_1\ x_2\ \ldots\ x_n) = cons[x_1;cons[x_2;\ldots$
$cons[x_n;NIL]\ldots]]$

Examples

$list[A] = (A)$
$list[A;B] = (A\ B)$
$list[X;(Y\ Z)] = (X\ (Y\ Z))$

Exercises

75. null[A]
76. null[NIL]
77. null[( )]
78. null[car[(A)]]
79. null[cdr[(A)]]
80. null[cdr[((A B C))]]
81. list[A;B;C]
82. list[(A);(B);(C)]
83. null[cdr[list[A]]]
84. null[cdr[cons[A;B]]]

Write expressions involving only cons, list, and atoms which will generate the following expressions: (Example: (A B) = list[A;B])

85. (A . B)
86. (X Y)
87. ((A B) (C D))
88. ((A . B) (C . D))
89. (((A)))
90. ((B . C))
91. (( ) ( ))
92. (A . (B C D))

Evaluate:

93. eq[car[(A)];car[(B)]]
94. eq[cdr[(A)];cdr[(B)]]

95. atom[cdr[(X Y)]]
96. atom[cdr[cdr[(X Y)]]]
97. car[(A B C)]
98. cadr[(A B C)]
99. caddr[(A B C)]
100. cddr[(A B C)]
101. cadr[(A (B C) D)]
102. caadr[(A (B C) D)]
103. cadadr[(A (B C) D)]
104. cddadr[(A (B C) D)]
105. cddr[(A)]
106. eq[car[(A)];cadr[(B A)]]
107. cadr[list[A;B;C]]
108. caddr[list[A;B;C]]
109. atom[cadr[(A (B) C)]]
110. atom[cadr[(A B C)]]
111. cdr[cons[A;(B . C)]]
112. atom[car[cons[A;(B C)]]]
113. atom[cdr[cons[A;NIL]]]
114. null[list[A]]
115. eq[(A . B);cons[A;B]]
116. car[cons[car[(A)];NIL]]
117. eq[A;car[(A)]]
118. eq[(A B);list[A;B]]

H. The LISP Meta-Language. Read Section 1.4. (Skip Section 1.5.)

119. [T→A;T→B]
120. [F→A;T→B]
121. [F→A;F→B]
122. [eq[A;A]→car[(A)];T→ cdr[(B)]]
123. [null[X]→Y;null[( )]→NIL;T→atom[A]]
124. [atom[x]→atom[x];T→eq[X;X]]

In exercises 125-133 the variables f, m, n, t, x, y, and z are bound by the following table. All other variables are unbound and therefore undefined.

f = F	t = T	z = AB
m = AB	x = ((AB))	
n = (AB . C)	y = ((AB . C))	

125. [eq[m;z]→n;T→x]
126. [f→A;T→B;T→c]
127. [eq[AB;m]→A;T→B]
128. [atom[m]→A;T→B]
129. [atom[n]→A;T→B]
130. [eq[m;caar[x]]→y;T→w]
131. [[T→F;F→T]→F;T→[F→T;T→T]]

132. [[eq[cdr[n];cdar[y]]→F;T→T]→y;T→w]
133. [[eq[m;z]→eq[f;t];T→null[cdr[x]]]→B;T→C]

I. Lambda Notation and Lambda Conversion.

The lambda notation (also written "λ-notation") is necessary for specifying the order in which a function is to receive its arguments. Understand the difference between a function and a form.

Examples

form: cons[x;y]
function: λ[[x;y];cons[x;y]]
form: λ[[x;y[;cons[x;y]]][A;B]

The process of pairing the elements of a list of variables following a λ with their respective arguments to form a table of bindings such as in the recent exercise, and then evaluating the form inside the λ expression, is called λ-conversion.

Example: λ-conversion

form:	table:
λ[[x;y];cons[y;x]][A;B] = cons[y;x] = cons[B;A] = (B . A)	x = A y = B

Evaluate:

134. λ[[x];x][A]
135. λ[[y];y][(A)]
136. λ[[y];A][B]
137. λ[[x];car[x]][(A)]
138. λ[[x];car[(A)]][(B)]
139. λ[[u;v];u][A;B]
140. λ[[u;v];u][B;A]
141. λ[[u;v];v][A;B]
142. λ[[x;y];cons[car[y];cdr[x]]][(A . B);(C . D)]
143. λ[[x];λ[[y];car[y]][x]][(A)]
144. λ[[x;y];[atom[x]→y;T→A]][R;S]
145. cons[A;[null[Q]→B;T→C]]
146. cons[A; [[x];car[x]][(B)]
147. cons[A;[λ[[x;y];y][T;F]→B;T→C]]

J. Function Definition.

To evaluate the forms below, use the following table:

```
u = (A B)
v = X
w = (T U V)
pred = λ[[x;y];eq[x;car[y]]]
test = λ[[x];[atom[x]→ T;atom[car[x]]→ F;T→ F]]
```

Example

form:	table:
pred[A;(B)] = λ[[x;y];eq[x;car[y]]][A;(A B)] = eq[x;car[y]] = eq[A;car[(A B)]] = eq[A;A] = T	x = A y = (A B)

148. test[A]
149. test[(u B C)]
150. test[(v . u)]
151. pred[X;(A B)]
152. pred[F;(F T F)]
153. pred[test[v];w]

K. Recursive Functions.

Use the definition of jj given below to evaluate the following forms. Write the argument for jj at each recursive step in the evaluations.

```
jj = λ[[x];[atom[x]→ x;T→ jj[cdr[x]]]]
```

154. A
155. (A . B)
156. ((X . Y) . (X . Z))
157. (A B C)
158. (A (C . E))

Use this definition to evaluate the following forms:

```
match = λ[[kk;m];[null[kk]→ NO;null[m]→ NO;eq[car[kk];
            car[m]]→ car[kk];T→ match[cdr[kk];cdr[m]]]]
```

159. match[(X);(X)]
160. match[(A B E);(J O E)]
161. match[(K A Y);(S V E)]

Use the following definition to evaluate these forms:

```
twist = λ[[s];[atom[s]→ s;T→ cons[twist[cdr[s]];
                twist[car[s]]]]]
```

162. twist[A]
163. twist[(A . B)]
164. twist[((A . B) . C)]
165. twist[(A B C)]
166. twist[((A . B))]

L. list vs. S-expression Recursions.

Recursive LISP functions are generally terminated in either of two ways. In operations which deal with data which are lists (that is can be expressed in list-notation), the termination is by null when a datum is exhausted. In functions dealing with dot notation data (that is with data in which atoms other than NIL may occur in the cdr part), the corresponding terminating condition is atom.

Examples:

null-termination is used in a recursion down a list in the function among which decides whether or not an atom is among those on a list:

```
among = λ[[a;kk];[null[kk]→ F;eq[a;car[kk]]→ T;T→among
                  [a;cdr[kk]]]]
```

atom-termination is used in the recursion in inside which decides whether or not an atom appears anywhere in an S-expression:

```
inside = λ[[a;s];[atom[s]→ eq[a;s];
                  inside[a;car[s]]→ T;
                  T→ inside[a;cdr[s]]]]
```

Exercises (167-175 are list type; 176-178 are S-expression type.)

Write LISP functions for the following purposes:

167. to determine whether an atom is a member of a list.

e.g. member[B;(A B C)] = T
member[X;(A B C)] = F
member[A;(B (A B) C)] = F

168. to produce a table (list of dotted pairs) given two lists,

one of the references, the other of values.

e.g. pair[(ONE TWO THREE);(1 2 3)] = ((ONE . 1)(TWO . 2)
(THREE . 3))
pair[(PLANE SUB);(B47 THRESHER)] = ((PLANE . B47)
(SUB . THRESHER))

169. to append one list onto another.

e.g. append[(A B C);(D E F)] = (A B C D E F)
append[((A B) C (D (E)));((A))] = ((A B) C (D (E))
(A))

170. to delete an element from a list.

e.g. delete[Y;(X Y Z)] = (X Z)
delete[X;((U V) X Y)] = ((U V) Y)

171. to reverse a list. (Hint: use append.)

e.g. reverse[(A B C)] = (C B A)
reverse[(A (B C) D)] = (D (B C) A)

172. to produce a list of all the atoms which are in either of two lists.

e.g. union[(U V W);(W X Y)] = (U V W X Y)
union[(A B C);(B C D)] = (A B C D)
union[(A B C);(A B C)] = (A B C)

173. to produce a list of all the atoms in common to two lists.

e.g. intersection[(A B C);(B C D)] = (B C)
intersection[(A B C);(A B C)] = (A B C)
intersection[(A B C);(D E F)] = NIL

174. to find the last element on a list.

e.g. last[(A B C)] = C
last[((A B)(C))] = (C)

175. to reverse all levels of a list.

e.g. superreverse[(A B (C D))] = ((D C) B A)
superreverse[((U V)((X Z) Y))] = ((Y (Z X))(V U))

176. to determine whether a given atomic symbol is some part of an S-expression.

e.g. part[A;A] = T
part[A;(X . (Y . A))] = T
part[A;(U V (W . X) Z)] = F

177. to substitute one atomic symbol for another in an S-expression.

e.g. subst[X;Y;(U V W X Y Z)] = (U V W X X Z)
subst[X;Y;((U Y) X ((Y) Z))] = ((U X) X ((X) Z))

178. to produce a list of all the atoms in an S-expression.

e.g. listofatoms[(A (B C) D)] = (A B C D)
listofatoms[((B (C D)) E)] = (B C D E)
listofatoms[(A . (B . C))] = (A B C)

M. A Differentiation Program.

A simple differentiation program can easily be written in LISP which will serve two purposes for us: it will be an example of an application of LISP; and will illustrate the usefulness of prefix notation in representing algebraic expressions.

The differentiation rules which we will be implementing are:

$$\frac{dx}{dx} = 1 \qquad \text{(rule 1)}$$

$$\frac{dy}{dx} = 0, (y \neq x) \qquad \text{(rule 2)}$$

$$\frac{d(u + v)}{dx} = \frac{du}{dx} + \frac{dv}{dx} \qquad \text{(rule 3)}$$

$$\frac{d(u \cdot v)}{dx} = v\frac{du}{dx} + u\frac{dv}{dx} \qquad \text{(rule 4)}$$

The LISP function diff[e;x] is to differentiate the algebraic expression e with respect to the variable x. Clearly some way of representing algebraic expressions as S-expressions must be invented, since the arguments of diff must be S-expressions.

There are many possible ways to represent an algebraic expression as an S-expression, e.g., y + zw might become (Y PLUS Z TIMES W) or (Y SUM (Z PRDCT W)) or (PLUS Y (TIMES Z W)).

We shall choose this last representation (called Polish prefix notation) for reasons of convenience, and will specify it as follows:

1. Algebraic constants and variables shall become atoms, e.g.,

$$y \rightarrow Y,\ \rho \rightarrow RHO,\ 37 \rightarrow THIRTYSEVEN$$

(The programmed versions of LISP all handle numbers more conveniently than this!)

2. The operators + and . shall be limited to two operands by associative grouping, e.g.

$$x + y + z \rightarrow x + (y + z)$$
$$u \cdot v \cdot w \rightarrow u \cdot (v \cdot w)$$

3. We will use the notation convention that an expression followed by * means the translation of that expression. All expressions will be in <u>prefix</u> notation, i.e.

$$x + y \rightarrow (PLUS\ x^*\ y^*)$$
$$x \cdot y \rightarrow (TIMES\ x^*\ y^*)$$

The following examples should clarify these rules.

<u>Algebraic Expression</u>	<u>S-expression Representation</u>
a + b	(PLUS A B)
a . b	(TIMES A B)
a . (b + c)	(TIMES A (PLUS B C))
$\alpha + \beta + \gamma x$	(PLUS ALPHA (PLUS BETA (TIMES GAMMA X)))
$2\pi r$	(TIMES TWO (TIMES PI R))

The program for <u>diff</u> using this representation for algebraic expressions is straightforward:

```
diff = λ[[e;x];
[atom[e] → [eq[e;x] → ONE;                                  (Rule 1)
            T       → ZERO];                                (Rule 2)
eq[car[e];PLUS] → list[PLUS;diff[cadr[e];x];                (Rule 3)
   diff[caddr[e];x]]
eq[car[e];TIMES] → list[PLUS;                               (Rule 4)
      list[TIMES;caddr[e];diff[cadr[e];x]]
      list[TIMES;cadr[e];diff[caddr[e];x]]]]]
```

<u>Exercises</u>. Evaluate these forms:

```
179.  diff[(PLUS A X);X]
180.  diff[(TIMES A X);X]
181.  diff[(TIMES THREE X);Y]
182.  diff[(PLUS Y Y);Y]
```

183. diff[(TIMES TWO (TIMES Z Z));Z]

Adding the following translation rules for converting algebraic expressions to S-expressions allows us to add new rules to increase the power of diff.

Algebra	S-expression
-x	(MINUS x*)
1/x	(RECIP x*)
x	
sin[x]	(SIN x*)
cos[x]	(COS x*)

Additional clauses can be added to the main conditional expression of diff which implement these additional rules.

Example:

$\frac{d(-u)}{dx} = -(\frac{du}{dx})$ is implemented by adding this clause to diff:

eq[car[e];MINUS] → list[MINUS;diff[cadr[e];x]]

Exercise.

Implement the following differentiation rules by writing a clause for diff to handle each:

184. $\frac{d(\frac{1}{u})}{dx} = \frac{\frac{du}{dx}}{u \cdot u}$

185. $\frac{d(\sin u)}{dx} = (\cos u) \cdot (\frac{du}{dx})$

186. $\frac{d(\cos u)}{dx} = -(\sin u) \cdot (\frac{du}{dx})$

N. S-expression Representation of LISP Expressions. Read Section 1.6.

Exercises. Translate these M-expressions into S-expressions.

187. a
188. x
189. A
190. T
191. NIL
192. ((A B))
193. QUOTE
194. (QUOTE A)
195. car

196. car[x]
197. car[A]
198. atom[x]
199. ff[x]
200. ff[car[x]]
201. [atom[x]→ x;T→ ff[car[x]]]
202. λ[[x];x]
203. λ[[x];car[x]]
204. λ[[x];[atom[x]→ x;T→ ff[car[x]]]
205. label[ff;λ[[x];[atom[x]→ x;T→ ff[car[x]]]
206. λ[[x];λ[[y]; car[y]][x]]

O. Table Building and Searching.

Evaluate the following forms. Use the definitions of pairlis and assoc given in Section 1.6, changing the occurrence of equal in assoc to eq.

207. pairlis[NIL;NIL;((Y . B))]
208. pairlis[(X);(A);NIL]
209. pairlis[(X Y);(A B);((Z . C))]
210. assoc[Y;((X . A) (Y . B) (Z . C))]
211. assoc[X;pairlis[(X);(A);((Z . C))]]

P. A LISP Interpreter

Using the Section 1.6 definitions of apply, eval, pairlis, evcon and evlis and with a = ((X . M) (Y . T) (Z . (M N)) (T . T)) evaluate the following.

212. assoc[Z;a]
213. assoc[Y;a]
214. eval[(QUOTE A);a]
215. eval[T;a]
216. eval[X;a]
217. evlis[(X);a]
218. evlis[(X Y Z);a]
219. evcon[((T X));a]
220. eval[(COND (T X));a]
221. apply[CAR;((A));a]
222. apply[CONS;(A B);a]
223. apply[CONS;((A) (B));( )]
224. apply[CAR;(A);( )]
225. eval[(ATOM X);a]
226. evcon[(((ATOM X) X) (T (FF (CAR X))));a]
227. eval[(COND ((ATOM X) X) (T (FF (CAR X))));a]
228. apply[(LAMBDA (X) (CAR X));((A));( )]
229. apply[(LABEL GARP (LAMBDA (X) (CAR X)));((A));( )]

230. apply[FIRST;((A));((FIRST . CAR))]
231. apply[(LAMBDA (X) (COND ((ATOM X) X) (T (FF (CAR X)))));
(A);( )]
232. apply[(LABEL FF (LAMBDA (X) (COND ((ATOM X) X) (T (FF
(CAR X))))));(A);( )]
233. apply[(LABEL FF (LAMBDA (X) (COND ((ATOM X) X) (T (FF
(CAR X))))));(((Q . R) (S . T)));( )]

Q. Fifteen Minute Problems

Write the LISP functions described below. You need not redefine any functions you use which appeared in an earlier exercise. Assume that integers are atomic symbols.

234. infix[polish;table] is to convert an expression from Polish to infix notation, using a table which gives the correspondence between prefix and infix operators.

e.g. infix[(TIMES 3 A B);((PLUS . +) (TIMES . X))] =
(3 X A X B)
infix[(DIVIDE (TIMES$\alpha\beta$) (PLUS 3 (TIMESα 2) A B));
((PLUS . +) (TIMES . X) (DIVIDE . /))] =
((α X α X β)/(3 + (α X 2) + A + B))

235. polish[infix;table] is to do the corresponding reverse transformation.

236. permut[string] is to produce a list of all the permutations of a given string, assuming that all the elements of the original string are unique.

e.g. permut[(A B C)] = ((A B C) (A C B) (B A C) (B C A)
(C A B) (C B A))

237. permutations[string] is to produce a list of all the permutations of a given string, whether or not the elements are unique. Note: It is difficult to do this in an efficient way.

e.g. permutations[(A A B)] = ((A A B) (A B A) (B A A))

238. Invent the necessary S-expression representation and write clauses for diff to implement Leibnitz' rule:

$$\frac{d}{d\alpha}\int_{a(\alpha)}^{b(\alpha)} f(x,\alpha)dx = \int_{a}^{b}\frac{df}{d\alpha}dx + f(b,\alpha)\frac{db}{d\alpha} - f(a,\alpha)\frac{da}{d\alpha}$$

239. Given the predicate greater[x;y] which orders atomic symbols, i.e., if greater[x;y] = T, then greater[y;x] = F where x and y are any two different atoms, write the predicate larger [x;y] which orders S-expressions, i.e., if larger[x;y] = T, then larger[y;x] = F.

240. One S-expression is a <u>factor</u> of another S-expression if the second includes the first as a sub-expression. An S-expression equivalent to a greatest common divisor is a common sub-expression which is not a factor of any other common sub-expression. (It is not necessarily unique.) Write functions to find a greatest common divisor of two S-expressions.

e.g. gcd[(A . ((B . C) . D));(E . (B . C))] = (B . C)

Write functions to find all the g.c.d.'s of two S-expressions.

e.g. allgcds[(X . (Y . (A . B)));(B . (X . (A . B)))] =
((A . B) X)

ANSWERS

1. yes
2. no
3. yes
4. no
5. yes
6. A
7. B
8. (FOO . CROCK)
9. GLITCH
10. undefined
11. (POOT . TOOP)
12. (X . (Y . Z))
13. T
14. F
15. undefined
16. T
17. F
18. T
19. A
20. (A . B)
21. B
22. F
23. T
24. T
25. F
26. T
27. B
28. true
29. true
30. false
31. true
32. false
33. false
34. true
35. A
36. ((PLOOP . NIL) . (FLOP . (TOP . NIL)))
37. ((X . (GLITCH . NIL)) . ((Y . (CROCK . NIL)) . NIL))
38. (((X . NIL) . NIL) . NIL)
39. (SNAP . ((CRACKLE . ((POP . NIL) . NIL)) . NIL))
40. ((A) (B))
41. (A B C)
42. (NIL)
43. ((A B) (C))
44. not list
45. A
46. (B C)
47. AB
48. A
49. NIL
50. undefined
51. undefined
52. (A B)
53. (A B C)
54. (A . B)
55. (A)
56. (NIL . A)
57. ((A) B C)
58. ((A B) C D)
59. ((A B) . C)
60. ((A . B) (C . D) (E . F))
61. ((A B))
62. ((A))
63. (T F T F)
64. T
65. car[x]
66. cadr[x]
67. cdr[x]
68. caddr[x]
69. cadddr[x]
70. caar[x]
71. cdar[x]
72. caaadr[x]
73. caadr[x]
74. caaaar[x]
75. F
76. T
77. T
78. F
79. T
80. T
81. (A B C)
82. ((A) (B) (C))
83. T
84. F
85. cons[A;B]
86. list[X;Y]
87. list[list[A;B];list[C;D]]
88. list[cons[A;B];cons[C;D]]

```
89.  list[list[list[A]]]
90.  list[cons[B;C]]
91.  list[list[ ];list[ ]]
92.  list[A;B;C;D]
93.  F
94.  T
95.  F
96.  T
97.  A
98.  B
99.  C
100. (C)
101. (B C)
102. B
103. C
104. NIL
105. undefined
106. T
107. B
108. C
109. F
110. T
111. (B . C)
112. T
113. T
114. F
115. undefined
116. A
117. T
118. undefined
119. A
120. B
121. undefined
122. A
123. NIL
124. T
125. (AB . C)
126. B
127. A
128. A
129. B
130. ((AB . C))
131. T
132. undefined
133. C
134. A
135. (A)
136. A
137. A
138. A
139. A
140. B
141. B
142. (C . B)
143. A
144. S
145. (A . C)
146. (A . B)
147. (A . C)
148. T
149. F
150. F
151. F
152. T
153. T
154. jj[A] = A
155. jj[(A . B)] = jj[B] = B
156. jj[((X . Y) . (X . Z))]
       = jj[(X . Z)] = jj[Z]
       = Z
157. jj[(A B C)] = jj[(B C)]
       = jj[(C)] = jj[NIL] =
       NIL
158. jj[(A (C . E))] =
       jj[((C . E))] =
       jj[NIL] = NIL
159. X
160. E
161. NO
162. A
163. (B . A)
164. (C . (B . A))
165. (((NIL . C) . B) . A)
166. (NIL . (B . A))
167. member = λ[[x;m];null[m]→
       F;eq[x;car[m]]→ T;T →
       member[x;cdr[m]]]]
168. pair = λ[[m;n];[null[m]→
       NIL;T→ cons[cons[car[m];
       car[n]];pair[cdr[m];
       cdr[n]]]]]
169. append = λ[[m;n];[null[m]
       → n;T→ cons[car[m];
       append[cdr[m];n]]]]
170. delete = λ[[x;m];[null[m]
       → NIL;eq[x;car[m]]→
```

```
      cdr[m];T→ cons[car[m];delete[x;cdr[m]]]]]
171.  reverse = λ[[m];[null[m]→ NIL;T→ append[reverse[cdr[m]];
      list[car[m]]]]]
172.  union = λ[[m;n];[null[m]→ n;member[car[m];n]→ union[cdr[m];
      n];T→ cons[car[m];union[cdr[m];n]]]]
173.  intersection = λ[[m;n];[null[m]→ NIL;member[car[m];n]→
      cons[car[m];intersection[cdr[m];n]];T→ intersection[cdr[m];
      n]]]
174.  last = λ[[m];[null[cdr[m]]→ car[m];T→ last[cdr[m]]]]
175.  superreverse = λ[[m];atom[m]→ m;null[m]→ NIL;T→
      append[superreverse[cdr[m]];list[superreverse[car[m]]]]]]
176.  part = λ[[x;s];[atom[s]→ eq[x;s];part[x;car[s]]→ T;T→
      part[x;cdr[s]]]]
177.  subst = λ[[x;y;s];[atom[s]→ [eq[y;s]→ x;T→ s];T→
      cons[subst[x;y;car[s]];subst[x;y;cdr[s]]]]]
178.  listofatoms = λ[[s];[atom[s]→ list[s];T→
      union[listofatoms[car[s]];listofatoms[cdr[s]]]]]
179.  (PLUS ZERO ONE)
180.  (PLUS (TIMES X ZERO) (TIMES A ONE))
181.  (PLUS (TIMES X ZERO) (TIMES THREE ZERO))
182.  (PLUS ONE ONE)
183.  (PLUS (TIMES (TIMES Z Z) ZERO) (TIMES TWO (PLUS (TIMES Z ONE)
      (TIMES Z ONE))))
184.  eq[car[e];RECIP]→ list[TIMES;list[RECIP;list[TIMES;cadr[e];
      cadr[e]]];diff[cadr[e];x]]
185.  eq[car[e];SIN]→ list[TIMES;list[COS;cadr[e]];diff[cadr[e];
      x]]
186.  eq[car[e];COS]→ list[MINUS;list[TIMES;list[SIN;cadr[e]];
      diff[cadr[e];x]]]
187.  A
188.  X
189.  (QUOTE A)
190.  (QUOTE T)
191.  (QUOTE NIL)
192.  (QUOTE ((A B)))
193.  (QUOTE QUOTE)
194.  (QUOTE (QUOTE A))
195.  CAR
196.  (CAR X)
197.  (CAR (QUOTE A))
198.  (ATOM X)
199.  (FF X)
200.  (FF (CAR X))
201.  (COND ((ATOM X) X) (T (FF (CAR X))))
202.  (LAMBDA (X) X)
203.  (LAMBDA (X) (CAR X))
204.  (LAMBDA (X) (COND ((ATOM X) X) (T (FF (CAR X)))))
```

```
205.  (LABEL FF (LAMBDA (X) (COND ((ATOM X) X) (T (FF (CAR X))))))
206.  (LAMBDA (X) ((LAMBDA (Y) (CAR Y)) X))
207.  ((Y . B))
208.  ((X . A))
209.  ((X . A) (Y . B) (Z . C))
210.  (Y . B)
211.  (X . A)
212.  (Z . (M N))
213.  (Y . T)
214.  A
215.  T
216.  M
217.  (M)
218.  (M T (M N))
219.  M
220.  M
221.  A
222.  (A . B)
223.  ((A) . (B)) = ((A) B)
224.  undefined
225.  T
226.  M
227.  M
228.  A
229.  A
230.  A
231.  A
232.  A
233.  Q
```

Notes on the Debugging of LISP Programs

Elaine Gord

Information International, Inc.

The purpose of these brief remarks is to provide some guidance to those persons whose programs expressed in LISP do not run on the computer. Most of the suggestions here stated were learned the hard way, from experience.

Parentheses

The most frequently occurring errors in LISP are parenthetical errors. It is thus almost imperative to employ some sort of counting or pairing device to check parentheses every time that a function is changed.[1] The spelling of words and the possible confusion of the numbers 1 (one) and 0 (zero) and the letters I (i) and O (o) respectively can be checked at the same time. When the program is to be run, any number of the functions being defined can be put between the two punch cards DEFINE((and)). An unpaired parenthesis can be found most quickly, however, by putting the functions concerned within separate "DEFINE" statements. No read-in error will occur until the parentheses do not pair. It is however necessary but not sufficient that there be the same number of left and right parentheses, because pairs of unnecessary parentheses may be present or pairs of necessary ones absent.

Parentheses must be correctly located. A left parenthesis must appear immediately to the left of a function name, when the function is applied to arguments. For example (CAR CDR X) is incorrect because there should be a left parentheses immediately to the left of CDR (and a matching right parentheses after X), that is, it should be (CAR (CDR X)). The expression (CAR (X)) is incorrect because the left parentheses next to X indicates incorrectly to the LISP interpreter that X is a function name. The correct expression is (CAR X). When a function name such as CDR is meant to be an argument, it should be expressed in the form (FUNCTION CDR).

Functions and Arguments

For each appearance of a function, the number and order of its arguments should be checked. In contrast with forms, which have an indefinite number of arguments, each function has a fixed number of arguments. The order of arguments may not be important for a few functions (such as EQUAL or INTERSECTION), but it is extremely important for most. MEMBER is a good example. Not only would the answer be incorrect if the order of arguments were reversed; but if the proper first argument were an atom, the function could not be evaluated.

It is clear that each argument must be of the right kind. This is most obvious when the distinction is between lists and atomic symbols. If a list is expected, the argument should not be atomic. If an atomic symbol or a number is expected, the argument should not be a list. Also, there are different kinds of lists, and some functions demand certain types of lists as arguments. For example, a function that calls CAAR or CDAR presupposes a list containing lists as elements. If a function is meant to operate on a list of the form ((A . B) (C . D) (E . F)), it should not be given a list of the form (A B C D E).

List Making Functions

Among the list-making functions are CONS and APPEND. The type of output desired will determine the function chosen since each handles parentheses and the value NIL in a different way. APPEND will not list NIL as a first element, but CONS will. In contrast to LIST, neither will list NIL as a last element. All of these functions are used in recursive functions in which the value of the terminating condition is NIL. Any terminating value other than NIL will introduce dot notation, except in the case of the function LIST.

Terminating Conditions

Terminating conditions are almost always the most important aspects of recursive functions. They must be of the right kind, they must terminate in the right place, and they must cover all possibilities. Often several terminating conditions are needed. Output containing expressions such as PNAME or many NILs often indicates that the function did not terminate correctly.

No Duplicate Names

In addition to checking parentheses, arguments, and conditions for each function, it is a good idea to make sure that the name of the function has not already been used for a function or form the system tape, expecially if the two functions do not have the same purpose. In effect, changing the definition might make it impossible for other functions to work. For example, since DEFI is called by DEFINE, if a different DEFI is introduced the DEFINE will no longer work correctly.

After a program has been run, errors can often be localized by noting the offending value and those functions in the backtrace following the error diagnostic which have been entered but not completed.

---

[1]Information International Inc. has a program for the PDP-1 which counts parentheses as follows: The count, which appears in red on the listing beside each parenthesis, begins with a zero at the first left parenthesis and is indexed for subsequent left parentheses until a right parenthsis occurs. For the first right parenthesis the count retains the value of the preceding left parenthesis and is decreased by one for succeeding right parentheses until a left parenthesis occurs, for which the count remains the same, and so on. The count at the final parenthesis of an expression should, then, be zero. If an error is indicated, it can be found by matching numbered pairs.

Styles of Programming in LISP

Fischer Black

Bolt Beranek and Newman, Inc.
Cambridge, Mass.

When programmer A writes a program that compiles into 10 instructions and takes 1 minute to run, and programmer B writes a program to do the same job that compiles into 100 instructions and takes 10 minutes to run, we say that programmer A is better than programmer B. But when programmers A and B write programs that compile into about the same number of instructions, and have about the same running time, but look very different, we say that programmer A has a different style from that of programmer B.

Programming style is not a matter of efficiency in a program. It is a matter of how easy it is to write or read a program, how easy it is to explain the program to someone else, how easy it is to figure out what the program does a year after you've written it; and above all, style is a matter of taste, of aesthetics, of what you think looks nice, of what you think is elegant.

Although style is mainly a matter of taste, a programmer with a "good" style will find his programs easy to write, easy to read, and easy to explain to others. If you prefer a complicated style, you may be penalizing yourself in the long run. What Charles Horton Cooley said about fiction applies to programming too: "An elaborate style has perils like those of an elaborate house. It commits the author, by habit and the expectation of others, to a special and costly way of living, which will become a burden when he loses the wish or the power to keep it up."

In particular, you may have acquired special programming tricks that you are very fond of, and that aren't used by other programmers, but that don't make your programs much more efficient. I urge you to stop using those tricks. As Samuel Johnson once said, "Read over your compositions, and when you meet with a passage which you think is particularly fine, strike it out."

In other words, make your style simple, not complicated, even though the complicated style may seem to have some abstract virtues.

Your style is developed by a series of choices. Which do you write, "()" or "NIL"? They compile into identical data structures; so choosing one over the other is a matter of style. Do you put commas in your lists, or do you separate the items with spaces? The lists will look the same inside the machine whether you put the commas in or leave them out; so it's not a matter of efficiency, it's a matter of style. If you had to write the value of "cons[(A B);(C D)]", which way would you write it:

```
((A B) . (C D))
((A B) C D)
```

And which of these expressions would you write:

```
eq[x; NIL]
null[x]
```

All of these choices contribute to your style.

Choice of the Program Feature or Functionals

To help in the developing of a good style, I am going to discuss two of the more complex choices you can make in LISP. (1) Should you use the program feature? (2) Should you use functionals?

I'm not saying you should always use the program feature, and I'm not saying you should never use it. But it is desirable to make the choice consciously in each case, without feeling that it must be written one way or the other. I happen to favor using the program feature and not using functionals, but the choices are up to you.

The program feature is not given much attention in the manual; the first mention of it is on page 29. But any function that can be written without the program feature is a function that can be written with the program feature. A trivial way is to take a

function defined as "λ[[x]; ---]" and rewrite it as "λ[[x]; prog[[]; return[---]]]". The resulting function will be equivalent to the original. The reverse is also true — any function written with the program feature can be written without.

I will give below some examples of functions written with and without the program feature, and then some examples of functions written with and without functionals. The final examples illustrate changes in both aspects of style at the same time. In the appendix, I have included S-expression translations of the first 5 examples.

You may use the program feature or you may not use it; you may use functionals or you may not use them. The examples below are to help you choose your style.

MEMBER

For illustration, let us take the function "member". It is built into the LISP system, and is described on pages 11 and 62 of the manual. A couple of examples may make it clearer:

```
member[C; (A B C)]= *T*
member[D; (A B C)]= NIL
```

Below I have written "member" using the program feature, and for comparison have put in the definition given in the manual.

Example 1

```
member[x;y]= prog[[];
A       [null[y]→ return[F];
        equal[x; car[y]]→ return[T]];
        y:= cdr[y];
        go[A]]
```

Example 2

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

FACTORIAL

The function "factorial" is not built into the LISP system in the manual. As a numerical function it has values like

```
factorial[1]= 1
factorial[2]= 2
factorial[3]= 6
factorial[4]= 24
     ....
```

Writing "factorial" using the program feature is certainly longer than writing it without, but this is a clear example of the difference in spirit between using the program feature and not using it.

Example 1

```
factorial[x]= prog[[y];
        y:= 1;
A       [onep[x]→ return[y]];
        y:= times [x; y];
        x:= subl[x];
        go[A]]
```

Example 2

```
factorial[x]=
        [onep[x]→ 1;
        T→ times[x; factorial[subl[x]]]]
```

REVERSE

An example of the use of the function "reverse" is given on page 62 of the manual. Here are two more:

```
reverse[(A B C)]= (C B A)
reverse[(A (B C))]= ((B C) A)
```

To write "reverse" without using the program feature, you must put in an extra function definition.

Example 1

```
reverse[x]=prog[[y];
A       [null[x]→ return[y]];
        y:= cons[car[x]; y];
        x:= cdr[x];
        go[A]]
```

Example 2

```
reverse[x]= reverse2[x; NIL]
```

```
reverse2[x; y]=
        [null[x]→ y;
        T→ reverse2[cdr[x]; cons[car[x]; y]]]
```

YDOT

An artificial but simple example of the use of functionals is the function "ydot" described on page 78 of the manual. The example of the application of the function given there is

```
ydot[(A B C D); X]= ((A . X) (B . X) (C . X) (D . X))
```

The function "ydot" is not a functional and is not supplied in the system, but the function "maplist", which can be used in defining "ydot", is a functional and is supplied in the system. I show below how "ydot" can be defined either using or not using the function "maplist". It's just a matter of style.

Example 1

```
ydot[x; y]=
        maplist[x; λ[[j]; cons[car[j]; y]]]
```

Example 2

```
ydot[x; y]=
        [null[x]→ NIL;
        T→ cons[cons[car[x]; y]; cons[cons[car[x]; y];
            ydot[cdr[x]; y]]]
```

SUBLIS

The function "sublis" is built into the LISP system, and can be illustrated by

```
sublis[((U . (PLUS X Y)) (V . (PLUS Z W)));
       (TIMES U V)]= (TIMES (PLUS X Y) (PLUS Z W))
```

Another illustration is given on page 12 of the manual. Two definitions of "sublis" are given in the manual, one on page 61 which uses the functional "search", and one on page 12 which doesn't use any functionals. I reproduce them both below.

Example 1

```
sublis[x; y]=
        [null[x]→ y;
        null[y]→ y;
```

```
T→ search[x;
     λ [[j]; equal[y; caar[j]]];
     λ [[j]; cdar[j]];
     λ [[j]; [atom[y]→ y;
             T→ cons[sublis[x; car[y]];
                     sublis[x; cdr[y]]]]]]]
```

Example 2

```
sublis[a; y]=
       [atom[y]→ sub2[a; y];
       T→ cons[sublis[a; car[y]]; sublis[a; cdr[y]]]]
sub2[a; z]=
       [null[a]→ z;
       eq[caar[a]; z]→ cdar[a];
       T→ sub2[cdr[a]; z]]
```

PRINTPROP

The function "printprop" is supplied in the LISP system and is described on page 68 of the manual. To illustrate its use, suppose you have executed

```
deflist[((REQUEST (PASS MUFFINS))); VALUE]
```

which puts "(PASS MUFFINS)" on the property list of "REQUEST" under the indicator "VALUE". Then the instruction

```
printprop[REQUEST]
```

will cause the following to be printed.

```
(PROPERTIES OF REQUEST)
VALUE
(PASS MUFFINS)
```

If you're puzzled by the extra "cdr[k]" in the examples below, don't forget that "car" of an atom is always -1 (that's how you know it's an atom), so "cdr" of an atom is its property list.

The first example uses a functional but not the program feature; the second uses the program feature but no functionals.

Example 1

```
printprop[k]=
      prog2[print[list[PROPERTIES; OF; k]];
           printpl[cdr[k]]]
```

```
printpl[k]=
        [null[k]→ NIL;
        T→ prog2[print[car[k]];
                 search[(SUBR FSUBR PNAME SYM);
                        λ[[j]; equal[car[j]; car[k]]];
                        λ[[j]; printpl[cddr[k]]];
                        λ[[j]; printpl [cdr[k]]]]]]]
```

Example 2

```
printprop[k]= prog[[m];
        print[list[PROPERTIES; OF; k]];
        k:= cdr[k];
A       [null[k]→ return[NIL]];
        m:= car[k];
        k:= cdr[k];
        print[m];
        [member[m; (SUBR FSUBR PNAME SYM)]→
                 k:= cdr[k]];
        go[A]]
```

Differentiating

The next function diff is used to differentiate algebraic functions formed from variables by addition and multiplication. A sample expression is

```
(TIMES (PLUS X Y Z) (PLUS U Y))
```

The derivative that the function gives is not in the form a person would write — for example, the derivative of "(PLUS X Y)" with respect to "X" is given by

```
diff[(PLUS X Y); X]= (PLUS 1.0 0.0)
```

The first version of "diff" is written using the functional "maplist" but not the program feature. The second version uses the program feature but not "maplist". If you can't figure out one version, try the other.

Example 1

```
diff[s; v]=
        [atom[s]→ [eq[s; v]→ 1.0; T→ 0.0];
        eq[car[s]; PLUS]→ cons[PLUS;
             maplist[cdr[s]; λ[[si]; diff[car[si]; v]]]];
        eq[car[s]; TIMES]→ cons[PLUS;
             maplist[cdr[s]; λ[[si];
```

```
        cons[TIMES; maplist[cdr[s];
        λ[[sj]; [equal[si; sj]→ diff[car[sj]; v]
                T→  car[sj]]]]]]]]]
```

Example 2

```
diff[s; v]=
        [atom[s]→ [eq[s; v]→ 1.0; T→ 0.0];
        eq[car[s]; PLUS]→ cons[PLUS; diffp[cdr[s]; v]];
        eq[car[s]; TIMES]→ cons[PLUS; difft[cdr[s]; v]]]
diffp[s; v]= prog[[m];
A       [null[s]→ return[reverse[m]]];
        m:= cons[diff[car[s]; v]; m];
        s:= cdr[s];
        go [A]]
difft[s; v]= prog[[m; k; r];
A       [null[s]→ return[reverse[k]]];
        m:= conc[reverse[r]; list[diff[car[s]; v]]; cdr[s]];
        k:= cons[cons[TIMES; m]; k];
        r:= cons[car[s]; r];
        s:= cdr[s];
        go[A]]
```

PAIRMAP

The functional "pairmap" is used in the LISP compiler. It applies its functional argument to pairs of items from two lists of the same length, and then attaches its last argument to the end of a list of the resulting values. By way of illustration,

```
pairmap[(A B); (X Y); cons; ((C . Z))]=
        ((A . X) (B . Y) (C . Z))
```

I'm giving examples of two ways of writing "pairmap" partly to show a subtle difference in style, and partly because "pairmap" can be used in defining the main function of my last example, "progiter". Both examples of "pairmap" use the program feature, and neither uses functionals, though a functional is being defined.

Example 1

```
pairmap[k; m; farg; z]= prog[[a; b];
        [null[k]→ return[z]];
        a:= b:= cons[farg[car[k]; car[m]]; z];
A       k:= cdr[k];
        m:= cdr[m];
        [null[k]→ return[a]];
```

```
        b:= cdr[rplacd[b; cons[farg[car[k]; car[m]]; z]]];
        go[A]]
```

Example 2

```
pairmap[k; m; farg; z]= prog[[b];
A       [null[k]→ return[conc[reverse[b]; z]]];
        b:= cons[farg[car[k]; car[m]]; b];
        k:= cdr[k];
        m:= cdr[m];
        go[A]]
```

PROGITER

The function "progiter" is used in the LISP compiler to convert a function definition that doesn't use the program feature into a definition that does use the program feature and will run faster when compiled than the original definition. You won't be able to tell exactly how "progiter" works from the examples, because I haven't included definitions of the functions "pil" and "pi3". But you can follow the transformation from the original version of "progiter", through an intermediate version that uses the program feature, to a final version that uses the program feature but does not use the functionals "pairmap" or "maplist". To avoid using the functionals I have defined two subfunctions, "pairmapl" and "gensyml". "Pairmapl" is like "pairmap" with a specific function "pi2" put for its functional argument. "Gensyml" generates a list of new atomic symbols.

Example 1

```
progiter[name; exp]=
        [eq[caaddr[exp]; COND]∧pil[cdaddr[exp]]→
             λ[[g1; g2; vs; gs];
             list[LAMBDA; vs;
             cons[PROG; cons[gs; cons[g1;
             pi3[cdaddr[exp]; NIL; cons[g2;
             pairmap[vs; gs; pi2; list[list[GO; g1]]]]]]]]]]
                [gensym[]; gensym[]; cadr[exp];
                maplist[cadr[exp]; gensym]];
        T→ exp]
pi2[j; k]= list[SETQ; j; k]
```

Example 2

```
progiter[name; exp]= prog[[g1; g2; vs; gs; x];
        [~[eq[caaddr[exp]; COND]∧pil[cdaddr[exp]]]→
              return[exp]];
```

```
        g1:= gensym[];
        g2:= gensym[];
        vs:= cadr[exp];
        gs:= maplist[cadr[exp]; gensym];
        x:= list[list[GO; g1]];
        x:= pairmap[vs; gs; pi2; x];
        x:= pi3[cdaddr[exp]; NIL; cons[g2; x]];
        x:= cons[PROG; cons[gs; cons[g1; x]]];
        return[list[LAMBDA; vs; x]]]
pi2[j; k]= list[SETQ; j; k]
```

Example 3

```
progiter[name; exp]= prog[[u; g1; g2; vs; gs; x];
        u:= eq[caaddr[exp]; COND] ∧pil[cdaddr[exp]];
        [- u→ return[exp]];
        g1:= gensym[];
        g2:= gensym[];
        vs:= cadr[exp];
        gs:= gensym1[length[vs]];
        x:= list[list[GO; g1]];
        x:= pairmap1[vs;gs;x];
        x:= pi3[cdaddr[exp]; NIL; cons[g2; x]];
        x:= cons[PROG; cons[gs; cons[g1; x]]];
        return[list[LAMBDA; vs; x]]]
gensym1[x]= prog[[y];
A       [zerop[x]→ return[y]];
        y:= cons[gensym[]; y];
        x:= sub1[x];
        go[A]]
pairmap1[l; m; a]= prog[[b];
A       [null[l]→ return[conc[reverse[b]; z]]];
        b:= cons[list[SETQ; car[l]; car[m]]; b];
        l:= cdr[l];
        m:= cdr[m];
        go[A]]
```

Appendix

S-EXPRESSION TRANSLATIONS OF FIVE EXAMPLES

Member

```
(MEMBER (LAMBDA (X Y) (PROG ()
A       (COND ((NULL Y) (RETURN F))
        ((EQUAL X (CAR Y)) (RETURN T)))
        (SETQ Y (CDR Y))
        (GO A))))
(MEMBER (LAMBDA (X Y) (COND
        ((NULL Y) F)
        ((EQUAL X (CAR Y)) T)
        (T (MEMBER X (CDR Y))))))
```

Factorial

```
(FACTORIAL (LAMBDA (X) (PROG (Y)
        (SETQ Y 1)
A       (COND ((ONEP X) (RETURN Y)))
        (SETQ Y (TIMES X Y))
        (SETQ X (SUB1 X))
        (GO A))))
(FACTORIAL (LAMBDA (X) (COND
        ((ONEP X) 1)
        (T (TIMES X (FACTORIAL (SUB1 X)))))))
```

Reverse

```
(REVERSE (LAMBDA (X) (PROG (Y)
A       (COND ((NULL X) (RETURN Y)))
        (SETQ Y (CONS (CAR X) Y))
        (SETQ X (CDR X))
        (GO A))))
(REVERSE (LAMBDA (X) (REVERSE1 X NIL)))
(REVERSE1 (LAMBDA (X Y) (COND
        ((NULL X) Y)
        (T (REVERSE1 (CDR X) (CONS (CAR X) (CONS (CAR X) Y)))))))
```

YDot

```
(YDOT (LAMBDA (X Y)
        (MAPLIST X (FUNCTION (LAMBDA (J)
```

```
        (CONS (CAR J) Y))))))
(YDOT (LAMBDA (X Y) (COND
        ((NULL X) NIL)
        (T (CONS (CONS (CAR X) Y) (YDOT (CDR X) Y))))))
```

Sublis

```
(SUBLIS (LAMBDA (X Y) (COND
        ((NULL X) Y)
        ((NULL Y) Y)
        (T (SEARCH X
        (FUNCTION (LAMBDA (J) (EQUAL Y (CAAR J))))
        (FUNCTION (LAMBDA (J) (CDAR J)))
        (FUNCTION (LAMBDA (J) (COND
        ((ATOM Y) Y)
        (T (CONS (SUBLIS X (CAR Y))
                 (SUBLIS X (CDR Y)))))))))))))
(SUBLIS (LAMBDA (A Y) (COND
        ((ATOM Y) (SUB2 A Y))
        (T (CONS (SUBLIS A (CAR Y)) (SUBLIS A (CDR Y)))))))
(SUB2 (LAMBDA (A Z) (COND
        ((NULL A) Z)
        ((EQ (CAAR A) Z) (CDAR A))
        (T (SUB2 (CDR A) Z)))))
```

Techniques Using LISP for Automatically Discovering Interesting Relations in Data

Edward Fredkin

Information International, Inc.

(Note: This paper, which was written June 1963, should be considered in the nature of a prelude to the paper by Malcolm Pivar, written in December 1963, which reports on further development of the ideas discussed here.)

The utility of a computer program that could automatically "discover" interesting features of a large mass of data, would unquestionably be great. Basically, that which one wants to know is interesting — that which one does not want to know is uninteresting. The separation — and elucidation — of the relations involved is a problem of great complexity. Undoubtedly the complete solution to the problem — even in the restricted context of data reduction and analysis — will await the creation of a vast and very well programmed computing facility.

In the meantime, a somewhat less omni-analytic machine would be a most useful aid not only to the experimenter but, more generally, to anyone who had to quickly see and comprehend the important relations to be found in a continual stream of incoming data. A classical example is the Field of Battle, where a commander must exercise control on the basis of incoming raw data.

A Model Situation

A feasible intermediate goal, however, would be the creation of a computer program that would operate on numerical data, characterize the data in some appropriate fashion, automatically discover interesting relationships in the data, and inform the user of these relationships.

Initially, such a program would be limited to simple situations and simple forms of data. However, with time, experience and effort, the capabilities of such a program could be developed to the point where it would represent a significant advance in the current state of computer technology.

What is "Interesting"?

The first step in this development process is the definition of "interesting" in a way sufficiently objective, precise, and simple to allow the formulation of an algorithmic definition. Such a definition, in turn, serves as the basis of a computer program capable of distinguishing between "interesting" and "uninteresting" data.

In expressing such a definition, however, it is desirable to keep in mind the context or background in relation to which an individual datum is to be considered "interesting" or "uninteresting". For example:

(i) A penguin in Antarctica is not interesting
(ii) A penguin in a zoo is not interesting
(iii) A zoo in Boston is not interesting
(iv) A penguin in Boston and not in a zoo is, however, somewhat interesting.

Thus a penguin may be interesting or not according to context.

Assume that the context that we are considering is a continual stream of numbers. The fact that these are numbers will, in itself, be uninteresting. The only possible point of interest is the characteristics of the numbers themselves. For example, in the following sample of a sequence:

....., 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, ...

the only interesting statement is "the sequence consists of a series of sevens". It is not, for example, interesting to say

"...the 6359th integer is a seven, the 6360th integer is a seven, the 6361st integer is a seven, the 6362nd integer is a seven..."

In general, then, we may assume that the interesting relations that one finds in data are implicit in the shortest description of the data. Both statements above describe the original data; however, one is more concise than the other.

The general objective in differentiating interesting information from uninteresting information is to be able to retain the interesting information in memory and dispense with the uninteresting information. Anyone who has used computers knows of the great ease with which a computer remembers arbitrarily long list of random numbers (on magnetic tape, for example). However, the human mind, which performs so many feats of memory and recall that are beyond the capability of any present computer system, can remember only a very short list of random numbers, say, seven; and then only for a short while. To remember a longer list for a longer period of time, the numbers must be made interesting, by building an associative memory chain involving the numbers in some other information.

At present, then, the human mind remembers information selectively. Since its information storage capacity is, probably, limited, it does not remember all data. It tends to remember, basically, only information which is "interesting", either because it promoted survival*[1], because it avoids discomfort (e.g., remembering that bees can sting), promotes pleasure, or for other similar reasons. Thus, in the above sense, we may be said to analyze information which we perceive into "interesting" and "uninteresting" categories for purposes of selective information storage.

What we perceive may be represented in many different forms. In the series:

....., 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, ...

we could remember the actual visual pattern created by the dots, the angular lines (the sevens), the curved lines (the commas), etc. The visual pattern would require, perhaps, 50,000 bits of information. But if we abstract the various lines and dots into characters, with an eight-bit code for each character, then a total of 512 bits would be sufficient. However, this type of encoding would require the knowledge of what characters, periods, commas, and sevens are. The encoding of information required to recognize such characters would undoubtedly add more than 50,000 bits to the 512 required for encoding of the characters. It is not necessary, of course, to write off all of the investment in

the knowledge and recognition of characters on this one case. For example, a reasonable allocation of the total requirement to this specific case might be only 75 bits (in addition to the 512 bits required for encoding of the characters themselves).

The essence, then, of the above pattern is that it is an arbitrarily long sequence of sevens. (The purpose of the periods to the right and to the left of the sequence is to indicate that the sequence continues indefinitely.) If we code the original sequence as follows:

sequence of "7"

This would require approximately 128 bits. In addition, the fair share of applicable overhead, as discussed above, might be approximately 100 bits.

Cost

In general, since there is a cost associated with remembering things, we should only remember what is worth the cost. For example, it is necessary to be very sophisticated about the amount of investment that is made on schemes to re-encode more efficiently, since it requires storage space to remember the re-encoding scheme itself.

In addition, it is necessary to note that what is remembered is both a collection of the memories of individual objects and abstractions and, in addition, the relationships between these objects and their properties. "Bees exist" is not usually an important fact; "Bees sting and stings hurt" <u>is</u> usually an important fact. The relations are perhaps much more numerous than the items so related, and the storage scheme used may take this into account.

A way to approach the difficult problem of discovering interesting relations in data is to invent a model situation that has most of the properties of the general situation yet which has a clearer and simpler solution to the problem. In addition we would like the model situation to have sufficient capacity to become more complex and general so as to lead to the development of useful applications and general principles.

The Sequence of Integers

Let us consider finite sequences of integers. Specifically, let us first consider what may be coded as finite sequences of

integers. By numbering the letters and other marks used in written communications, we can encode, for example, English text into finite sequences of integers.

This is commonly done in computer systems. There is for example the IBM Binary Coded Decimal, or BCD, standard encoding scheme. On the Model 33 teletypewriter, a standard code is used to represent the characters found on the keyboard, and the necessary controls; this code consists of seven bits, plus one additional parity bit making 8 bits. Thus we can represent written matter by a sequence of integers, each integer representing one letter or other mark.

Consider the output of an analog-to-digital converter: It consists of a sequence of integers. If we pass human speech into such a device, the speech is encoded into a sequence of integers. The same may be done, for example, with the output of a television show. The process may be inefficient, but it is highly general. For example, one can imagine encoding the successive states of the universe as a sequence of integers, each N-tuple, describing the type, position, velocity, time, and any other desired parameters for every fundamental particle in the universe. Storing the resulting sequence could not be done in this universe, however, unless it was encoded (as it is) into the actual situation.

Simple Sequences: Segmentation

One of the most crucial problems encountered in sequence analysis is that of segmentation. Given a sequence, how can it be segmented into a more meaningful (i.e., one better suited for coding) sequence of sequences? We can, with no loss in generality, consider the terms of a sequence to be elemental — i.e., not capable of being broken down. Thus, we will always build up from simpler sequences, never down.

Sequences will be allowed that consist of sequences of sequences, for example,

S35: S1, S2, S3, S4, .

In Backus normal form, we may define a sequence in the following manner

term::= <element> <,>
simple sequence::= <term> / <simple sequence> <term>
sequence::= <simple sequence> / <.> <sequence> /
<sequence> <.>

Operations on Sequences

We shall now consider a set of basic operations that can be performed on sequences.

(1) Local Operations. A local operation is defined as an operation involving a limited range of terms. Most important is the successor operation.

(2) The Successor Operation. If an operation on a term involves only that term and its successor (the next term in the sequence) we shall call it a successor operation. Probably the most interesting successor operation on the integers is the successive difference.

Let us adopt here a formal notation. The notation will be that of the LISP system developed by John McCarthy of Stanford University (see "LISP 1.5 Programmers Manual", August 17, 1962, M.I.T.).

In particular, we shall use M-expressions, since this type of expression is a most powerful method of describing the types of functions that we have in mind. In addition, it is a simple matter to transcribe LISP M-expressions into S-expressions, which are in a form ready to operate on a computer.

We shall aim to express the original sequence in a function which when evaluated for successive integers will result in the original sequence. The function will itself be a LISP expression and will be able to operate as a computer program to produce, as output, the original sequence, plus additional terms. For example, given the sequence

$$3, 5, 7, 9, 11, 13 \ldots$$

the program would operate on the sequence to produce a function, $Fn[n]$, equivalent to:

$$Fn[n] = [2 \cdot n + 1]$$

Thus, as n takes on the successive values of the integers, $Fn[n]$ would take on the successive values of the sequence.

The general plan of operation will be to use successive differences as an analytical tool.

Facts about Differences

If a sequence can be represented by a polynomial of degree n, then the n^{th} successive difference sequence is made up of terms all of which have the same value. That value is, in fact, a.n!, where a is the coefficient of the highest order term of the polynomial. The sequence of all n^{th} differences can be defined by a single element. For example, see the sequence of cubes n^3 and their differences shown in Table 1. The arrows "A⟶" indicate the direction of "differences. Summations would be in the other direction.

We can be sure that we understand the rule governing a sequence when our analysis has reduced it to a very simple form. For example, if the 3rd difference yields all constant terms, the sequence can be represented by a polynomial of degree 3.

In order to test a sequence to see if it is made up of terms all of which are the same, we can define the following LISP function:

P1[x]=null[x] ∨ P2[car[x]; cdr[x]]
P2[a;y]=null[y] ∨ [equal[a;car[y]] ∧ P2][a;cdr[y]]]

In the above expression, P1[x] represents a function of x, where x is a sequence, or list, of integers. If all terms in the sequence are equal, the function P1[x] will have the value T (true); otherwise it will have the value F (false).

null[x] is an expression which itself may have the value T (true) or F (false). It is "true" if the list, x, is empty, i.e., has no terms. It is "false" otherwise. For example, null[(1,2, 3)]=F; null[(7)]=F; null[( )]=T.

The symbol ∨ is a Boolean symbol meaning "or".

The expression P2[car[x]; cdr[x]] also may have the value T (true) or F (false).

Thus it may be seen that the function P1[x] will have the value T (true) if either the expression null[x] is true or the expression P2[car[x]; cdr[x]] is true for a given list x. If neither of these expressions is true, the function P1[x] will have the value F (false).

The expression P2[car[x]; cdr[x]] may be described as follows:

car[x] represents the first element in a given list;

Table 1

	n^3	1st Difference	2nd Difference	3rd Difference
n=0	0			
		1		
n=1	1		6	
		7		6 *2
n=2	8		12	
		19		6
n=3	27		18	
		37		6
n=4	64		24	
		61		6
n=5	125		30	
		91		
n=6	216			
		A→	A→	A→

*2 It may be noted that the list of numbers 0,1,6,6 is sufficient to define the whole array, and thus the complete sequence n^3; a similar short list of numbers will define any polynomial of degree 3.

cdr[x] represents all the rest of the elements of the list. Thus,

car[(1,2,3)] = 1
car[(17,29,31,103,74)] = 17
car[(19)] = 19
cdr[(1,2,3)] = (2,3)
cdr[(17,19,31,103,74)] = (19,31,103,74)
cdr[(19)] = NIL

The function P2[car[x]; cdr[x]] may be represented in a more general form as P2[a;y], where a represents the first element in an original list x, and y represents all successive elements of the list. The function P2[a;y] is defined in the second line of the above statement:

$$P2[a;y]=null[y]\vee[equal[a;cdr[y]]\wedge P2[a;cdr[y]]]$$

The function P2[a;y] may have the value T (true) or F (false). Similarly the expressions

(i) null[y],
(ii) [equal[a;car[y]], and
(iii) P2[a;cdr[y]]]

may each have the values T (true) or F (false). The Boolean symbol $\vee$, as noted above, represents "or"; the sumbol $\wedge$ represents "and". The function P2[a;y] will be true, then, if null[y] is true or ($\vee$) [equal[a;car[y]] and ($\wedge$) P2[a;cdr[y]] are both true.

This may be illustrated by the following table:

null[y]	F	F	F	T	T
$\vee$(OR)					
equal[a;car[y]]	T	F	T	F	F
$\wedge$(AND)					
P2[a;cdr[y]]	F	T	T	T	F
P2[a;y]	F	F	T	T	T

Null[y], then, will be true if the value of y(=cdr[x]) is NIL; i.e., if the list y has no terms. equal[a;car[y]] will be true only if a equals car[y]. For example,

equal[7;car[(7,12,13,14)] = T

since car[(7,12,13,14)] = 7, and 7 equals 7.

equal[13;car[(3,13,13,13,13)]=F

since car[(3,13,13,13,13)] is 3, and 13≠3

P2[a;cdr[y]] will be true if the function P2[a;y] is true when y is replaced by cdr[y].

Im summary, then, the program will operate in the following manner:

1. P1[x]=null [x] ∨ P2[car[x]; cdr[x]]
   a. If null[x] is true, P1[x] is true.
   b. If null[x] is not true, P1[x] is true only if P2[car[x]; cdr[x]] is true. This may be determined from the following statement, where P2[car[x]; cdr[x]] is represented by P2[a;y].
2. P2[a;y]=null[y] [equal[a;car[y]] P2[a;cdr[y]]]
   P2[a;y] will be true only if null[y] is true (the list is empty) or if both:
   a. a is equal to car[y] and
   b. P2 of a and cdr[y] is true.

In general, if all elements of the original list, x, are equal, all elements will be equal to the first element, car[x]. The program will compare every element of x, in sequence, with the first element, car[x], to determine whether they are equal. If they are (or if there are no elements in the list), P1 will be true; otherwise it will be false. Examples of the evaluation of actual lists are presented in Appendix B.

We shall now define a general function, s, of a list. This function is unusual in that one of its arguments is a function. For example, in arithmetic one could define a function operate [op;a;b], where op is a functional argument that specified which arithmetic operation is to be performed on a and b. Thus

operate[+;3;4]=7 and
operate[x;3;4]=12

The operator, s, which is a function of two variables, Fn and x, is given below:

s[Fn;x]=if null[cdr[x]] then NIL
else cons[Fn[cadr[x]; car[x]]; s[Fn;cdr[x]]]

Fn is a function of two variables, x is a list of at least two

members. s[Fn;x] is a list created by performing Fn on successive pairs of x. The first difference of x, d[x] is then:

```
d[x]=[s[difference;x]]
```

The n^{th} difference of sequence x, nd[n;x] is then:

```
nd[n;x]= if equal [n;0] then x; else nd[n-1;d[x]]
```

To encode a sequence x into a defining element, as discussed earlier, we have encode[x]:

```
encode[x]= if null[cdr[x]]vPl[x]
           then cons[car[x];NIL]
else cons[car[x]; encode[d[x]]]
```

In order to expand a series, given the defining element e, we first define a function st[e] which is the next element in the series of elements in the difference array:

```
st[e]= if null[cdr[e]]
       then cons[car[e]; NIL
else cons[car[e]+cadr[e];[st[cdr[e]]]
```

Then, using the first term of each element, we expand the first n terms of the original sequence, x, by means of expand[n;x]:

```
expand[n;x]= if n=0 then NIL;
else cons[car[x]; expand[n-1; st[x]]]
```

The n^{th} term of the original sequence is:

```
nthterm[n;e]= if n=1 then car[e];
else nthterm[n-1; st[e]]
```

We can now perform an interesting set of calculations.

1. Given a sequence, x, that we wish to encode as a polynomial function, we write:

```
encode[x]
```

2. To predict the next term in a sequence, assuming a polynomial representation, we write:

```
predictnext[x]=[nthterm[length[x]+1; encode[x]]]
```

thus

```
predictnext[(1,8,27,64,125,216,343)]=512
predictnext[(16,25,36,49,64)]=81
predictnext[(3,8,13,18,23,28,33)]=38
```

To illustrate the efficiency of encoding, we have:

(a) encode[(3,8,13,18,23,28,33,38,43)]=expand[9;(3,5)]
(b) encode[(16,25,36,49,64,81,100,121,144,169)]=
expand[10;(16,9,6)]

Now let us see if we have generated any "interesting" results. In (a) above, we may translate the expression to the right, [9;(3,5)], as follows:

There are 9 terms; the first term is 3; and each successive term is 5 larger than the preceding term.

In the following sequence:

(c) x=(1,2,3,4,5,6,7,38,9,10,11,12)

the first difference sequence

d[x]=(1,1,1,1,1,1,31,-29,1,1,1)

Since not all terms are equal (though most of them are 1) we must continue the differencing process as follows.

```
 1
     1
 2       0
     1       0
 3       0       0
     1       0       0
 4       0       0       0
     1       0       0       30
 5       0       0       30      -240
     1       0       30      -210    1080
 6       0       30      -180    840     -3600
     1       30      -150    630     -2520   9900
 7       30      -120    -450    -1680   6300
    31       -90     300     -1050   3780
38      -60     180     -600    2100
   -29      -90     -300    1050
 9       30      -120    450
     1       -30     150
10       0       30
     1       0
11       0
     1
12
```

In fact, predictnext[1,2,3,4,5,6,7,38,9,10,11,12]=23,773.

This may not seem very likely; however, it merely illustrates the inability of a polynomial approximation to deal with noise!

What we would like to say is that the sequence resembles the integers except for the 8th term, which is 38 instead of 8.

The series of 12 integers starting with 1 is represented by the expression:

```
expand[12;(1,1)]
```

what we would like is the sequence in (c) above, which may be represented by the expression:

```
if n=8 then 38,
if n≤12 last [expand[n;(1,1)]] (undefined for n>12)
```

"last[x]" is a function which produces the last item of a list x:

```
last[x]= if null[x] then NIL; if atom [x] then x;
if null[cdr[x]] then car[x]; else last[cdr[x]]
```

Consider the following sequence:

(d) (1,2,3,4,5,6,8,10,12,14,16,18)

we would like to realize that this is really two sequences and to encode it as follows:

```
if n≤6 then last[expand[n;(1,1)]];          (undefined
if n≤12 then last[expand[n;(8,12)]]          for n>12)
```

Thus (c) above, when encoded, would automatically point out the interesting facts: "The series is the sequence of integers except that 8 is replaced by 38" or in (d) above, "The series consists of integers up to 6; from then on it consists of the even numbers."

By encoding the sequences into many possible representations, we can choose among them the best one by choosing the one whose representation length is least. This follows from the concept expressed earlier, that the most efficient encoding scheme is the most interesting statement about the data. The function derived may be used to handle sequences of sequences, or sequences of variables, functions, etc., with only slight modification.

APPENDIX

In evaluating the list (1,2,3) the program would proceed as follows:

1. P1[(1,2,3)]=null[(1,2,3)] P2[car[(1,2,3)];cdr[(1,2,3)]]

2. null[(1,2,3)] is <u>false</u> (since the list is not empty)

3. In evaluating whether P2[car[(1,2,3)];cdr[(1,2,3)]] is true or false, it would substitute these values in the second statement as follows:

   P2[car[(1,2,3)];cdr[(1,2,3)]=null[(2,3)]∨
   [equal[1;car[(2,3)] P2[1;cdr[(2,3)]]]

4. In the above expression,

   a. car[(1,2,3)]=1
   b. cdr[(1,2,3)]=(2,3)
   c. car[(2,3)]=2
   d. cdr[(2,3)]=(3)

5. Thus, P2[1;(2,3)]=null[(2,3)]∨
   [equal[1;2] P2[1;3]]

6. In this expression,

   a. null[(2,3)] is <u>false</u> (the list is not empty)
   b. equal[1;2] is <u>false</u> (1≠2)*4
   c. P2[1;(3)] may be evaluated, again, by the formula P2[a;y]=null[y]∨[equal[a;car[y]]∧P2[a;cdr[y]]].

      (1) Substituting these values in this formula, we have P2[1;(3)]=null[(3)]∨[equal[1;car[(3)]∧ P2[1;cdr[(3)]]].
      (2) null[(3)] is <u>false</u> (the list is not empty)
      (3) equal[1;car[(3)]] is <u>false</u> (since car[(3)] is 3, and 1≠3.
      (4) P2[1;cdr[(3)]] may be evaluated as follows:
          (i) P2[1;cdr[(3)]=P2[1;NIL]=Null[NIL]∨ [equal[1;NIL]∧P2[1;NIL]
          (ii) Since null[NIL] is true (the list is empty) it is not necessary to evaluate the expression further; and the value of P2[1;cdr[(3)]] is also true.

(5) Thus, substituting c(2),c(3) and c(4)(ii) in c(1), above, we have P2[1;3]=F∨F∧T

(6) Thus P2[1;3]=F.

7. Substituting 6a, 6b and 6c in 5, we have P2[1;(2,3)]=F∨F∧F

8. Thus P2[1;(2,3)]=F

9. Substituting 8 in 3, we have P2[car(1,2,3);cdr(1,2,3)]=F

10. Substituting 2 and 9 in 1, we have P1[(1,2,3)]=F∨F

11. Thus P1[(1,2,3)]=F, i.e., the list is not composed of equal integers.

As a second example, in evaluating P1 for the list (2,2,2), the program would proceed as follows:

1. P1[(2,2,2)]=null[(2,2,2)]∨P2[car[(2,2,2)];cdr[(2,2,2)]

2. null[(2,2,2)] is <u>false</u> (since the list is not empty).

3. In evaluating whether P2[car[(2,2,2)];cdr[(2,2,2)]] is true or false, it would substitute these values in the second statement as follows:

P2[car[(2,2,2)];cdr[(2,2,2)]]=null[(2,2)]∨
[equal[2;car[(2,2)]∧P2[2;cdr[(2,2)]]]

4. In the above expression,

a. car(2,2,2)=2
b. cdr(2,2,2)=(2,2)
c. car[(2,2)]=2
d. cdr[(2,2)]=(2)

5. Thus, P2[2;(2,2)]=null[(2,2)]∨
[equal[2;2]∧P2[2;2]]

6, In this expression,

a. null[(2,2) is <u>false</u> (the list is not empty)
b. equal[(2;2)] is <u>true</u> (2=2)
c. P2[2;2] may be evaluated, again, by the formula P2[a;y]=null[y]∨[equal[a;car[y]]∧P2[a;cdr[y]]].

(1) Substituting these values in this formula, we have P2[2;(2)]=null[(2)]∨[equal[2;car[(2)]∧P2[2;cdr[(2)]]]]

(2) null[(2)] is false (the list is not empty)

(3) equal[2;car[(2)]] is true since car[(2)]=2 and 2=2).

(4) P2[2;cdr[(2)]] may be evaluated as follows:

(i) P2[2;cdr[(2)]=P2[2;NIL]=null[NIL]∨[equal[2;NIL]∧P2[2;NIL]

(ii) Since null[NIL] is true (the list is empty), it is not necessary to evaluate the expression further; and the value P2[2;cdr[(2)]] is also true.

(5) Thus, substituting c(2), c(3) and c(4) (ii) in c(1), above, we have P2[2;(2)]=F∨T∧T

(6) Thus P2[2;(2)]=T

7. Substituting 6a, 6b and 6c in 5, we have P2[2;(2,2)]=F∨T∧T

8. Thus P2[2;(2,2)]=T

9. Substituting 8 in 3, we have P2[car[(2,2,2)];cdr[(2,2,2)]]=T

10. Substituting 2 and 9 in 1, we have P1[(2,2,2)]=F∨T

11. Thus P1[(2,2,2)]=T, i.e., the list is composed of equal integers.

---

*1We are including so-called "racial memory" or "instinct" in this category — e.g., an infant "remembering" the ability or instinct to suck.

*2It may be noted that the list of numbers 0,1,6,6 is sufficient to define the whole array, and thus the complete sequence n^3; a similar short list of numbers will define any polynomial of degree 3.

*3See Page 20, LISP 1.5 PROGRAMMER'S MANUAL. (This manual contains a detailed description of LISP terms (car, cdr, etc.) used in this report.)

*4In actual practice, the program could determine at this point that the value of this expression is F (false), and therefore

would not continue its evaluation further. (That is, if null[(2,3)] is false and equal[1;3] is false, the total expression P2[1;(2,3)] will be false no matter whether the final term, P2[1;3], is true or false. Or, in other words, F∨F∧?=F.) However we will here continue the evaluation somewhat further for the sake of illustration.

Automation, Using LISP, of Inductive Inference on Sequences

Malcolm Pivar and Mark Finkelstein

Information International, Inc.

(Note: This paper should be considered in the nature of a sequel to the preceding paper by Edward Fredkin.)

This report deals with the problem of programming a computer to perform induction on certain general kinds of data in a manner superior to the majority of human beings.

The kinds of data dealt with here consist of lists or sequences of symbols taken from the following groups:

1) integers;
2) letters;
3) words;
4) symbols made up of meaningless arrangements of letters, that is, nonsense syllables.

Computer analysis of the data presented will give either one of the following two results:

1) A representation of the data which utilizes the patterns found in it by the computer. (Any

representation of the data will be known, in accordance with current usage, as an encoding of the data. Thus, an encoding of "(1,1,1,1,1)" might be "five 1's)". As a limiting case, we may regard "(1,1,1,1,1)" as an encoding of itself.

2) Prediction of the next symbol of a given sequence.

In order to compare the computer's capacities with human capacities, some of the problems submitted for the type of inductive inference considered here will be taken from human intelligence tests. However, we do not mean to imply, by such a comparison, that the computer programs developed here can handle all or even a majority of the varied types of inductive thinking of which the human mind is capable. But we shall show that they can handle certain kinds of problems which are used in tests of general intelligence. Some examples are given in an appendix.

The programs we have been developing may be further classified according to the extent of their dependence on the conventional meanings of the symbols dealt with. For example, in their search for patterns, the less sophisticated programs utilize the conventional order relationships among integers (i.e., "greater than", "less than") and the ordering of letters according to the alphabet.

The more advanced programs, however, make no such assumptions, and are, therefore, able to handle far more general kinds of data. They instead start with a collection of sequences which one may think of as having been acquired from previous experience. The program, in coming to grips with a new sequence, will then make whatever use it can of this "previous experience", whether for predicting the next member, or for producing an encoding of the new sequence which is more economical (with regard to length or storage space) than the sequence itself.

The Basic Programs

The programs written for this project were done in the programming language LISP 1.5 and run on the IBM 7094 computer. Four basic programs are involved: OUTFCN (Outfunction), NEXT, PERTEST, and TEST.

OUTFCN is a program which, given a sequence of integers, produces a LISP 1.5 program capable of generating the same sequence. TEST, given a numerical sequence, will predict the next element in the sequence. PERTEST does the same for alphabetic

sequences. TEST is a similar but somewhat more sophisticated program which can work with either numerical or alphabetic sequences.

OUTFCN

OUTFCN, given a sequence of integers, produces a program in LISP 1.5, which is capable of generating the given sequence.

For example, if the sequence were:

(1) 1, 2, 3, 4, 5, ...

then OUTFCN would produce a program which would say

(2) f(n) = n .... (LAMBDA (N) N)

Given a sequence such as

(3) 3, 5, 7, 9, 11, 13,

it would produce

(4) f(n) = 2n plus 1....(LAMBDA (N)
(PLUS 1 (TIMES 2N)))

And in general the output would be a polynomial evaluation program which given an integer "n" produces f(n).

It may happen that the majority of the numbers in a sequence fit a simple polynomial equation while one or more other numbers in the sequence are exceptions, as in the following sequence:

(5) 1, 4, 9, 16, 26, 36, 49

A strict polynomial interpretation of (5) would need to have as many coefficients as there are numbers in (5) itself and so would hardly constitute an economical encoding. We have therefore designed OUTFCN to take account of the possibility that, but for a few numbers, a sequence may have a simple pattern. For a series like (5) then, OUTFCN will not produce a program to evaluate a 6th degree polynomial, but will produce instead, using an Algol-like notation,

(6) If n = 5, then 26; else f(n) = n^2.... (LAMDA (N)
(COND ((EQUAL N5)26) (T (POWER N 2))))

The remaining possibility is that the sequence fits neither a simple polynomial equation nor a simple polynomial with excep-

tions. (By simple polynomial we mean one in which the number of coefficients is significantly smaller than the length of the given sequence. The exact criterion of when this is so depends on a parameter set by the programmer; it is not something we have an ultimate criterion for as yet.)

For example:

(7) 1, 5, 6, 19, 327, 66

In this type of sequence OUTFCN would regard the sequence (7) as an encoding of itself, and as such, superior to any that it could produce, and would, therefore, refrain from producing any alternative encoding.

We anticipate programs which will construct many different programs to compute a given sequence and will compare their lengths to determine the most economical encoding. This procedure is related to the suggestion of R. J. Solomonoff[1] that the measure of the worth of an encoding could be taken as the number of bits in a computer program which uses the encoding to generate the sequence. That this type of measure is self-consistent has been demonstrated mathematically by H. P. Kramer.[2]

NEXT

It has been stated that the problem of predicting the next member of a sequence is basic to many of the problems of scientific research. Some confirmation of this claim may be derived from the present study, in which, without directly attempting to do so, we have been led to constructing a model of that part of scientific inquiry which involves: making a hypothesis, seeing where it goes wrong, making additional hypotheses to explain the failures of the first hypothesis, continuing to amend one provisional explanation after another, and refraining from attempting explanation when the amount of data is too small.

However, somewhat more than the modeling of scientific activity is involved in the program, since it would not be difficult to imagine sequences which the machine could encode, but for which it would be practically impossible for a human being to discover an optimal pattern, not because of the size of the sequence (though it would have to be fairly long) but because of the complexity of their pattern.

The action of NEXT may be illustrated by means of the following example. Suppose NEXT is given the sequence:

(1) 0, 1, 0, 0, 2, 0, 0, 0, 3, 0, 0, 0, 0, 4, 0,
0, 0, 0, 0, 5, 0, 0, 0, 0, 0, 0, 6, 0, 0, 0,
0, 0, 0, 0,.

It would proceed as follows in attempting to predict the next member.

First it would apply a function called IDSEQ to the given sequence. IDSEQ (from "ideal sequence") takes differences until half or so of the resulting list of numbers are constant, and then produces an ideal sequence, that is, it produces the sequence which would be obtained if all the numbers were constant.

For example, given the sequence:

(1b) 1, 2, 3, 17, 5, 6, 7, 8, 617

the function IDSEQ would take the first differences, namely 1, 1, 14, -12, 1, 1, 1, 609. It would see that more than half of this list were ones; it would then produce the sequence that would result if all numbers in the difference list had been ones, namely 1, 2, 3, 4, 5, 6, 7, 8, 9, which is the "ideal sequence" with respect to the given sequence.

Returning to the sequence (1) above, we see that IDSEQ will produce a list of thirty-four zeroes (the list has 34 members). A compact description of the pattern of the list produced by IDSEQ could be printed out at this point. Such a description would represent a common occurence in scientific explanation: namely, the formulation of a provisional hypothesis which explains much but not all of the data.

The next step, of course, is to try to understand the exceptions to the hypothesis. To accomplish this, NEXT will compare the members of the original sequence (1) to the output sequence of IDSEQ. In particular it will make a list of the positions at which exceptions occur.

(2) 2, 5, 9, 14, 20, 27 (exception list)

and a corresponding list of the values of the exceptions

(3) 1, 2, 3, 4, 5, 6 (value list)

The program then performs the recursive operation of calling itself to see if it can predict the next member of (2); this operation, if successful, would indicate the position where we would expect the next exception to occur. In this case it would dis-

cover that second differences of (2) are constant; so that IDSEQ of (2) would be identical to (2), that is, the exception list itself has no exceptions. Therefore it would confidently predict the next member of (2) to be 35, basing its prediction on the assumption that the second differences of (2) ought to remain constant when new terms are added.

Having discovered where the next exception could be expected to occur (namely, at position 35), IDSEQ then looks to see if this position is the same as the position at which we are required to predict. In other words, is it the same as the length of the original sequence plus one? Finding this to be the case, the program calls itself with the list of values which occurred at the exceptional positions, using (3) as its argument, and by means of the techniques described above would obtain 7 as the next member of (3). It would now declare 7 as the next member of (1).

It is necessary to explain what would have happened if certain of the questions asked by the program had been answered in the negative. Suppose, for example, that the next member of (2) had been different from 35. In this case it would not expect an exception to occur at position 35 and would therefore apply itself to the ideal sequence of (1) and declare zero as output. It would also produce a zero if it had been unable to find any pattern whatsoever in (2). Another possibility would occur if NEXT applied to (2) had yielded 35 but the values at exceptional positions, (3), had been anomalous or irregular. In this event the program would conclude that an exception is expected at position 35 but that its value could not be determined and it would therefore refrain from making any prediction at all and would simply say "no pattern found".

Concerning the recursive aspect of the above program, it should be noted that when NEXT is called to predict the next member of (2), it treats (2) exactly as it treats (1); that is, if (2) had instead been

(2') 1, 5, 9, 14, 20, 28

it would have formed an exception list (2a') and a value list (2b')

(2a') 1, 6	(exception list)
(2b') 1, 28	(value list)

and if necessary would have done the same again for (2a') and for (2b') and similarly for (3) and in fact apply its whole range of techniques to every sequence so derived (until the length of resulting lists fell below specified criteria).

PERTEST

The program PERTEST is designed to accept letter sequences and return a prediction of the next letter. The program was written as a result of seeing a previous program developed by Simon.[3] Simon's program was developed for the purpose of simulating the observed behavior of people when trying to solve problems of predicting letter sequences from an intelligence test. The program PERTEST, on the other hand, was oriented towards the automation of inductive thinking rather than the simulation of human beings; therefore, we developed somewhat simpler though perhaps more mathematical ways of dealing with the problem. We can describe fairly accurately the class of letter sequences which the program will handle.

1) Cyclic sequences, such as: a, b, a, b, a, b, or b, a, c, b, a, c, b, a, c
2) Sequences with a constant skip, such as: a, b, c, d, or u, r, o, l
3) Sequences which are made up of intertwinings[4] of the above two types, such as: a, u, b, r, c, o, d, l, or a, b, a, c, d, b, e, f, c, g, h, d

The program illustrates the following method of attack on the problem of prediction.

When a series is periodic, then we can always predict the next member. Therefore, whenever we can reduce a series to one that is periodic by means of transformations which may be reversed, then we can predict the next member of the cyclic series that was obtained from the original, predict its next member, and then apply the transformations that produced the periodic series in reverse order and thus determine the next member of the original series.

In the case of OUTFCN (described above) this principle is used by taking differences. Note that if the series is 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 then the first difference is periodic; i.e., 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 is of period one. Note also that this transformation is reversible; that is, given the first member of the original series, "1" and the difference list, one can produce the original series. Thus, the operation of taking differences is reversible. In the present program, PERTEST, we used in this a generalization of the operation of taking differences. This is called the shift difference. If we think of first differences as arising from taking a list of numbers, setting a duplicate of the list directly below itself, shifting the

lower list left once, removing the righthand number from the lower list, and the lefthand number from the upper list, and then subtracting members in the lower list from the ones directly above them in the top list, then, in this fashion, the list of differences produced will be identical to the first differences. Now if instead of shifting the lower numbers left one space we shift them two, three, or n spaces, and remove the n righthand numbers projecting out on the left portion of the upper list, and the n numbers projecting out on the right of the lower list, then by subtracting members of the lower list from corresponding members of the upper list we obtain a shift difference with a shift n of the original list. If we denote a list by L and the result of shifting n places and taking differences by Sn, then the first difference becomes S1(L). The Sn's then become a class of transformations which may be used to transform a series into one that is periodic.

Now we can explain how the letter sequence program works. It first converts the given series of letters which we denote by L into numbers, taking A as 0, B as 1, and so forth, Z being 25. After this operation it applies S1, S2, etc., to L. Each operation is applied to the list L, not to the results of previous transformations of L, though such a compounding of operations is necessary when dealing with more complex sequences. After each application of an Sn, the program tests the resulting list to see if it is cyclic. When it discovers a cyclic series in this fashion it predicts its next member, adjoins it to the cyclic series and then reverses the transformation to obtain the next member of the original series L.

In this as in other pattern-finding programs the question arises as to how well established the apparent patterns must be for the program to feel justified in making a prediction. In our program this is a parameter which the programmer sets at present; in more sophisticated versions the parameter can be set by the program itself, especially when the program uses some technique for measuring the goodness of a pattern (such as ratio of length of original form to encoded form.)

TEST

TEST is a somewhat more sophisticated program for sequence analysis. In TEST we are dealing with essentially two different methods for solving a given sequence. One of these is called elliptic differencing. The method is as follows: suppose we are given a two-element frequency distribution created by a human being or by a computer program. Presumably the pairs of elements

(1 2), (2 3), and in general (n n+1) will have much higher frequencies than others. More precisely, the conditional probability of n+1 given n would be expected to be higher than any other element's conditional probability.

Now suppose we are given a sequence (1, 3, 5, 7) and asked to predict the next element without using any of the knowledge we have about the properties of the numbers in the sequence, using just the information in the frequency distribution. Then if we ask for the most probable path, based on the frequency distribution, from 1 to 3 (as indicated by the first two elements in the sequence) we find that it is (1, 2, 3). Thus, the sequence has left out one element between the first two. Repeating the process for the second and third elements of the sequence we find that again one element has been left out, that is, that the ellipsis between them is 1. By continuing to search for the ellipsis between successive pairs of elements in the sequence and noting the magnitude of the ellipsis, we can form the elliptic first difference for the sequence, which for the sequence (1, 3, 5, 7) would be (1, 1, 1). Note that the method employed here would work equally well on letters. In any event, the elliptic first difference will be numerical, and thus we may use numerical methods for the remainder of the solution.

In the actual computer program the routines involved in the above discussion are E and G. These may call upon one another recursively to determine the best (most probable) path between two elements. The routine E is called from the outside with the following parameters: P, the pair (m n) which represents the endpoints of the path for which we are searching; L, the frequency distribution list of pairs of elements (a discussion occurs below describing the formal of L); as a fourth parameter, M, another copy of the list L. The routines E and G produce as a result a list which represents the best possible path, say (1, 2, 3) perhaps for the input P of (1 3), or if no such path exists, then the value is NIL. These two routines call on several other routines. One such routine VALUE, gives as a result the probability of the path it is given as an argument, based on the frequency distribution. TRANS, another routine, gives as a result the probability of the pair given it. (1 2) might, for instance, give a result of 0.75. (Obviously, VALUE will call upon TRANS.)

The routines E and G operate basically as follows. First, they ask: Does the next element which may be appended to the end of the path created so far bring the VALUE of that path to less than a specified threshold? If so, that element is skipped and the next looked at. If not, it then asks, does that element complete the path (i.e., is that element equal to the n of (m n)?

If so, the completed path is noted; the routine continues to scan the remainder of the possibilities, but with a higher threshold, namely the VALUE of the already completed path.

The second method used in TEST is the ORACLE method, which is as follows: for each pair of elements in the system, ORACLE (another routine) tells us of interesting relationships between them. Thus ORACLE might point out, with the pair (3 27), that the first can be transformed into the second by the functional: +24, x9, or $(\ )^3$; in other words, 3+24=27, 3x9=27, 3^3=27. Of the pair (41 43) it might point out that the second is the successor prime to the first. ORACLE is a two argument program, the two arguments being the elements m and n of (m n). Currently the program ORACLE confines them to be integers; however, a trivial modification will allow more generalization. The result of ORACLE is a list of possible transformations which will transform the first element into the second. If the arguments of ORACLE were 4 and 16, the result would be, in LISP notation:

((TIMES X 4) (PLUS X 12) (POWER X 2))

PREDICT

We are now in a position to discuss the main routine of TEST, which is PREDICT. TEST itself is used to construct the pair frequency table with ORACLE values attached to each pair. PREDICT will operate as follows: given the sequence S and the frequency table L, it asks first if the sequence can be solved by a simple cycle test, by calling CYCLES. CYCLES is a small routine which examines whether a sequence is cyclic, and if so, gives as the value the next element.) If it is not cyclic, then the value is NIL. CYCLES is a general program and will work on any list which is purported to be a sequence — that is, the elements of the sequence may be any general expression or lists themselves. If the value of CYCLES is not NIL, then we are done, and PREDICT gives its answer. More likely this is not the case, and then PREDICT operates as follows.

PREDICT attempts to extrapolate from the sequence of lists or ORACLE values a sequence of transformations from which the original sequence was derived. An example at this point may make this clear. Suppose the original sequence was (1, 2, 6, 24, 120), that is, the factorial sequence. Forming the list of lists of ORACLE values for the successive pairs, (1 2), (2 6), (6 24), etc., we would obtain, again in LISP notation, (((TIMES X 2) (PLUS X 1)) ((TIMES X 3) (PLUS X 4)) ((TIMES X 4) (PLUS X 18)) ((TIMES X 5) (PLUS X 96))). This is a list of 4 elements, each

of which is a list of transformations. Note that within each list the transformation (TIMES X n) occurs, for different n. We would wish to note that fact, and take as a sequence to be solved the successive n's, namely (2, 3, 4, 5). Solving this sequence and obtaining 6 as a result, we should like to say that the sequence of transformations yielded the value (TIMES X 6), which would be applied to the last element in the original sequence, 120, to obtain the result 720. This of course would be the correct answer.

In order to accomplish the operations described in the paragraph above, the program PREDICT calls upon a program SBAR, which performs these operations. The main routine of SBAR is FORM, which seeks similarities between two expressions and notes their differences. FORM applied to (A, B, C), (C, D, E) would produce ((A C) (B D) (C E)), i.e., a list of substitutions which when applied to the first argument will yield the second. SBAR uses FORM to determine if any of the ORACLE values in the lists are similar, and rejects all but those whose differences occur only in numbers. (This is only a temporary restriction and will be lifted when we go to more generalized expressions). Thus, if each element in the sequence were a multiple of its predecessor, then the functional (TIMES X n) would appear in each of the lists, and the routine SBAR would note this. It would then set up as a sequence to be solved the n's in the functionals; and in this way it would solve the factorial sequence.

Structure of the Frequency Distribution List in TEST

In considering the structure of the frequency distribution list, suppose the number of occurrences of the pair (i j) is a_{ij}. Then the list shall be structured as follows:

$$((1\ j\ a_{1j}\ (2\ a_{1,2})(3\ a_{1,3})\ldots(k\ a_{1,k}))(2\ j\ a_{2j}(1\ a_{2,1})(2\ a_{2,2})\ldots()))$$

That is, the frequency distribution is a list, each element of which begins with a number i, followed by the total number of pairs that begin with i, followed by a sequence of lists, each one corresponding to an occurrence of a pair beginning with i but ending with j, and within this list is the pair $(j\ a_{ij})$. An optional feature, which is used in the operating versions of this program, is to put the ORACLE value for i, j at the end of the pair, so that the element on the i-list will be $(j\ a_{ij}$ (list of ORACLE(i,j))).

We close this report with a significant quotation from Attneave[5]. It comes from a survey by the same author of a whole

cluster of related studies.

"The view that a basic function of psychoneural activity is the economical encoding of experience may be elaborated in many ways. It has extraordinary generality, applying to the most complex scientific thinking as well as to the basic processes of perception and memory."

---

[1]Solomonoff, R.J., A preliminary report on a general theory of inductive inference, Zator Co., Report ZTB-138.

[2]Kramer, H.P., A note on the self-consistency of definitions of generalization and inductive inference. Journal of the ACM, April 1962, pp 280-281.

[3]Simon, Herbert A. and Kotovsky, Kenneth, "Human Acquisition of Concepts for Sequential Patterns", unpublished.

[4]By intertwining we mean series in which every nth element is from a series of type (1) or (2), every n + 1th element of type (1) or (2), and so forth.

[5]Attneave, F., University of Oregon, "Applications of Information Theory and Psychology", Holt Dryden, 1959, pp 87-88.

Application of LISP to Checking Mathematical Proofs

Paul W. Abrahams

ITT Data and Information Systems Division

I. Introduction

One of the things to which LISP has been applied is the construction of an experimental computer program known as the Proofchecker for checking mathematical proofs. The idea was originally suggested by McCarthy [3] , and the investigations are described by Abrahams [1] . This paper is intended to show how LISP was used in the construction of the Proofchecker and what properties of LISP were particularly useful; little emphasis will be placed on those aspects of the Proofchecker that relate primarily to mathematical logic.

The Proofchecker was primarily directed towards the verification of textbook proofs, i.e., proofs resembling those that normally appear in mathematical textbooks and journals. Although textbook proofs deal with widely varying subject matter, they are quite similar in terms of the kinds of inference they employ. The differences lie in terms of the objects being discussed, the nature and number of the postulates used, and the level of detail provided. The Proofchecker was designed to operate impartially with respect to these differences, just as a human mathematician does.

If a computer were to check a textbook proof verbatim, it

would require far more intelligence than is possible with the present state of the programming art. Therefore, the user must create a rigorous, i.e., completely formalized, proof that he believes represents the intent of the author of the textbook proof, and use the computer to check this rigorous proof.

It is a trivial task to program a computer to check a rigorous proof; however, it is not a trivial task to create such a proof from a textbook proof. Therefore, a primary function of the Proofchecker was to aid in translating from a textbook proof to a rigorous proof. To this end, a language for input proofs was specified; this language bears a close resemblance to programs written using the LISP program feature PROG, and is based upon the construction of "macro-steps." The input proof language was intended to represent a halfway point between textbook proofs and rigorous proofs, the translation from textbook proof to input proof being performed by a mathematician and from input proof to rigorous proof by the Proofchecker.

The formal system used for stating rigorous proofs was based upon the natural deduction system of Suppes [7]. It also included features for handling definitions and for introducing the results of calculations into a rigorous proof. Though the set of rules of inference was fixed, the rules were such that it was possible to introduce additional rules via axioms; λ-conversion, for instance, was handled in this way.

Both the checking of a rigorous proof within this formal system and the generation of the rigorous proof from an input proof were performed by the Proofchecker. The Proofchecker was written in LISP, and consequently the well-formed formulas of the formal system were LISP S-expressions. The Proofchecker accomplished the translation from an input proof to a rigorous proof through the use of macro-steps, which bear the same relation to rigorous proof steps that macro-instructions in a computer macro-assembly language bear to machine instructions in that language. A library of macro-steps was developed to demonstrate the potential capabilities of a computer in making the input proof as close to the textbook as possible, and in expanding steps of the input proof by heuristic techniques.

The Proofchecker was actually applied to Chapter II of Principia Mathematica, by Russell and Whitehead [5]. The proofs from this chapter are all in propositional calculus, and use only the connectives $\vee$, $\sim$, and $\supset$. Macro-steps were constructed so that the textbook proofs could be transcribed practically verbatim, except for the translation from infix notation (e.g., "p $\vee$ q $\supset$ q $\vee$ p") to LISP notation (e.g., "(IMPLIES (OR P Q)

(OR Q P))"). During this transcription process, a number of logical gaps were found in the proofs given in Principia.

The Proofchecker was able to check most of these proofs on the IBM 7090 in a few seconds; but a notable and unpleasant exception concerned those proofs involving replacement, i.e., the substitution of $p \supset q$ for $\sim p \vee q$ and conversely. Because the facilities of the Proofchecker were used for handling definitions, it turned out to require 50 rigorous steps and about 30 seconds for each replacement step in the input proof. Furthermore, in textbook proofs involving more than one replacement operation, the program exhausted the available storage and the rigorous proof could not be completed. An attempt was also made to apply the Proofchecker to some elementary group theory proofs, but this required a larger library of macro-steps, and consequently the program ran out of storage on the initial test cases.

A notable feature of the Proofchecker as far as LISP is concerned is the use of the syntax of LISP on a metamathematical level. In metamathematics, one deals not with logical formulas but rather with the names of these formulas. For instance, one employs theorems stating that if the expression α has such-and-such a syntactic property, then α denotes a true statement. One frequently reasons from properties of symbolic expressions to properties of the formulas that they denote. In practice, a great deal of what is called mathematics is actually metamathematics. Often one requires metamathematical statements that are mechanically verifiable through the use of the LISP function eval, and the ability to perform this verification is one of the most significant features of the Proofchecker.

II. Rigorous Proofs in the Proofchecker

The Proofchecker can best be conceived of as divided into two parts: a part that generates rigorous proofs and a part that checks them. These two parts operate in tandem; the proof generator spews out a step from time to time, whereupon the proof checker examines the step, either accepts or rejects it, and then returns control to the proof generator.

The Proofchecker makes use of a list of previously proven theorems. This list initially contains all the axioms of the Proofchecker, as well as the postulates of the theory under examination. Each theorem is stored as a list of three elements: the name of the theorem, the list of its substitutable variables, and its form, i.e., its statement (as an S-expression). When we wish to use a previously proven theorem or an axiom, we give its

name and a list of the quantities to be substituted for its substitutable variables. There are restrictions on the substitutions since we must prevent the quantifiers of the theorem from capturing free variables in the substituted quantities, and we must also avoid substitutions into the middle of a quotation.

To see how a theorem is represented, consider the tautology

$$p \wedge q \supset q \quad .$$

If we name this tautology CONJ, then it would be stored on the theorem list as

(CONJ (P Q) (IMPLIES (AND P Q) Q))

A rigorous proof is specified by a sequence of steps. Each step consists of the step type followed by a set of parameters. There are a fixed number of step types, and for each step type there are restrictions on the number and form of the parameters. The Proofchecker examines the steps one by one, and if a step is legitimate the Proofchecker generates a line of proof. Thus, the step together with its parameters constitutes an instruction to the Proofchecker; if the instruction is in proper form, it results in a line of proof.

A line of proof represents a proposition that has been demonstrated. Each line of proof is a list of three elements: the line number (simply a serial number); the text of the line, i.e., the proposition demonstrated on that line; and a list of antecedents of the line (as line numbers). The antecedents specify the assumptions on which the line depends. Unlike Suppes' system, there is no provision for flagging of variables. The proof is correct if the text of the last line corresponds to the theorem and has no antecendents. In this case, the theorem is added to the theorem list.

There are various times when the Proofchecker needs an atomic symbol which has not been used before, as for an ambiguous name generated by specializing an existential quantifier. Since it is difficult (though not impossible) in LISP to distinguish among different types of atomic symbols in the way that we distinguish Latin and Greek letters, we must be sure that these new symbols do not appear in a part of the proof which we have not yet examined. There is a LISP function gensym[ ] which can be used for this purpose. Each time gensym is evaluated, it yields a freshly created atomic symbol, constructed in such a way that no other symbol can be equal to it, not even one with the same name! This is accomplished by not putting the symbol on the object list.

When we refer to a generated symbol, we will hereafter mean that the symbol has been obtained by evaluating gensym.

When the Proofchecker checks the rigorous proof, it first constructs a substitution list, in which each substitutable variable of the proof is paired with a generated symbol. This provides for unintended constants among these variables. During the checking of the proof, each expression taken from the step list is relabeled by substituting generated symbols for the substitutable variables according to this list. This substitution list (referred to as the relabeling list) will be augmented during the proof whenever an existentially quantified variable is specified via the Suppes rule ES. We also keep a list of print labels; this list corresponds initially to the relabeling list, but is treated differently from the relabeling list when we perform an ES step. The print labels are needed in printing out the lines of proof, since otherwise the generated symbols would make the proof harder to read.

We can illustrate how the relabeling works, leaving aside for the moment the question of labels arising from ES. Suppose that the substitutable variables of the theorem are (G H). Then the relabeling list might initially be

((G . G00157) (H . G00158)) ,

where G00157 and G00158 are typical generated symbols. If the expression

(SUBGROUP H G)

were to appear in one of the steps, it would be relabeled and so become

(SUBGROUP G00158 G00157) .

The relabeling process operates even inside a quotation.

We also have a list of ambiguous names, initially NIL. Whenever we do an ES step, we add a name to the list. We need this list to prevent universal generalization on an ambiguous name.

The Proofchecker employs ten rules of inference. There is one rule, USETHM, concerned with introducing axioms and previous theorems into the proof. There are four rules, CONJOIN, MODUS, ADDPREM, and MAKEIMP for handling the purely propositional aspects of a proof. There are two rules, CALCEQUAL and VALEQ, concerned with calculation. There are two rules for handling quantifiers,

GENERALIZE and SPECIALIZE. And finally, there is one rule, USEDEFN, for using definitions. We shall now describe these rules in detail.

(1) USETHM has two parameters: the name of a theorem and a list of substitutions. The theorem must appear on the theorem list and the substitutions must satisfy certain logical criteria. Each of the substituted expressions is relabeled before the substitution is made, but the theorem itself is not relabeled. If the step is legitimate, a line of proof is generated. The text of the line of proof is the result of the substitution. The line has no antecedents, i.e., its antecedent list is set to NIL.

Example: Suppose that the theorem list contains the entry

```
(TRANSEQ (X Y Z) (IMPLIES (AND (EQUAL X Y) (EQUAL Y Z))
        (EQUAL X Z)))     ,
```

and step 18 is:

```
(USETHM TRANSEQ ((GMULT A B) (GMULT A C) (GMULT D C))) .
```

We will ignore the relabeling in this example. In this case, the substitutions into TRANSEQ are:

```
X:  (GMULT A B)
Y:  (GMULT A C)
Z:  (GMULT D C)
```

The resulting line of proof is

```
(18 (IMPLIES (AND (EQUAL (GMULT A B) (GMULT A C)) (EQUAL
      (GMULT A C) (GMULT D C))) (EQUAL (GMULT A B)
      (GMULT D C))) NIL)     .
```

(2) CONJOIN has as its single parameter a list of integers which represent line numbers. The text of the resulting line is the logical conjunction of the texts of the given lines, in the same order. The antecedents of the resulting line consist of the union of the antecedents of the given lines.

(3) MODUS is a rule which takes two line numbers as parameters. If the text of the first line is α , the text of the second line must be (IMPLIES α β), for some β . The resulting line has text β , and its antecedents are the union of the antecedents of the parameter lines.

(4) ADDPREM has one parameter, which is an expression.

The resulting line has this expression as text, and its single antecedent is the line itself.

(5) MAKEIMP has two parameters. The first is either a single line number or a list of line numbers, and the second is a line number. Each line in the first parameter must be a premiss, i.e., a line introduced by ADDPREM. If the first parameter is a single line whose text is α, and the second parameter has text β, then the resulting line has text (IMPLIES α β). Otherwise, the first parameter will be a list of lines $\alpha_1, \alpha_2, \ldots, \alpha_n$; in this case, the resulting line has text (IMPLIES (AND α_1 α_2 ... α_n) β). In either case, the antecedents of the resulting line are obtained by removing the line numbers in the first parameter from the antecedents of the line given by the second parameter.

Example: Suppose that we already have the lines

```
(2 (EQUAL (G X) (G Y)) (2))
(4 (EQUAL (G X) (F X)) (4))
(11(EQUAL (H X) (G Y)) (1 2 9 4))
```

and step 12 is

```
(MAKEIMP (4 2) 11)    .
```

The resulting line will be

```
(12 (IMPLIES (AND (EQUAL (G X) (F X)) (EQUAL (G X)
    (G Y))) (EQUAL (H X) (G Y))) (1 9))    .
```

On the other hand, if we have the step

```
(MAKEIMP 4 11)
```

the resulting line will be

```
(12 (IMPLIES (EQUAL (G X) (F X)) (EQUAL (H X) (G Y)))
    (1 2 9 ))    .
```

(6) CALCEQUAL has one parameter, which is an expression to be evaluated. If the parameter is α and eval[α; NIL] = β, then the text of the resulting line is (EQUAL α (QUOTE β)). If eval[α; NIL] is undefined, then the step fails. The resulting line has no antecedents. This step is called the calculation rule.

(7) VALEQ has one parameter, which is an expression α.

The resulting line has text (EQUAL α (EVAL1 (QUOTE α))), and no antecedents.

(8) GENERALIZE has two parameters: a line number and a list of atomic symbols. Let the atomic symbols of the list be $\alpha_1, \alpha_2, \ldots, \alpha_n$. If appropriate conditions are satisfied, a line is generated with text (FORALL α_1 (FORALL α_2 ... (FORALL α_n β)) ...), where β is the text of the given line, and the new line has no antecedents.

Example: If we already have the line

(13 (IMPLIES (EQUAL X Y) (EQUAL (PLUS X Z) (PLUS Y Z)))
NIL)

and step 15 is

(GENERALIZE 13 (X Y)) ,

the resulting line is

(15 (FORALL X (FORALL Y (IMPLIES (EQUAL X Y)
(EQUAL (PLUS X Z) (PLUS Y Z))))) NIL) .

(9) USEDEFN has one parameter, which must be a defined term. The effect of this step is, essentially, to replace the term by its defining expression.

(10) SPECIALIZE is a step used for existential specification. Its operation is quite complicated and its explanation would entail detailed familiarity with the Suppes system; as it is not necessary in the sequel, further description is omitted. The reader is referred to Abrahams [1] for a complete description of SPECIALIZE.

These rules of inference by themselves are not as comprehensive as those of the Suppes system. However, when appropriate axioms are introduced, a logical system can be obtained which is more powerful than that of Suppes, and in particular makes provision for handling quotations, λ -expressions, and non-propositional functions. Although rigorous proofs still are stated in terms of the ten elementary rules, the translational machinery of the Proofchecker is used to in effect simulate the more powerful system.

III. Calculations and λ -conversion

There are two distinct ways to derive statements about the objects involved in a proof: by deduction and by calculation. In deduction, the traditional rules of inference are used to obtain new properties of objects from old ones; these rules include the introduction of previously proven theorems and the use of modus ponens. In calculation, the properties of objects are determined by actual observation. The observing apparatus in the Proofchecker is the LISP function eval1, and in order to make an observation, this function is applied to an appropriate argument. eval1 is defined by

$$\text{eval1}[x] = \text{eval}[x; \text{NIL}] \quad .$$

The rule of inference used for this purpose is CALCEQUAL; VALEQ is needed in order to simplify the results of CALCEQUAL in a manner to be described below.

In order to describe the application of the calculation rule, we need to examine the significance and properties of eval1. Clarification of the terminology used in connection with eval is in fact of interest for its own sake. If we assign the word "value" its usual mathematical connotation, then the value of the S-expression x is not eval1[x], but rather x itself. Since eval is short for "evaluate," the name might seem to be a misnomer. In fact, the name of the function eval stems from its application, not to S-expressions, but to M-expressions. For instance, we can say that

$$\text{cons}[\text{car}[(A)]; (B)] = (A\ B) \quad .$$

The left side of this equation is clearly an M-expression; the right side must also be interpreted as an M-expression for the statement to be correct. Now the transform of the left side into an S-expression is

$$\text{(CONS (CAR (QUOTE (A)) (B)))} \quad ,$$

and

$$\text{eval1}\ [\text{(CONS (CAR (QUOTE (A)) (B)))}] = (A\ B).$$

This result illustrates the general rule: To evaluate an M-expression, i.e., to find its value, transform the M-expression into an S-expression, apply eval1 to this S-expression, and interpret the result as an M-expression. The result is not to be converted to an M-expression by applying the transform rules in reverse; and in fact, it is not in general even possible to apply the transform rules in reverse. For instance, there is no inverse transform for (A . B); i.e., there is no M-expression whose S-expression transform is (A . B).

Applying the function eval1 to an S-expression α yields the denotation of α if eval1 [α] exists. This statement partially defines what we mean by the denotation of an S-expression, and this partial definition has the necessary property that the denotation of the name of any S-expression is the S-expression itself, i.e.,

$$\text{eval1} \quad [(\text{QUOTE } \alpha)] = \alpha$$

for any S-expression α. In the Proofchecker, the S-expressions that appear within a proof may have implicit denotations as well as explicit ones. An implicit denotation is one that is not obtainable through eval1.

We can define the standard value of an S-expression α to be that unique S-expression β such that β is of the form (QUOTE γ) and γ = eval1 [α]. Thus, the standard value of (QUOTE α) for any α is again (QUOTE α); and the standard value of

(CONS (CAR (QUOTE (A)))(B))

is (QUOTE (A B)). The calculation rule says that if an S-expression possesses a standard value, then we may obtain a line of proof stating what its standard value is. In general, the standard value of an expression is much simpler than the expression.

The value of an S-expression will be defined (non-uniquely) to be either the standard value of the S-expression or an atomic symbol with the same standard value as the S-expression. This ambiguous usage is convenient in dealing with LISP constants such as integers and truth-values. Thus, both 3 and (QUOTE 3) are values of (PLUS 1 2).

The principal application of the calculation rule in the Proofchecker is to the verification of syntactic statements. These statements take the form of propositions concerning specific S-expressions, and thus are quite suitable for verification through calculation in LISP. As an example of the technique, we will consider the problem of λ-conversion. The discussion will illustrate both the solution to a particular logical requirement on the Proofchecker, namely, the ability to perform λ-conversions, and the general technique used for syntactic calculations.

λ-conversion is the rule by which we obtain the equality

$$\lambda[[x_1; \cdots; x_n]\mathcal{E}\{x_1; \cdots; x_n\}][a_1; \cdots; a_n] = \mathcal{E}\{a_1; \cdots; a_n\}$$

The λ-expression represents a function; the quantities

a_1 ... a_n are arguments for that function. The conversion is accomplished by substituting each a_i for the corresponding x_i in the expression $\mathcal{E}$. For example, we should be able to obtain the result

(1) (EQUAL ((LAMBDA (X Y) (AND (P X) (Q X Y))) (H1 U) U)

(AND (P (H1 U)) (Q (H1 U) U))) .

by λ-conversion.

In order to perform a λ-conversion, we must satisfy several conditions. The list of dummy variables (the x_i) cannot contain any constants; there must be the same number of dummy variables and arguments; and when we substitute the arguments for the dummy variables in $\mathcal{E}$, we must satisfy certain legitimacy criteria. All of these conditions are syntactic; that is to say, they are conditions on the form of the expressions involved, and have nothing to do with the meaning of those expressions.

The λ-conversion axiom is called LAM1. Its substitutable variables are all intended to be names of S-expressions. The axiom is:

ℓaml [vblist; fm; arglist] = same length [vblist; arglist] $\wedge$ frenlist [vblist] $\wedge$ oksublis [pair [vblist; arglist] ; fm] $\supset$ evalℓ1 [list [EQUAL; cons [list [LAMBDA; vblist; fm] ; arglist] ; sublis [pair [vblist; arglist]; fm]]]

For the example in (1), the substitutions are:

vblist: (QUOTE (X Y))
fm: (QUOTE (AND (P X) (Q X Y)))
arglist: (QUOTE ((H1 U) U)) .

If we write the axiom as an S-expression, we obtain

```
(2)   (LAM1 (VBLIST FM ARGLIST) (IMPLIES (AND
          (SAMELENGTH VBLIST ARGLIST) (FRENLIST  VBLIST)
          (OKSUBLIS (PAIR VBLIST ARGLIST) FM))
          (EVAL1 (LIST (QUOTE EQUAL)
              (CONS (LIST (QUOTE LAMBDA) VBLIST FM)
              ARGLIST) (SUBLIS (PAIR VBLIST ARGLIST)
              FM)))))    .
```

In order to perform a λ -conversion, it is necessary to start with this axiom and then proceed through a standardized sequence of inferences to obtain the result. The axiom LAM1 takes the form of an implication, in which the antecedent represents the syntactic requirements for the λ -conversion, and the consequent represents the desired result. When the appropriate substitutions are made, the antecedent becomes an expression that can be evaluated, and the result of the evaluation will be *T* (the value of T). From this we can deduce the consequent; and after evaluating the appropriate sub-expression of the consequent; we get the result we want.

Let us first examine what happens when we evaluate the antecedent of (2) which, after the substitutions are made, becomes

```
(3)     (AND (SAMELENGTH (QUOTE (X Y)) (QUOTE ((H1 U) U)))
             (FRENLIST (QUOTE (X Y)))
             (OKSUBLIS (PAIR (QUOTE (X Y)) (QUOTE ((H1 U) U)))
                 (QUOTE (AND (P X) (Q X Y))))))     .
```

If we denote this expression by α, then the evaluation consists of computing eval[α], which we will call β (and is *T*).

α consists of a conjunction of three terms; if each of them evaluates to *T*, the entire expression will evaluate to *T*. The first term states that the argument list and the variable list have the same length (though of course the function samelength must have an appropriate LISP definition). Similarly, the function frenlist tests whether its argument is a list of atomic symbols, each of which has at most PNAME and SPECIAL on its property list, and none of which are numbers, i.e., none of the symbols are constants. Thus the second term guarantees that the dummy variable list is grammatically correct. The function oksublis[x; y] expects its first argument x to be a list of pairs, each of which specifies a substitution and is of the form (a . e), where e is to be substituted for the atomic symbol a. oksublis tests whether all the substitutions given in x are legitimate; its value is *T* if they are. In (3), the first segment of oksublis is

```
pair[(X Y); ((H1 U) U)]  = ((X . (H1 U)) (Y . U))
```

and the second argument is

```
(AND (P X) (Q X Y))     ;
```

these two arguments do satisfy the criterion, so that the third term evaluates to *T* also and hence α evaluates to *T*.

Had the substitutions been incorrect, the evaluation of α could have failed in either of two ways. Obviously, α might have evaluated out to NIL instead of to *T* in such a case. Another possibility is that the evaluation might have led to a LISP error complaint. The latter is in fact the more likely possibility. Suppose, for instance, that we had failed to quote the substituted expressions. The result would have been a complaint from the LISP system that X was an undefined function.

Let us now examine the consequent of the implication in (2). After the substitutions are made, we have:

(4) (EVAL1 (LIST (QUOTE EQUAL) (CONS (LIST (QUOTE LAMBDA) (QUOTE (X Y)) (QUOTE (AND (P X) (Q X Y)))) (QUOTE ((H1 U) U)))
(SUBLIS (PAIR (QUOTE (X Y)) (QUOTE ((H1 U) U)))
(QUOTE (AND (P X) (Q X Y))))))) .

Let us denote the expression following EVAL1 in (4) by γ, and the expression in (1) by δ. Then δ is the desired result, and furthermore,

$$eval1[\gamma] = \delta \quad .$$

Thus, from the calculation rule, we obtain

(5) (EQUAL γ (QUOTE δ)) .

We now need an axiom for substitution. The axiom is:

substit [fn; arg1; arg2] = equal [arg1; arg2] ⊃
equal [fn [arg1]; fn [arg2]]

We use this axiom with the arguments

fn: EVAL1
arg1: γ
arg2: (QUOTE δ) .

Using MODUS on (5) and this axiom, we obtain

(6) (EQUAL (EVAL1 γ) (EVAL1 (QUOTE δ))) .

Using the rule of inference VALEQ, we obtain

(7) (EQUAL δ (EVAL1 (QUOTE δ))) .

VALEQ must be a rule of inference because we cannot in general

substitute into a quotation within an axiom. In this case δ is substituted within a quotation. We now use an axiom for the transitivity of equality, together with CONJOIN and MODUS, to obtain the result

(8) (EQUAL (EVAL1 γ) δ) .

Now since we have shown that the antecedent of (2) is true, i.e., we have obtained it as a line of proof, we can deduce the consequent, i.e., (4). But (4) is just (EVAL1 γ). Since this is logically equivalent to δ by (8), we can deduce δ from (8), and δ is the desired result.

Now that we have presented the entire argument, let us review it briefly. If we wish to perform a λ -conversion, we must first substitute appropriate arguments into the axiom LAM1, and these arguments are the names of various parts of the desired result. LAM1 is in the form of an implication; by calculating the antecedent, we determine that the conversion is legitimate, and by calculating the consequent, we obtain the actual expression of the result. Since the antecedent is true, the consequent must be true, and so we have the result.

The use of the λ -conversion axiom is an example of a technique that is employed repeatedly by the Proofchecker. It is possible only because of the existence of the calculation rule. The Proofchecker has been set up in such a way that it is easy to create macro-steps that follow this pattern; and in actual practice, we can do an entire λ -conversion with one macro-step.

IV. Macro-steps and the macro-step interpreter

There is a very strong analogy between the problem of composing a rigorous proof for a theorem and writing a machine-language program. In each case, we have available certain operators which alter our data. In the Proofchecker, the operators are the ten rules of inference, and the data is the list of previously derived results. In a machine-language computer program, the operators are the machine instructions and the data is the set of machine registers which have been set aside for storage. In each case, we are faced with the difficulty that a single application of an operator produces only a small change in the data, and that a great many applications of the operators are needed in order to obtain significant results.

A feature common to both situations is that there are recurring patterns of application of the operators. In the case

of machine coding, there are two kinds of approaches which have been used to take advantage of this situation. One of these is the analytic approach; we develop an artificial programming language (e.g., FORTRAN, or for that matter, LISP) which enables us to describe the calculations we wish to perform, but bears little or no relationship to machine language. More often than not, this language will be machine-independent. Then we build a translator which reads a program in the artificial language and creates a corresponding machine-language program.

The other approach, which is the one we are interested in, is the synthetic approach, and is represented by macro-assemblers such as SCAT and MACRO-FAP [4]. These macro-assemblers grew out of the older-style assembly programs, and in fact they almost invariably contain a traditional assembly program as part of their operation. They are designed so that frequently used sequences of instructions may be abbreviated by a single instruction.

In the Proofchecker, the role of the macro-assembler is played by the macro-step interpreter meth 1. Macro-step definitions, which closely resemble LISP PROG-type programs, play the role of macro-instruction definitions. The macro-step definitions may contain actual proof steps, calculations to be performed, or calls to other macros. meth 1 uses the LISP function errorset to supervise most calculations so that in the case of error, meth 1 still retains control.

As a proof is generated, two records are kept of it: a list of rigorous steps, known as STEPLIST, and a corresponding list of lines of proof, known as LINESET. Each step is in fact an instruction on how to generate the corresponding line. Whenever a rigorous proof step is encountered by meth 1 in its interpretation of a macro, meth 1 calls upon a function called stepcheck. stepcheck checks the step for legitimacy, using the appropriate rulecheck lr (there is one for each rule of inference). If the step is legitimate, stepcheck creates the appropriate line of proof, adds it to LINESET, and adds the step to STEPLIST. If the step is illegal, stepcheck communicates an error indicator to meth 1. Generated lines of proof are numbered consecutively, and the variable LINECOUNT records the current count.

Since rigorous proofs are difficult to handle, meth 1 serves as an interface between the mathematician and the rigorous proof. For instance, the mathematician who uses a macro-step does not ordinarily know how many rigorous lines of proof it will generate. Thus he cannot refer to previous results, since these are indexed by line number. Hence provision is made in the Proofchecker to accept input proof steps with attached labels; the

labels are then tied to the last line generated by the macro-step via a symbol table known as LABEL EDLINES. At any stage, the result of the immediately previous macro-step is given the implicit label NIL. This arrangement makes it possible to refer to the immediately preceding result without labeling it; since in many proofs a result is used only to support the immediately following step, this arrangement eliminates a great many labels.

Another useful aid to the mathematician is a list called KEYLINES. Any line of the input proof may be indicated as a key line; in addition, at any stage, the result of the last executed macro-step, i.e., the one implicitly labeled NIL, will be automatically included in KEYLINES. The information in KEYLINES may be referred to implicitly as well as explicitly. This effect is achieved by having macros that search through KEYLINES when an additional result is needed to support a conclusion. In other words, the user, at least in some cases, need not cite all the steps upon which a given line depends; he can let the Proofchecker find them by a search through KEYLINES. Of course, if too many entries appear in KEYLINES the search becomes woefully inefficient.

The Proofchecker programs in general, and meth 1 in particular, make extensive use of the LISP function errorset. errorset performs the same calculation as eval, with the following important difference: if the evaluation succeeds, list of the result is returned, and if it fails due to a LISP system error, NIL is returned via a system error recovery routine. Thus, errorset makes it possible for the program to attempt evaluations even when they lead to system errors, without control being lost by the function that called errorset. With this function, it is possible for a program to try an evaluation and see whether it is legitimate without losing control. Since such things as evaluating unsatisfied conditional expressions or unbound variables will cause system errors, this type of error recovery is frequently desirable. Although errorset was explicitly constructed for the sake of the Proofchecker, it is useful in other applications also. It is explained in detail in the LISP 1.5 Manual [2] .

Macro-steps are of two kinds: external macro-steps and internal macro-steps. Only external macros can be used in the input proof. An external macro-step receives its parameters from meth 1 in a particular format which differs from that used by most other macro-steps. Normally, however, an external macro-step will call one or more internal macro-steps, which are not suitable for external use. An example of the specification of an internal macro-step is the following:

```
EQL1:    ((L1 L2) (L3)
          (SETQ L3 (FINDLINE1 L2))
          (CONJOIN (LIST L1 L2))
          (USETHM (QUOTE EQL1) (LIST (CADR L3) (CADDR
              L3)))
          (MODUS (LCM 2) (LCM 1))      )
```

This definition is attached to the property list of EQL1 following the indicator APVAL. The macro makes use of the axiom

(EQL1 (P Q) (IMPLIES (AND P (EQUAL P Q)) Q))

We use this macro-step when we have two lines of proof, the first being an expression α and the second being (EQUAL α β). Using this macro-step, we derive the line β . The function findline [ℓ] takes a line number as argument, and its value is the text of that line. If ℓ does not correspond to a line number, a LISP error occurs. The function lcm[k] has the value LINECOUNT - k.

On the first line of the definition, we find the parameters of the macro-step, i.e., its arguments, and its program variables. In this case the macro-step name is EQL1, its parameters are L1 and L2, and its program variable is L3. L1 and L2 will have as values the line numbers of α and (EQUAL$\alpha\beta$) respectively. The first step of the macro is

(SETQ L3 (FINDLINE1 L2)) .

This step causes the program variable L3 to be set equal to the value of (FINDLINE1 L2), i.e., (EQUAL α β). The next line is

CONJOIN (LIST L1 L2)) .

Since the symbol in the function position here is one of the rules of inference, meth 1 does not merely evaluate it; instead, it evaluates the parameter of CONJOIN, giving a list of two integers, and then uses stepcheck to attempt to CONJOIN step, with the list of two integers as parameter. The next step is

(USETHM (QUOTE EQL1) (LIST (CADR L3) (CADDR L3))) .

(QUOTE EQL1) is used rather than EQL1 because the parameters of USETHM are evaluated before they are given to stepcheck. Evaluating the second parameter or USETHM gives the list (α β). Thus, USETHM will introduce the axiom EQL1 with α as P and β as Q. The final step is

(MODUS (LCM 2) (LCM 1)) .

This combines the results of the two previous lines so that the new line is β . To illustrate: suppose that we already have the lines

```
(9 (EQUAL  (P X) (Q Y)) (3 4))
(12 (P X) (12))
```

and that the last line generated is 15. Then evaluating meth 1 (EQL1 12 9) will cause the following steps to be generated:

```
(16 (CONJOIN (12 9)))
(17 (USETHM EQL1 ((P X) (Q Y))))
(18 (MODUS 16 17))     .
```

The corresponding lines are:

```
(16 (AND (P X) (Q Y)) (12 3 4))
(17 (IMPLIES (AND P X) (EQUAL (P X) (Q Y))) (Q Y)) NIL)
(18 (Q Y) (12 3 4))     .
```

Although this macro is occasionally needed on the top level of a proof, its greatest value is for use within other macros. For the top level, its inputs are not in the necessary format, so we still need an external macro to call it.

A macro definition is written in the same form as a LISP program. There is a preamble specifying the name and arguments of the macro, followed by a sequence of S-expressions. Any atomic symbols within this list are taken to be labels of the first following non-atomic S-expression. Each line of the macro-step is examined in turn. If the first element of the line is the name of a rule of inference, then stepcheck is invoked. If the step fails, an error condition is established; otherwise a line of proof is generated by stepcheck. If the first element of the line is MACRO, then meth 1 is called recursively to interpret the macro (with parameters) specified by the remainder of the line. Any error condition within the called macro will in turn set up an error condition in the calling macro. Lines beginning with GOTO or CONDl are treated like GO and COND in the usual PROG function definition; for technical reasons, different names must be used. RETURN acts like RETURN in a PROG function definition. With the exception of lines beginning with CONTINUE, any other line is merely evaluated under the control of errorset; if the evaluation causes a system error, an error condition is set up within the macro.

One interesting aspect of meth 1 is its provision for error recovery. It was pointed out that errors can occur when meth 1

operates on a line. However, before any action is taken, the line following the erroneous one is examined. If it is of the form (CONTINUE x), where x is an atomic symbol, then control is transferred to the line of the macro-step labeled by x, just as if the CONTINUE were a GOTO. If CONTINUE is encountered under any other circumstances, it is ignored. Ordinarily, when a line of a macro-step creates an error condition, interpretation of the macro-step ceases and control passes to the next higher macro. Since this macro in turn detects an error condition, control passes all the way up to the topmost level of interpretation, i.e., interpretation of an input proof step. However, the appearance of CONTINUE following a macro call at any level of the hierarchy will cause the macro at that level to retain control. Thus it is possible to attempt proof steps, either elementary or on the macro level, even if it is not certain they will work. It would in fact appear that this principle -- namely, of being able to try things that cause system errors and act on the basis of whether these errors occur -- is a generally useful one in programming systems design.

When an error condition is passed all the way up to the top level of interpretation, a backtrace is generated and printed out. This backtrace shows all macros entered at the intermediate levels between the error and the input proof step, and what the parameters of each macro were. Since an error in an input proof step frequently does not show up until some macro (or an elementary proof step) several levels down is examined, this backtrace is invaluable in determining why a step is erroneous. It is also quite helpful as a debugging aid for macro-step definitions. When a step of the input proof fails to check, the result is assumed and the proof continued. Consequently, it is possible to learn of many errors in one pass.

One example of a macro-step is the macro LAMCON1, which is used to mechanize the λ-conversion procedure discussed in the previous section. The single parameter of LAMCON1 is an expression of the form $(g\ a_1 a_2 \ldots a_n)$, where g is a λ-expression for a function and $a_1, a_2, \ldots, a_n$ are its arguments. Executing the macro performs the λ-conversion and generates the line

$$(\text{EQUAL}\ (g\ a_1\ a_2 \ldots\ a_n)\ \mathcal{E}) \quad .$$

where $\mathcal{E}$ is the result of the λ-conversion. The interpretation of LAMCON1 generates 17 rigorous steps; the last one of these is the desired result.

Two other macros of interest are ONE-STEP-DED and MODUS-PONENS. ONE-STEP-DED is an internal macro that makes certain "obvious" deductions, which are called one step deductions. It

has three parameters: a conclusion, a premiss, and the line number of the premiss, which must be a previous result. ONE-STEP-DED attempts to derive the conclusion from the premiss. There is a related external macro ONESTEP which has a label as parameter; ONESTEP uses the label to obtain both the text and the line number of the premiss, and then calls ONE-STEP-DED with the desired result as the conclusion and the premiss and the line number corresponding to the given label.

ONE-STEP-DED itself determines what sort of deduction is appropriate to obtain the conclusion. It handles the following cases:

(a) The premiss is a conjunction of several terms, and the conclusion is one of them.

(b) The premiss is an implication whose antecedent is either among KEYLINES or is calculably true, and whose consequent corresponds to the conclusion.

(c) The premiss is preceded by one or more universal quantifiers, and the consequent may be obtained by a sequence of specialization of these quantifiers.

(d) The premiss is a defined term, and either the conclusion can be obtained from the definiens of the premiss by a one step deduction, or the definiens of the premiss is a conjunction of several terms, and the conclusion may be obtained by a one step deduction from one of them.

These cases were selected on the basis of observation as to the sort of thing which is frequently needed in a proof. Although there are many other kinds of deductions which might be included in this step, adding them would lengthen the execution time of the step. This effect would be particularly pronounced because of the recursive nature of the deduction in case (d). The recursive nature of the step has the effect of permitting some deductions which are actually several steps in length.

The comparison of case (a) with the subcase of case (d) where the definiens is a conjunction of several terms illustrates the heuristic basis of the choice of permitted deductions. If the premiss is a defined term, we will check each component of the definiens to see if the conclusion follows by a one step deduction; but if the premiss is already a conjunction, we limit ourselves to an identity check. This situation is based on the observation that conjunctions of terms in a proof arise frequently from definitions, and that in this case we are quite likely to be able

to achieve the desired result by another application of ONE-STEP-DED.

The macro MODUS-PONENS is used to handle citations of theorems. Its first parameter is the name of a theorem; its second parameter is a list containing two types of elements; suggested substitutions into the theorem, and previous results appearing in the hypotheses of the theorem. The desired result is also provided. MODUS-PONENS first compares the desired result with the consequent of the cited theorem (which must be in the form of an implication or equivalence) to determine any substitutions left unspecified. Substitutions not appearing in the consequent must be specified, though in practice all substitutable variables appear in the consequent more often than not. After determining the entire set of substitutions, the macro can then find precisely what hypotheses must be established. It then attempts to establish these one at a time, by using ONE-STEP-DED on pairings between the cited previous results and the necessary hypotheses. Hypotheses that are either calculably true (i.e., that evaluate to T) or that are among KEYLINES need not be substantiated. The last line generated by MODUS-PONENS is the desired result; the intermediate lines are those needed to establish the hypotheses, followed by the actual application of the theorem via USETHM and MODUS.

The macro-step mechanism we have described makes it possible to achieve a great deal of abbreviation in specifying a proof. Unabbreviated proofs are extremely tedious to compose, and hence their composition is highly vulnerable to error. Furthermore, composing such proofs is a task which few people are willing to undertake; the proof of this lies in the fact that the task has been undertaken only for the most elementary logical theorems. Macro-steps are no more than an abbreviative device, but nevertheless such a device seems essential.

Macro-steps avoid the necessity of proving many derived rules of inference. Each derived rule is represented by a macro, and this macro actually generates the steps corresponding to the derived rule. The Proofchecker is set up in such a way than an error in a macro-step simply causes the proof generator to fail (and simultaneously to pour forth a wealth of diagnostic information). Thus, if the conception of a derived rule has been in error, this will reflect itself in an error in a macro-step definition which in turn will show up when an attempt is made to check proofs using this derived rule.

Macro-steps can also be used to perform heuristic searches. In the present implementation of the Proofchecker, this ability was utilized in only a limited way, to cut down the amount of

detail that the user need state as in the MODUS-PONENS macro. However, searching can also be used to fill in gaps in a proof that the mathematician has not thought of, or in conjunction with proof procedures. These extensions remain to be explored.

A further property of the macro-step arrangement is its open-endedness. It is always possible to add new macros to the existing library, and thus to augment the kinds of inference that the Proofchecker is capable of performing. One disadvantage of this must be noted, however: the macro library tends to consume storage. Storage capacity turned out to be one of the chief limitations on what the Proofchecker could do.

V. Conclusions

The construction and application of the Proofchecker demonstrate that the macro-language concept is a powerful and useful one in areas other than macro-assemblers. The macro-steps of the Proofchecker permitted a notable condensation in specifying proofs. These macro-steps saved not only the labor of writing things down, but also the labor of figuring many things out. Furthermore, the subroutine kind of organization inherent in a macro language permits the rapid construction of an extensive hierarchy of available devices and methods; an internal macro such as ONE-STEP-DED, for instance, can be used by many other macros and yet needs to be defined only once.

A further conclusion is that the type of control provided by errorset is a useful one in computer executive systems. The LISP function meth 1 was effectively an executive program that decided what needed to be done and then called upon the appropriate apparatus to do it. Within these programs, it was possible to try a step and if it failed, to try something else without losing control. The ability to do this in general is something which is missing in most existing executive and monitor systems; but the results of the Proofchecker suggest that such features would be well worthwhile.

The self-descriptive properties of LISP do indeed have more than theoretical interest. The calculation rule, which was based on the LISP function eval, made possible the construction of a somewhat unusual but very powerful mathematical system, and in particular permitted the development of the whole apparatus of syntactic methods that was illustrated for the particular case of λ-conversion. In particular, it was possible for LISP to create expressions to be evaluated via the calculation rule, and then carry out the evaluation by considering these expressions

to be LISP programs.

While LISP is still the best programming language for this type of application, it has serious drawbacks, both in terms of convenience and in terms of programming efficiency. The task of writing out S-expressions to define programs is a tedious one, especially since the LISP notation seems to run counter to the natural way that people think of mathematical expressions. For instance, in a sequence of composed functions, we would like the function to be performed first to appear first; but in the LISP notation, it is just the opposite. Furthermore, the rate at which LISP consumes free storage, combined with the limited amount of storage available, greatly restricts the kinds of things that can be done in LISP.

The problem of excessive storage requirements seems to be a general one with list-processing languages. The requirements of computations requiring list processors are a strong argument for increasing the capacities of our computers. (The same problems that plagued the Proofchecker were encountered by Slagle [6]). Because of the random access needed to virtually all parts of both program and data, magnetic tape and other auxiliary serially-accessed storage media have been of little help in easing the space shortage in LISP; although IPL has facilities for storing information on a drum, for instance, these are of limited usefulness.

The fact that the macro-steps had to be interpreted considerably degraded the performance of the Proofchecker. It ought to be possible to build a special-purpose compiler, perhaps using some components of the present LISP compiler, to convert macro-steps into machine-language programs. This would do two things at once: it would speed up the operation of the macro-steps, and it would cut down the amount of storage they require (since the interpreter itself uses significant amounts of free storage).

REFERENCES

1. Abrahams, P., Machine Verification of Mathematical Proof, Doctoral Dissertation in Mathematics, Massachusetts Institute of Technology, June 1963.

2. LISP 1.5 Programmer's Manual, Computation Center and Research Laboratory for Electronics, Massachusetts Institute of Technology, August 1962.

3. McCarthy, J., Computer Programs for Checking Mathematical Proofs, Proceedings of the American Mathematical Society on Recursive Function Theory, held in New York, April 1961.

4. McIlroy, M.D., Macro-Instruction Extensions of Compiler Languages, Communications of the Association for Computing Machinery, April 1960, pp. 184-195.

5. Russell, B., and Whitehead, A.N., Principia Mathematica, Vol. 1, New York, Chelsea, 1938 (2nd edition).

6. Slagle, J. R., A Heuristic Program that Solves Symbolic Integration Problems in Freshman Calculus, Journal of the Association for Computing Machinery, October 1963, pp. 507-520.

7. Suppes, P., Introduction to Logic, Princeton, Van Nostrand, 1957.

METEOR: A LISP Interpreter for String Transformations

Daniel G. Bobrow

(The work reported here was supported by the Research Laboratory of Electronics and the Computation Center of the Massachusetts Institute of Technology.)

I. Introduction

Conditional expressions, composition, and recursion are the basic operations used in LISP[1] to define functions on list structures. Any computable function of arbitrarily complex list structures may be described using these operations, but certain simple transformations of linear lists (strings) are awkward to define in this notation. Such transformations may be characterized (and caricaturized) by the following instructions for a transformation:

> Find in this string the substring consisting of the three elements immediately following the first occurrence of an A, and find the element just before an occurrence of a B which follows these three elements; if such elements exist, exchange the position of the three elements and the one element, delete the A, and replace the B by a C.

A notation for expressing such transformations is the basis for the COMIT programming language of Yngve.[2] It consists of a

formal method for selecting substrings from a string, and then indicating the structure of the transformed string. It is easy to write COMIT rules which perform string transformations such as rearrangement, deletion, insertion, and selection of elements from context. However, COMIT does not easily allow the general list processing that can be done in LISP.

A language in which statements in both LISP functional notation and COMIT prototype notation are interpretable would be desirable. As a compromise, to allow easy string manipulation within LISP, a LISP function, METEOR, has been written which will interpret COMIT-type rules, and perform the indicated string transformations. METEOR notation is similar to that used in COMIT, with some additional features, such as use of mnemonic names for certain expressions. This report describes in detail the types of program statements that can be interpreted by METEOR, and is hopefully independent of any knowledge of COMIT, although some knowledge of LISP is assumed at times. Similarities to COMIT will be obvious to the knowledgeable reader, and occasional warnings about differences between METEOR and COMIT are inserted.

Section II of this report is an introduction to METEOR by examples. Most of the features of the system are illustrated. Section III is a complete, exact specification of a METEOR program and its interpretation. Section IV is a collection of warnings for the unwary about the foibles of the system — a combination of the worst of LISP and COMIT — and some paternal advice about how one can best use the system.

II. Operations with METEOR Rules

In this section the structure and operation of some types of METEOR rules will be illustrated by simple examples. Figure 1 is a listing of a sample program as run in the LISP system under the METEOR Interpreter. The output from this program is shown in Figure 2. The entire program will be discussed in some detail.

First, the LISP interpreter must be informed that it is to use the METEOR program. The punch card expressed in line 0 of Figure 1 performs this function. The two left parentheses open, respectively, the list of arguments for METEOR (it has two) and the list of rules (the first argument).

Lines 1-22 state rules, which will be applied successively to transform the linear list (called the workspace) given in line 23, and the "workspace", in this example, is "(A ROSE IS A ROSE IS A ROSE)."

FIGURE 1

```
 0.  METEOR((
 1.  (*   (ROSE) (FLOWER)  * (SIMPLE REPLACEMENT))
 2.  (*    ((*P THE WORKSPACE IS))         *   (DEBUG PRINTOUT))
 3.  (*  (IS A ROSE)    (0     *  (DELETION))
 4.  (*   (A FLOWER IS)  (3 1 2)           *  (REARRANGEMENT))
 5.  (*  ((*P WS2))              *)
 6.  (*  ( FLOWER)  ( 1 OF RED)       *     (INSERTION) )
 7.  (*   (A FLOWER)  (THE 2)   * (REPLACEMENT IN CONTEXT))
 8.  (*  ((*P WS3))              *)
 9.  (*   (FLOWER)                        *  (NO OPERATION))
10.  (*     (RED)   (1 1)         * (DUPLICATION))
11.  (*  ((*P WS4))              *)
12.  (*   (OF ($.1))   (1)     *(SINGLE UNKNOWN CONSTITUENT))
13.  (*    (($.1))    (QUESTION 1)              *  (FIRST CONSTITUENT))
14.  (*    ( (*P WS5))         *)
15.  (*   ( ($.2)  FLOWER ($.3))   (3 2 1) * (N CONSECUTIVE CONSTITUENTS))
16.  (*  ((*P WS6))              *)
17.  (*  (FLOWER $ ROSE)  (1 3)    *   (UNKNOWN NUM OF CONSTITUENTS))
18.  (*    ( (*P WS7))        *)
19.  (*    ($)   (START C A B D)  * (REPLACING ENTIRE WORKSPACE))
20.  (*  (START ($.1)  $ D)   (1 3 2 4)  *)
21.  (*  ((*P WS8))
22.  (*   ($)          END)
23.  )(A ROSE IS A ROSE IS A ROSE))
```

FIGURE 2

```
(THE WORKSPACE IS)
(A FLOWER IS A ROSE IS A ROSE)
(WS2)
(IS A FLOWER A ROSE)
(WS3)
(IS THE FLOWER OF RED A ROSE)
(WS4)
(IS THE FLOWER OF RED RED A ROSE)
(WS5)
(QUESTION IS THE FLOWER OF RED A ROSE)
(WS6)
(QUESTION OF RED A FLOWER IS THE ROSE)
(WS7)
(QUESTION OF RED A FLOWER ROSE)
(WS8)
(START A B C D)
```

Replacement

Items in the workspace may be replaced. The rule in line 1 finds the first occurrence in the workspace of "ROSE" and replaces it by an occurrence of "FLOWER". The list of one element "(ROSE)" is called the left half of this rule, and the list of one element "(FLOWER)" is called the right half of this rule. The "left half" selects elements from the workspace. The "right half" indicates what the result is to be from transforming these selected elements.

Printout of the Workspace

The contents of the workspace may be printed out with an identifying message by a rule such as that in line 2. The list "(*P message)" as the only element of the left half of a rule will cause a printout of the "message" first, and then the contents of the workspace. Line 2 prints out "(THE WORKSPACE IS)" and then the contents of workspace. This printout is shown in Figure 2. The workspace remains unchanged.

Deletion

Items may be deleted from the workspace. The rule in line 3 finds the first occurrence in the workspace of the three words "IS A ROSE". These are specified by the left half of this rule. The atom 0 (zero) as the right half of this rule specifies that these words should be deleted from the workspace, and nothing inserted in their place.

Rearrangement

Items in the workspace may be rearranged. The rule in line 4 finds the first occurrence of "A FLOWER IS" and reorders it as

"IS A FLOWER". This is specified by the order of the elements in the right half, the list "(3 1 2)". In this right half, 3 refers to the element matched by the third element of the left half, 1 to the first, and 2 to the second. Deletion and rearrangement can be done simultaneously by not mentioning in the right half an item matched in the left half. For example, if the right half of this last rule were (3 2), the string "IS FLOWER" would be substituted for "A FLOWER IS" and this occurrence of "A" would no longer appear in the workspace.

The rule in line 5 causes a printout of the message WS2 followed by the modified workspace. (See Figure 2.)

Insertion

New material may be inserted into the workspace by a METEOR rule. The rule in line 6 finds the first occurrence of "FLOWER" in the workspace, and inserts, just after it, the elements "OF RED". The 1, of course, refers to the occurrence of "FLOWER" as the first (in this case, the only) left half constituent (pattern element). Insertions can be made before, after, or between elements of the workspace identified (matched) by the left half of the rule.

Replacement in Context

Suppose we wish to replace the article "A" by the article "THE" when it appears immediately before the word "FLOWER". The left half "(A)" cannot be used to locate this occurrence of "A". This left half will locate the first occurrence of "A" in the workspace. However, the "A" found by "(A FLOWER)" is the appropriate one, i.e., the one immediately preceding "FLOWER", and the stated transformation is performed by the rule in line 7. Line 8 prints out "WS3" and this transformed workspace.

If only a left half appears in a rule, as in line (9), an occurrence of "FLOWER" is found in the workspace, but no transformation is made, and the workspace remains unchanged.

Duplication

Items matched by the left half may be duplicated in the right half by mentioning them more than once. For example, rule 10 inserts another copy of "RED" into the workspace immediately succeeding the first occurrence, and rule 11 prints out the resulting workspace following the message "WS4".

Unknown Constituents

Sometimes only the context of a desired item is known. To locate such an item we need a notation for an unknown item. METEOR uses the symbol "($.1)" (that is, left paren, dollar, period, 1, right paren), which is similar to the COMIT notation. This

symbol represents any single unknown constituent. In rule 12, the "($.1)" is used to find the item immediately after "OF" in the workspace. Since 2 (which would refer to this item found by the left half) does not appear in the right half, this item after "OF" (i.e., the element "RED") is deleted by this rule.

Left half searches are made from left to right in the workspace: therefore "($.1)" can be used to find the first constituent in the workspace. Rule 13 finds this first constituent and inserts "QUESTION" immediately before it. Line 14 prints out the contents of the workspace after this transformation.

The notation ($.n) is used to represent n consecutive unknown constituents, where n may be any integer. Thus in rule 15, "($.2)" refers to the 2 constituents immediately before "FLOWER" and ($.3) to the 3 constituents just after. This rule rearranges these constituents and the modified workspace is printed out by the rule in line 16, following the message (WS6).

For an unknown number of constituents the symbol "$" is used. In 17 the "$" will match all items of the workspace between "FLOWER" and "ROSE". The right half specifies that these items are to be deleted. The result is shown by the output produced by 18, following the WS7. Since "$" will match any number of constituents, including 0 (zero), it can be used alone in the left half to match the entire workspace, and for example, as in 19, used to replace the entire workspace by a new string.

Line 20 contains a slightly more complex rule which moves the element after "START" to the position before "D". The result is printed out by the rule in line 21.

Line 22 is a standard rule to terminate a METEOR program. A METEOR program will come to a normal halt if it executes a rule for which there is a left half match ($ always gives such a match) and whose "go-to" is the atom "END". If no such rule ends the program an error occurs, METEOR complains, and prints out an error statement.

Comments Inside a METEOR Program

The list to the right of the second "*" in some rules are comments which are ignored by the interpreter. However, they take valuable space within computer storage (they are read in) and thus comments should be used sparingly if space is tight.

Flow of Control

In the program shown in Figure 1 the rules were executed sequentially, and no rule was executed more than once. However, repeated operations can be done by a single rule. In order to apply

the same rule to the workspace several times, we give this rule a name, which we write instead of the left-hand "*". The name of the first rule in the program in Figure 3 is CHANGE. This name is also inserted instead of the right-hand "*". As long as the left half finds a match in the workspace, the transformation indicated by the right half will be made, and control goes to the rule named by the atom replacing the right hand "*", in this case CHANGE again. The workspace is searched from left to right each time the rule is entered. CHANGE finds the first occurrence of "A ROSE", changes it to "THE FLOWER" and repeats. When there are no more occurrences of "A ROSE" in the workspace, the left half "fails", no transformation is made and control goes to the next rule in sequence. RULE1 is this next rule. RULE1 tests to see if there is an occurrence of "FLOWER" in the workspace. If there were not, RULE2 would be interpreted next. However, in this case, since we have just inserted into the workspace several occurrences of "FLOWER", the left half succeeds and control goes to RULE3.

RULE3 tests for the occurrence of "ROSE" in the workspace, and if there were such an occurrence, control would go back to CHANGE. There are none in this case and control passes to the next sequential rule — in line 5 (an unnamed rule).

It is sometimes desirable to reverse the normal flow of control, that is, to go to the next rule when the left half finds a match and the transformation is made, and conversely, to go to the rule specified in the "go-to" on failure of the left half. For example, this allows the program to exit from the middle of a multi-rule loop on failure of a condition. This type of flow is indicated in line 5 and line 6 by the "*" which is the second element of the rule. Since line 5 is this reverse-flow type of rule, and since there are no occurrences of "ROSE" in the workspace, the left half fails and control goes to RULE2, which prints out the workspace, and terminates the program.

To review, normal flow is to the specified rule on left half success, and to the next sequential rule on left half failure. If the second element of a rule (immediately following the name or first "*") is a "*", this flow of control is reversed. "*" in the go-to specifies the next sequential rule.

FIGURE 3

```
0.  METEOR((
1.  (CHANGE ( A ROSE) (THE FLOWER)  CHANGE (FLOW OF CONTROL))
2.  (RULE1   (FLOWER   )              RULE3  )
3.  (RULE2 * ((*P WSP))   END)
4.  (RULE3    (ROSE)                  CHANGE)
```

```
5.  (* * (ROSE)    (FLOWER)          RULE2)
6.  (* *   ( (*P WSEND))    END)
7.  )(A ROSE IS A ROSE IS A ROSE))
```

a. Program

```
(WSP)
(THE FLOWER IS THE FLOWER IS THE FLOWER)
```

b. Printout

Temporary Storage (Shelves)

In the example in Figure 3, each time the rule CHANGE is entered, the entire workspace is searched. Successive searches can be made more efficient by removing material already searched, and placing it on a temporary storage area called a shelf. The program in Figure 4 stores material in one such area called "SHELF1".

Temporary storage on these shelves is controlled by instructions in the "routing" section of the rule; the routing section is a list starting with the atom "/". The routing instruction in line 1 of Figure 4 queues onto the right end of SHELF1 (in a first-in first-out list) the items associated with 1 by the left half (i.e., the ones matched by the $), and the word "PRETTY". This is repeated as many times as the left half match succeeds.

When control is finally transferred to the rule in line 2 of Figure 4, All the material on this shelf is re-inserted into the workspace. It is called in by the list "(*A SHELF1)" in the right half of the rule. A "(*N SHELF1)" would retrieve the Next or first item only of this shelf.

Any type of item which can be inserted into the workspace by a right half rule can be put onto a shelf through a routing instruction. This is illustrated by the rule on lines 7 and 8.

Another type of routing instruction is illustrated in line 2. This "*D" routing instruction makes RULE3 the "go-to value" of PNTRET. Thus after transferring to PRNTWS, and interpreting this rule, the interpreter treats the go-to "PNTRET" as if it were "RULE3". This "go-to value" for PNTRET is reset again in line 8. Setting the "go-to value" of a return allows easy access and return from standard routines such as print routines or read routines.

Figure 5 shows a method used for communicating with recursive subroutines. The "$" in the go-to causes the first element of the

FIGURE 4

```
0.  METEOR((
1.  (CHANGE    ($ ROSE) (FLOWER) (/ (*Q SHELF1  1 PRETTY)) CHANGE)
2.  (*    ($)   ((*A SHELF1) 1)   (/(*D PNTRET RULE3))   *)
3.  (PRNTWS * ((*P THE WORKSPACE IS))    PNTRET)
4.  (RULE2  ($)                          END)
5.  (RULE3 (($.1) ($.1)) 0 (/(*S ODD 1) (*Q EVEN 2) (*D PNTRET RULE3))
6.        PRNTWS  (THIS IS A CONTINUATION OF THE PREVIOUS CARD))
7.  (*     ($)   ((*A ODD)   (*N EVEN))   (/ (*Q ODD (*N EVEN) ONLY)
8.        (*P ODD EVEN)   (*D PNTRET RULE2))     PRNTWS)
9.  )(A ROSE IS A ROSE IS A ROSE))
```

a. Program

```
(THE WORKSPACE IS)
(A PRETTY FLOWER IS A PRETTY FLOWER IS A PRETTY FLOWER)
(THE WORKSPACE IS)
(FLOWER IS A PRETTY FLOWER IS A PRETTY FLOWER)
(THE WORKSPACE IS)
(A PRETTY FLOWER IS A PRETTY FLOWER)
(THE WORKSPACE IS)
(FLOWER IS A PRETTY FLOWER)
(THE WORKSPACE IS)
(A PRETTY FLOWER)
(THE WORKSPACE IS)
(FLOWER)
(SHELF ODD CONTAINS (IS ONLY))
(SHELF EVEN CONTAINS (PRETTY IS PRETTY))
(THE WORKSPACE IS)
(A FLOWER A FLOWER A PRETTY)
```

b. Printout

FIGURE 5

```
METEOR((
(*  (($.1))      $)
(BLAH ($)  END)
(RULE ($)  END)
)(RULE BLAH))
```

FIGURE 6

```
0.  METEOR((
1.  (DEAL  ($1  $1)  ((FN ADD1 1)) (/(*S * 1 2)) *)
2.  (* *   ($2)                PRINT)
3.  (*   (5)  ((QUOTE 1))       DEAL)
4.  (*   ($)                    DEAL)
5.  (PRINT  ($)  (/(*P /))       END)
6.  )(1 H1 H2 H3 H4 C1 C2 C3 C4 D1 D2 D3 D4 S1 S2 S3 S4))
```

a. Program

```
( 4  (S4 D4 C4 H4) 3 (S3 D3 C3 H3) 2 (S2 D2 C2 H2) 1 (S1 D1 C1 H1))
```

b. Printout

workspace (not its subscript as in COMIT) to be taken as the "go-to" for this rule. If a stack of returns is stored on a shelf (in a pushdown list) and inserted into the workspace, successive returns can be made from recursive subroutines. This of course leaves a residue (the return) in the workspace. In Figure 5, control would pass from the first rule to the one labeled RULE.

Figure 6 is the listing and printout from a program which "deals cards" onto shelves 1, 2, 3 and 4, starting with the "deck of cards" in the workspace. Note that integer 1, which is the first element of the workspace, is not a "card" but will be used as an indicator in shelving (this is explained in more detail below).

In the first rule "$1" has been used instead of the "($.1)" previously described. Strictly as a typing convenience, because they are used so often, $1, $2, and $3 have been made special symbols which are interpreted as ($.1), ($.2) and ($.3) respectively. Thus the first rule matches the first two elements of the workspace. The right half of this rule indicates that these two elements are to be transformed into a single element which is the result of applying the LISP function ADD1 to the first element matched by the left half, which is, the first time, the number 1, and, the second time, the number 2. The LISP function is tagged by a "FN" for the METEOR interpreter.

In the routing instruction, the *S stores a card, the second element matched (labeled 2), onto a shelf. The shelf name is specified indirectly. The "*" immediately following the *S indicates this indirect addressing. The element of the workspace indicated by the "1" following the "*" specifies the shelf name; that is, the shelf name will be the number in the workspace which is matched by the first $1 in the left half.

The rule on line 3 illustrates the use of a quoted number in a right half. A left half match will be found for this rule if 5 is the first element of the workspace. If the match is found, the 5 will be replaced by the numeral 1. If 1 were not quoted, this "1" in the right half would be interpreted as the first element matched in the left half, which is not the transformation intended.

In the routing section of the rule labeled "PRINT", the routing instruction "(*P /)" causes the contents of all shelves to be printed, as shown in Figure 6b. The contents of shelves 4, 3, 2, and 1 immediately follow the label in the printout.

FIGURE 7

Full Printout Given by LISP for a Sample METEOR Program

```
FUNCTION EVALQUOTE HAS BEEN ENTERED, ARGUMENTS..
METEOR
(((* DICT (BOY ((BOY / NOUN HE)))
          (GIRL ((GIRL / NOUN SHE))))
(LOOKUP ((WORD ($.1))) 0 (/ (*Q SENT (FN GETDCT WORD DICT))
     (*P SENT)) LOOKUP)
     (* ($) ((*A SENT)) END))
(THE BOY AND GIRL))

(SHELF SENT CONTAINS (THE))
(SHELF SENT CONTAINS (THE (BOY / NOUN HE)))
(SHELF SENT CONTAINS (THE (BOY / NOUN HE) AND))
(SHELF SENT CONTAINS (THE (BOY / NOUN HE) AND (GIRL / NOUN SHE)))

END OF EVALQUOTE, VALUE IS..
 THE (BOY / NOUN HE) AND (GIRL / NOUN SHE)))
```

FIGURE 8

```
 FUNCTION EVALQUOTE HAS BEEN ENTERED, ARGUMENTS..
METEOR
(((* ((BOY / NOUN SING)) ((*/ AND 1 (DOG / NOUN MALE))
(*/ OR 1 (BOY / SMALL MALE)) (*/ SUBST 1 (MAN / MALE)))
END)) (THE (BOY / NOUN SING SMALL) AT HOME))

END OF EVALQUOTE, VALUE IS..
(THE (BOY / NOUN) (BOY / NOUN SING SMALL MALE) (BOY / MALE) AT HOME)
```

FIGURE 9

```
METEOR
(((* (($.1) IS ($.2) $ THERE) ((*K 1 2 3 4)) END))
(WHO IT IS AT MY DOOR IS THERE NOW))

END OF EVALQUOTE, VALUE IS..
(WHO (IT IS AT MY DOOR IS) NOW)
```

FIGURE 10

```
METEOR
((( * (IS ($.1)) (1 (*E 2)) END))
(IS (ANYBODY AT HOME) NOW))

END OF EVALQUOTE, VALUE IS..
(IS ANYBODY AT HOME NOW)
```

Dictionary Rule and Retrieval

Figure 7 illustrates a type of rule which stores definitions of words for very fast, hash-code retrieval. This type of rule is indicated by a second element which is atomic (not a list) and which is not "*". The remainder of the list is interpreted as a list of associated pairs, and retrieval done by the LISP function GETDCT obtains the second member, given the first. The retrieval method is very fast and new definitions can be added at any time. In this program the "(*P SENT)" causes the printout of the contents of the shelf SENT.

Figure 8 illustrates the use of subscripted (labeled) atoms and the three modes by which subscripts can be merged; i.e., intersection, union, and substitution. The "*/" indicates that subscripts of the constituents should be merged.

The program in Figure 9 shows how several elements of the workspace can be "compressed" (by the *K in the right half) so as to be treated as a single item, a list of these elements. The program in Figure 10 performs the inverse of this compression operation and expands (with the *E) a list which is a single item in the workspace, and brings the elements of this list to the "top level" of the workspace.

A list of characters can be compressed (with a *C operator on the right) to form a single atom whose print name is this string of characters. An atom can be expanded into a list of its characters by a *E operation in the right half. These operations are illustrated in the two programs in Figure 11.

This has been a very brief survey, by example, of some of the types of operations that can be done within a METEOR program. The remainder of the report gives exact specifications for the program.

III. Specifications for a METEOR Program

METEOR is a LISP function of two arguments, RULES, a list of the transformation rules to be applied, and WORKSPACE, the list

or string to which these transformations are to be applied. The flow of control from one rule to another will be described below. Figure 1 is an example of a METEOR program, and its use under the LISP system.

FIGURE 11

```
METEOR
(((*(($.1))  ( (*E 1 ))    END))   (  GARBAGE PILE))
END OF EVALQUOTE, VALUE IS..
(G A R B A G E PILE)

METEOR
((( * ( ($.1 ) ($.2))   ( (*C 1 2))   END)) (F O O ON ME))
END OF EVALQUOTE, VALUE IS..
(FOO ON ME)
```

FIGURE 12

```
(RULE1  *  ( ($.1) $ ) (2 1)  (/ (*S SHOT 1))  NEXT  (COMMENT) )
```

Rule name	Go-to reverse	Left-half pattern	Right-half	Routing Section	Go-to	comment

A. A METEOR Rule

An individual METEOR rule is a list of not less than two nor more than seven elements. An example of a complete rule is found in Figure 12. The first element of the rule is called the "rule name", and it must be present. It must be a LISP atom, and is either the atom "*" or a unique atom (i.e., no other rule may have this name).

The second element of the list is optional. It too may be "*", a "*T", a "*U", or another LISP atom. If it is "*", normal flow of control is reversed. Normal flow of control in a METEOR program is as follows:

1. If a match is found between the pattern element of the rule and the workspace, control is passed to the rule specified by the atom in the "go-to" section of this rule.

2. Otherwise control passes to the next rule in the list RULES.

If this second element is a *T, tracing is turned on; that is, before any rule is executed, the contents of the workspace and this rule are printed out. This continues for every rule every time it is executed until a rule whose second element is *U is encountered. This *U turns off the tracing. Encountering *T in trace mode, or a *U in nontrace mode has no effect.

If the second element of a rule is a *M, then this rule will be traced each time it is encountered but general tracing will not be turned on. This means that when the rule is encountered, the workspace is printed out followed by a printout of the rule and then followed by a printout of the resulting workspace if the workspace has been changed.

If the second element of the rule is an atom other than "*", "*T", or "*U", for example "DICT", the remainder of the rule is interpreted as a list of dictionary entries to be made. When the entries are made, control will automatically pass to the following rule. The list of dictionary entries is a list of pairs which is used as an argument for the LISP function DEFLIST. The first member of each pair must be an atom and the second a dictionary entry for this atom. The dictionary entry is stored permanently (for the entire LISP run) on the property list of the indicated atom. The element which introduced the dictionary rule, in this case "DICT", is used as a flag to mark this entry on the property list of the atom. Thus several dictionary entries with different flags can be made for a single atom, and each may be retrieved later (by the function GETDCT, described below). Retrieval from the dictionary is very fast because LISP uses a hash-coded "bucket sort" on readin to find the property lists of atoms, and there is immediate access to these property lists when dictionary look-up is performed.

The third element of the rule is a pattern statement, which is used to select relevant items from the workspace. This third element must be present, and must be a list. If the workspace "matches" this pattern (how a match is achieved is described below), the rest of the rule is interpreted. If not, immediate transfer is made to another rule.

The fourth element of the rule is the atom 0 (zero) or a list, which describes the transformed workspace. This element is optional. If it is not present, the workspace remains unchanged.

The fifth element is an optional list which is identified by its initial element, the atom "/". This list is called the rout-

ing section of the rule, and it controls temporary storage of data and multiple branching of flow of control.

The sixth element is the "go-to" section and specifies to which rule in the list RULES control will pass. It must be an atom which is the name of some rule in the program, one which has been given a "go-to value" in the routing section of some previously executed rule, or the atom "END". If this sixth element is not present, it is assumed to be "*".

The seventh element is an optional list ignored by the interpreter. It may be present only if the sixth element is present. Since it is ignored it may be used to insert comments on the program.

B. The Pattern Section of a METEOR Rule (the "left half")

This section, the third element of the rule list, is a list of patterns which must be matched in the workspace. A match is achieved if each of the individual patterns matches some element or elements in the workspace. These matched elements must be in the same order as the patterns appearing in the list of patterns and form a single contiguous substring in the workspace. Search is done from left to right, and the first match obtained is used.

The LISP function which obtains the match is called COMITMATCH. It is a function of two arguments, RULE and WORKSPACE. RULE is the list of patterns to be matched in the list WORKSPACE. Each pattern is associated with a name and if a match is achieved, the value of COMITMATCH is a list of pairs containing the name and the substring of the workspace matched by the pattern corresponding to this name. If no match is achieved, the value of COMITMATCH is NIL.

B1. Direct Match

If an element of the pattern list is an atom, it will match the first identical atom in the workspace. It will also match an item in the workspace which is a list, but whose first element is this atom and whose second element is "/". This latter match is useful if one wishes to label atoms in the workspace by attaching "subscripts" to them. COMITMATCH will match subscripted and unsubscripted items. The usual form for a labeled atom is a list of the form

(atom / subscript1, subscript2...subscriptk)

For example, the atom "BOY" as a pattern element will match the list "(BOY / NOUN SINGULAR)" appearing in the workspace. However "(BOY / NOUN SINGULAR)" as a pattern element will not match "BOY"

in the workspace. (See section B3.) This type of direct match can be done for any list structure which can be considered a single element. For example, "(A B C)" will match, as a single element in the workspace, the list "(A B C)". If the workspace were (M N (A B C) P), a match would also be found for the sublist (A B C).

B2. "Dollars" Match

The pattern word "($.1)" will match any single element of the workspace. In general, the form "($.n)", where n is any integer greater than 0, will match n consecutive constituents of the workspace. The atom "$" alone will match an indefinite number of elements of the workspace, including zero. Thus if "$" is the only member of the pattern list, there will always be a match, even if the workspace is null (empty).

As mentioned, $1, $2, and $3 are accepted abbreviations for ($.1), ($.2) and ($.3); but $4, $5, ... will not be understood by the METEOR interpreter.

The symbol $0 has a special function in the left half of a rule. If it is the left-most element of a rule, a match will only be found if the succeeding elements are at the left end of the workspace. For example, the left half ($0 A) will only succeed in matching if the first element of the workspace is an A. If the workspace were (B A C) this left half would fail.

Similarly, if $0 is the last symbol in a left half, the match will succeed if the immediately preceding symbols are matched at the right end of the workspace. For example, the A in the left half (A $0) will match only the last A (underlined) in the workspace (A B A C A). The search is still made from left to right.

There are four other special "dollars" matches. The left half element ($. ATOM) will match any atomic (non-list) element. A subscripted element is a list, for example. Conversely, the left half element ($.LIST) matches any non-atomic (list) elements. The pattern element ($.NUMBER) will match any element which is a number. Finally, the left half element ($. item) where item is anything except "ATOM" or "LIST", will match any workspace not identical to item. For example, the left half (A ($.B)) will match the first occurrence of an A not followed by a B; that is, in the workspace (C A B A A D), the A in the left half will match the second A in the workspace.

B3. Subscript Match

A pattern word of the form

(element1 / subscript1 subscript 2...)

will match a constituent in the workspace which has the same first element, then "/" and then a list of subscripts which include those mentioned in the pattern word. Additional subscripts may be present in the workspace element. Order of the subscripts is unimportant.

The first element "element1" of this pattern word may be either an atom which must be matched exactly by the workspace item, or be $1, or "($.1)", which will match any element. One can thus find a word which has specified labelling (subscripting) without knowing the words itself. For example, "($1 / NOUN)" will match any element in the workspace labeled as a noun, such as "(PLATO / MAN NOUN GREEK)".

B4. Names for Left-Half Elements

Each individual pattern in the left half is associated with a name. When a match is found for this individual pattern, the matched portion of the workspace is paired with the associated name. The names associated with individual patterns in the pattern list are successive integers, i.e., "1" with the first pattern element, "2" with the second, etc. For example, in Figure 12 the name associated with the "($.1)" is "1" and with BE is "2". If a match were found for this pattern, 1 would be associated with the element in the workspace immediately preceding "BE", and "2" would be associated with the occurrence of "BE" in the workspace.

B5. Matching with Left Half Names

Matching with the left half pattern is done from left to right in the workspace. To determine if an element appears twice in the workspace, a match can be found for a single element, and the name associated with this matched element can be used later in the left half to obtain a second match for this element. For example, if the left half pattern were "($1 BE $ 1)", the "$1" would match the word preceding "BE" in the workspace, and the fourth element of this pattern would match a later occurrence of this same word. The first occurrence of the word would be associated with the name "1", and the second occurrence with the name "4". If the workspace were "(THIS COULD BE THE WORD COULD)" then the left half pattern above would match the workspace, and the list of associated pairs would be: ((1 COULD) (2 BE) (3(THE WORD)) (4 COULD)).

B6. LISP Functions for Matching

In addition to these standard patterns for matching, LISP functions can be used to determine a match. These may be functions of any number of arguments, where the first argument is the workspace, and the rest are items found previously by the match. This function is used in the left half pattern in the following format:

(FN function name1, name2,..., namek)

"FN" is the signal used by the interpreter to mark this type of function match. For "function" one may insert the name of a function previously defined in LISP. Name1,..., namek are names associated with elements previously matched in this left half pattern. If there are k such names, the defined function must have k+1 arguments. When matching, the first argument will be the remainder of the workspace (the part not yet used in the left half match), and the other arguments are previously matched elements of the workspace associated with name1,..., namek. The value of this function should be NIL if no match is found. It should be cons[list[m]; w] if there is a match, where m is the portion of the workspace that is matched, and w is the remainder of the workspace past the matching elements. Figure 13 illustrates the definition and use of a match function. CARMATCH will find a match if the first element of the remaining workspace is the same as the first list element of the list whose name is given as the second argument of CARMATCH. In use, the unmatched portion of the workspace is the implicit first argument of CARMATCH.

FIGURE 13

Function Definition

```
DEFINE((
     (CARMATCH (LAMBDA (WKSPACE LT) (COND
          ((EQUAL (CAR LT) (CAR WKSPACE))
           (CONS (LIST (CAR WKSPACE)) (CDR WKSPACE)))
     (T NIL))))
      ))
```

Rule

```
(* ($1  (FN CARMATCH 1)) (2 1) *)
```

B7. Printing the Workspace

If "((*P message))" is used on the left side, no match will be found, i.e., the left half will always fail, but the "message" will be printed out, followed by a printout of the workspace. This is useful in debugging, and for obtaining intermediate printouts. The workspace remains unchanged.

B8. Matching Special Characters

Seven characters will not be seen as characters by the LISP read routine because they have syntactic meaning within LISP. They are:

( ) , - + . [blank or space]

Thus they cannot appear in METEOR rules. These characters will be read in however by the METEOR reader, and can appear in the workspace (see below). To write a rule which tries to match one of these characters in the workspace — or to insert such characters in the workspace — use the following lists respectively "(* LPAR)", "(* RPAR)", "(* COMMA)", "(* DASH)", "(* PLUS)", "(* PERIOD)", and "(* BLANK).

The inner workings of METEOR are as follows for this case, for those who are interested. The second item in a list started with a "*" is evaluated by the LISP function EVAL. The workspace is then matched against the correct unspeakable item, or said item is inserted into the workspace — whichever is appropriate. The "*" in this case is acting as an "unquote" operator. The * operator can be used on the right to cause evaluation of arguments of a function. Normally arguments of functions tagged with FN are not evaluated.

C. Right Hand Side (Transformed Workspace)

If a match is obtained by the left half pattern, the interpreter then uses the right half (the fourth element of the rule) to determine in what way the workspace is to be transformed. Only those elements of the workspace utilized in the left half match can be affected by the right half transformation. If the right half is the atom "0" (zero) then all those items which were matched will be deleted from the workspace.

If the right half is not the atom "0", then if the right half appears at all, it must be a list. If the right half is not present in a rule, the workspace remains unchanged.

The right half is a list of elements. Each element will be replaced by the item or items it names, and the resulting string put in the workspace in place of the substring matched by the left half. The only way to affect the contents of the workspace is through the right half rule. This differs from COMIT, which allows additions to and deletions from the workspace under control from the routing section. This is not allowed in a METEOR program. All additions from temporary storage "shelves" are done directly from instructions in this right half.

C1. Substitutions, Insertions, and Rearrangement

The names which appear in the right hand side list can be of several types. The first is a name associated with a matched element. Its value is its matching substring in the workspace. The same name may appear any number of times on the right hand side. Thus elements of the workspace may be duplicated. The names may appear in any order, and thus elements can be rearranged.

The second type of name is an atom or list which is not a name of this first type, and not a list beginning with one of the seven control characters "*K, *C, *E, */, *N, *A, *W". Such an element is a name for itself, and will be inserted directly into the workspace; an atom will be inserted and a list will be concatenated in the workspace in the position in which it appears in the right half list. For example, see Figure 14 below:

FIGURE 14

Rule

(* (DID $1 GO) (2,DOES,3) *)

Workspace

Before	(DID HE GO HOME TODAY)
After	(HE DOES GO HOME TODAY)

The element associated with the name "2" (i.e., "HE") becomes the first element, the atom DOES is then inserted, and then the element associated with "3" is placed in the workspace. Since the name "1" is used on the left side but not on the right, the element paired with 1 is deleted from the workspace. Since "HOME TODAY" is not mentioned in the pattern, it is not affected and remains at the end of the workspace.

To reiterate, the names associated with matched left half patterns are the integers which specify their position in this pattern. To insert a given integer, say 3, in the workspace, quote it in the right half, i.e., use the element "(QUOTE 3)".

C2. Compression and Expansion

Sometimes it is desirable to compress several elements of the workspace into a single unit, or list. This is accomplished by using a right half element which is a list whose first element is "*K". The succeeding elements of this list are of any type

which can be found in a right half rule including other lists starting with "*K". These items are all placed in the workspace in a single list. Figure 15 (below) gives an example of this type of rule and its operation:

FIGURE 15

(* (WILL ($.1)($.1)) (2 (*K 1 3)) *)

Workspace

Before	(WHERE WILL HE STAY TONIGHT)
After	(WHERE HE (WILL STAY) TONIGHT)

An item on the right may be of the form:

"(*C namel name2...namek)"

where namel,...,namek identify elements which are single characters. This entire item on the right side will be replaced by an atom whose print name is the list of characters specified. An error will result if one of the elements specified is not a single alphanumeric character.

The inverse of these compression operations can be performed by a list of the form

"(*E namel)".

If namel specifies an atom, then "(*E namel)" will be replaced by a list of the letters in the print name of this atom. If namel specifies a list, the list will be concatenated into the workspace. Figure 16 gives an example of this:

FIGURE 16

(* (IS $1) (1(*E 2)) *)

Workspace

Before	(WHAT IS (A METEOR PROGRAM))
After	(WHAT IS A METEOR PROGRAM)

C3. Reading and Writing Operations

Strings of atoms may be written out by an element in the right half which is a list of the form:

(*W namel name2...namek).

The last part of this list is treated as if it were a right half rule. When it is evaluated, it is a list. The "*W" then causes the atoms in this list to be printed out without spaces. If these atoms are individual characters, then any sequence of characters can be printed. Remember that special characters such as "(" and ")" etc., must be referred to by "(* LPAR)" and "(* RPAR)" etc. To end the printline the last atom in the string must be "EOR". Any characters in the string past the "EOR" will not be printed out. The value of this operator (*W namel...namek) is NIL. Thus although it appears in the right half, it contributes nothing to the transformed workspace. For any element of the list to be printed which is not atomic (i.e., a list), the string "***" will be printed.

Characters on cards may be read into the workspace (or onto shelves) by a list containing the single element *R; the list "(*R)" in a right half will be replaced by a list of the characters on the next card read. The card image will be read from tape if tape input is used. The list will end after the last non-blank character, and is a maximum of 72 columns. The first 72 columns are read from a card. If the card is blank this list will be empty. If an "end of file" is encountered, the element EOF will be inserted into the workspace.

For example, the rule:

(* ($) ((*R)) *)

will replace the contents of the workspace by a list of characters on the next card. If the card is blank, the workspace would then be the null. The rule

(* ($) (1 (*R)) *)

will concatenate this input list onto the right end of the workspace. As mentioned (* BLANK) will match blanks read in by this read operation, etc.

C4. Additions from Temporary Storage

Sometimes it is convenient to remove temporarily certain elements from the workspace. For example, while doing certain long searches, previously searched items may be placed on temporary storage areas called shelves (from the COMIT terminology). Material is placed on the shelves through instructions given in the routing section of a rule. Items can be returned from these shelves and deposited in the workspace by instructions in the right half rule.

A list in the right half consisting of the elements "*A" followed by a shelf name will be replaced by the entire contents of the shelf, and the shelf will be emptied. For example, "(*A SH1)" in a right half rule will bring to the workspace the entire contents of shelf named "SH1".

A list of the form "(*N SH1)" will bring into the workspace the next (first) element of this shelf, i.e., the first element of the list which this shelf contains. This first item is removed from the shelf by the operation. If the shelf is empty a "(*N SH1)" or "(*A SH1)" is ignored. These operations are performed from left to right, so that a right half ((*N SH) MK (*A SH)) would place the first item from shelf SH before the MK and the remainder after.

C5. Subscript Combination in the Right Half

An item of the workspace may have a subscript, or subscripts for labeling purposes. The format for such subscripted items is:

(atom / subscript1,...,subscriptk).

An item may have its subscripts modified by operations in the right half. Modification is controlled by elements in the right half which are lists starting with the atom "*/". The second item of the modifier list specifies the method of combination for two sets of subscripts. The specifying item may be one of the atoms, "AND", "OR", or "SUBST". The third and fourth elements, s_3 and s_4, of the modifier list are the names of the items whose subscripts are to be merged. The third item may specify an atom in the workspace or a subscripted element, but the fourth must name an element which is a list whose second element is "/". An example of a modifier list which could appear in a right half is "(*/ OR BOY (MAN / MALE NOUN))". The resulting element in the workspace would be "(BOY / MALE NOUN)". If the second element in the modifier list is "AND", the subscripts of s_3 and s_4 will be merged by logical conjunction. If the intersection of the two subscript sets is empty, the resulting item would have no subscripts, and the item is made atomic, e.g., "BOY" instead of the list "(BOY /)".

If the second item is "OR", logical disjunction is used to combine the subscripts of s_3 and s_4. If the second item is a "SUBST" the subscripts of s_4 are substituted for those of s_3. Other methods of combination can easily be added to the program by modifying the LISP function SBMERGE. Figure 8 gives an example of the three types of subscript modification.

In COMIT subscripts of two elements are merged by a special process. The intersection of the two sets of subscripts is found.

If this intersection is not empty, it is used as the subscript of the resulting element. If the intersection is empty, the subscripts of the second element are used with the "head" of the first. This merging operation can be performed in METEOR by using a second element "MERGE", instead of "AND" or "OR". This merging operation is also performed if an element such as (2 / BOY TALL) appears on the right. The subscripts of the element named 2 are combined by a "MERGE" with an element whose only subscripts are BOY and TALL. If 2 were (TOM / BOY PERSON), then the element resulting from the merge would be (TOM / BOY). If 2 were (TOM / PERSON), the resulting element would be (TOM / BOY TALL).

D1. The Routing Section of a Rule

The routing section is a list whose first element is the atom "/". Each subsequent element is a list, called a routing instruction, which begins with one of the atoms "*S", "*Q", "*X", "*P", or "*D". The next element after "*S", "*Q", OR "*X" in a routing instruction names a shelf to be used. For example (*S SH1 2) will store on the shelf SH1, the items which are named by 2. Except for the atom "*", shelf names may be any LISP atom, including numbers or atoms already used for rule names. Shelves and their contents are stored as pairs of the form (name, contents) on a list associated with the free variable SHELF. Each time a shelving operation is done, a search is made through the list SHELF. Thus shelving operations will be done more rapidly if fewer shelf names are used.

"*S" will cause items to be stored on the front (beginning) of the shelf named. The item last stored by a "*S" instruction will be the first obtained by a "*N" item in a right half instruction. A shelf built up by "*S" instructions is a push down list with the last item in first out.

A queue (a first-in first-out list) can be built up using the "*Q" routing instruction. "*Q" queues items onto the "back end" of the shelf named.

The name of the shelf to be used may be specified directly as previously indicated or may be obtained from the workspace. If the second element of the routing instruction is "*", then the third item is the name of a workspace item (the name as found by the match routine). This workspace item is the name of the shelf to be used. (This differs from the COMIT method for indirect addressing. COMIT uses a subscript of the workspace item to specify the name of a shelf indirectly. METEOR may be easily changed in this respect by changing the function named "INDIRECT".)

The items to be placed on the shelf named are specified by the remainder of the routing instruction. An item to be shelved may be described in any way used to describe an item to be put into the workspace by a right half rule. In fact, the same LISP function (COMITRIN) is used both to collect on a list the items to be shelved, and to arrange the transformed portion of the workspace. Copies of material in the workspace, new atoms, and material from other shelves may be placed directly onto a shelf in the same way that they are placed into the workspace. This is again different from COMIT, where all items to be shelved must be moved through the workspace.

A routing instruction starting with the atom "*X" will exchange the contents of the shelf named with the contents of the workspace. Remember that names used in the routing instruction in this rule still refer to names associated with items found by the left half match with the original workspace.

A routing instruction starting with the atom "*P" will print out the message (SHELF s CONTAINS c) for each shelf "s" named in the list following the "*P". Only direct addressing of shelves is allowed. "c" is the contents of the shelf named. This instruction is useful in debugging programs.

As an example, "(*P SH1 17 3.2 *Q)" in the routing instruction of a rule will print out the contents of the shelves named SH1, 17, 3.2, and *Q, in that order. (Note the variety of allowable names.) "(*P /)" prints out the contents of all shelves.

D2. The Dispatcher Instruction in the Routing Section

The instruction "*D" in a routing instruction is the only one which actually affects the routing of control in the program. "*D" must be followed by two atoms. This pair of atoms is put as a pair on the list DISPCH. This list is searched each time a name is found in the "go-to" section of a rule, and the second member of the pair substituted for the first. If the first atom were, for example, BRANCH, and the second were RULE1, then each time BRANCH occurs in the "go-to" section of a rule it is interpreted as a "go-to" to the rule named RULE1. Thus, "*D" provides a method of setting a switch on an n-way branch. It may be reset at any time to any value. Any atom in the "go-to" section for which no value has been set by a "*D" routing instruction is assumed to be its own "go-to" value.

E. The Go-To Section of a Rule

The sixth (optional) element of a rule is an atom which tells which rule is to be used next. If it is the atom "*" (or is absent) control passes to the next rule in the list RULES. If it is an atom, which has not been given a "go-to" value by a "*D" ins-

truction in some routing instruction, then this atom is the name of the next rule to be used. If it has been paired with another atom by a "*D" instruction, this paired atom is the name of the next rule interpreted. If this atom is "$", the first item in the workspace is used as the name of the next rule executed. (This latter differs from COMIT; COMIT uses a subscript on the first element.) Normal flow of control is to the rule named by the "go-to" of a rule, if a match is achieved by the left half. If no match is achieved, the next rule, in the list RULES, is used. However, if the second item of a rule is "*", this normal flow pattern is reversed. No match implies transfer of control to the rule specified in the "go-to", and a successful match transfers control to the succeeding rule. If there is a match the transformation given by the right half of this rule and its routing instructions are always interpreted before control is transferred.

F. Comment Field

The seventh (optional) section is a list which may contain any comments on this rule or the program, etc. It is ignored by the METEOR interpreter. These comments will use up space within the computer and therefore should be used sparingly.

IV. Warnings and Advice

METEOR is based on COMIT and written in LISP and has foibles for both. The LISP system is built on the parenthesis and is very stubborn about having the correct number of parentheses in the proper place. Thus for the want of one parenthesis or a pair, an entire METEOR program may fail. Following is a list of warnings and advice:

1. The left half of a METEOR rule must be a list of patterns. If the only element of this left half is a list, for example ($.1) or (*P THE WORKSPACE IS), make sure these elements are enclosed in parentheses, i.e., the left halves are respectively "(($.1))" and "((*P THE WORKSPACE IS))".

2. A similar warning holds for the right half. If the only element in the right half is, for example, "(FN ADD1 2)", then the right half is "((FN ADD1 2))" (note the number of parentheses). To perform a deletion, however, using "0" (zero) as the right half, this zero must not be enclosed in parentheses. This is the only exception to the rule that the right half is a list.

3. Check the number of parentheses at the end of the routing section. Remember this section is a list of lists.

4. Remember that "(*R)" not "*R" reads in a card, i.e., "(*R)" is a list of one element. Thus a right side containing just a "(*R)" must be "((*R))".

5. Comments may be used, but they take up space. If space is a problem, delete or shorten your comments.

6. A linear search through a list of shelves is made each time a shelf reference is made. The name of any shelf ever mentioned is put on this list. To speed searches, use fewer shelf names.

7. Since METEOR is a LISP function, it may be used recursively within itself. However, METEOR resets "SHELF" and "DISPCH" to NIL. To keep the same shelf contents and dispatcher settings, instead of using "(METEOR RULES WORKSPACE)", use "(METRIX2 RULES WORKSPACE)".

8. If METEOR is used within a LISP program, and not executed as a pair for EVALQUOTE, remember to quote the list of rules, because LISP evaluates arguments inside functions (this is because of a LISP peculiarity).

9. Best use of METEOR can be obtained by using it with the LISP system and expressing each operation in the language best suited for it. Remember LISP functions can call METEOR and vice versa.

10. Placing items on a shelf does not automatically remove them from the workspace. To remove matched items from the workspace, there must be a right half.

11. The LISP read program does not distinguish between commas and blanks; therefore, a "," and a " " (blank) are interchangeable in a METEOR program.

12. Another foible of the LISP read program is that "/" can appear as a character in the middle of an atomic symbol. Therefore, when using "/" to separate an element from subscripts, the "/" must have a blank (or a comma) on each side of it.

13. COMIT and METEOR differ in the following ways not previously mentioned explicitly: (a) COMIT subscripts have values. In METEOR an element can have subscripts, but values are not explicitly provided for. (b) A "*N" instruction is ignored (brings in a null element) if the shelf is empty in METEOR. It does not cause a rule failure in METEOR as it does in COMIT. (c) There is no random element available in any form in METEOR — as opposed to the random rule selection feature available in COMIT.

14. Recall that the atom "*P" is used in two different contexts. When used in the left half it is followed by a message to be printed to label the following workspace printout.

In the routing instruction *P is followed by a shelf name or series of shelf names, and the contents of these shelves will be printed. Followed by a "/", *P will cause the list "SHELF" to be printed. This is a list of pairs,.the first of each pair being the shelf name, and the second the contents of the shelf.

15. METEOR is presently buried in the LISP parenthesis system. However, using METEOR as a bootstrap, a read program can be built which can read and convert from any fixed-format parenthesis-free (or more free) notation to METEOR. Since they are kept symbolically within the LISP system, METEOR programs can be generated or modified at run time by a METEOR program (or LISP function) and then executed.

16. The function which assembles the right half of a rule, i.e., "COMITRIN", effectively strips one pair of parentheses from all non-atomic symbols named in the right half. Thus, since "FOO" is a name for itself, "COMITRIN" will treat both "FOO" and "(FOO)" in exactly the same way. If the workspace were "(ON YOU)", then both

```
"(* ($) (FOO 1) *)"
```

and

```
"(* ($) ((FOO) 1) *)"
```

would change the workspace to "(FOO ON YOU)", and both

```
"(* ($) (FOO FOO 1) *)"
```

and

```
"(* ($) ((FOO FOO) 1) *)"
```

change the workspace to

```
"(FOO FOO ON YOU)"
```

In setting up dictionary entries, care must be taken to provide the proper parenthesis level. For example, the pair "(TWICE (TWO TIMES))" as a dictionary entry, when inserted by "GETDCT" in "(ALMOST TWICE THE NUMBER)" will change it to "(ALMOST TWO TIMES THE NUMBER)", which is presumably what was wanted. Similarly, if "(BOY (BOY / MALE))" is entered into "(THE BOY WINS)" the result is "(THE BOY / MALE WINS)" which however is not what was intended at all. Always put one more pair of parentheses than you wish to appear in the workspace, around the second element of a dictionary pair.

17. The *E instruction expands an atom into a list of the characters in the print name of the atom. It concatenates into the workspace a list which is a single element of the workspace. However, note the following eccentricity; if the workspace were "(THE GOOD BOY)", the rule "(* (($.2)) (*E 1)) *)" would transform the workspace to "(THE BOY)". The *E operator, operating on a substring from the workspace (in this case the two elements matched by ($.2)) specifies the first element of this substring, and the others are deleted from the workspace.

18. I would appreciate hearing about any bugs or eccentricities found in the METEOR program, and any applications found for this language. My address is care of the publisher of this book.

Notes on Implementing LISP for the M-460 Computer

Timothy P. Hart and Thomas G. Evans

Air Force Cambridge Research Laboratories

This article describes the process used to implement LISP 1.5 on the Univac M 460 (an early military version of the Univac 490). This machine, which has been available to us on an open-shop basis, has 32000 registers of 30-bit, 8 microsecond memory. It has an instruction set which is quite convenient for LISP, e.g., it is possible to load an index register (of which, incidentally, there are seven) from either the left or right half of the word addressed by the same index.

The external language is compatible with LISP 1.5 for the IBM 7090. There are some unimplemented features (such as arrays and "$$" quoting in the read program) and some additional features.

Steps

The steps taken to produce this LISP were:

(1) Decide on internal conventions and write the garbage collector (at least mentally).
(2) Write all required machine-language subroutines (surprisingly few).
(3) Modify the S-expression version of the LISP compiler to produce code for the new machine

(in a list-structure form analogous to LAP) and use it to compile a LISP interpreter (this "bootstrap" compilation can be done with an existing LISP, in this case, the 7090 version).

Garbage Collection and Storage Conventions

The M 460 garbage collector (g.c.) packs active list structure into one end of the space allocated for list structure, abandoning a contiguous block. This property of the g.c. is taken advantage of in M 460 LISP to avoid allocating two separate areas of memory, one for list structure, and the other for a push-down block. Instead, one block is used for both these purposes, list structure being created from one end of this area and push down from the other.

There are no full words in this system. Numbers and print names are placed in free storage using the device that sufficiently small (i.e., less than 2^{10}) half-word quantities appear to point into the bit table area and so don't cause the garbage collector any trouble. A number is stored as a list of words (a flag-word and from 1 to 3 number words, as required), each number word containing in its CAR part 10 significant bits and sign. Thus an integer whose absolute value is less than 2^{11} will occupy the same amount of storage (2 words) as in 7090 LISP 1.5. Print names are stored as character-string lists (see below for a description of the character-handling features of this LISP), one character to a word.

At first it seemed that this scheme would be relatively costly of free-storage space. However an analysis of a typical object list gave about 530 words for print names stored in string form, versus 470 words in 7090 style packed form. The saving which could be made by packing the print names is offset by the simplicity of this method and by the drastic reduction in the size and complexity of the garbage collector which it allows.

The garbage collector has four phases:

(1) mark and count;
(2) list available cells;
(3) move and save new location;
(4) modify pointers.

Marking and Counting

The base pointers for marking are the address part of every compiler-produced program word and the address part of the push-down list. The addresses of program words may contain:

(1) references to other program locations such as jump addresses;
(2) small positive and negative constants (magnitude less than 100);
(3) references to quoted S-expressions and to value cells, just as in 7090 LISP 1.5. These are the ones which are significant for the marking phase. No QUOTELIST is needed.

Free-storage half words may contain any quantity except an address in the free storage area which is not really pointing to list structure (these would get marked, which isn't disastrous, and would be possibly modified at relocation time, which is disastrous).

Since a machine word holds exactly two machine addresses all garbage-collection marking is done in a table of bits.

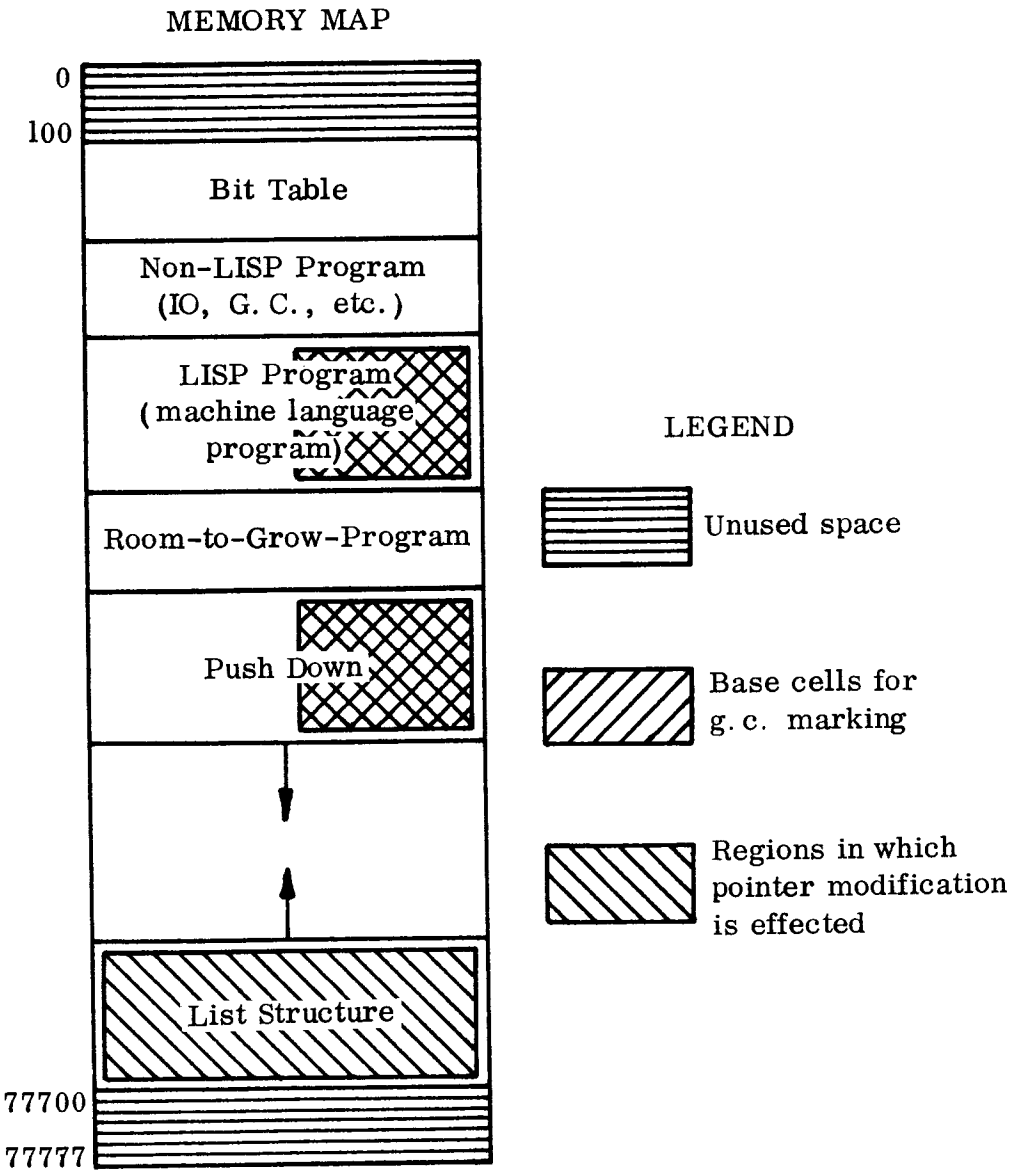
During the marking phase the number of active cells is counted.

Listing Available Cells

At this point during a garbage collection the list structure area whose length will be called l contains a active cells and $l-a$ unmarked cells, where a is the number of active cells counted during phase one. It is clear that if the active structure is packed toward one end of the list structure area, it will occupy the a cells nearest that end we will call the no-relocation area. Only the active list cells in the other part of the list structure area, called the relocation area, will need to be relocated. As a matter of fact, there must be exactly the same number of active cells in the relocation area as abandoned cells in the no-relocation area. During phase two a list is made of all the available cells in the no-relocation area.

Move and Save New Location

The relocation area is now methodically searched for active words. When one is found, its contents are moved to an empty


MEMORY MAP
0
100
Bit Table
Non-LISP Program
(IO, G. C., etc.)
LISP Program
(machine language
program)
Room-to-Grow-Program
Push Down
List Structure
77700
77777
LEGEND
Unused space
Base cells for
g. c. marking
Regions in which
pointer modification
is effected

cell in the no-relocation area (obtained from the list made during phase 2), and this new location is recorded in the old cell.

Modify Pointers

The references to words which have been relocated must now be updated. The address part of all LISP program and push down words, plus both left and right halves of all no-relocation area words are inspected for pointers into the relocation area. When one is found it is updated by replacing it with the new location, which can be found in the word in the relocation area where it points.

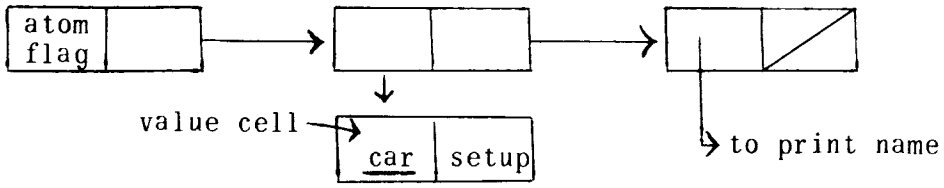
Subroutine Entry and Special Machine Language Subroutines

Arguments and the values of program variables and internal lambda variables are stored on the pushdown list. Every atom has a value word on its property list. Values of special variables are posted in the left half of the value cell just as in LISP 1.5. There are no SUBR pointers on property lists — the value cell is used for this purpose also, as suggested by Paul W. Abrahams. This means one probably won't get away with declaring, e.g., a special variable CAR. (Only one case of this trouble exists in the 7090 compiler; LENGTH is the name of both a function and a special variable.) In spite of this defect, this approach seems elegant: LISP does allow explicit binding of functionals in certain restricted syntactic positions — why not allow this generally?

The value of a function currently returns in the accumulator. It would be more sensible to have the value return in an index register except that this would constrain values to be of index-register size (15 bits) thus disallowing full-word answers; the ideal situation would be to have full-word length index registers as in the PDP-6.

Subroutines are called by a return jump indirect through the right half of the value cell on the property list of the routine's name (Note: return jump amounts to "deposit PC in Y and transfer to Y + 1", just like the jms instruction on the PDP-4). To explain our subroutine linkage scheme, we give an example:

The property list of CAR looks like this:



The left half of the value cell contains the address of car, the machine language subroutine for CAR; the right half contains the address of a machine language routine called setup, which does approximately the job of *MOVE in 7090 LISP 1.5.

When some program calls CAR, control goes first to setup, which by some fiddling can locate the address of CAR in the left half of the value cell. CAR contains some parameters in its heading which setup can use to do the following:

(1) Check whether car is an S-expression, and if it is call apply; otherwise
(2) Push down (i.e., with our conventions, add to index register 1) by the length of the block specified for CAR, thus protecting its argument;
(3) Check for "out of pushdown list" and initiate garbage collection, if necessary.
(4) Put the return location and number of words in the pushdown block in the last word of that block.
(5) Transfer to car.

It shortened considerably the coding for functions to permit the last argument for a function being called to remain in the accumulator and have setup store it on the pushdown list. So setup also:

(6) Puts C(AC) (contents of the accumulator) in the last argument position on pushdown list.

All these things can be done with one heading word for car, containing (a) the number of words in the pushdown block for car (namely 2) and (b) the number of arguments of car (namely 1) (a variant of these was actually used for efficiency in the coding of setup). In addition, more header words are provided containing currently only a pointer to the property list (and hence the name) of the function for backtrace purposes. Eventually bits may be provided to tell the garbage collector what cells in the pushdown block contain full-word quantities.

Notice that tracing may be turned on by replacing the address of setup in the value cell with the address of a tracing routine.

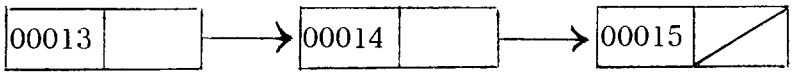
Subroutines exit by jumping to a routine called escape which:

(1) pushes up (i.e., decrements index register 1 appropriately);
(2) returns to the location specified in the last word of the pushdown block;
(3) clears the abandoned push down block.

Functional arguments work if the atoms binding them are declared SPECIAL (as opposed to the COMMON declaration in 7090 LISP 1.5). It appears that the necessity for even this declaration may be removable with a little more work.

Character-Handling

Each character in the character set available on the M 460 (including tab, carriage return, and others) is represented internally by an 8-bit code (6 bits for the character (up to case), 1 bit for case, and 1 bit for color). To facilitate the manipulation of character strings within our LISP system, we permit such character literals to appear in list structure as if they were atoms, i.e. pointers to property lists. These literals can, where necessary, be distinguished from atoms since they are less than 2^8 in magnitude and hence, viewed as pointers, don't point into free storage (where, as in 7090 LISP, property lists are stored). The predicate charp simply makes this magnitude test. This scheme seems preferable to the character-handling in 7090 LISP in that we save the free storage space that would be required to give each 7090 LISP-style character-representing atom a property list, while still permitting character-containing lists to be manipulated by LISP functions as if the characters were atoms. To read S-expressions containing these character literals, we have made the convention of preceding each character literal by a slash, e.g. (/a /b /c) reads in as the "character-string" list structure:



Of course, the output programs also know about the slash convention and print S-expressions in the same form. Furthermore, there exists a routine which, given a character list, simply prints the corresponding string of characters.

Compiler Modification

The changes made in the 7090 LISP 1.5 compiler were extensive. This compiler consists of two main pieces, passone and

phase 2. passone eliminates most special forms by regarding them as macro instructions and expanding them (select is a good example). Changes were required in this part to eliminate all interpreter references (such as eval or $ALIST). This part can probably stand essentially as is in any further compilers which are produced for LISP 1.5. In 7090 form it is suitable for interpreter-cognizant and in M 460 form for compiler-only systems.

phase 2 was first changed to permit go's to appear anywhere inside a prog rather than only on the top level or inside a cond. Otherwise, the changes relate to producing the appropriate code for the M 460 instead of the 7090. Major changes were necessary in call to produce the proper calling sequences and to provide open coding for car-cdr chains.

The result of a compilation using the M 460 version of the LISP compiler is an S-expression analogous to that given to lap in the 7090 version. Our lap is similar to the 7090 one; however, all but a small piece of this minimal assembler is written in LISP. This small machine language piece is lapl [l], where the first element of l is a pair, consisting of a function name and its relative address, and the rest of the elements are also pairs, to be inserted one to a word in successive cells. The car part of these pairs is always an octal number, and the cdr part specifies the address of the word as follows:

(1) NIL no modification
(2) T relocation
(3) (QUOTE s) insert quoted S-expression
(4) (SPECIAL a) insert special cell address

Interpreter

The interpreter we use posts the value of variables in the value cell of the atom corresponding to the variable's name. The function set and value post and retrieve, respectively.

Following is the listing of the interpreter.

```
DEFINE((
    (APPLY (LAMBDA (FN ARGS) (COND
     ((ATOM FN) (COND

      ((EQ FN (QUOTE UNBOUNDVAR)) (ERROR2 (QUOTE A2) FN))
      ((SEXPRP (VALUE FN)) (COND
       ((MEMBER (QUOTE TRACE) FN) (PROG (H)
      (TERPRI) (PRINT (CONS FN (QUOTE (ARGS))))

        (PRINLIS ARGS) (SETQ H (APPLY (VALUE FN) ARGS))
        (TERPRI) (PRINT (CONS FN (QUOTE (VALUE))))

        (RETURN (PRINT H))))
      (T (APPLY (VALUE FN) ARGS))))
    (T (APP2 FN ARGS))))
 ((EQ (CAR FN) (QUOTE LAMBDA)) (PROG (A H) (SETQ A

    (SAVELIS (CADR FN))) (SETLIS (CADR FN) ARGS) (SETQ H
       (EVAL (CADDR FN)
))

    (RESTORLIS A) (RETURN H)))
   ((EQ (CAR FN) (QUOTE LABEL)) (PROG (A H) (SETQ A (LIST
    (CONS (CADR FN) (VALUE FN))))  (SET (CADR FN) (CADDR
       FN))
    (SETQ H (APPLY (CADDR FN) ARGS)) (RESTORLIS A) (RETURN
       H)))

(T (APPLY (EVAL FN) ARGS))
)))

(EVAL (LAMBDA (FORM) (COND
 ((NULL FORM) NIL)
((NUMBERP FORM) FORM)
((CHARP FORM) FORM)
((ATOM FORM) (COND

 ((EQ (VALUE FORM) (QUOTE UNBOUNDVAR)) (ERROR2 (QUOTE A8)
```

```
   FORM))
 (T (VALUE FORM))))

((ATOM (CAR FORM)) (COND
 ((EQ (CAR FORM) (QUOTE QUOTE)) (CADR FORM)) ((EQ (CAR
    FORM) (QUOTE LIST))
 ((EQ (CAR FORM) (QUOTE FUNCTION))
    (CADR FORM))
 ((EQ (CAR FORM) (QUOTE COND))  (EVCON (CDR FORM)))
    (EVLIS (CDR FORM)))

 ((EQ (CAR FORM) (QUOTE PROG))  (EVPROG (CDR FORM)))
 ((EQ (CAR FORM) (QUOTE SETQ))  (SET (CADR FORM) (EVAL
    (CADDR FORM))))

 ((EQ (CAR FORM) (QUOTE GO))  (EVGO (CADR FORM)))
 ((EQ (CAR FORM) (QUOTE RETURN))(PROG NIL
      (SETQ PROGRETURNSWITCH T) (RETURN
      (SETQ PROGVALUE V))))
 ((EQ (CAR FORM) (QUOTE AND))  (EVAND (CDR FORM)))
 ((EQ (CAR FORM) (QUOTE OR)) (EVOR (CDR FORM)))
 (T (APPLY (CAR FORM)  (EVLIS (CDR FORM))))))

(T (APPLY (CAR FORM) (EVLIS (CDR FORM)))))))


  (SAVELIS (LAMBDA (L) (MAPLIST L (FUNCTION (LAMBDA (J)
     (CONS
  (CAR J) (VALUE (CAR J))))))))

  (RESTORLIS (LAMBDA (L) (MAP L (FUNCTION (LAMBDA (J)

  (SET (CAAR J) (CDAR J)))))))
```

```
(SETLIS (LAMBDA (L V) (COND
 ((NULL L)  (COND
 ((NULL V) NIL) (T (ERROR2 (QUOTE F2) L))))
 ((NULL V) (ERROR2 (QUOTE F1) L))

 (T (PROG NIL (SET (CAR L) (CAR V)) (SETLIS (CDR L)
    (CDR V)))))))


(EVLIS (LAMBDA (L) (MAPLIST L (FUNCTION (LAMBDA (J)
  (EVAL (CAR J)))))))
(EVCON (LAMBDA (L) (COND
 ((NULL L) (ERROR2 (QUOTE A3 NIL))

 ((EVAL (CAAR L)) (EVAL (CADAR L)))
 (T (EVCON (CDR L))))))


(EVAND (LAMBDA (L) (COND
 ((NULL L) T)
((EVAL (CAR L)) (EVAND (CDR L)))

(T F))))


(EVOR (LAMBDA (L)  (COND
 ((NULL L) F)
 ((EVAL (CAR L)) T)
 (T (EVOR (CDR L))))))


(EVPROG (LAMBDA (FORM) (PROG

  (P PR A R C PROGGOSWITCH PROGRETURNSWITCH PROGVALUE)
(SETQ P (CAR FORM))
(SETQ PR (CDR FORM))
(SETQ A (SAVELIS P))

(MAP P (FUNCTION (LAMBDA (J) (SET (CAR J) NIL))))
(SETQ R PR)

ML (SETQ PROGGOSWITCH NIL)
(COND
     ((NULL R) (GO RET))
     ((ATOM (CAR R)) (GO ADV))

   ((EQ (CAAR R) (QUOTE COND)) (GO CON)))
(EVAL (CAR R))
```

```
TEST  (COND

       (PROGGOSWITCH (GO GOS))
      (PROGRETURNSWITCH (GO RET)))
ADV  (SETQ R (CDR R))
  (GO ML)

GOS  (SETQ R PR)
GOL  (COND

       ((NULL R) (ERROR2 (QUOTE A6) PROGGOSWITCH))
        ((EQ (CAR R) PROGGOSWITCH) (GO ML)))
  (SETQ R (CDR R))
  (GO GOL)

RET (RESTORLIS A)
  (RETURN PROGVALUE)

CON  (SETQ C (CDAR R))
CONL  (COND
      ((NULL C) (GO ADV))
      ((EVAL  (CAAR C)) (GO CONV)))

  (SETQ C (CDR C))
  (GO COND)

CONV  (EVAL (CADAR C))
  (GO TEST)
   )))


  (EVGO (LAMBDA (A) (SETQ PROGGOSWITCH A)))



(ERROR2 (LAMBDA (X Y) (PROG2 (PRINT X) (ERROR Y))))
(PRINLIS (LAMBDA (X) (MAP X (FUNCTION (LAMBDA (J)
   (PRINT (CAR J)))))
  ))
```

*	L	compiler		numberp
	L	lap		plus
		interpreter		times
	L	setup		greaterp
		escape		eqp
		lister	*	read
		bind	*	prinl
		unbind		terpri
	*	reclaim		punl
		eq		terpun
		car		evalquote
		cdr		lapl
		cons		charp
		rplaca		app2
		atom		set
		gensym		value

---

* - big job
L - written in LISP

LISP as the Language for an Incremental Computer

L. A. Lombardi and Bertram Raphael

1. General

The following two characteristics are commonly found in information systems for the command and control of complex, diversified military systems, for the supply of information input for quantitive analysis and managerial decision making, and for the complementation of computer and scientist in creative thinking ("synnoesis") [10]:

1) the input and output information flows from and to a large, continuous, on-going, evolutionary data base;
2) the algorithms of the process undergo permanent evolution along lines which cannot be predicted in advance.

Most present day information systems are designed along ideas proposed by Turing and von Neumann. The intent of those authors was to automate the execution of procedures, once the procedures were completely determined. Their basic contributions were the concepts of "executable instructions", "program" and "stored program computer". Information systems based on this conventional philosophy of computation can handle effectively only an information process which (1) is "self-contained", in the sense that its data have a completely predetermined structure,

and (2) can be reduced to an algorithm in "final" form, after which no changes can be accomodated but those for which provision was made in advance. Consequently, the current role of automatic information systems in defense, business and research is mainly confined to simple routine functions such as data reduction, accounting and lengthy arithmetical computations. Such systems cannot act as evolutionary extensions of human minds in complex, changing environments.

List-processing computer languages [7] have introduced a flexible and dynamically-changeable computer memory organization. While this feature permits the manipulation of new classes of data, it does not solve the basic communication problems of an evolutionary system. Each program must still "know" the form of its data; and before any processing takes place, a "complete" data set containing a predetermined amount of data must be supplied.

Multiple-access, time-shared, interactive computers [8] cannot completely make up for the inadequacies of conventional and list-processing systems. With time-sharing, changes in systems being developed can be made only by interrupting working programs, altering them, and then resuming computation; no evolutionary characteristics are inherent in the underlying system of a multiple-access, time-shared computer. Thus, as preliminary usage confirms, multiple-access time-sharing of conventional computers is useful mainly in facilitating debugging of programs. While such physical means for close man-computer interaction are necessary for progress in information systems, they are not sufficient alone to produce any substantial expansion in the type of continuous, evolutionary, automatic computer service with which this paper is concerned.

2. The Problem

A new basic philosophy is under development for designing automatic information systems to deal with information processes taking place in a changing, evolutionary environment [1,5,6]. This new approach requires departure from the ideas of Turing and von Neumann. The problem is no longer "executing determined procedures", but rather "determining procedures". Open-endedness, which was virtually absent from the Turing-von Neumann machine concept, must lie in the very foundations of the new philosophy.

The basis of the new approach is an "incremental computer" which, instead of executing frozen commands, evaluates expressions under the control of the available information context. Such evaluation mainly consists of replacing blanks (or unknowns) with data, and performing arithmetical or relational reductions. The

key requirements for the incremental computer are:

(1) The extent to which an expression is evaluated is controlled by the currently-available information context. The result of the evaluation is a new expression, open to accommodate new increments of pertinent information by simply evaluating again within a new information context.
(2) Algorithms, data, and the operation of the computer itself are all represented by "expressions" of the same kind. Since the form of implementation of an expression which describes an evaluation procedure is irrelevant, decisions of hardware *vs*. software can be decided case by case.
(3) The common language used in designing machines, writing programs, and encoding data is directly understandable by untrained humans.

While the Turing-von Neumann computer is computation-oriented, the incremental computer is interface-oriented. Its main function is to catalyze the open-ended growth of information structures along unpredictable guidelines. Its main operation is an *incremental data assimilation* from a variable environment composed of information from humans and/or other processors. (Still, the incremental computer is a universal Turing machine, and can perform arithmetical computations quite efficiently.)

Current research on the incremental computer is aimed at designing it with enough ingenuity to make the new principles as fruitful as the ones of Turing and von Neumann (see [1] and [5]). Some of the main study areas are: the design of the language; the class of *external* recursive functions and a mechanism called a *discharge stack* [2] for their fast evaluation; the design of a suitable memory and memory addressing scheme (the latter problem is being attacked by means of *higher-order* association lists); saving on transfers of information in memory and the use of *cyclic* lists; avoidance or repetition of identical strings within different expressions through the use of *shorthands*, and related problems of maintenance of free storage.

The following will present a quite elementary, restricted and perhaps inefficient version of the incremental computer based on LISP. LISP is the currently available computer language which most closely satisfies the requirements of an incremental computer system. The purpose of this presentation is to demonstrate some of the concepts of incremental data assimilation to scientists who are familiar with LISP. Features of a preliminary LISP

implementation can be used as a guide in the development of the ultimate language for the incremental computer.

3. Aspects of the Proposed Solution

Various structures have been proposed for the language of the incremental computer, mainly stressing closeness to natural language (for preliminary studies see [3] and [4]). Here, however, we will consider the case in which this language is patterned on LISP. In this case a simplified version of the incremental computer will be represented by an extension of the normal LISP "EVALQUOTE" operator. This operator, itself programmed in LISP, will evaluate LISP expressions in a manner consistent with the principles of the incremental computer which are presented below. The LISP representations and programs for implementing these principles will be discussed in section 4 of this paper. The LISP meta-language will be used for all examples in the following sections.

(i) Omitted arguments:

Suppose *func* is defined to be a function of *m* arguments. Consider the problem of evaluating

$$(1)\quad \text{func}[x_1; x_2; \ldots ; x_n] \qquad (n < m)$$

Regular LISP would be unable to assign a value to (1). However, for the *incremental* computer (1) has a value which is itself a function of (m-n) arguments. This latter function is obtained from (1) by replacing the *appropriate* *n* arguments in the definition of *func* by the specified values $x_1, x_2, \ldots, x_n$.

For example, consider the function

$$\text{list3} = \lambda[[x;y;z];\text{cons}[x;\text{cons}[y;\text{cons}[z;\text{NIL}]]]]$$

If A and (B,C) are somehow specified to correspond to the first and third arguments in the *list3* definition, then the incremental computer should find the value of list3[A;(B,C)] to be

$$\lambda[[u];\text{cons}[A;\text{cons}[u;((B,C))]]]$$

(ii) Indefinite arguments:

In regular LISP a function can be meaningfully evaluated only if each supplied argument is of the same *kind* — such as S-expression, functional argument, or number — as its corresponding variable in the definition of the function. In contrast,

the incremental computer permits any argument of a function to be itself a function of further, undetermined arguments. (If these latter arguments were known, then the inner function could be evaluated before the main function, as LISP normally does.) The value of a function with such indefinite arguments should be a new function, all of whose unspecified arguments are at the top level.

For example, consider again the function list3 defined above. In the incremental computer,

$$list3[D;\lambda[[u];cons[E;u]];\lambda[[u];car[u]]]$$

should evaluate to

$$\lambda[[r;s][cons[D;cons[cons[E;r];cons[car[s];NIL]]]]]$$

(iii) Threshold conditions

Consider for example the function sum = $\lambda[[x;y];x+y]$. We say that the "threshold condition" for evaluating a sum is that both arguments of sum be supplied and that they both be numerical atoms. In general, a threshold condition is a necessary and sufficient condition for completing, in some sense, the evaluation of a function. In regular LISP, it is considered a programming error to request the evaluation of an expression involving a function whose threshold condition cannot be satisfied. In the incremental computer, on the other hand, expressions may be evaluated even though they involve indefinite or omitted arguments (as in (i) and (ii) above). In these cases the evaluation is not complete in the sense that the values are themselves functions which will require additional evaluation whenever the appropriate missing data are supplied.

Occasionally the threshold condition for a function does not require the presence of all the arguments. For example, the threshold condition associated with the logical function and is, "either all arguments are present and are truth-valued atoms, or at least one argument is present and it is the truth-valued atom representing falsity." The incremental computer must "know" the threshold conditions for carrying out its various levels of evaluation. One of the most challenging problems in the theoretical design of the new incremental computer is that of determining efficient threshold conditions for arbitrary functions designed by a programmer.

The illustrative program described in the next section employs only the most obvious threshold conditions.

4. The Program

Let us consider some of the problems of representation and organization which must be faced in the course of implementing a LISP version of the incremental computer.

(i) Omitted arguments:

Since LISP functions are defined by means of the lambda notation [9], the role of an argument of a function is determined solely by its relative position in the list of arguments. If an argument is omitted, the omission must not change the order number of any of the supplied arguments. This can be accomplished only if each omitted argument is replaced by some kind of marker to occupy its position. Therefore in the present LISP formalism for the incremental computer, each function must always be supplied the same number of arguments as appear in its definition; however, some of these arguments may be the special atomic symbol "NIL*" which indicates that the corresponding argument is not available for the current evaluation.

The evaluation of a function, some of whose arguments are NIL*'s, is approximately as follows: Each supplied argument (i.e., each argument which is not NIL*) is evaluated, the value substituted into the appropriate places in the definition of the function, and the corresponding variable deleted from the list of bound variables in the definition of the function. What remains is just the definition of a function of the omitted variables.

(ii) Indefinite arguments:

An indefinite argument, as discussed in Section 3 above, is an argument which is itself a function of new unknown arguments. The present program assumes that any argument which is a list whose first element is the atom "LAMBDA" is an indefinite argument. This convention does not cause any difficulty in the use of functional arguments, since they would be prefixed, as S-expressions, by the symbol "FUNCTION." However, there is an ambiguity between indefinite arguments and functional arguments in the meta-language. Also, it is illegal to have an actual supplied argument be a list starting with a "LAMBDA." A more sophisticated version of this program should have some unique way to identify indefinite arguments (perhaps by consing a NIL* in front of them).

The treatment of indefinite arguments is straightforward if one remembers that a main function and an indefinite argument are both λ-expressions, each consisting of a list of variables and a

form containing those variables. The process of evaluating a function fn of an indefinite argument arg involves, then, identifying the variable v in the variable-list of fn which corresponds to arg; replacing v by the string of variables in the variable-list of arg; and substituting the entire form in arg for each occurrence of v in the form in fn. (The treatment of a conditional function containing an indefinite argument is similar although somewhat more complicated.)

(iii) Conflicts of variables:

The same bound variables used in different λ-expressions which appear one within another "conflict" in the sense that they make the meaning of the over-all expression ambiguous. The use of indefinite arguments frequently leads to such conflicts. This problem is avoided in the present system by replacing every bound variable, as soon as it is encountered, by a brand new atomic symbol generated by the LISP function gensym.

(iv) Threshold conditions:

Certain program simplifications can be made automatically by the incremental computer, if corresponding threshold conditions are satisfied. In particular, if every argument of a function is the symbol NIL*, then the function of those arguments is replaced by the function itself.

The incremental computer is represented by the LISP function evalquotel. This function is similar to the normal evalquote operator except that evalquotel first checks to see if any incremental data processing, of the kinds discussed above, is called for. If so, evalquotel performs the appropriate "partial" evaluations. If the given input is a normal LISP function of specified arguments, on the other hand, the effects of evalquotel and evalquote are identical.

Appendix I is a listing of the complete deck for a test run, and includes the definitions of evalquotel and all its subsidiary functions. The results of the run, showing examples of incremental data assimilation with the substl function (which is identical to the normal LISP subst function), are given in Appendix II. The curious reader can understand the detailed operation of the programs by studying these listings.

5. Conclusions

We can now make the following observations concerning the use of LISP as the language for the incremental computer:

(i) Although perhaps too inefficient to be a final solution, LISP is still a very useful language with which to illustrate the features of a new concept of algorithm representation. It is especially easy to use LISP to design an interpreter for a language similar to, but different in significant ways from, LISP itself.

(ii) The program described in this paper is quite limited with regard to its implementation of both LISP and the incremental computer. If a more complete experimental system were desired, the present system could easily be extended in any of several directions. For example, in LISP, allowance could be made for the use of functions defined by machine-language subroutines, and the use of special forms. In the incremental computer, threshold conditions could be inserted to allow partial evaluation and simplification of conditional expressions.

(iii) Replacing all bound variables by new symbols is too brutal a solution to the "conflict" problem; the resulting expressions become quite unreadable. Bound variables frequently have mnemonic significance, and therefore should not be changed unless absolutely necessary. A more sophisticated program would identify those symbols which actually caused a conflict, and then perhaps replace each offending symbol with one whose spelling is different but similar.

(iv) When a function of an indefinite argument is evaluated, the form in the argument is substituted for each occurrence of a variable in the form in the function definition. Similarly, when a function has omitted arguments, those arguments which were not omitted are each evaluated and substituted for each occurrence of variables in the form in the function definition. In the interest of saving computer space, we must be sure that what is substituted is a reference to an expression, not a copy of the expression. In the interest of readability, perhaps the print-outs should similarly contain references to repeated subexpressions, e.g., in the form of λ-expressions, rather than fully-expanded expressions.

Bibliography

[1] L. A. Lombardi and B. Raphael: Man-computer information systems, lecture notes of a two week course UCLA Physical Sciences Extension, July 20-30, 1964.

[2] L. A. Lombardi: Zwei Beitrage zur Morphologie und Syntax deklarativer Systemsprachen, Akten der 1962 Jahrestagung der Gesellschaft für angewandte Mathematik Mechanik (GAMM), Bonn (1962); Zeitschr. angew. Math. Mech. (42) Sonderheft, T27-T29.

[3] _____: On the Control of the Data Flow by Means of Recursive Functions, Proc. Symp. "Symbolic Languages in Data Processing", International Computation Center, Roma, Gordon and Breach, 1962, 173-186.

[4] _____: On Table Operating Algorithms, Proc. 2nd IFIP Congress, Munchen (1962), section 14.

[5] _____: Prospettive per il calcolo automatico, Scientia (in Italian and French) Series IV (57) 2 and 3 (1963).

[6] _____: Incremental data assimilation in man-computer systems, Proc. 1st Congress of Associazione Italiana Calcolo Automatico (AICA), Bologna, May 20-22, 1963 (in press).

[7] D. G. Bobrow and B. Raphael: A Comparison of List-processing Computer Languages, Comm. ACM, expected publication April or May, 1964.

[8] MIT Computation Center: The Compatible Time-Sharing System: A Programmer's Guide, MIT Press, Cambridge, Mass., 1963.

[9] A. Church: The Calculi of Lambda-Conversion, Princeton University Press, Princeton, N.J., 1941.

[10] L. Fein: The computer-related science (synnoetics) at a University of the year 1975, American Scientist (49) (1961), 149-168; Datamation (7) 9 (1961), 34-41.

[11] Work reported herein was partly supported by Project MAC, an MIT research program sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract Number Nonr-4102(01). Reproduction in whole or in part is permitted for any purpose of the United States Government.

[12] Assoc. Prof. of Industrial Management, MIT.

[13] Research Assistant, Mathematics Dept., MIT.

```
       TEST     B RAPHAEL   M948
DEFINE ((
(EVALQUOTE1  (LAMBDA  (FN  X)
    (APPLY1  FN  X  NIL) ))
  (APPLY1  (LAMBDA  (FN  X  A)  (COND
   ((ATOM  FN)  (COND
      ((GET  FN  (QUOTE  EXPR))  (APPLY1  (GET  FN  (QUOTE EXPR)) X A))
      ((EQ  FN  (QUOTE  CAR))  (COND
       ((NULL*  (CAR  X))  (QUOTE  CAR))
        ((LAM1  (CAR  X))  (APP2  X  (QUOTE  CAR)))
        (T  (CAAR  X))  ))
      ((EQ  FN  (QUOTE  CDR))  (COND
       ((NULL*  (CAR  X))  (QUOTE  CDR))
        ((LAM1  (CAR  X))  (APP2 X (QUOTE  CDR)))
        (T  (CDAR  X))  ))
      ((EQ  FN  (QUOTE  CONS))(COND((LAM2 X)(APP3 X A  (QUOTE CONS)))
        (T  (CONS  (CAR  X)  (CADR  X)))  ))
      ((EQ  FN  (QUOTE  ATOM))  (COND  ((NULL*  (CAR  X))(QUOTE ATOM))
        ((LAM1 (CAR X)) (APP2 X (QUOTE ATOM))) (T (ATOM (CAR X))) ))
   ((EQ  FN  (QUOTE  EQ)) (COND ((LAM2 X)(APP3 X A (QUOTE  EQ)))
        (T  (EQ (CAR  X) (CADR X)))  ))
       (T  (ERROR (LIST  (QUOTE  APPLY1)  FN  X  A)  ))  ))
   ((EQ (CAR  FN)  (QUOTE  LAMBDA)) (APPLY2 (LAMS
        (CONS  (QUOTE  LAMBDA)  (UNFLICT  (CDR  FN)))  X)  A))
    (T  (ERROR  (LIST  (QUOTE  APPLY1)  FN  X  A)))  ))  )
 (LAM1  (LAMBDA  (X)
   (AND  (NOT (ATOM  X))  (EQ (CAR  X)  (QUOTE  LAMBDA)))  ))

 (APP2  (LAMBDA  (X  A)
     (LIST  (CAAR  X)  (CADAR  X)  (LIST  A  (CADDAR  X))  ))  )
```

```
(NULL*  (LAMBDA  (X)  (EQ  X  (QUOTE  NIL*))  ))
(LAM2  (LAMBDA  (X)  (OR
  (MEMBER (QUOTE NIL*)  X)  (LAM1 (CAR X))  (LAM1 (CADR X)) )))

(APP3  (LAMBDA (X A F) ((LAMBDA  (U  V)  (APPLY1
    (LIST (QUOTE LAMBDA) (LIST U V)(LIST  F  U  V))  X  A))
        (GENSYM)  (GENSYM))))

(LAMS  (LAMBDA  (FN  X)  (PROG  (VAR1 ARG1 VARS ARGS ARG2 M  L)
  (SETQ  M  (CADDR  FN))
  (SETQ  ARGS  X)
  (SETQ  VARS  (CADR  FN))
LOOP  (SETQ  L  (CAR  ARGS))
  (COND  ((LAM1  L)
         (GO FLICT)))
  (SETQ  VAR1  (CONS  (CAR  VARS)  VAR1))
  (SETQ  ARG1  (CONS  L  ARG1))
LOOP1  (SETQ  VARS  (CDR  VARS))
  (COND  ((NULL  VARS) (RETURN  (LIST  (REVERSE  VAR1)  M
              (REVERSE ARG1)))))
  (SETQ  ARGS  (CDR  ARGS))
  (GO LOOP)
FLICT  (SETQ  L  (UNFLICT  (CDR  L)  ))
  (SETQ  ARG2  (CAR  L))
LOOP2  (SETQ  VAR1  (CONS  (CAR  ARG2)  VAR1))
  (SETQ  ARG1  (CONS  (QUOTE  NIL*)  ARG1))
  (SETQ  ARG2  (CDR  ARG2))
  (COND   (ARG2  (GO  LOOP2)))
  (SETQ  M  (SUBST  (CADR  L)  (CAR  VARS)  M))
  (GO LOOP1)  )))
(UNFLICT  (LAMBDA (Y  )  (PROG  (L  )
```

```
  (SETQ  L  (CAR  Y))
LOOP  (COND  ((NULL  L)  (RETURN  Y)))
  (SETQ  Y  (SUBST  (GENSYM)  (CAR  L)  Y))
  (SETQ  L  (CDR  L))
  (GO  LOOP)  )))

(APPLY2  (LAMBDA (L  A)  (COND  ((MEMBER  (QUOTE  NIL*)  (CADDR  L))
        (APPLY3  L  A))  (T  (EVAL1  (CADR  L)  (PAIRLIS (CAR L)
          (CADDR  L)  A)))  )))

(APPLY3  (LAMBDA  (L  A)  (SEARCH  (CADDR  L)
  (FUNCTION  (LAMBDA  (J)  (NOT  (EQ  (CAR  J)  (QUOTE  NIL*)))  ))
  (FUNCTION  (LAMBDA  (J)  (APPLY4  L  A)))
  (FUNCTION  (LAMBDA  (J) (LIST (QUOTE LAMBDA)(CAR L)(CADR L)))) )))

(APPLY4 (LAMBDA  (L  A)  (PROG  (VARS FORM ARGS M ARG1)
  (SETQ  VARS  (CAR  L))
  (SETQ  FORM  (CADR  L))
 (SETQ  ARGS  (CADDR  L))
LOOP  (SETQ  ARG1  (CAR  ARGS))
  (COND  ((EQ  ARG1  (QUOTE  NIL*))  (GO B))  )
  (SETQ  FORM  (SUBST  (LIST  (QUOTE  QUOTE)  ARG1)  (CAR  VARS) FORM))
LOOP1  (SETQ  ARGS  (CDR  ARGS))
  (COND ((NULL ARGS)(RETURN          (LIST (QUOTE LAMBDA) M FORM)   )))
  (SETQ  VARS  (CDR  VARS))
  (GO  LOOP)
 B (SETQ  M  (CONS  (CAR  VARS)  M))
  (GO  LOOP1)  )))

(EVAL1  (LAMBDA (E  A) (COND  ((ATOM  E)  (COND
    ((GET  E  (QUOTE  APVAL))  (EVAL  E  A))
```

```
      ((EQ  E  (QUOTE NIL*)) (QUOTE NIL*))(T (CDR (ASSOC E  A)) ) ))
   ((ATOM (CAR E))  (COND
     ((EQ  (CAR  E)  (QUOTE  QUOTE))  (CADR  E))
     ((EQ  (CAR  E)  (QUOTE  COND))  (EVCON1  (CDR  E)  A))
     ((EQ  (CAR  E)  (QUOTE  LAMBDA))  E)
     (T  (APPLY1  (CAR  E)  (EVLIS1  (CDR  E)  A)  A))  )   )
   (T    (APPLY1  (CAR  E)  (EVLIS1  (CDR  E)  A)  A))  ) ))

 (EVCON1  (LAMBDA  (C  A)  ((LAMBDA  (X)  (COND
   ((LAM1  X)  (LIST  (CAR  X)  (CADR  X)
      (CONS (QUOTE COND)(CONS (LIST (CADDR X)(CADAR C)) (CDR C))) ) )
   ((EVAL1  X  A)  (EVAL1  (CADAR C)  A))
   (T  (EVCON1  (CDR  C)  A)) ))  (CAAR  C))  ))

 (PAIRLIS  (LAMBDA  (X  Y  A)  (COND  ((NULL  X)  A)
    (T  (CONS  (CONS (CAR X)(CAR Y))(PAIRLIS (CDR X)(CDR Y) A)) )) ))

 (ASSOC  (LAMBDA  (X A)  (COND  ((EQUAL (CAAR A) X)  (CAR  A))
    (T  (ASSOC  X  (CDR  A)))  )))

 (EVLIS1  (LAMBDA  (M  A)  (COND  ((NULL  M)  NIL)
   (T  (CONS  (EVAL1  (CAR  M)  A)  (EVLIS1  (CDR  M)  A)))  )))
(  SUBST1  (LAMBDA  (X  Y  Z)  (COND  ((ATOM  Z)  (COND
      ((EQ  Z  Y)  X)  (T  Z))  )
     (T  (CONS  (SUBST1  X  Y  (CAR  Z))  (SUBST1 X  Y  (CDR  Z)))))))
    ))

EVALQUOTE1
  (SUBST1  ((A  B)  C  NIL*)  )
EVALQUOTE1
  (SUBST1  ((LAMBDA  (X)  (CONS  X  (QUOTE  (B))))  C  (C  Y  (C  D))))
```

```
EVALQUOTE1
  (SUBST1  (NIL*  NIL*        NIL*))
EVALQUOTE1
  (SUBST1  (ONION  NIL*  (LAMBDA  (X  Y)  (CONS  X  Y))  ))
EVALQUOTE1
  SUBST1  ((A  B)  C  (C  Y  (C  D))  )              )
      STOP ))))))))))
      FIN
```

Appendix II: Results of Computer Run

```
FUNCTION   EVALQUOTE   HAS BEEN ENTERED, ARGUMENTS..
EVALQUOTE1
(SUBST1 ((A B) C NIL*))


END OF EVALQUOTE, VALUE IS ..
(LAMBDA (GOOOO3) (COND ((ATOM GOOOO3) (COND ((EQ GOOOO3 (QUOTE C)) (QUOTE (A B))) (T GOOOO3)))
                                                                    (T (CONS (SUBST1
(QUOTE (A B)) (QUOTE C) (CAR GOOOO3)) (SUBST1 (QUOTE (A B)) (QUOTE C) (CDR GOOOO3))))))



FUNCTION   EVALQUOTE   HAS BEEN ENTERED, ARGUMENTS..
EVALQUOTE1
(SUBST1 ((LAMBDA (X) (CONS X (QUOTE (B)))) C (C Y (C D))))


END OF EVALQUOTE, VALUE IS ..
(LAMBDA (GOOOO7) (COND ((ATOM (QUOTE (C Y (C D)))) (COND ((EQ (QUOTE (C Y (C D))) (QUOTE C))
                                                                    (CONS GOOOO7 (
QUOTE (B)))) (T (QUOTE (C Y (C D)))))) (T (CONS (SUBST1 (CONS GOOOO7 (QUOTE (B))) (QUOTE C)
                                                                    (CAR (QUOTE (C
Y (C D))))) (SUBST1 (CONS GOOOO7 (QUOTE (B))) (QUOTE C) (CDR (QUOTE (C Y (C D)))))))))



FUNCTION   EVALQUOTE   HAS BEEN ENTERED, ARGUMENTS..
EVALQUOTE1
(SUBST1 (NIL* NIL* NIL*))
```

```
END OF EVALQUOTE, VALUE IS ..
(LAMBDA (G00008 G00009 G00010) (COND ((ATOM G00010) (COND ((EQ G00010 G00009) G00008) (T G00010)))
                                                                    (T (CONS
(SUBST1 G00008 G00009 (CAR G00010)) (SUBST1 G00008 G00009 (CDR G00010))))))


FUNCTION   EVALQUOTE   HAS BEEN ENTERED, ARGUMENTS..
EVALQUOTE1
(SUBST1 (ONION NIL* (LAMBDA (X Y) (CONS X Y))))


END OF EVALQUOTE, VALUE IS ..
(LAMBDA (G00015 G00014 G00012) (COND ((ATOM (CONS G00014 G00015)) (COND ((EQ (CONS G00014 G00015)
                                                                    G00012) (
QUOTE ONION)) (T (CONS G00014 G00015)))) (T (CONS (SUBST1 (QUOTE ONION) G00012 (CAR (CONS G00014
                                                                    G00015)))
(SUBST1 (QUOTE ONION) G00012 (CDR (CONS G00014 G00015)))))))


FUNCTION   EVALQUOTE   HAS BEEN ENTERED, ARGUMENTS..
EVALQUOTE1
(SUBST1 ((A B) C (C Y (C D))))


GARBAGE COLLECTOR ENTERED AT 04050 OCTAL.        FULL WORDS  1438 FREE   5329  PUSH DOWN DEPTH 105

GARBAGE COLLECTOR ENTERED AT 04050 OCTAL.        FULL WORDS  1426 FREE   4856  PUSH DOWN DEPTH 304

END OF EVALQUOTE, VALUE IS ..
((A B) Y ((A B) D))
```

The LISP System for the Q-32 Computer

Robert A. Saunders

Information International, Inc

The LISP system for the time-shared AN/FSQ-32/V computer at System Development Corp., Santa Monica, Calif., is based on a compiler written in LISP and compiled by itself on an IBM 7090. The system is, in this respect, similar to the system developed by T.P. Hart for the M-460 computer at Air Force Cambridge Research Labs (1). The Q-32 compiler is based on the Hart compiler for the M-460, and it is fitting here to acknowledge the considerable assistance rendered to the Q-32 project by Mr. Hart. Assistance was also given by D. Edwards and M. Levin of the Massachusetts Institute of Technology, and Prof. J. McCarthy and S. Russell of Stanford University. Computer time on the Stanford 7090 and PDP-1 was used in conjunction with Stanford's contract with Advanced Research Projects Agency for research in time sharing and artificial intelligence.

The Q-32 is a 1's complement binary computer with a 48-bit word length and 65,536 words of storage. Core speed is about 2 microseconds, and some instructions overlap. It has an accumulator, an accumulator extension called the B-register, eight index registers, and various other electronic registers. Peripheral equipment includes 16 tape drives (729 IV), about 350,000 words of drum storage, a card reader, card punch, and a line printer. A PDP-1 is used as a peripheral processor to service time-shared Teletypes. When run in time sharing, the lowest 16,384 registers are used by the executive system.

The external language is compatible with LISP 1.5. Some features (arrays, character objects, and "$$" quoting in READ) are not implemented at present. Most programs which will run on 7090 LISP will run on Q-32 LISP without change.

From the user's point of view, the Q-32 system is seen through a version of EVALQUOTE which reads a pair and executes it. As in 7090 LISP *(2), the pair is a function and a list of

---

*The reader is advised to familiarize himself with this reference if he has not done so.

arguments. If the function is an atom carrying a functional definition, that definition, in the form of compiled code, is applied to the arguments. If it is a functional expression, the expression is compiled and then executed.

The function DEFINE takes a list of pairs, making the first element of a pair the name of the function represented by the second element. The function is compiled at the time it is defined.

Q-32 LISP uses the macro system developed by Hart (3). The function MACRO takes a list of pairs similar to DEFINE. The functions, which are compiled by MACRO, are applied to a list of their arguments whenever they appear in a DEFINE.

There are two classes of variables in Q-32 LISP, local and special. Local variables, which may be referred to only within the function binding them, are stored on the pushdown list. Variables used free (i.e., used in a function in which they are not explicitly bound) must be declared special. As in 7090 LISP, this is done with a declaration of the form:

```
SPECIAL (( A B C ))
```

If an un-special variable is used free, it will be treated as special, and an error comment will appear. A local variable binding is visible only in the function that binds it, and the binding is lost when the function is completed. A special variable binding is also lost when the function binding it is left, and the old special value, saved upon entering the function, is restored. The special value of a variable bound by no function can be considered a constant, and thus is the Q-32's equivalent of the 7090 APVAL. Such a binding can be established by CSET. It is also established by DEFINE, and the value of the binding is a number (called the function descriptor). Functional arguments work by binding the function atom to a function descriptor. The call of a function looks for a function descriptor binding on the SPECIAL cell, so functional arguments must be declared SPECIAL.

```
DEFINE ((

(MAPLIST (LAMBDA (L FN) (COND ((NULL L) NIL)

    (T (CONS (FN L) (MAPLIST (CDR L) FN))) )))

))
```

The functional argument FN must be declared SPECIAL when MAPLIST

is defined. (This has been done, of course, in the system version of MAPLIST.) Then when MAPLIST is entered, the functional argument is attached to FN, and when the function FN is called, the right thing happens.

Atoms in Q-32 LISP are of two types: numeric and literal. Numeric atoms begin with a digit, plus, or minus; any other legal character as the first character denotes a literal atom (e.g., CAR). Numbers are fixed point unless containing a decimal point, which denotes floating point. Scaling of either fixed or floating point decimal numbers can be done with an E scale factor. Examples:

1E17 1.0E-250 .347E12

Negative scaling of a fixed point number is not meaningful.

Octal integers are denoted by a letter Q following the integer. A decimal integer following the Q denotes an octal scale factor. Only positive scalings are meaningful. The following are equivalent in value:

3641lQ2 36411000Q 1000000 1E6

They will print differently, however; the first two as 36411Q2 and the other two as 1000000.

The system contains an incomplete set of Hollerith objects. The object names below are bound to special values which print as indicated.

Object	Value	Object	Value
LPAR	(	COLON	:
PERIOD	.	LARR	→
BLANK		UPARR	↑
RPAR	)	LSTHAN	<
DOLLAR	$	GRTHAN	>
STAR	*		
SLASH	/		
EQSIGN	=		

Because the Q-32 LISP system is compiler-oriented rather than interpreter-oriented, the user should expect:

1. Programs to run faster on the Q-32 than (uncompiled) on the 7090.
2. To have to pay more attention to variable declarations than in an interpretive system, where free

variable bindings are available automatically.

3. To have less thorough error-checking by the system. In particular, there is no check that functions are supplied the right number of arguments. (To put this feature in would require two words instead of one for function calls.)

Inside the System

The foregoing section will suffice to guide the casual user familiar with 7090 LISP. What follows is a fairly complete description of the inner workings of the system.

Storage Allocation

Storage in the computer is laid out roughly as follows:

OCTAL	ITEM
40000 - 44000	Maching language code
44000 - 54400	Compiled code
54400 - 55400	Scratch function area
55400 - 70000	Binary program space
70000 - 74000	Pushdown list
74000 - 100000	Quote cells and atom heads
100000 - ?	Full word space
? - 170000	Free storage

Full word space is assigned from the bottom up. Free storage is assigned from the top down. Garbage collection is initiated when the total storage is exhausted. The full word space is compacted downward, and the free storage is compacted upward. The floating boundary permits better utilization of storage.

Compiled references to list structure are made through the quote cells, the pushdown list, and the atom heads. This allows the garbage collector to avoid tracing through compiled code.

Atomic Structure

Atoms in Q-32 LISP are of two types; literal and numeric. Numeric atoms occupy 3 words in the computer. See Figure 1.

The one in bit 1 of the free storage word is the atom indicator. Bit 0 is used by the garbage collector. The decrement and remainder of the prefix are not used. The tag indicates a

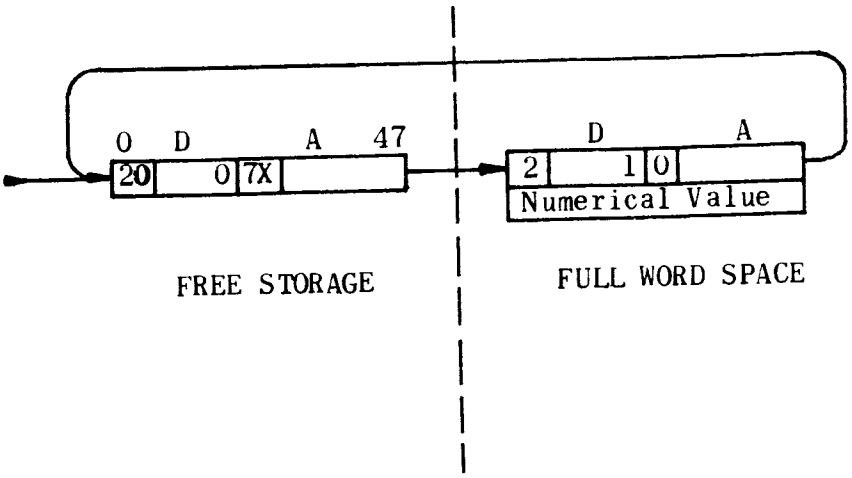


Figure 1. Structure of a LISP Number

number, of type defined by X, while the address points to the structure in full word space. X is 1 if the number is a fixed point integer, 2 if it is floating point, and 5 if the number is to be treated as an octal integer.

The structure in full-word space is actually an array of one element. The one in bit 4 of the array head indicates non-list-type elements in the array (presently the only allowed type). Bit 5 is a 1 if the array elements are BCD (binary coded decimal) characters. Bit 0 is used by the garbage collector, and the remaining bits of the prefix are unused. The decrement contains the number of words in the array (excluding the head word). In numbers, the tag is not used. The address is a pointer to the associated word in free storage.

A typical atom (e.g., CAR) looks like the diagram in Figure 2.

The atom head cell is assigned when the atom is first read (or generated, in the case of GENSYM's) and is not presently recoverable by garbage collection. The prefix contains a 1 in bit 1 as the atom indicator. A 1 in bit 4 would indicate that the function represented by the atom was being traced. The decrement points to the property list. The first item on the property list is always the print name, so the available property list begins at CDDR of the atom. The system does not use the property list; it is entirely available to the programmer.

The print name is a numerical array of as many words as are required at 8 characters per word. The 3 in the tag of the array head denotes 3 characters in the last word of the print name.

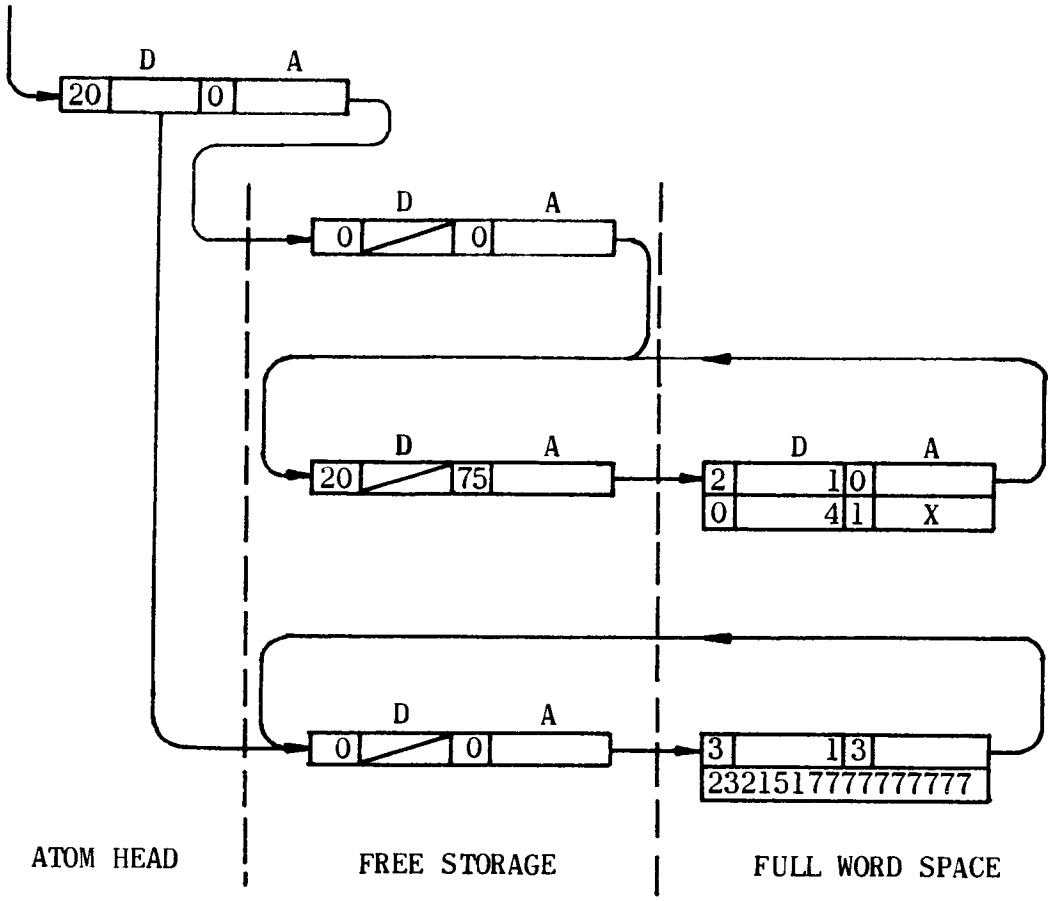


Figure 2. Structure of a Typical Atom (CAR)

The address and tag of the atom head give the location of the special cell, if any. In the case of functional bindings, the special cell is a pointer to a number. The number gives in its address the location of the machine language code for the function. The tag gives the number of arguments the function expects to get, and the decrement gives the amount of space required on the pushdown list. Obviously the number of arguments to a function cannot exceed 63. Since the pushdown list requires two words per entry, and an extra entry for control information, $D \geq 2T+2$.

An atom is considered to have been declared SPECIAL if bit 3 of the atom head is a 1. This bit is set by the function SPECIAL and cleared by UNSPECIAL . This bit is examined by the compiler when an atom is used as a free variable. Because this bit is independent of the actual existence of a SPECIAL binding, an atom which has a SPECIAL or functional value can be used as a local variable even if the special value is present during compilation of a function.

Binding a special variable consists of saving on the pushdown list the old address part of the atom head, i.e. the previous special binding (if any). The address part of the atom is then caused to point to the location on the pushdown list where the value of the new binding is stored. This is the reason for two words per entry on the pushdown list - the first is used for saving old special bindings, and the second has the value of the associated variable.

Function Calls

The majority of compiled function calls are of the form

(BSX *CALL 4 (E FN)),

i.e., branch to *CALL and set index 4 to point to the head of the atom FN. *CALL is a section of the assembled part of the system that:

1. Examines the atom for a functional binding, i.e., a pointer to a number in CAR of the atom. If such a binding does not exist, an error is indicated. If it does, the subroutine entry address, number of arguments, number of cells on the pushdown list, and whether it is being traced, are determined. Then,
2. Unless the function is of no arguments, the accumulator is stored on the pushdown list as the last argument.

3. The pushdown pointer is incremented, and the old pushdown pointer, the return address, and the atom being called are saved on the pushdown list.
4. If the function is being traced, its name and arguments are printed.
5. Control passes to the subroutine.

Returns from functions generally are transfers to *RETRN which:

1. If the function is being traced, prints its name and value.
2. Finds the return address.
3. Restores the pushdown pointer
4. Returns control to the calling routine.

Certain routines have special calls. SPECBIND and SPECRSTR, the routines that bind and restore special variables, are called

by, for example:

```
        (BUC SPECBIND 0 4)
G14707  (0 (2 *N) 1 (E V1))
        (CAS (4 *N) 1 (E V2))
```

The parameter list is a list of special variables to be bound, and of where the old bindings are to be saved. *N is minus the total number of cells of pushdown list used by the function. The prefix is 0 except for the last entry, where it is 40 octal. The address and tag denote the place on the pushdown list where the old binding is saved, and the decrement gives the atom to be bound. After binding, (0 (2 *N) 1) has in its address a pointer to the previous binding, and the decrement has the atom name so that the special values can be restored by UNWND in case an error occurs.

Restoration of special variables is handled by a call of the form

```
(BSX SPECRSTR 4 G14707)
```

SPECRSTR uses the parameter list used by SPECBIND to restore the previous bindings.

LIST is a rather common function, and instead of having it defined as a macro to generate lots of CONS's, a special subroutine (which calls CONS) is used. Calls are of the form:

```
(BSX *LIST 2 N)
(0 L1)
(0 L2)
(0 LN)
```

where N is the number of items to be listed. The parameter list contains the locations of the items to be listed.

Some machine-language-coded functions (e.g. NUMBERP) are called both as LISP functions through *CALL and from inside the system. These return to the location following that from which they are called. *RETRN immediately follows the end of *CALL so that returns from LISP calls work.

Input-Output

Primary input-output is via time-shared Teletype. The read routine reads one S-expression at a time, extending over as many lines of input as are required. For fast reading, a block of 125 registers is chained in the CDR-direction into a list called

the object list (OBLIST). The CAR of each of these points to a list of atoms, all of which have the same remainder when the numerical value of the first word of the print name is divided by 125. These lists are set up by a subroutine called BUCSRT when the system is first loaded. When an atom is read, a subroutine called INTERN determines whether it is one seen before or not. If not, the newly created atom is hung onto the right part of the object list. More precisely, INTERN is a function of one argument, a just read atom, whose value is either that atom or another with the same print name. This insures a unique representation for literal atoms.

Printing is done by PRINT, which calls PRINØ and TERPRI. PRINØ breaks list structure into atoms and passes them to PRIN1, which puts the print name into the output buffer. Punctuation (dots and parentheses) are outputted by giving PRIN1 the correct Hollerith object. TERPRI puts a carriage return into the buffer and dumps it. The buffer is also dumped when full or when an object ends within 16 characters of the end of the buffer.

Garbage Collection

Garbage collection is done in a four phase process.

1. All list structure is marked, starting from the quote cells, the object list, and the push-down list. Full words are marked with a bit in the array head, so a bit table is not required.
2. Full word space is compacted downward. Two pointers start at the beginning of full word space. The first pointer is advanced over all full words, and those marked are copied into the location indicated by the second pointer, which is advanced for each array copied. The pointer in the array head is used to update the list pointers to relocated arrays.
3. Free storage is compacted upward by a scheme attributed to D. Edwards. Two pointers are set, one to the top of free storage and one to the bottom. The top pointer scans words, advancing downward, looking for one not marked. When one is found, the bottom pointer scans words, advancing upward, looking for a marked word. When one is found, it is moved into the location identified by the other pointer. A pointer is left at the location the word was moved from, pointing to where it was moved to. The top pointer is then again advanced as before. The process terminates when

the two pointers meet.

4. List references to the vacated free storage are fixed up by looking at CAR and CDR of every word on the pushdown list, on the oblist, and in the compacted free storage. Any pointers to the vacated area are replaced with pointers to the relocated words, using the pointers left there in Step 3.

The process is substantially slower than on the 7090 for two reasons: a much larger free storage (approximately 30,000 words), and more computation.

Compilation

Compilation is done in a five-step process:

1. The macros present in the definition, if any, are expanded.
2. Pass 1 of the compiler proper alters the definition to take care of special variables and various odds and ends.
3. Pass 2 of the compiler generates symbolic machine code from the altered S-expressions.
4. Pass 1 of LAP constructs a symbol table.
5. Pass 2 of LAP uses the resulting symbol table to translate the machine code into binary, which is stored in memory.

The code that is compiled is not particularly efficient, but it is serviceable, and not less than half as fast as hand-compiled code would be. It consists almost entirely of subroutine calls, only EQ, NOT, and NULL in predicates being compiled open. COND's and PROG's are expanded in a straightforward way. The definition of EQ which appears in the system, to wit:

```
(EQ (LAMBDA (A B) (COND ((EQ A B) T) (T NIL) )))
```

may appear redundant or infinitely recursive, but in fact it is neither; the appearance of EQ inside is compiled open; as it is in a predicate; and the function is used for calls of EQ outside of predicates.

LAP

The Q-32 version of LAP (the LISP assembly program) is

similar to that on the 7090. It is written in S-expressions and is part of the system compiled on the 7090. The language is identical to that used on the 7090 except for the origin statement. The legal types include literal atoms and numbers, which are identical in effect to the same types on the 7090; and lists of the form

(NAME SUBR X Y)

except for Y, are the same as on the 7090. Y is the number of cells (not pairs) required on the pushdown list. As stated before, $Y \geq 2X+2$.

Words consist of lists of one to four elements, in the order operation, address, tag, decrement. The address and decrement are taken modulo $2^{18}-1$. The fields present are OR'ed together, with the tag and decrement fields shifted left 18 and 24 bits, respectively. The orders in the permanent vocabulary include those used by the compiler. These are:

BSX	73Q14	Branch and set index
BUC	14Q13	Branch unconditionally; set index register designated by decrement to location counter
STZ	51Q13	Store zero
STA	5Q15	Store accumulator
BOZ	66Q15	Branch on accumulator zero
BNZ	601Q13	Branch on accumulator not zero
LDA	2Q15	Load accumulator
XOR	43003Q11	Exclusive OR to accumulator
CAS	4Q15	Compare accumulator with storage (used as flag only)
LDB	22Q24	Load B - register
STB	504Q13	Store B - register

As many others as are desired can be defined, e.g. CSET (BAX 74Q14).

Building the System

The system is written about half in SCAMP, the Q-32 assembly language, and half in LISP and LAP. Each half comprises about 1900 cards. The LISP part is compiled on a 7090, taking about 40 minutes of computer time. Various parameters set in advance control the layout of the system. About a dozen references to the assembled code are handled by use of a transfer vector at the beginning of the assembled code. Various locations in the resulting compiled code are put into the SCAMP deck as assembly parameters.

The output from the LISP compilation consists of an assembly listing of the compiled code, and a tape containing some 6000 card images, each of which has an octal address in cols. 1-8, and an octal word (instruction or permanent list structure) in cols. 9-24. The 48 bit Q-32 word is handled in the 7090 by treating it as two words. The 90 code correctly assembles list structure, instructions, positive integers up to 36 bits, and negative integers up to 24 bits. At present writing, this is adequate for everything in the system except one number (in EQUAL), which is corrected with a patch.

The SCAMP deck contains the basic functions used in the system, a code to read the octal card-image tape, and a code to write a time-sharing compatible tape. The assembled code is loaded during non-time-sharing, and the program reads the octal tape and plants the words where they belong. A routine then plants patches as required, and the time-sharing "load tape" is then written. It is essentially a core dump, taken 960 words at a time.

References

1. Hart, T.P., and T.G. Evans, "The M-460 LISP 1.5 System", Air Force Cambridge Research Laboratory memorandum. (Reprinted in this volume.)

2. McCarthy, John, et al., LISP 1.5 Programmer's Manual The M.I.T. Press, Cambridge, Mass., 1962.

3. Hart, T.P., "MACRO Definitions for LISP", Artificial Intelligence Project, Research Laboratory of Electronics and MIT Computation Center Memo 57, Oct. 1963.

APPENDIX 1

Atoms with Zero-Level Bindings in Q-32 LISP

This appendix is essentially a list of the reserved atoms in Q-32 LISP.

The following atoms represent the same functions as in 7090 LISP. Numbers refer to pages in the LISP 1.5 Manual.

ADD1	26, 64
APPEND	11, 61
ATOM	3, 57
CAR, CDR, CAAR-CDDDDR	2, 3, 56
CONS	2, 56
CSET	17, 59
DEFLIST	41, 58
EQ	3, 23, 57
EQUAL	11, 26, 57
ERROR	32, 66
FIXP	26, 64
FLOATP	26, 64
GENSYM	66
GET	41, 59
GREATERP	26, 64
LAP	65, 73
LENGTH	62
LESSP	26, 64
MAP	63
MAPCON	63
MAPLIST	20, 63
MEMBER	11, 62
MINUS	26, 63
MINUSP	26, 64
NCONC	62
NUMBERP	26, 64
NULL	11, 57
PAIR	60
PRIN1	65, 84
PRINT	65, 84
PROG2	42, 66
PROP	59
READ	65, 84
REVERSE	62
RPLACA	41, 58
RPLACD	41, 58
SASSOC	60
SPECIAL	64, 78
SUB1	26, 64
TERPRI	65, 84
TRACE	32, 66, 79
UNSPECIAL	64, 78
UNTRACE	32, 66
ZEROP	26, 64

The following atoms represent functions different from those in the 7090:

(LSHIFT A B)	Same as (LEFTSHIFT A B) on 7090.
(DEFINE L)	Compiles the list of functions L. Similar to combined effects of 7090 DEFINE, COMPILE.
(DIFFER A B)	Same as (DIFFERENCE A B) on 7090.

The following atoms represent mathematical functions. They differ from the related 7090 functions in that these require precisely two arguments:

(*PLUS A B)	(*LOGOR A B)	(*MIN A B)
(*TIMES A B)	(*LOGXOR A B)	
(*LOGAND A B)	(*MAX A B)	

The functions PLUS, TIMES, LOGAND, LOGOR, LOGXOR, MAX, MIN can be defined as macros using the above functions.

The following atoms represent functions either unique to the Q-32 or not described in the LISP 1.5 Manual:

(MACRO L)	Defines a list of functions L as macros.
(MAPCAR L FN)	Same as MAPLIST except FN is applied to each element of L: i.e., CAR of what it is applied to in MAPLIST.
(EQP A B)	Tests two numbers for equality within a tolerance (3.0 E-6). Used by EQUAL.
(ABSVAL A)	Absolute value of A. Used by EQP.
(PRINØ L)	The principal part of PRINT; does all of PRINT except a final TERPRI.
(PRINOCT A B)	Prints in unsigned octal the B lowest significant octal digits of A, including leading zeros, if any.
(BLANKS N)	Enters N blanks into the print buffer.
(READ1)	Part of READ.
(LAST L)	Gives last element of list L.

The compiler and related code comprise the following functions:

DEF1	PA7	STORE
MDEF	PA8	CALL
COM2	PA9	LAC
PASS1	PA12	ATTACH
PROGITER	DELETEL	LOCATE
PAIRMAP	COMP	LAPEVAL
PI1	PASS2	JUST
PI3	COMVAL	*CALL
PALAM	COMPROG	*RETRN
PAFORM	COMCOND	*LIST
PA1	COMBOOL	SPECBIND
PA2	COMPACT	SPECRSTR
PA3	CEQ	*SPECIND
PA4	COMPLY	TEREAD
PA5	COMLIS	

The following atoms have zero-level bindings used by the compiler:

BPORG	SCRACH	TBPS	PRINLIS
BSX	BUC	STA	STZ
BNZ	LDA	BOZ	XOR
CAS	STB	LDB	

The following atoms are bound to character objects:

LPAR	SLASH	COLON	not installed at present
BLANK	EQSIGN	LARR	
PERIOD	DOLLAR	UPARR	
RPAR	STAR	LSTHAN	
	T	GRTHAN	

Various atoms represent miscellaneous service routines. These atoms all begin with an asterisk and are not listed here.

Certain functions in the compiler have generated symbols for names. These names, although appearing on the oblist, will not conflict with symbols generated by Q-32 GENSYM, even if they have the same print name, as GENSYM does not enter the generated atoms on the oblist. However, a conflict could arise if the wrong GENSYM's were read in and then bound on the zero level. Any subsequent version of the Q-32 LISP will use different internal GENSYM's to avoid this problem.

The following atoms, although not bound as functions, are treated as special functions by the compiler:

LAMBDA	NIL	COND	GO	NOT
PROG	T	QUOTE	CSETQ	SET
LABEL	F	FUNCTION	SELECT	CONC
	LIST	RETURN		

2. Sample Output from the Q-32 LISP Compilation

The following output is a sample portion of the output from the 7090 produced when compiling LISP for the Q-32. The first part is the compilation of LENGTH, MAP, MAPCON, MAPLIST, ADD1, SUB1, and ZERCP. The second part is the atomic structure of ADD1, MAP, MAPCON, and SUB1, as produced by PUNOBJ. The constants 0, 1, and -1 also appear.

*N is always the number of cells used on the pushdown list (in the code of LENGTH, 6).

The compiler listing should be consulted and the function definitions compared with the compiled code given here.

a. Part 1

```
                                    (LENGTH SUBR 1 6)
045314 0510000001777776                (STZ (5 *N) 1)
045315 2000000000074146                (LDA (QUOTE 0))
045316 5000000001777776                (STA (5 *N) 1)
                          G00730
045317 2000000001777774                (LDA (3 *N) 1)
045320 6010000000045323                (BNZ G00734)
045321 2000000001777776                (LDA (5 *N) 1)
045322 7307410004040011                (BSX *RETRN 4 (E LENGTH))
                          G00734
                          G00733
045323 2000000001777776                (LDA (5 *N) 1)
045324 7307414704040010                (BSX *CALL 4 (E ADD1))
045325 5000000001777776                (STA (5 *N) 1)
045326 2000000001777774                (LDA (3 *N) 1)
045327 7307400504040010                (BSX *CALL 4 (E CDR))
045330 5000000001777774                (STA (3 *N) 1)
045331 0140000000045317                (BUC G00730)

                                    (MAP SUBR 2 10)
045332 0140000400044024                (BUC SPECBIND 0 4)
                          G00741
045333 4007414001777771                (CAS (4 *N) 1 (E FN))
045334 0510000001777776                (STZ (9 *N) 1)
045335 2000000001777770                (LDA (3 *N) 1)
045336 5000000001777776                (STA (9 *N) 1)
                          G00746
```

```
045337 2000000001777776                (LDA (9 *N) 1)
045340 6010000000045342                (BNZ G00750)
045341 0140000000045350                (BUC G00745)
                                G00750
                                G00749
045342 2000000001777776                (LDA (9 *N) 1)
045343 7307414004040010                (BSX *CALL 4 (E FN))
045344 2000000001777776                (LDA (9 *N) 1)
045345 7307400504040010                (BSX *CALL 4 (E CDR))
045346 5000000001777776                (STA (9 *N) 1)
045347 0140000000045337                (BUC G00746)
                                G00745
045350 5000000001777774                (STA (7 *N) 1)
045351 7304533304044041                (BSX SPECRSTR 4 G00741)
045352 2000000001777774                (LDA (7 *N) 1)
045353 7307415004040011                (BSX *RETRN 4 (E MAP))

                                (MAPCON SUBR 2 12)
045354 0140000400044024                   (BUC SPECBIND 0 4)
                                G00760
045355 4007414001777767                (CAS (4 *N) 1 (E FN))
045356 2000000001777766                (LDA (3 *N) 1)
045357 6010000000045361                (BNZ G00763)
045360 0140000000045376                (BUC G00762)
                                G00763
045361 2000000001777766                (LDA (3 *N) 1)
045362 7307414004040010                (BSX *CALL 4 (E FN))
045363 5000000001777774                (STA (9 *N) 1)
045364 2000000001777766                (LDA (3 *N) 1)
045365 7307400504040010                (BSX *CALL 4 (E CDR))
045366 5000000001777776                (STA (11 *N) 1)
045367 2000000020074140                (LDA (SPECIAL FN))
045370 2200000001777776                (LDB (11 *N) 1)
045371 5040000001000003                (STB 3 1)
045372 7307415104040010                (BSX *CALL 4 (E MAPCON))
045373 2200000001777774                (LDB (9 *N) 1)
045374 5040000001000003                (STB 3 1)
045375 7307414504040010                (BSX *CALL 4 (E NCONC))
                                G00765
                                G00762
045376 5000000001777772                (STA (7 *N) 1)
045377 7304535504044041                (BSX SPECRSTR 4 G00760)
045400 2000000001777772                (LDA (7 *N) 1)
045401 7307415104040011                (BSX *RETRN 4 (E MAPCON))

                                (MAPLIST SUBR 2 12)
045402 0140000400044024                   (BUC SPECBIND 0 4)
                                G00774
```

```
045403 40074140001777767                (CAS (4 *N) 1 (E FN))
045404 20000000001777766                (LDA (3 *N) 1)
045405 60100000000045407                (BNZ G00777)
045406 01400000000045424                (BUC G00776)
                                G00777
045407 20000000001777766                (LDA (3 *N) 1)
045410 73074140040040010                (BSX *CALL 4 (E FN))
045411 50000000001777774                (STA (9 *N) 1)
045412 20000000001777766                (LDA (3 *N) 1)
045413 73074005040040010                (BSX *CALL 4 (E CDR))
045414 50000000001777776                (STA (11 *N) 1)
045415 20000000200741400                (LDA (SPECIAL FN))
045416 22000000001777776                (LDB (11 *N) 1)
045417 50400000001000003                (STB 3 1)
045420 73074104040040010                (BSX *CALL 4 (E MAPLIST))
045421 22000000001777774                (LDB (9 *N) 1)
045422 50400000001000003                (STB 3 1)
045423 73074011040040010                (BSX *CALL 4 (E CONS))
                                G00779
                                G00776
045424 50000000001777772                (STA (7 *N) 1)
045425 73045403040440410                (BSX SPECRSTR 4 G00774)
045426 20000000001777772                (LDA (7 *N 1)
045427 73074104040040011                (BSX *RETRN 4 (E MAPLIST))

                                (ADD1 SUBR 1 4)
045430 50000000001000003                   (STA 3 1)
045431 20000000000741520                   (LDA (QUOTE 1))
045432 73074032040040010                   (BSX *CALL 4 (E *PLUS))
045433 73074147040040011                   (BSX *RETRN 4 (E ADD1))

                                (SUB1 SUBR 1 4)
045434 50000000001000003                   (STA 3 1)
045435 20000000000741541                   (LDA (QUOTE -1))
045436 73074032040040010                   (BSX *CALL 4 (E *PLUS))
045437 73074153040040011                   (BSX *RETRN 4 (E SUB1))

                                (ZEROP SUBR 1 4)
045440 50000000001000003                   (STA 3 1)
045441 20000000000741460                   (LDA (QUOTE 0))
045442 73074131040040010                   (BSX *CALL 4 (E EQUAL))
045443 73074155040040011                   (BSX *RETRN 4 (E ZEROP))
```

b. Part 2

```
074147 20167342001673410        ADD1
167342 00000000001005520
100553 21242401777777777
```

```
100552 0300000104167342
167340 2000000075100554      40104543Q1
100555 0000000401045430
100554 0200000100167340

167341 0000000000167340

074150 2016733700167336      MAP
167337 0000000000100556
100557 4421477777777777
100556 0300000103167337
167335 2000000075100560      12020453320
100561 0000001202045332
100560 0200000100167335

167336 0000000000167335

074151 2016733400167333      MAPCON
167334 0000000000100562
100563 4421472346457777
100562 0300000106167334
167332 2000000075100564      14020453540
100565 0000001402045354
100564 0200000100167332

167333 0000000000167332

167331 2000000075100566      1
100567 0000000000000001
100566 0200000100167331

074152 0000000000167331

074153 2016733000167327      SUB1
167330 0000000000100570
100571 6264220177777777
100570 0300000104167330
167326 2000000075100572      4010454340
100573 0000000401045434
100572 0200000100167326

167327 0000000000167326

167325 2000000075100574      -1
100575 7777777777777776
100574 0200000100167325

074154 0000000000167325
```

An Auxiliary Language for More Natural Expression — the A–Language

William Henneman

Information International, Inc.

Although LISP is one of the most powerful tools available to the research worker in many fields of programming, the format of the input language (S-expression) is awkward to use -- so awkward that many errors in programming for LISP stem directly from the fact that S-expressions are the only allowable form of input. The inherent difficulty of producing correctly working S-expressions is tacitly recognized by anyone who uses M-expressions in place of them, when not communicating directly with the computer.

The most striking example of this difficulty is the extreme number of parentheses used in writing S-expressions. Thus, the function of "firstatom" is defined in M-expression notation as

```
firstatom [x] = [null [x] → NIL;
                 atom [car  [x]] → car  [x] ;
                 T → firstatom   [car  [x]]] .
```

The corresponding S-expression is

```
DEFINE (((FIRSTATOM (LAMBDA (X) (COND
       ((ATOM (CAR X)) (CAR X))
       (T (FIRSTATOM (CAR X))))))))
```

This simple function has 26 parentheses, and a long compli-

cated program would have many more. This proliferation of parentheses makes it very difficult to write such complicated programs without an error in placement or pairing of parentheses. Indeed, in some quarters, "LISP" is considered to be an acronym for "Lots of Irritating Single Parentheses".

The worst feature of this plethora of parentheses is the fact that a localized error in parenthesization can cause global errors in the LISP system. For example, an unpaired right parenthesis on any line can cause reading of the input to terminate at that line. An unpaired left parenthesis can cause most of the program to be unexecuted.

Another inconvenience associated with S-expression is the fact that Polish notation is mandatory. Now it is harder to read Polish notation than infix notation, if only because we were all taught infix notation in school. In logic and in other programming systems, it is worthwhile to pay the price of readability for prefix notation since one is rewarded by having the need for parentheses eliminated; in LISP, we are actually punished -- the contrast between the M-expression a $\vee$ b $\wedge$ c and the corresponding S-expression (OR A (AND B C)) illustrates this point.

The names of the most basic functions have no mnemonic value to the programmer at the level at which he wishes to think when he is programming. This feature is worst for beginners, of course, but all LISP programmers must at one time be beginners, and why make life harder for them than it need be?

All this points toward the need for a more natural input language; this language is called the A-Language, and is described in succeeding paragraphs.

The A-Language is a language which allows expressions to be written as a mixture of M-expressions, A-expressions, and S-expressions. The reader is presumed to be familiar with S-expressions and M-expressions.

A-expressions are basically similar to Algol statements. As an example, the expression

```
(DEFINE FIRSTATOM (IN) (6) (FIRSTATOM IN X)
(IF X IS EMPTY THEN NIL ELSE IF FIRST OF X IS
ATOMIC THEN FIRST OF X ELSE FIRSTATOM IN FIRST OF X))
```

is the A-expression equivalent of the function defined above. This example is explained in detail below.

The first element of this A-expression is the word DEFINE,

and indicates to the A-Language compiler that this is a definition statement.

The second element is the name of the function being defined.

The third element of the expression is a list of auxiliary words; their only function is to make the program more readable, and they are edited out during compilation.

As an example, the function subst, familiar to readers of the LISP 1.5 Manual, can be defined in A-Language as follows:

```
(DEFINE SUBSTITUTING (THE RESULT OF FOR IN) (7)
(THE RESULT OF SUBSTITUTING X FOR Y IN Z) (SUBST X Y Z))
```

It would be used as follows:

```
(IF THE RESULT OF SUBSTITUTING X FOR Y IN Z EQUALS Z
THEN PRINT Z ELSE X)
```

The fourth element is the precedence value; this value is specified by the programmer, and determines the hierarchy of applications of functions. Its main function is to eliminate parentheses. Thus if the precedence of + was set higher than •, the expression a+b•c would be translated as (a+b)•c. If there were no precedence relationship between + and •, a+b•c would be ambiguous.

There are times when one wishes to vary the precedence of operations to be applied. This is accomplished in the language in two ways -- by writing the special words "begin" and "end" before and after the unit which is to receive highest precedence, or by using parentheses. Thus, for example, both the expressions

```
(a + b) (c + d)
begin a+b end x begin c+d end
```

will be equivalent to

```
ac + bc + ad +bd
```

The fifth element is the example. This shows the form in which the function being defined will appear in use. It specifies the position in the expression occupied by the function name, the auxiliary words, and the variables.

The sixth and last element in the list is the definition of the function. This corresponds to the expression on the right hand side of the equal sign in the M-expression definition of a function.

If the definition makes use of conditionals, they are expressed in the IF, THEN, ELSE terminology of Algol; if there is no IF immediately following the final ELSE, a T is automatically supplied in the antecedent of the final condition.

The function FIRSTATOM defined above illustrates how conditionals in A-expressions are written. Note that A-system expressions FIRST OF X, X IS ATOMIC, X IS EMPTY, etc., are used rather than their S-expression counterparts (CAR X), (ATOM X), (NULL X), etc. These A-expressions are easier to remember and to comprehend than the corresponding S-expressions. A list of certain basic S-expressions and their A-Language equivalents is given in Table 1.

Table 1

S-Expression	A-Language	English
(CAR X)	(DEFINE FIRST (OF) (11) (FIRST OF X) (CAR X))	the first of list x
(CDR X)	(DEFINE REST (OF) (11) (REST OF X) (CDR X))	the rest of list x
(CONS X Y)	(DEFINE CONNECT (TO) (10) (CONNECT X TO Y) (CONS X Y))	the cons of x and y
(ATOM X)	(DEFINE ATOMIC (IS) (10) (X IS ATOMIC) (ATOM X))	x is atomic
(EQUAL X Y)	(DEFINE EQUALS (NIL) (4) (X EQUALS Y) (EQUAL X Y))	x is equal to y
n^{th} element	(DEFINE ELEMENT (OF TH) (2) (N TH ELEMENT OF X) (IF N EQUALS 1 THEN FIRST OF X ELSE N MS 1 TH ELEMENT OF REST OF X))	the nth element of list x
(MEMBER X A)	(DEFINE MEMBER (IS A OF) (6) (X IS A MEMBER OF Y) (IF Y IS EMPTY THEN NIL ELSE IF X EQUALS FIRST OF Y THEN T ELSE X IS A MEMBER OF REST OF Y))	x is a member of y
(UNION X Y)	(DEFINE UNION (OF AND) (8) (UNION OF X AND Y) (IF X IS EMPTY THEN Y ELSE IF FIRST OF X IS A MEMBER OF Y THEN UNION OF REST OF X AND Y ELSE	the union of x and y

S-Expression	A-Language	English
	CONNECT FIRST OF X TO BEGIN UNION OF REST OF X AND Y END))	
(INTERSECTION X Y)	(DEFINE INTERSECTION (OF AND) (8) (INTERSECTION OF X AND Y) (IF X IS EMPTY THEN NIL ELSE IF FIRST OF X IS A MEMBER OF Y THEN CONNECT FIRST OF X TO INTERSECTION OF REST OF X AND Y ELSE INTERSECTION OF REST OF X AND Y))	the intersection of x and y
(PLUS X Y)	(DEFINE PS (NIL) (5) (X PS Y) (PLUS X Y))	x plus y
(DIFFERENCE X Y)	(DEFINE MS (NIL) (5) (X MS Y) (PLUS X (MINUS Y)))	x minus y
(TIMES X Y)	(DEFINE * (NIL (6) (X * Y) (TIMES X Y))	x multiplied by y
(QUOTIENT X Y)	(DEFINE / (NIL (6) (X / Y) (QUOTIENT X Y))	x divided by y
(CSETQ Y X)	(DEFINE PUT (INTO) (0) (PUT X INTO Y) (CSETQ Y X))	let y equal x

In the A-Language, one may choose to use infix notation, Polish notation, or some other placement of the operator in relation to the variable, and specify this position in the defining example. By judicious choice, one may make the notation fit the problem at hand in the most natural manner.

An S-expression is allowed as any part of an A-expression. Thus, the definition of firstatom could just as well have been written (IF (NULL X) THEN NIL ELSE IF (CAR X) IS ATOMIC THEN FIRST OF X ELSE (FIRSTATOM (CAR X))),

or

(IF (NULL X) THEN NIL ELSE IF (ATOMIC (FIRST X)) THEN (FIRST X) ELSE (FIRSTATOM (FIRST X))).

(In the preceding expression, ATOMIC is in A-Language, and has the same meaning and function as the predicate ATOM in S-expressions.)

The Translator of A-Language

The translator (the system for translating A-Language into

LISP) was designed to eliminate the possibility of a programmer-caused fatal error in compilation. This causes some loss of efficiency in compile time, but this will hopefully be more than offset by having the number of debugging efforts reduced.

Aside from checking for obvious stupidities (omitted parentheses, dropped auxiliary words, etc.), there is a fairly elaborate function which checks to see if the translation is "making any sense" -- i.e., to see if functions have the proper number of variables, if auxiliary words have all been edited out, etc. The one great cause of most of the incorrect results obtained in practice is an incorrect precedence being assigned to a function. For example, when the functions THIRD and REST have precedences 10 and 9, respectively, the function

THIRD OF REST OF X

was incorrectly translated as

((THIRD REST) X).

The function that checks for incorrect arguments of functions finds that REST is a function, and signals the translator.

A function is planned (and is partially running) which will search for an interpretation of the offending statement that produces meaningful output, and which will inform the programmer of this "hunch" on the part of the compiler. If the compiler was correct, the programmer may then forget the error -- if not, compilation will not be halted. In most cases of incorrectly assigned precedences, the translator will assign a consistent meaning to an expression. If, however, it cannot, a fatal error occurs, and compilation is halted.

Several programs have been successfully compiled with the current version of the translator. Surprisingly, the compile times have been just a fraction more than the corresponding compile times from the corresponding S-expression format. Since the translator is written entirely in S-expressions, and since the number of auxiliary functions was held to a minimum, this is a rather remarkable feat, and speaks well of the innate power of LISP 1.5.

In view of our experience, it would appear that the goal of providing a special-purpose language to workers in each area of science is quite attainable with LISP and A-Language.

An Illustrative Example

In order to illustrate the various points discussed in the preceding pages, let us define the various functions needed to trace a path from the entrance to the exit of a maze, such as the one in Figure 1.

For convenience, we shall replace the letters that label the branch points in the maze by their numeric positions in the alphabet -- thus an "a" will become 1, a "b", 2, etc.; and we shall represent the maze by a list whose n^{th} entry is the list of points that can be reached in one step from the point n of the maze. The maze in the figure is then represented by the list

((22) () () (5 6 8) () (7) (6) () (10) (9 11) (10 12)
(13 14 15) (2 19 20) () (16 17 18) () () () () () ()
(3 4 9)).

We shall denote the n^{th} element of the list by $\Gamma[n]$, GAMMA of n.

It will be useful in the sequel if we have a convenient way of talking about the set of points that can be reached from one or the other of the points belonging to a set A in one step; we shall denote this set by $G[A]$, and define it by the equation:

$$G[A] = \bigcup_{x \text{ in } A} \Gamma x$$, which is the union of all the gamma's for each x in A.

There is one other function that will have great interest for us, and that is the transitive closure of a point n. This is the set of all points that can be reached in any number of moves starting from the point n of the maze. We use the notation $\hat{\Gamma}[n]$ to stand for the transitive closure of n. The usefulness of this function is seen upon observing that a path between the points a and b exists if and only if b is a member of the transitive closure of a. Since, for any point x in a maze,

$$\Gamma[\hat{\Gamma}[n]] = \Gamma[n]$$

an effective computation procedure for the function $\hat{\Gamma}$ is to compute $\Gamma[n]$, $\Gamma[n \cup \Gamma[n]]$, ..., until the result is equal to the argument.

The algorithm for finding the path from a to b in a maze is based upon the remark connecting the concepts of transitive

*This problem, but not the solution outlined above, is taken from Brege, The Theory of Graphs and its Applications, New York, 1962.

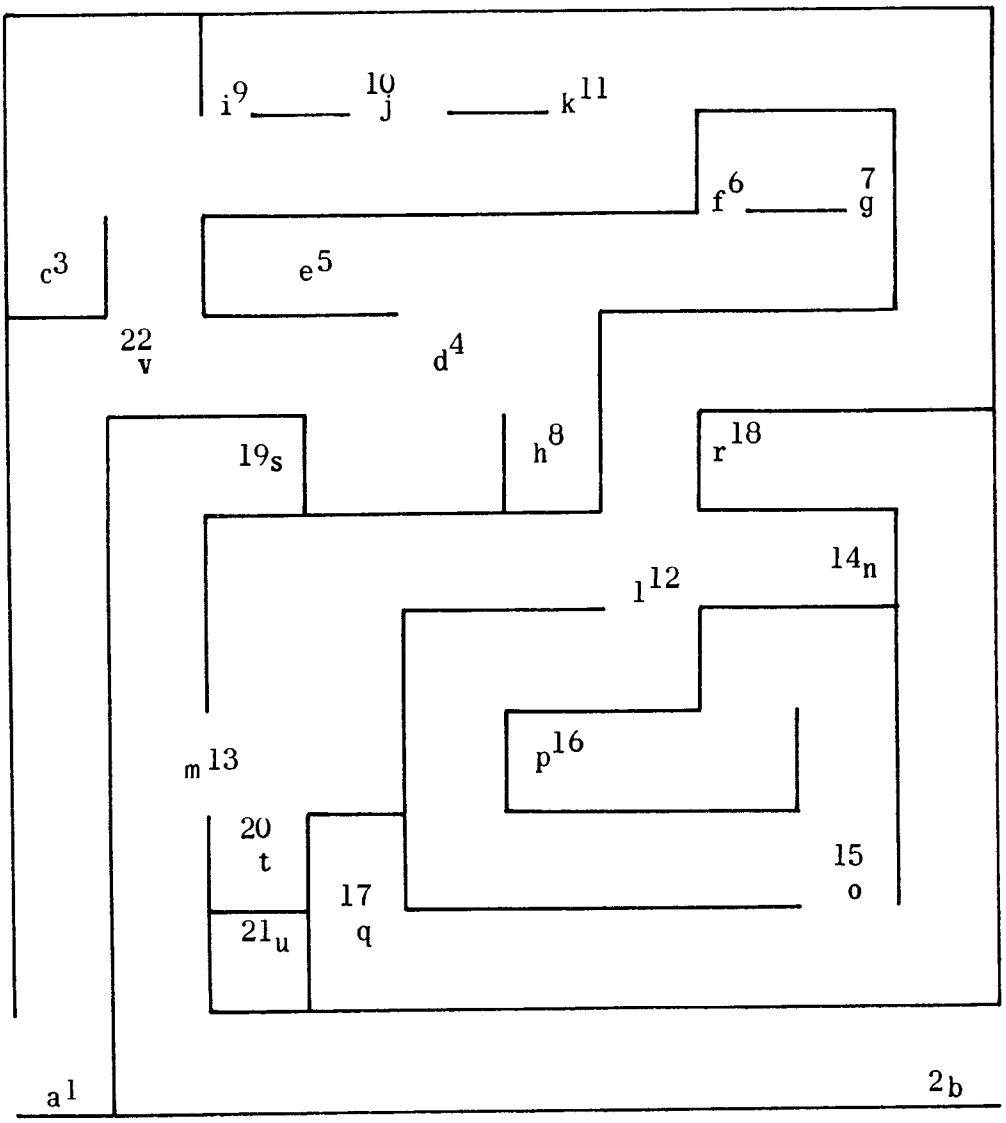

i 9
10 j
k 11
f 6
7 g
c 3
e 5
22 v
d 4
19 s
h 8
r 18
14 n
l 12
m 13
p 16
20 t
15 o
17 q
21 u
a 1
2 b

Figure 1

closure and path made above. The only complication comes from the fact that the presence of loops in the path chosen may cause infinite recursion if care is not taken to avoid it. However, it is clear that if a path with loops links the points a and b in a maze, then there is a path without loops linking them as well.

We are now in a position to write the main functions for our program. These are:

```
(DEFINE PATH (THE FROM TO SAVING) (1) (THE PATH FROM A
   TO  B SAVING C) (IF B IS A MEMBER OF GAMMA OF A
   THEN CONNECT B TO C ELSE IF B IS A MEMBER OF THE
   TRANSITIVE CLOSURE OF FIRST OF A THEN IF FIRST OF
   A IS A MEMBER OF C THEN THE PATH FROM REST OF A TO
   B SAVING C ELSE THE PATH FROM GAMMA OF FIRST OF C
   TO B SAVING CONNECT FIRST OF A TO C ELSE THE PATH
   FROM REST OF A TO B SAVING C));

(DEFINE CLOSURE (THE TRANSITIVE OF) (9)(THE TRANSITIVE
   CLOSURE OF X) (IF GAMMA OF X EQUALS X THEN X ELSE
   THE TRANSITIVE CLOSURE OF THE UNICN OF X AND GAMMA
   OF X));

(DEFINE GAMMA (OF) (7) (GAMMA OF X) (IF X IS EMPTY THEN
   NIL ELSE THE UNION OF THE FIRST OF A TH ELEMENT OF
   M AND GAMMA OF REST OF X)).
```

The precedence values were assigned by observing that:

1. CLOSURE must have a precedence value somewhere between that of member and union;
2. GAMMA must have a precedence value somewhere between connect and union;
3. PATH must have a precedence value less than any other function;

referring to the precedence values in table 1, we see that the values assigned just turn the trick. The input and translated program, with the final results of the run, are given below.

Input

```
(DEFINE PATH (THE FROM TO SAVING) (1) (THE PATH FROM A TO B
     SAVING C) (IF B IS A MEMBER OF GAMMA OF A THEN CONNECT B
     TO C ELSE IF B IS A MEMBER OF THE TRANSITIVE CLOSURE OF
     FIRST OF A THEN IF FIRST OF A IS A MEMBER OF C THEN THE
     PATH FROM REST OF A TO B SAVING C ELSE THE PATH FROM GAMMA
     OF FIRST OF A TO B SAVING CONNECT A TO C ELSE THE PATH FROM
     REST OF A TO B SAVING C))
```

```
(DEFINE CLOSURE (THE TRANSITIVE OF) (9) (THE TRANSITIVE CLOSURE
     OF X) (IF GAMMA OF X EQUALS X THEN X ELSE THE TRANSITIVE
     CLOSURE OF THE UNION OF X AND GAMMA OF X))

(DEFINE GAMMA (OF) (7) (GAMMA OF X) (IF X IS EMPTY THEN NIL
     ELSE THE UNION OF THE FIRST OF A TH ELEMENT OF M AND GAMMA
     OF REST OF X))

(PUT (LIST ((22) () () (5 6 8) () (7) (6) () (10) (9 11) (10 12)
     (13 14 15) (2 19 20) () (16 17 18) () () () () () ()
     (3 4 9))) INTO M)

(PATH FROM (1) TO (2) SAVING NIL)

(STOP)
```

Translated Program

```
DEFINE (((PATH (LAMBDA (A B C) (COND
        ((MEMBER B (GAMMA A)) (CONNECT B C))
        ((MEMBER B (CLOSURE (FIRST A))) (COND
                    ((MEMBER (FIRST A) C) (PATH (REST A) B C))
                    (T (PATH (GAMMA (FIRST A)) B (CONNECT A C)))))
         (T (PATH (REST A) B C)))))))

DEFINE (((CLOSURE (LAMBDA (X) (COND
          ((EQUAL (GAMMA X) X) X)
          (T (CLOSURE (UNION X (GAMMA X)))))))))

DEFINE (((GAMMA (LAMBDA (X) (COND
          ((NULL X) NIL)
          (T (UNION (ELEMENT (FIRST A) M) (GAMMA (REST X))))))))))

(CSETQ (LIST ((22) () () (5 6 8) () (7) (6) () (10) (9 11) (10 12)
     (13 14 15) (2 19 20) () (16 17 18) () () () () () ()
     (3 4 9))) M)

(PATH (1) (2) NIL)

(STOP)
```

Answer

```
(2 13 12 11 10 9 22 1)
```

Appendix 1

The LISP Program for METEOR

Daniel G. Bobrow

This listing is current as of February 20, 1964 and has all known errors removed. At the top of the first page is a list of free or "special" variables used in the compiled program. This listing was made with the aid of a format printing program written by Daniel Edwards of Massachusetts Institute of Technology, and the lines giving a value and the label $/ FUNCTION DEFINITIONS/ are artifacts of this program.

The principal functions in METEOR are:

1. METRIX2: which performs transfer of control from rule to rule;

2. COMITRULE: which performs tracing and flow of control through each rule;

3. CMATCH (called from COMITMATCH): which performs the left half match;

4. COMITRIN (called from COMITR): which creates the transformed workspace and lists for the shelves;

5. SHELVE: which performs the shelving operations.

```
((MPAIRS) (PRS) (WORKSPACE) (DISPCH) (SHELF) (TRACK))


$/FUNCTION DEFINITIONS/


(METEOR
   (LAMBDA (RULES WORKSPACE) (METRIX RULES WORKSPACE NIL NIL
 NIL)))

(METRIX
   (LAMBDA (RULES WORKSPACE SHELF DISPCH TRACK) (METRIX2 RULES
 WORKSPACE)))

(METRIX2
   (LAMBDA (RULES WORKSPACE) (PROG (PC GT A)
             (SETQ PC RULES)
       START (COND
                ((NULL PC) (RETURN (PROG2 (PRINT (QUOTE (OVER
 END OF PROGRAM))) WORKSPACE))))
             (SETQ GT (DISPATCH (COMITRULE (CAR PC))))
             (COND
                ((EQ GT (QUOTE *)) (GO NEXT))
                ((EQ GT (QUOTE END)) (RETURN WORKSPACE))
                ((EQUAL GT (CAAR PC)) (GO START)))
             (SETQ A (TRANSFER GT RULES))
             (COND
                ((EQ (CAR A) (QUOTE NONAME)) (RETURN (PROG2
(PRINT (LIST (CADR A) (QUOTE (UNDEFINED GO-TO IN)) (CAR PC)
)) WORKSPACE))))
             (SETQ PC A)
             (GO START)
       NEXT  (SETQ PC (CDR PC))
             (GO START))))

(COMITRULE
   (LAMBDA (RULE) (PROG (A B C D E G M LEFT)
             (SETQ G RULE)
       TOP   (SETQ RULE (CDR RULE))
             (SETQ A (CAR RULE))
             (SETQ E (QUOTE *))
             (COND
                ((NOT (ATOM A)) (GO START))
                ((EQ A (QUOTE *)) (GO STAR))
                ((EQ A (QUOTE *M)) (GO *M))
                ((EQ A (QUOTE *T)) (GO *T))
                ((EQ A (QUOTE *U)) (GO *U)))
             (DEFLIST (CDR RULE) A)
             (RETURN (QUOTE *))
       STAR  (SETQ RULE (CDR RULE))
             (SETQ E (FSTATM RULE))
       START (COND
                ((AND
                   (NULL TRACK)
                   (NULL M)) (GO TRACK)))
             (PRINT (QUOTE WORKSPACE))
             (PRINT WORKSPACE)
             (PRINT (QUOTE RULE))
             (PRINT G)
```

```
       TRACK (SETQ LEFT (COMITMATCH (CAR RULE) WORKSPACE))
             (COND
                ((NULL LEFT) (RETURN E)))
       LOOP  (SETQ RULE (CDR RULE))
             (SETQ A (CAR RULE))
             (COND
                ((NULL RULE) (RETURN E))
                ((EQ A (QUOTE $)) (GO DOLL))
                ((EQUAL A 0) (GO ON))
                ((ATOM A) (GO SW))
                ((EQ (CAR A) (QUOTE /)) (GO SHELVE)))
       ON    (SETQ WORKSPACE (COMITR LEFT A))
             (COND
                (M (PROG2 (PRINT (QUOTE WORKSPACE)) (PRINT WORKSPACE
))))
             (GO LOOP)
       DOLL  (SETQ A (CAR WORKSPACE))
       SW    (COND
                ((EQ E (QUOTE *)) (RETURN A)))
             (RETURN (QUOTE *))
       SHELVE (SHELVE LEFT A)
             (GO LOOP)
       *M    (SETQ M A)
             (GO TOP)
       *T    (SETQ TRACK A)
             (GO TOP)
       *U    (SETQ TRACK NIL)
             (GO TOP))))

(TRANSFER
   (LAMBDA (GT RL) (PROG NIL
      START (COND
                ((NULL RL) (RETURN (LIST (QUOTE NONAME) GT))
)
                ((EQ GT (CAAR RL)) (RETURN RL)))
            (SETQ RL (CDR RL))
            (GO START))))

(DISPATCH
   (LAMBDA (GT) (PROG (A)
            (COND
               ((EQ GT (QUOTE *)) (RETURN GT)))
            (SETQ A (GTPAIR GT DISPCH))
            (COND
               ((NULL A) (RETURN GT)))
            (RETURN (CAR A)))))

(GTPAIR
   (LAMBDA (NAME X) (PROG (A)
      START (COND
               ((NULL X) (RETURN NIL))
               ((EQUAL (CAR X) NAME) (RETURN (CDR X))))
            (SETQ X (CDDR X))
            (GO START))))
```

```
(FSTATM
   (LAMBDA (RULE) (PROG (A)
      START (SETQ A (CAR RULE))
            (COND
               ((NULL RULE) (RETURN (QUOTE *)))
               ((EQUAL A 0) (GO ON))
               ((ATOM A) (RETURN A)))
      ON    (SETQ RULE (CDR RULE))
            (GO START))))

(SHELVE
   (LAMBDA (PAIRS INST) (PROG (A B C D)
      START (SETQ INST (CDR INST))
            (COND
               ((NULL INST) (RETURN SHELF)))
            (SETQ A (CAR INST))
            (SETQ B (CAR A))
            (SETQ C (CADR A))
            (SETQ D (CDDR A))
            (COND
               ((EQ B (QUOTE *P)) (GO PR))
               ((EQ B (QUOTE *D)) (RETURN (SETDIS C (CAR D)
)))
               ((NOT (EQ C (QUOTE *))) (GO GETD)))
            (SETQ C (INDIRECT (CAR D) PAIRS))
            (SETQ D (CDR D))
      GETD  (SETQ D (COMITRIN PAIRS D))
            (SETQ A (GTSHLF C))
            (COND
               ((EQ B (QUOTE *S)) (GO ST1))
               ((EQ B (QUOTE *Q)) (GO QU1))
               ((EQ B (QUOTE *X)) (GO EX)))
            (PRINT (LIST (QUOTE (SHELVING ERROR IN)) (CAR INST
)))
            (GO START)
      PR    (COND
               ((EQ C (QUOTE /)) (RETURN (PRINT SHELF))))
      PR1   (PRINT (LIST (QUOTE SHELF) C (QUOTE CONTAINS) (
CAR (GTSHLF C))))
            (COND
               ((NULL D) (GO START)))
            (SETQ C (CAR D))
            (SETQ D (CDR D))
            (GO PR1)
      EX    (SETQ B (CAR A))
            (RPLACA A WORKSPACE)
            (SETQ WORKSPACE B)
            (GO START)
      QU1   (RPLACA A (NCONC (CAR A) D))
            (GO START)
      ST1   (RPLACA A (APPEND D (CAR A)))
            (GO START))))

(SETDIS
   (LAMBDA (X Y) (PROG (A)
            (SETQ A (GTPAIR X DISPCH))
            (COND
               ((NULL A) (SETQ DISPCH (CONS X (CONS Y DISPCH
))))
               (T (RPLACA A Y)))
            (RETURN DISPCH))))
```

```
(GETDCT
   (LAMBDA (X Y) (PROG (A)
            (COND
               ((NOT (ATOM X)) (RETURN (LIST X))))
            (SETQ A (GET X Y))
            (COND
               ((NULL A) (RETURN X)))
            (RETURN A))))

(INDIRECT
   (LAMBDA (X PAIRS) (GTNAME X PAIRS)))

VALUE
(METEOR METRIX METRIX2 COMITRULE TRANSFER DISPATCH GTPAIR FSTATM
SHELVE SETDIS GETDCT INDIRECT)
$/FUNCTION DEFINITIONS/


(COMITR
   (LAMBDA (LEFT ORDER) (PROG (A B C)
            (SETQ A (GTNAME 0 LEFT))
            (COND
               ((EQUAL A 0) (SETQ A NIL))
               ((NULL A) (GO ON))
               ((ATOM A) (SETQ A (LIST A))))
      ON    (SETQ B (GTNAME (QUOTE WSEND) LEFT))
            (COND
               ((EQUAL ORDER 0) (SETQ C NIL))
               (T (SETQ C (COMITRIN LEFT ORDER))))
            (RETURN (APPEND A (APPEND C B))))))

(COMITRIN
   (LAMBDA (LEFT ORDER) (PROG (A B)
      START (COND
               ((NULL ORDER) (RETURN A)))
            (SETQ B (GTNAME (CAR ORDER) LEFT))
            (COND
               ((NULL B) (GO ON))
               ((ATOM B) (SETQ B (LIST B))))
            (SETQ A (NCONC A B))
      ON    (SETQ ORDER (CDR ORDER))
            (GO START))))

(GTNAME
   (LAMBDA (NAME PRS) (PROG (A B C)
            (SETQ C (CAR NAME))
            (COND
               ((ATOM NAME) (GO START))
               ((EQ C (QUOTE FN)) (RETURN (COPYTP (APPLY (CADR
 NAME) (COMITRIN PRS (CDDR NAME)) NIL))))
               ((EQ C (QUOTE *K)) (RETURN (LIST (COMITRIN PRS
 (CDR NAME)))))
               ((EQ C (QUOTE *C)) (RETURN (COMPRESS (COMITRIN
 PRS (CDR NAME)))))
               ((EQ C (QUOTE *)) (RETURN (COPYTP (EVAL (CADR
 NAME) NIL))))
               ((EQ C (QUOTE *W)) (RETURN (WRITES (COMITRIN
 PRS (CDR NAME)))))
               ((EQ C (QUOTE *E)) (RETURN (EXPAND (GTNAME (
CADR NAME) PRS))))
               ((EQ C (QUOTE */)) (RETURN (LIST (SBMERGE (CDR
```

```
 NAME)))))
                  ((EQ C (QUOTE *N)) (RETURN (NEXT (CDR NAME))
))
                  ((EQ C (QUOTE *R)) (RETURN (MTREAD)))
                  ((EQ (CADR NAME) (QUOTE /)) (RETURN (LIST (SBMERGE
 (LIST (QUOTE MERGE) C (CONS (QUOTE G99999) (CDR NAME))))))
)
                  ((EQ C (QUOTE *F)) (RETURN (CAAR (GTNAME (CADR
 NAME) PRS))))
                  ((EQ C (QUOTE *A)) (RETURN (ALL (CDR NAME)))
)
                  ((EQ C (QUOTE QUOTE)) (RETURN (CADR NAME))))
      START (COND
                  ((NULL PRS) (RETURN NAME)))
            (SETQ A (CAR PRS))
            (COND
               ((EQUAL NAME (CAR A)) (RETURN (COPYTP (CDR A
)))))
            (SETQ PRS (CDR PRS))
            (GO START))))

(COPYTP
   (LAMBDA (X) (COND
      ((ATOM X) X)
      (T (APPEND X NIL)))))

(EXPAND
   (LAMBDA (X) (COND
      ((ATOM X) (MAPCON (GET (CDR X) (QUOTE PNAME)) (FUNCTION
 (LAMBDA (Y) (UNPACK (CAR Y))))))
      (T (CAR X)))))

(COMPRESS
   (LAMBDA (X) (PROG NIL
            (CLEARBUFF)
            (MAP X (FUNCTION (LAMBDA (X) (PACK (CAR X)))))
            (RETURN (INTERN (MKNAM))))))

(MTREAD
   (LAMBDA NIL (PROG (A B C)
            (SETQ A (STARTREAD))
            (GO A)
      START (SETQ A (ADVANCE))
      A     (COND
               ((EQ A (QUOTE $EOF$)) (RETURN A))
               ((EQ A (QUOTE $EOR$)) (RETURN B))
               ((EQ A (QUOTE  )) (SETQ C (NCONC C (LIST A))
))
               (T (GO B)))
            (GO START)
      B     (SETQ B (NCONC B (NCONC C (LIST A))))
            (SETQ C NIL)
            (GO START))))

(ALL
   (LAMBDA (X) (PROG (A B)
            (COND
               ((EQ (CAR X) (QUOTE *)) (SETQ X (INDIRECT (CADR
```

```
  X) PRS)))
                    (T (SETQ X (CAR X))))
                 (SETQ A (GTSHLF X))
                 (SETQ B (CAR A))
                 (RPLACA A NIL)
                 (RETURN B))))

(NEXT
   (LAMBDA (X) (PROG (A B C)
                 (COND
                    ((EQ (CAR X) (QUOTE *)) (SETQ X (INDIRECT (CADR
 X) PRS)))
                    (T (SETQ X (CAR X))))
                 (SETQ A (GTSHLF X))
                 (SETQ C (CAR A))
                 (COND
                    ((NULL C) (RETURN NIL)))
                 (SETQ B (CAR C))
                 (RPLACA A (CDR C))
                 (RETURN (LIST B)))))

(GTSHLF
   (LAMBDA (X) (PROG (A)
                 (SETQ A (GTPAIR X SHELF))
                 (COND
                    ((NULL A) (GO A)))
                 (RETURN A)
       A         (SETQ A (CONS NIL SHELF))
                 (SETQ SHELF (CONS X A))
                 (RETURN A))))

(SBMERGE
   (LAMBDA (X) (PROG (A B C D E G)
                 (SETQ A (CAR X))
                 (SETQ B (CADR X))
                 (COND
                    ((EQ (CADR B) (QUOTE /)) (GO BX)))
                 (SETQ B (GTNAME B PRS))
                 (COND
                    ((NOT (ATOM B)) (SETQ B (CAR B))))
       BX        (SETQ C (CADDR X))
                 (COND
                    ((EQ (CADR C) (QUOTE /)) (GO CX)))
                 (SETQ C (GTNAME C PRS))
                 (COND
                    ((NOT (ATOM X)) (SETQ C (CAR C))))
       CX        (COND
                    ((OR
                       (ATOM C)
                       (NOT (EQ (CADR C) (QUOTE /)))) (SETQ C NIL
))
                    (T (SETQ C (CDDR C))))
                 (COND
                    ((OR
                       (ATOM B)
                       (NOT (EQ (CADR B) (QUOTE /)))) (GO B)))
                 (SETQ D (LIST (CAR B) (QUOTE /)))
                 (SETQ B (CDDR B))
                 (GO D)
       B         (SETQ D (LIST B (QUOTE /)))
                 (SETQ B NIL)
```

```
      D     (COND
               ((EQ A (QUOTE AND)) (GO AND))
               ((EQ A (QUOTE MERGE)) (GO AND))
               ((EQ A (QUOTE OR)) (GO OR))
               ((EQ A (QUOTE SUBST)) (GO SUBST)))
      ERROR (PRINT (QUOTE (SUBSCRIPT ERROR)))
            (PRINT X)
            (RETURN (GTNAME (CADR X) PRS))
      AND   (COND
               ((NULL B) (GO RETURN))
               ((MEMBER (CAR B) C) (SETQ G (CONS (CAR B) G)
)))
            (SETQ B (CDR B))
            (GO AND)
      OR    (SETQ G C)
      OR1   (COND
               ((NULL B) (GO RETURN))
               ((NOT (MEMBER (CAR B) G)) (SETQ G (CONS (CAR
 B) G))))
            (SETQ B (CDR B))
            (GO OR1)
      SUBST (SETQ G C)
      RETURN (COND
                ((AND
                   (EQ A (QUOTE MERGE))
                   (NULL G)) (SETQ G C)))
             (COND
                ((NULL G) (RETURN (CAR D))))
             (RETURN (NCONC D G)))))

VALUE
(COMITR COMITRIN GTNAME COPYTP EXPAND COMPRESS MTREAD ALL NEXT
GTSHLF SBMERGE)
$/FUNCTION DEFINITIONS/


(COMITMATCH
   (LAMBDA (RULE WORKSPACE) (PROG (A B)
             (SETQ A (CMATCH (NAMER RULE) WORKSPACE NIL))
             (COND
                ((NULL A) (RETURN NIL))
                ((EQ A (QUOTE $IMP)) (RETURN NIL)))
             (SETQ B (CONS (QUOTE WSEND) (CDR A)))
             (RETURN (ADDLAST (CAR A) B)))))

(CMATCH
   (LAMBDA (RULE WORKSPACE MPAIRS) (PROG (RNAME A B C D E G
 H)
             (SETQ RNAME (CAR RULE))
             (SETQ RULE (CDR RULE))
             (SETQ B (CAR RULE))
             (COND
                ((NULL RULE) (RETURN (CONS MPAIRS WORKSPACE)
))
                ((EQ B (QUOTE $0)) (GO $0))
                ((EQ B (QUOTE $)) (GO PDOLL)))
             (SETQ H (CAR B))
             (COND
                ((EQ H (QUOTE *P)) (GO PRINT))
                ((EQ H (QUOTE FN)) (GO FN))
                ((NULL WORKSPACE) (RETURN (QUOTE $IMP))))
```

```
              (SETQ G 0)
              (COND
                 ((EQ B (QUOTE $1)) (SETQ G 1))
                 ((EQ B (QUOTE $2)) (SETQ G 2))
                 ((EQ B (QUOTE $3)) (SETQ G 3)))
              (COND
                 ((NOT (EQUAL G 0)) (GO NDOLL2)))
              (GO TEST)
       $0     (COND
                 ((AND
                    (NOT (NULL WORKSPACE))
                    (NULL (CDR RULE))) (SETQ B NIL))
                 (T (SETQ B (CONS NIL WORKSPACE))))
              (GO WATB)
       TEST   (COND
                 ((EQ H (QUOTE $)) (GO NDOLL))
                 ((EQ H (QUOTE *)) (GO EVAL))
                 ((EQ H (QUOTE QUOTE)) (GO ATB1))
                 (T (GO ATB)))
       FN     (SETQ B (CDR B))
              (SETQ E (CONS WORKSPACE (COMITRIN MPAIRS (CDR B
))))
              (SETQ B (COPYTP (APPLY (CAR B) E NIL)))
       WATB   (COND
                 ((NULL B) (RETURN NIL))
                 ((EQ B (QUOTE $IMP)) (RETURN B))
                 (T (RETURN (CMATCH (CONS (CDR RNAME) (CDR RULE
)) (CDR B) (ADDLAST MPAIRS (CONS (CAR RNAME) (CAR B)))))))
       PDOLL  (SETQ D (CDR RNAME))
              (SETQ RULE (CDR RULE))
              (COND
                 ((NULL RULE) (RETURN (LIST (ADDLAST MPAIRS (
CONS (CAR RNAME) WORKSPACE))))))
       DLOOP  (SETQ B (CMATCH (CONS D RULE) WORKSPACE MPAIRS)
)
              (COND
                 ((NULL WORKSPACE) (RETURN NIL))
                 ((EQ B (QUOTE $IMP)) (RETURN B))
                 (B (RETURN (CONS (ADDLAST (CAR B) (CONS (CAR
 RNAME) C)) (CDR B)))))
              (SETQ C (ADDLAST C (CAR WORKSPACE)))
              (SETQ WORKSPACE (CDR WORKSPACE))
              (GO DLOOP)
       SUBMCH (SETQ B (SUBMCH B WORKSPACE))
              (GO WATB)
       PRINT  (PRINT (CDR B))
              (PRINT WORKSPACE)
       $IMP   (RETURN (QUOTE $IMP))
       EVAL   (SETQ B (EVAL (CADR B) NIL))
              (GO ATB2)
       ATB1   (SETQ B (CADR B))
              (GO ATB2)
       ATB    (COND
                 ((ATOM B) (SETQ B (GTNAME B MPAIRS))))
       ATB2   (SETQ H (CAR WORKSPACE))
              (COND
                 ((ATOM B) (GO B))
                 ((EQ (CADR B) (QUOTE /)) (GO SUBMCH))
                 ((EQUAL B H) (SETQ B (CONS (LIST B) (CDR WORKSPACE
```

```
))))
                (T (SETQ B NIL)))
             (GO WATB)
       B     (COND
                ((EQUAL B H) (SETQ B WORKSPACE))
                ((AND
                   (EQUAL B (CAR H))
                   (EQ (CADR H) (QUOTE /))) (SETQ B (CONS (LIST
 H) (CDR WORKSPACE))))
                (T (SETQ B NIL)))
             (GO WATB)
       NDOLL (SETQ G (CDR B))
       NDOLL2 (SETQ B (DOLNM G WORKSPACE))
             (GO WATB))))

(NAMER
   (LAMBDA (X) (PROG (A B C D E)
             (SETQ A (CAR X))
             (SETQ D 1)
             (SETQ B X)
             (COND
                ((OR
                   (EQ A (QUOTE $))
                   (EQ A (QUOTE $0))) (GO START)))
             (SETQ B (CONS (QUOTE $) X))
             (SETQ E (LIST 0))
       START (COND
                ((NULL X) (RETURN (CONS E B))))
             (SETQ E (ADDLAST E D))
             (SETQ X (CDR X))
             (SETQ D (ADD1 D))
             (GO START))))

(SUBMCH
   (LAMBDA (X Y) (PROG (A B C)
             (SETQ A (CAR X))
             (SETQ B (CAR Y))
             (COND
                ((NOT (OR
                   (EQ A (QUOTE $1))
                   (EQUAL A (CAR B))
                   (EQUAL A (QUOTE ($ . 1))))) (RETURN NIL))
)
             (COND
                ((EQ (CADR B) (QUOTE /)) (GO ON))
                (T (RETURN NIL)))
       ON    (SETQ A (CDR X))
             (COND
                ((EQ (CAR A) (QUOTE /)) (GO A)))
             (PRINT (LIST (QUOTE (SUBSCRIPT ERROR SUBMCH)) X
))
             (RETURN NIL)
       A     (SETQ A (CDR A))
             (SETQ C (CDDR B))
       START (COND
                ((NULL A) (RETURN (CONS (LIST B) (CDR Y))))
                ((MEMBER (CAR A) C) (SETQ A (CDR A)))
                (T (RETURN NIL)))
             (GO START))))
```

```
(DOLNM
   (LAMBDA (NUM WSPACE) (PROG (A B)
            (SETQ B (CAR WSPACE))
            (COND
               ((NUMBERP NUM) (GO NUM))
               ((EQ NUM (QUOTE NUMBER)) (GO NUMBER))
               ((EQ NUM (QUOTE ATOM)) (GO ATOM))
               ((EQ NUM (QUOTE LIST)) (GO LIST)))
            (COND
               ((OR
                  (EQUAL NUM B)
                  (EQUAL NUM (CAR B))) (GO RNIL)))
      $1    (COND
               ((ATOM B) (GO B)))
      LST   (RETURN (CONS (LIST B) (CDR WSPACE)))
      NUMBER (COND
               ((NOT (NUMBERP B)) (GO RNIL)))
      B     (RETURN WSPACE)
      ATOM  (COND
               ((ATOM B) (GO B)))
      RNIL  (RETURN NIL)
      LIST  (COND
               ((ATOM B) (GO RNIL))
               (T (GO LST)))
      NUM   (COND
               ((EQUAL NUM 1) (GO $1)))
      START (COND
               ((EQUAL NUM 0) (RETURN (CONS A WSPACE)))
               ((NULL WSPACE) (RETURN (QUOTE $IMP))))
            (SETQ A (ADDLAST A (CAR WSPACE)))
            (SETQ WSPACE (CDR WSPACE))
            (SETQ NUM (SUB1 NUM))
            (GO START))))

(ADDLAST
   (LAMBDA (X Y) (APPEND X (LIST Y))))

(WRITES
   (LAMBDA (X) (PROG (A)
      START (SETQ A (CAR X))
            (COND
               ((NULL X) (RETURN NIL))
               ((EQ A (QUOTE $EOR$)) (GO ON))
               ((ATOM A) (PRIN1 A))
               (T (PRIN1 (QUOTE ***))))
            (SETQ X (CDR X))
            (GO START)
      ON    (TERPRI)
            (RETURN NIL))))

VALUE
(COMITMATCH CMATCH NAMER SUBMCH DOLNM ADDLAST WRITES)
READY.
```

Appendix 2

The LISP Programs for Inductive Inference on Sequences

Malcolm Pivar and Elaine Gord

1. Functions Common to Several Prediction Programs

LAST finds the last member of a list.
COMPOSE applies a function to a list n times.
NTHMEM finds the nth member of a list, beginning the count with O.
P1 is true if and only if all members of a list are identical.
S applies a function to successive members of a list, producing a list.
D finds the first difference list of a list of numbers.
ND finds the nth first difference list.
R finds the first ratio list of a list of numbers.
NR finds the nth first ratio list.
RATIO finds the (real) ratios between corresponding members of 2 lists.
NTHLIS produces a list containing every nth member of a given list.
NUMEQUAL finds the number of times a given member of a list appears.
DIFFLIST deletes a given member from a list whenever it appears.
DISTAB produces a distribution table of members of a list.
MODFREQ, given a list of pairs such as DISTAB produces, finds that pair with the largest second member.
SUMTEST produces a description in LISP of the additive relationship between 2 numbers.
SUBSTRINGTEST compares first difference and first ratio lists to a given list to see if they match. If they do, the next member of the given list is returned. Auxiliary functions used are DTEST, RTEST, and SUBSTRING.

RITEFORM is a list-making function.
BESTONE lists the best one of a list of pairs by finding the MODFREQ.
RETTIMES, RETTIMES1, RETPLUS, and RETPLUS1 are used to predict the next member of a list, given the next member of the nth first difference or first ratio list.
RETMEM returns that member of a list having a given position.

2. LISP Listing

```
FUNCTIONS COMMON TO SEVERAL PREDICTION PROGRAMS
(LAST (LAMBDA (S) (COND
      ((NULL (CDR S) ) (CAR S))(T (LAST (CDR S))       ))))
(COMPOSE (LAMBDA (EQ X N) (COND ((ZEROP N) X) ((NULL X) NIL)
      (T (COMPOSE EC (FC X) (SUB1 B))))))
(NTHEM (LAMBDA (A X)(COND ((NULL X) NIL) ((EQUAL A O)(CAR X))
      (T (NTHEM (SUB1 A) (CDR X))))))
(P1(LAMBDA(X)(COND((NULL( CDDR X))  (EQUAL (CAR X)(CADR X)))
      ((EQUAL (CAR X) (CADR X)) (P1 (CDR X))) (T F) )))
(S (LAMBDA(FN X) (COND ((NULL(CDR X)) NIL) (T (CONS (FN (CADR X)
      (CAR X)) (S FN (CDR X)))))))
(D (LAMBDA (X) (S (FUNCTION DIFFERENCE) X)))
(ND (LAMBDA (N X) (COND ((NULL X) NIL) ((ZEROP N) X)
      (T (ND (SUB1 N) (D X)))  )))
(R (LAMBDA (X) (COND ((NULL X) NIL) ((MEMBER O X) NIL)
      (T (S (FUNCTION QUOTIENT) X))  )))
(NR (LAMBDA (N X) (COND ((NULL X) NIL) ((ZEROP N) X)
      (T (NR (SUB1 N) (R X))) )))
(RATIO (LAMBDA (X Y) (COND ((NULL X) NIL) (T (CONS
      (QUOTIENT (TIMES 1.0 (CAR X)) (CAR Y))(RATIO (CDR X)
      (CDR Y))))  )))
(NTHLIS (LAMBDA (X N) (PROG (A)
      (COND ((NULL X) NIL))
      (SETQ A (COMPOSE (FUNCTION CDR) X (SUB1 N)))
      (RETURN
      (COND ((NULL A) NIL) (T (CONS (CAR A) (NTHLIS (CDR A)
      N))))  ))))
(NUMEQUAL (LAMBDA(A X) (COND ((NULL X) O) ((EQUAL A
      (CAR X)) (PLUS 1
      (NUMEQUAL A (CDR X)))) (T (NUMEQUAL A (CDR X))))))
(DIFFLIST (LAMBDA (A X) (COND ((NULL X) NIL) ((EQUAL A
      (CAR X))
```

```
      (DIFFLIST A (CDR X))) (T (CONS (CAR X) (DIFFLIST A
      (CDR X)))))))
(DISTAB (LAMBDA (X) (COND ((NULL X) NIL) ( T (CONS (CONS
      (CAR X)
      (NUMEQUAL (CAR X) X)) ( DISTAB (DIFFLIST ( CAR X) X)))))))
(MODFREQ (LAMBDA (X) (COND ((NULL (CDR X)) (CAR X))
      ((NULL X) NIL)
      ((GREATERP (CDAR X)(CDR (MODFREQ (CDR X)))) (CAR X)
      (T (MODFREQ (CDR X))))))
(SUMTEST (LAMBDA (X Y) (SUBST (DIFFERENCE Y X)(QUOTE Y)
      (QUOTE (PLUS X Y)
      ))))
(SUBSTRINGTEST (LAMBDA (X) (PROG (D)
      (SETQ D (DTEST X))
      (COND ((NULL D) (GO A)))
      (RETURN D)
      (RETURN (RTEST X)) )))
(DTEST (LAMBDA (X) ((LAMBDA (Y) (COND ((NULL Y) NIL)
      (T (PLUS (LAST X) Y)))) (SUBSTRING (D X) X))))
(RTEST (LAMBDA (X) (COND ((MEMBER O X) NIL)
(T ((LAMBDA (Y) (COND ((NULL Y) NIL) (T (TIMES (LAST X) Y))))
      (SUBSTRING (R X) X))))))
(SUBSTRING (LAMBDA (X Y) (PROG (U V OU OV M N PARM5)
      (SETQ U (REVERSE X))
      (SETQ OU U)
      (SETQ PARM5 (MAX 3 (TIMES 0.67 (LENGTH U))))
      (SETQ V (REVERSE Y))
      (SETQ OV V)
      (SETQ M O)
      (SETQ N 1)
A     (COND ((NULL V)(RETURN NIL))
      ((GREATERP N PARM5)(RETURN (NTHMEM (SUB1 M) OV)))
      ((EQUAL (CAR U)(CAR V))(GO B)))
      (SETQ N 1)
      (SETQ M (ADD1 M))
      (SETQ V (COMPOSE (FUNCTION CDR) OV M))
      (SETQ U OU)
      (GO A)
B     (SETQ N (ADD1 N))
      (SETQ U (CDR U))
      (SETQ V (CDR V))
      (GO A) )))
(RITEFORM (LAMBDA (X Y) (COND ((NULL X) Y) (T (CONS X Y)) )))
```

```
(BESTONE (LAMBDA (X) (COND ((NULL X) NIL) (T (LIST
     (MODFREQ X))) )))
(RETTIMES (LAMBDA (M V X) (COND ((NULL X) O) ((EQUAL M O) V)
     (T (RETTIMES1 (SUB1 M) V X)) )))
(RETTIMES1 (LAMBDA (M V X) (COND ((NULL X) 1) ((ZEROP
     (ADD1 M)) ))
     (T (TIMES V (LAST (NR M X)) (RETTIMES1 (SUB1 M) 1 X))) )))
(RETPLUS (LAMBDA (N V X) (COND ((NULL X) O) ((EQUAL N O) V)
     (T (RETPLUS1 (SUB1 N) V X)) )))
(RETPLUS1 (LAMBDA (N V X) (COND ((NULL X) O) ((ZEROP
     (ADD1 N)) O)
     (T (PLUS V (LAST (ND N X)) (RETPLUS1 (SUB1 N) O X))) )))
(RETMEM (LAMBDA (N X)(COND((ZEROP N) NIL)((NULL X) NIL)
     (T (CONS (CAR X)(RETMEM (SUB1 N)(CDR X)))))))
```

3. The Prediction Program NEXTOF

NEXTOF produces an ideal sequence corresponding to a given list, and, by comparing the two lists, predicts the next member. The ideal sequence is produced by expanding an encoding of a list.

ENCODE is a list of first members of a list and its successive first difference lists. CODER is the corresponding list for first ratio lists.

EXPAND and its auxiliary ST and RAX and its auxiliary RAST expand lists produced by ENCODE and CODER respectively.

DEGREE determines the number of times first differences must be taken in order to produce a list a set number of whose members are identical. RADEG is the corresponding function for first ratio lists.

LEFTLIST and LEFTLISTAUX (and RALEFT and RALEFTAUX) shift an encoded list to the left. Once a "perfect" encoding is found, this is done to insure that the proper ideal sequence is found. This avoids an improper prediction due to the first member of a list being an exception.

RITENDUM (RATIONUM) indicates the number of shifts to the right that is necessary in order to avoid confusion due to exceptions.

RITFIRSTVEC (RATIOVEC) produces the leftmost "perfect" encoding.

IDSEQ expands the encoding to the length of the original list.

ODPRFIN finds the exceptions to the ideal sequence. If there is no ideal sequence VALPOSPAIR returns a list of pairs of members of the original list and their positions. Otherwise ODPR returns a list of pairs of corresponding unlike elements.

PREDICTNEXT predicts the next member of a list specified by NEXTOF. VALIS produces a list of exceptions in a given list to its ideal sequence. POSLIS1 produces a list of positions of these exceptions. PATTERN determines whether a list has a pattern.

NEXTOF determines whether the next member of a list should be an exception to the ideal sequence. If not, the next member of the ideal sequence is returned by PREDICTNEXT. If it is an exception, NEXTOF determines whether there is a pattern to exception values and if so, returns the next member.

4. LISP Listing

```
THE PREDICTION PROGRAM NEXTOF
(ENCODE (LAMBDA (X) (COND ((OR (NULL (CDR X))(P1 X)) (LIST
      (CAR X)))
      (T (CONS (CAR X) (ENCODE (D X)))))))
(CODER (LAMBDA (X) (COND
      ((OR (NULL (CDR X)) (P1 X)) (LIST (CAR X)))
      (T (CONS (CAR X) (CODER (RATIO (CDR X) X)))) )))
(ST (LAMBDA (E) (COND((NULL (CDR E))E) (T (CONS (PLUS (CAR E)
      (CADR E))
      (ST (CDR E)))))))
(RAST (LAMBDA (X) (COND ((NULL (CDR X)) X)
      (T (CONS (TIMES 1.0 (CAR X)(CADR X)) (RAST (CDR X)))) )))
(EXPAND (LAMBDA (N LS) (COND ((ZEROP N) NIL) (T (CONS (CAR LS)
      (EXPAND
      (SUB1 N) (ST LS)))))))
(RAX (LAMBDA (N LS) (COND ((ZEROP N) NIL)
      (T (CONS (CAR LS) (RAX (SUB1 N) (RAST LS)))) )))
(DEGREE (LAMBDA (X) (PROG (PARM1 PARM2 W LENW LENW1)
      (SETQ PARM1 3)
      (SETQ PARM2 0.4)
      (SETQ LENW (LENGTH W))
      (SETQ LENW1 LENW)
A     (COND ((LESSP LENW PARM1) (RETURN -1))
      ((GREATERP (CDR (MODFREQ (DISTAB W)))
      (TIMES PARM2 LENW))
      (RETURN (DIFFERENCE LENW1 LENW))))
      (SETQ W (D W))
      (SETQ LENW (LENGTH W))
      (GO A) )))
```

```
(RADEG (LAMBDA (X) (PROG (PARM1 PARM2 W LENW LENW1)
      (SETQ PARM1 3)
      (SETQ PARM2 0.4)
      (SETQ W X)
      (SETQ LENW (LENGTH . W))
      (SETQ LENW1 LENW)
A     (COND ((LESSP LENW PARM1) (RETURN -1))
      ((GREATERP (CDR (MODFREQ (DISTAB W)))(TIMES PARM2
      LENW))
      (RETURN (DIFFERENCE LENW1 LENW))))
      (SETQ W (RATIO (CDR W) W))
      (SETQ LENW (LENGTH W))
      (GO A) )))
(LEFTLIST (LAMBDA(L) (COND((NULL (CDR L)) L) (T
      ( LEFTLISTAUX L
      (LEFTLIST (CDR L)))))))
(LEFTLISTAUX (LAMBDA (L X) (CONS (DIFFERENCE (CAR L)
      (CAR X)) X)))
(RALEFT (LAMBDA (X) (COND ((NULL (CDR X)) X)
      (T (RALEFTAUX X (RALEFT (CDR X)))) )))
(RALEFTAUX (LAMBDA (L X) (CONS (QUOTIENT (CAR L)
      (CAR X)) X)))
(RITENDNUM (LAMBDA (X Y) (PROG (U CONSTERM DEG1 P N)
      (SETQ U X)
      (SETQ DEG1 Y)
      (SETQ CONSTERM (CAR (MODFREQ (DISTAB (ND DEG1 U)))))
      (SETQ P O)
A     (SETQ N (NTHMEM DEG1 (ENCODE U)))
      (COND ((EQUAL N CONSTERM)(RETURN P)))
      (SETQ U (CDR U))
      (SETQ P (ADD1 P))
      (GO A))))
(RATIONUM (LAMBDA (X Y) (PROG (U CONSTERM DEG1 P N)
      (SETQ U X)
      (SETQ DEG1 Y)
      (SETQ P (MODFREQ (DISTAB (NR DEG1 U))))
      (COND ((NULL P)(RETURN (PROG2 (PRINT (APPEND
      (QUOTE (THERE IS NO
      CONSTERM FOR)) (LIST X))) O))))
      (SETQ CONSTERM (CAR P))
      (SETQ P O)
A     (SETQ N (NTHMEM DEG1 (CODER U)))
      (COND ((EQUAL N CONSTERM)(RETURN P)))
```

```
      (SETQ U (CDR U))
      (SETQ P (ADD1 P))
      (GO A))))
(RITFIRSTVEC (LAMBDA (X)(PROG (V RIT G1 DEG VEC)
      (SETQ V X)
      (SETQ DEG (ADD1 (DEGREE V)))
      (SETQ RIT (RITENDNUM V (SUB1 DEG)))
      (SETQ G1 (ENCODE (COMPOSE(FUNCTION CDR) V RIT)))
      (SETQ VEC (RETMEM DEG G1))
      (RETURN (COMPOSE (FUNCTION LEFTLIST) VEC RIT)))))
(RATIOVEC (LAMBDA (X) (PROG (V R G RAD VEC)
      (SETQ V X)
      (SETQ RAD (ADD1 (RADEG V)))
      (SETQ R (RATIONUM V (SUB1 RAD)) )
      (SETQ G (CODER (COMPOSE (FUNCTION CDR) V R)))
      (SETQ VEC (RETMEM RAD G))
      (RETURN (COMPOSE (FUNCTION RALEFT) VEC R)) )))
(IDSEQ (LAMBDA (X) (COND
      ((EQUAL (DEGREE X) -1) (COND
      ((MEMBER O X) NIL)
      ((EQUAL (RADEG X) -1) NIL)
      (T (RAX (LENGTH X) (RATIOVEC X)))))
      (T (EXPAND (LENGTH X) (RITFIRSTVEC X))) )))
(VALPOSPAIR (LAMBDA (X N)(COND ((NULL X) NIL)
      (T (CONS (CONS (CAR X) N)(VALPOSPAIR (CDR X)(ADD1 N)))))))
(ODPR (LAMBDA (X Y) (COND ((NULL X) NIL) ((EQUAL (CAR X)
      (CAR Y))
      (ODPR (CDR X) (CDR Y))) (T (CONS (CONS (CAR X) (CAR Y))
      (ODPR (CDR X) (CDR Y)))))))

(ODPRFIN (LAMBDA (X) (COND ((AND (EQUAL (DEGREE X) -1)
      (EQUAL (RADEG X) -1)) (VALPOSPAIR X 1))
      (T (ODPR X (IDSEQ X))))))
(PREDICTNEXT (LAMBDA (X L)(COND ((NULL X) NIL)
      ((EQUAL L 1) (CAR X))((EQUAL L 2)(DIFFERENCE
      (TIMES 2(CADR X))(CAR X)))
      (T (NTHMEM L (EXPAND (ADD1 L)(ENCODE X)))))))
(VALIS (LAMBDA (X) (COND ((NULL X) NIL)
      (T (CONS (CAAR X)(VALIS (CDR X)))))))
(POSLIS1 (LAMBDA (X Y) (COND ((NULL X) NIL)
      ((EQ (CAR X) (CAAR Y)) (CONS (CDAR Y) (POSLIS1 (CDR X)
      (CDR Y))))
      (T (POSLIS1 X (CDR Y))) )))
```

```
(PATTERN (LAMBDA (X) (COND ((NULL X) F)
      ((EQUAL (LENGTH (ODPRFIN X)) (LENGTH X)) F) (T T))))
(NEXTOF (LAMBDA (X) (PROG (B C D E LENA)
      (SETQ LENA (LENGTH X))
      (COND ((LESSP LENA 4)(RETURN (PREDICTNEXT X LENA))))
      (SETQ B (IDSEQ X))
      (SETQ E (COND ((NULL B)(VALPOSPAIR X 1))(T (ODPR X B))))
      (SETQ D (VALIS E))
      (SETQ C (POSLIS1 (VALIS E)(VALPOSPAIR X 1)))
      (COND ((NOT (PATTERN C))(GO Z)))
      ((PATTERN D)(GO Y))
      ((EQUAL (ADD1 LENA)(NEXTOF C))(RETURN NIL)))
      (GO Z)
Y     (COND ((EQUAL (ADD1 LENA)(NEXTOF C))
      (RETURN(NEXTOF D))))
Z     (RETURN (PREDICTNEXT B LENA)))))
```

5. The Program Producing Program OUTFCN

OUTFCN produces a LISP program describing the pattern of a given list and exceptions to this pattern. Given a number, this program will return the corresponding value. Exceptions to the pattern are determined by comparison with an ideal sequence (see the program NEXTOF). The pattern is determined by a numerical analysis of the encoding of the original list.

6. LISP Listing

```
(IDSEQ (LAMBDA (X) (COND ((EQUAL (DEGREE X) -1) NIL)
     (T (EXPAND (LENGTH X)(RITFIRSTVEC X))))))
(CONST (LAMBDA (X) (PROG (Z V LEN DEG RIT G1 VEC RITFIRST
     N L)
          (SETQ Z X)
          (SETQ LEN (LENGTH Z))
          (SETQ DEG (DEGREE Z))
          (SETQ RIT (RITENDNUM Z DEG))
          (SETQ G1 (ENCODE (COMPOSE (FUNCTION CDR) Z RIT)))
          (SETQ VEC (RETMEM (ADD1 DEG) G1))
          (SETQ RITFIRST (COMPOSE (FUNCTION LEFTLIST).
          VEC RIT))
          (SETQ V (EXPAND LEN RITFIRST ))
          (SETQ N 1)
     B    (COND ((AND (NULL L)(NULL Z))(RETURN (PP
          (COMP1 VEC) O)))
          ((NULL Z)(RETURN (CONS (QUOTE COND)(APPEND L
          (SUBST (PP (COMP1
          VEC) O) (QUOTE V) (QUOTE ((T V))))))))
          ((EQUAL (CAR Z)(CAR V))(GO A)))
          (SETQ L (APPEND L (SUBLIS (LIST (CONS (QUOTE M) N)
          (CONS(QUOTE P)(CAR Z)))(LIST(QUOTE((EQUAL X M)
          P)))) ))
     A    (SETQ Z (CDR Z))
          (SETQ V (CDR V))
          (GO B))))
```

```
(STTB (LAMBDA (N L) (COND ((NULL L) NIL) (T(CONS(TIMES
      N(CAR L))
      (STTB N (CDR L)))))))
(VADD (LAMBDA (L M) (COND ((NULL L) M)((NULL M) L)
      (T (CONS (PLUS (CAR L)(CAR M))(VADD (CDR L)(CDR M)))))))
(COMP1(LAMBDA (V) (CC V 1 1 (LIST 1))))
(CC (LAMBDA (V FACT COUNT XVEC) (COND ((NULL V) NIL)
      (T (VADD (STTB (QUOTIENT (CAR V) FACT) XVEC)(CC
      (CDR V) (TIMES FACT
      (COUNT()PLUS COUNT 1)(VADD(CONS O XVEC)(STTB
      (MINUS COUNT) XVEC))))))))
(PP (LAMBDA (L N) (COND ((NULL L) NIL)
      ((ZEROP (CAR L))(PP (CDR L) (ADD1 N)))
      ((EQ N O) (CONS (QUOTE PLUS)(CONS (CAR L) (PP (CDR L)
      (ADD1 N)))))
      ((EQUAL (CAR L) 1) (COND
      (  (NULL (CDR L)) (LIST (QUOTE EXPT) (QUOTE X) N))
      (T(CONS (LIST (QUOTE EXPT)(QUOTE X)N)(LIST(PP (CDR L)
      (ADD1 N)))))))
      ((NULL (CDR L))(LIST (QUOTE TIMES)(CAR L)(LIST
      (QUOTE EXPT)
      (QUOTE X) N)))
      (T (CONS (LIST (QUOTE TIMES) (CAR L) (LIST (QUOTE EXPT)
      (QUOTE X) N)) (LIST (PP (CDR L) (ADD1 N))))) )))
(OUTFCN (LAMBDA (X) (COND ((EQUAL (DEGREE X) -1)(PRINT
      (QUOTE (NO
      PATTERN))))(T(SUBST (CONST X)(QUOTE Z)(QUOTE (LAMBDA
      (X) Z)))) )))
```

7. The Prediction Program SEQUENTIAL PATTERNS

POSN assigns to a letter a number correponding to its position in the alphabet, beginning with A = O.

POSNLIS applies POSN to each member of a list of letters

D subtracts the members of one list from the corresponding members of another.

COMPOSE applies a function to a variable, then applies the function to the result, and so forth, a total of n times.

SN subtracts the first member of a list from the nth member, the second from the n + lst, and in general the kth member from the n + k − lst.

SNMOD puts a list of numbers into an equivalent list mod 26.

DIFFLIST strikes from a list all numbers equal to a given number.

NEXTMEM predicts the next member of a periodic series.

NTHEM locates the nth member of a list.

PATCONST produces two numbers indicating the nature of a periodic function. The first number, when used with the list in SN, produces a periodic series. The second number is the period of that series.

PERTEST1 predicts the next member of a list of numbers.

PERTEST2 is an auxiliary function for saving time.

PERTEST predicts the next letter of the original sequence using NUMMOD and RETLET.

NUMMOD transforms a number into an equivalent number mod 26.

RETLET finds the letter occupying the place in the alphabet indicated by a given number.

NEXTLET gives a list of letters to PERTEST. If no prediction is returned, it gives POSNLIS of that list to STUDY which, in turn calls the program NEXTOF.

8. LISP Listing

```
CSET (ALPHABET (A B C D E F G H I J K L M N O P Q R S T
U V W X Y Z))
CSET ( PARM2 0.5 )
CSET (PARM3 0.67)
DEFINE((
(POSN (LAMBDA (X A)(COND((EQUAL X(CAR A))O)(T(PLUS 1
(POSN X(CDR A)))))))
(POSNLIS (LAMBDA (X) (COND ((NULL X) NIL)(T (CONS (POSN
(CAR X)
```

```
      ALPHABET)(POSNLIS (CDR X)))))))
(D2(LAMBDA (X Y) (COND ((NULL X) NIL) (T
      (CONS (DIFFERENCE (CAR X)(CAR Y))(D2(CDR X)(CDR Y)))))))
(COMPOSE (LAMBDA (FC X N) (COND ((ZEROP N) X) ((NULL X) NIL)
      (T (COMPOSE FC (FC X) (SUB1 N))))))
(SN (LAMBDA (N X)(D2(COMPOSE (FUNCTION CDR) X N) X)))
(SNMOD (LAMBDA (X) (COND ((NULL X) NIL) (T (CONS (COND
      ((GREATERP (CAR X) 25) (DIFFERENCE (CAR X) 26))
      ((LESSP (CAR X) O)(PLUS (CAR X) 26))(T (CAR X)))(SNMOD
      (CDR X)))))))
(DIFFLIST (LAMBDA (A X) (COND ((NULL X) NIL) ((EQUAL A (CAR X))
      (DIFFLIST A (CDR X))) (T (CONS (CAR X) (DIFFLIST A
      (CDR X)))))))
(NEXTMEM (LAMBDA (N X)(NTHMEM1(ADD1 (REMAINDER(LENGTH X)
      N)) X)))
(NTHMEM1(LAMBDA (N X)(COND ((NULL X NIL)((EQUAL N 1)(CAR X))
      (T (NTHMEM1(SUB1 N)(CDR X))))))
(PATCONST (LAMBDA (Y)
      (PROG (P L HLENY N M LENL)
            (SETQ P 1)
            (SETQ HLENY (TIMES PARM2 (LENGTH Y)))
      C     (SETQ L (SNMOD (SN P Y)))
            (SETQ LENL (TIMES PARM3 (LENGTH L)))
            (SETQ N 1)
      B     (SETQ M (SNMOD (SN N L)))
            (COND ((NULL (DIFFLIST O M((*RETURN (CONS P N))))
            (COND ((GREATERP N LENL)(GO A)))
            (SETQ N (ADD1 N))
            (GO B)
      A     (COND ((GREATERP P HLENY)(RETURN NIL)))
            (SETQ P (ADD1 P))
            (GO C) )))
(PERTEST1 (LAMBDA (U Y) (COND ((NULL U NIL)
      (T (RETLET (NUMMOD (PLUS (NEXTMEM (CDR U)(SN (CAR U Y))
      (NTHMEM1(DIFFERENCE (LENGTH Y)(SUB1 (CAR U))) Y))))))))
(PERTEST2 (LAMBDA (X)(PERTEST1 (PATCONST X) X)))
(PERTEST (LAMBDA (X) (PERTEST2 (POSNLIS X))))
(NUMMOD (LAMBDA (X) (COND ((GREATERP X 25)(DIFFERENCE X 26))
      ((LESSP X O)(PLUS X 26))(T X))))
(RETLET (LAMBDA (N) (CAR (COMPOSE (FUNCTION CDR)
      ALPHABET N))))
(NEXTLET1 (LAMBDA (X Y) (COND ((NULL Y)(NEXTLET2 (STUDY
      (POSNLIS X))))
```

```
      (T Y))))
(NEXTLET2 (LAMBDA (X) (COND ((NULL X)(QUOTE (NO PATTERN)))
      (T (RETLET (NUMMOD X))))))
(NEXTLET (LAMBDA (X) (NEXTLET1 X (PERTEST X))))
```

9. The Prediction Program Test

ENTER, UPDATE, and STEP are the 3 functions that form the frequency distribution list for a given list. Before TEST can be called, FIXLIST must call these functions, apply the functions DOIT, DO1 and DO2 (which calls ORACLE), and CSETQ the final result as FREQDIST. FREQDIST is an argument of PREDICT, the function called by TEST. (ENTER is also called by the program NEXT – not the same as NEXTOF referred to in the article. NEXT is independent of the program TEST but uses some of the same functions; i. e., if CYCLES does not provide a prediction, ELIPSIS is called.)

ORACLE produces a list of LISP expressions of interesting relations between two expressions, if these expressions are numbers. The function ORVAL applies the tests PRIMTEST, EXPTEST, MULTEST and SUMTEST, each of which produces a LISP expression if applicable. If the two numbers are primes, determined by the functions PRIME and PRIM1, PRIMTEST produces a LISP expression that will predict the next prime if a value is substituted for its variable. The number of primes "up" or "down" to be sought is determined by the difference of the PRIMSEQs (determined by PS and NEXTPRIMEUP) of the two numbers. The LISP expression is in terms of PRIMEGEN, which calls NEXTPRIMEUP or NEXTPRIMEDN, until the desired prime is found. EXPTEST (with EX1) describes the exponential, SUMTEST the additive, and MULTEST the multiplicative relation between the two numbers. (ORACLE is called by the frequency distribution functions and at other times by the program TEST. It is also called by the program TEX, unconnected with TEST, and its connected functions TEX2, LOOKUP, and L2.)

PREDICT, called by TEST, has as its two arguments a list and FREQDIST. The first test that PREDICT applies to the list is CYCLES. CYCLES, C1, and LIKE determine if a list is perfectly cyclic. If so, it predicts the next member. If not, the second test is applied.

The second test of PREDICT is SBAR of LOOP of the list. LOOP applies the subroutine ORACLE to successive members of the list. Thus

the argument of SBAR is a list of S-expressions, each of which is either NIL or a list of S-expressions describing the numerical relations holding between a pair of numbers from the original list. SBAR's auxiliary function S1 applies SCAN to each of the descriptions of the first pair of numbers of the original list and to the descriptions of the remaining pairs. This is continued until all descriptions of the first pair have been tested by S1 and SCAN or until a solution based on these comparisons has been found. SCAN always returns a list whose first member is either T or NIL. This first member is used by S1 as the antecedent of a conditional whose consequent is CDR of the list returned by SCAN. SCAN searches the descriptions of the relations between two successive members of the original list until the predicate SEQ returns T. The value of SEQ is determined by that of FORQ which, with the functions QFORM, QF, FORM, and TLU, analyzes each part of the description. When SEQ returns T, SCAN continues with descriptions of the next pair. If no solution has been found by the second test, the third test of PREDICT is applied. If a value Y other than NIL has been found by SBAR, PREDICT of (CDR Y) and FREQDIST is sought. This value is substituted within the value Y, which is in turn applied to the last member of the original list given PREDICT.

The third test of PREDICT is the application of the function FOLLOW to the three arguments: 1) the last member of the original list; 2) the recursive application of PREDICT to ELIPSIS of the original list and frequency distribution list and a new frequency distribution list which adds to the original one an analysis of the list in question; and 3) the original frequency distribution list. ELIPSIS applies EP to successive members of the original list. The functions E and G search the frequency distribution list for a description whose first member is the same as that of the first of the two members being considered by EP. Probability levels for solutions are set by the functions VALUE, TRANS, and the frequency distribution list; solutions having a probability beneath a certain level are disregarded. FOLLOW searches the frequency distribution list for a description whose first member is the same as the last member of the original list. If the frequency distribution list is exhausted before such a description is found, PREDICT returns (IGIVEUP). Otherwise FOLLOW and FOL call one another, using the last member of the original list, the value returned as the second argument of FOLLOW and the description in the frequency distribution list to predict the next member of the original list.

10. LISP Listing

```
(UPDATE (LAMBDA (P L K) (COND
     ((NULL L) (LIST (LIST (CAR P) K (LIST (CDR P) K))))
     ((NOT (EQUAL (CAR P) (CAAR L)))
     (CONS (CAR L) (UPDATE P (CDR L) K)))
     (T (CONS (CONS (CAR P) (CONS (PLUS K (CADAR L))
     (STEP P (CDDAR L) K))) (CDR L))))))
(STEP (LAMBDA (N L K) (COND
     ((NULL L) (LIST (LIST (CAR N)K(ORACLE (CAR N)
          (CDR N)))))
     ((NOT (EQUAL (CAAR L) (CAR N))) (CONS (CAR L)
          (STEP N (CDR L) K)))
     (T (CONS (CONS (CAR N) (CONS (PLUS (CADAR L) K)
          (CDDAR L)))(CDR L))))))
(ENTER (LAMBDA (S L) (COND
     ((NULL (CDR S)) L)
     ((EQ (CAR S) (CADR S)) (ENTER (CDR S) L))
     (T (ENTER (CDR S) (UPDATE (CONS (CAR S) (CADR S))
          L 1))))))
(NEXT (LAMBDA (S L) ((LAMBDA (X) (COND
     ((CAR X) (CDR X))
(T ((LAMBDA (Y) (NEXT Y (ENTER Y L)))(ELIPSIS S L)))))
          (CYCLES S))))
(CYCLES (LAMBDA (S) (C1 S (CDR S) 1 (LENGTH S))))
(C1 (LAMBDA (L M N P) (COND
     ((GREATERP (TIMES 2 N) P) (LIST NIL))
(T ((LAMBDA (X) (COND
     ((CAR X) X)
     (T (C1 L (CDR M) (PLUS N 1) P)))) (LIKE L M))))))
(LIKE (LAMBDA (L M) (COND
     ((NULL M) (CONS T (CAR L)))
     ((NOT (EQUAL (CAR L) (CAR M))) (LIST NIL))
     (T (LIKE (CDR L) (CDR M))))))
(G (LAMBDA (N L TH M Q) (COND
     ((NULL L) NIL)
     ((LESSP (QUOTIENT (CADAR L) (PLUS Q O.O)) TH)
     (G N (CDR L) TH M Q))
(EQUAL N (CAAR L)) ((LAMBDA (X) (COND
     ((NULL X) (LIST N)) (T X)))
     (G N (CDR L) (QUOTIENT (CADAR L) (PLUS Q O.O))
          M Q)))
(T ((LAMBDA (X) (COND
```

```
     ((NULL X) (G N (CDR L) TH M Q))
(T ((LAMBDA (Y) (COND
     ((NULL Y) X) (T Y)))
     (G N (CDR L) (TIMES (QUOTIENT (CADAR L) (PLUS Q O.O))
          (VALUE X M)) M Q)))))
     (E (CONS (CAAR L) N) M (QUOTIENT (TIMES TH Q)
          (PLUS (CADAR L) O.O)) M))))))
(E (LAMBDA (P L TH M) (COND
     ((NULL L) NIL)
     ((LESSP TH O.O1) (E P L O.O5 M))
     ((NOT (EQUAL (CAR P) (CAAR L))) (E P (CDR L) TH M))
(T ((LAMBDA (X) (COND
     ((NULL X NIL)    (T (CONS (CAR P) X))))
     (G (CDR P) (CDDAR L) TH M (CADAR L)))))))
(VALUE (LAMBDA (S L) (V1 S 1 L)))
(V1 (LAMBDA (S A L) (COND
     ((NULL (CDR S)) A)
     (T (V1 (CDR S) (TIMES A (TRANS (CONS (CAR S)(CADR S))
     L))L)))))
(ORACLE (LAMBDA (X Y) (ORVAL X Y (QUOTE (PRIMTEST EXPTEST
     MULTEST SUMTEST)))))
(ORVAL (LAMBDA (X Y L) (COND
     ((NOT (AND (NUMBERP X) (NUMBERP Y))) NIL)
     ((NULL L) NIL)
(T ((LAMBDA (Z) (COND
     ((NULL Z) (ORVAL X Y (CDR L)))
     (T (CONS Z (ORVAL X Y (CDR L))))))   (APPLY (CAR L)
     (LIST X Y) NIL))))))
(EXPTEST (LAMBDA (X Y) (EX1 X Y 1 X)))
(EX1 (LAMBDA (X Y M P) (COND
     ((LESSP X 2) NIL)
     ((EQUAL P Y) (SUBST M (QUOTE Y) (QUOTE (POWER X Y))))
     ((GREATERP P Y) NIL)
     (T (EX1 X Y (PLUS M 1) (TIMES P X))))))
(MULTEST (LAMBDA (X Y) (COND
     ((ZEROP (REMAINDER Y X)) SUBST (QUOTIENT Y X) (QUOTE Y)
     (QUOTE (TIMES X Y))))  (T NIL))))
(SUMTEST (LAMBDA (X Y) (SUBST (DIFFERENCE Y X)(QUOTE Y)
     (QUOTE (PLUS X Y)))))

(PRIMTEST LAMBDA (X Y) (COND
     ((AND (PRIME X (PRIME Y)) (SUBST (DIFFERENCE (PRIMSEQ Y)
```

```
      (PRIMSEQ X)) (QUOTE Y) (QUOTE (PRIMEGEN X Y))))  (T NIL))))
(PRIME (LAMBDA (X) (COND
      ((LESSP X 2) NIL)
      ((EQUAL X 2) T)
      ((ZEROP (REMAINDER X 2)) NIL)
      (T (PRIM1 X 3)))))
(PRIM1 (LAMBDA (X Y) (COND
      ((GREATERP (TIMES Y Y) X) T)
      ((ZEROP (REMAINDER X Y)) NIL)
      (T (PRIM1 X (PLUS Y 2))))))
(PRIMSEQ (LAMBDA (Y) (PS Y 2 1)))
(PS (LAMBDA (Y P C) (COND
      ((GREATERP P Y) (ERROR (QUOTE PRIMSEQ)))
      ((EQUAL Y P) C)
      (T (PS Y (NEXTPRIMEUP (PLUS P 1))(PLUS C 1))))))
(NEXTPRIMEUP (LAMBDA (X) (COND
      ((PRIME X) X)
      (T (NEXTPRIMEUP (PLUS X 1))))))
(PRIMEGEN (LAMBDA (X C) (COND
      ((LESSP C O) (PRIMEGEN (NEXTPRIMEDN (SUB1 X)) (PLUS C 1)))
      ((GREATERP C O) (PRIMEGEN (NEXTPRIMEUP (PLUS X 1))
      (SUB1 C)))
      (T X))))
(NEXTPRIMEDN (LAMBDA (X) (COND
      ((PRIME X) X)
      ((EQUAL X 3) 2)
      (T (NEXTPRIMEDN (SUB1 X))))))
(TRANS (LAMBDA (P L) (COND
      ((NULL L) O)
      ((EQUAL (CAR P) (CAAR L)) (T1 (CDR P) (CDDAR L) (CADAR L)))
      (T (TRANS P ( CDR L))))))
(T1 (LAMBDA (N L K) (COND
      (( NULL L) O)
      ((EQUAL (CAAR L) N) (QUOTIENT (CADAR L) (PLUS K O. O)))
      (T (T1 N (CDR L) K)))))
(LOOKUP (LAMBDA (P L) (COND
      ((NULL L) (ORACLE (CAR P) (CDR P)))
      ((NOT (EQUAL (CAR P) (CAAR L))) (LOOKUP P (CDR L)))
      (T (L2 P (CDDAR L))))))
(L2 (LAMBDA (N L) (COND
      ((NULL L) (ORACLE (CAR N) (CDR N)))
      ((NULL L) (ORACLE (CAR N) (CDR N)))
      ((NOT (EQUAL (CAR N) (CAAR L))) (L2 N (CDR L)))
```

```
      (T (COND
      ((NULL (CDDR L)) (ORACLE (CAR N) (CDR N)))
      (T (CADDAR L)))
(TEX (LAMBDA (S L) (TEX2 (CDR S) L (LOOKUP (CONS (CAR S)
      (CADR S)) L))))
(TEX2 (LAMBDA (S L J) (COND
      ((NULL J) J) ((NULL (CDR S) ) J)
      (T (TEX2 (CDR S) L (INTERSECTION    J (LOOKUP (CONS (CAR S)
      (CADR S)) L)))))))

(ELIPSIS (LAMBDA (S L) (COND
      ((NULL (CDR S) ) NIL)
      (T (CONS (EP (CAR S) (CADR S) L) (ELIPSIS (CDR S) L))))))
(EP (LAMBDA (M N L) ((LAMBDA (X) (COND
      ((NULL X) 1) (T (SUB1 (LENGTH X)))
      )) (E (CONS M N ) L (TRANS (CONS M N) L)L))))
(PREDICT (LAMBDA (S L) (COND
      ((NULL S) NIL)
(T ((LAMBDA (X) (COND
      ((CAR X) (CDR X))
(T ((LAMBDA (Y) (COND
      ((NULL Y) (FOLLOW (LAST S)(PREDICT (ELIPSIS S L)
      (ENTER S L)) L))
(T ((LAMBDA (W) (COND
      ((NULL W) (ERROR (QUOTE BOTCHUP)))
      (T (APPLY (SUBST (SUBST W (CAAR Y)(CDAR Y))(QUOTE Z)
      (QUOTE
(LAMBDA (X) Z)))  (LIST (LAST S)) NIL)))) (PREDICT (CDR Y)
      L)))))
      (SBAR (LOOP S))))))
      (CYCLES S))))))
(INTERSECTION (LAMBDA (M N) (COND
      ((NULL M ) NIL)
      ((MEMBER (CAR M) N) (CONS (CAR M (INTERSECTION (CDR M)
      N)))
      (T (INTERSECTION (CDR M) N)))))
(LAST (LAMBDA (S) (COND
      ((NULL (CDR S) ) (CAR S))(T (LAST (CDR S))))))
(FOLLOW (LAMBDA (E N L) (COND
      ((EQUAL N O) E)
      ((NULL L) (QUOTE (I GIVE UP)))
      ((EQUAL (CAAR L) E) (FOL (CDDAR L) (QUOTE (O O)) N L))
      (T (FOLLOW E N (CDR L))))))
```

```
(FOL (LAMBDA (S B N L) (COND
      ((NULL S) (FOLLOW (CAR B) (SUB1 N) L))
      ((GREATERP (CADAR S) (CADR B)) (FOL (CDR S) (CAR S) N L))
      (T (FOL (CDR S) B N L)))))
(DOIT (LAMBDA (L) (COND
      ((NULL L) NIL)
      (T (CONS (DO1 (CAR L)) (DOIT (CDR L)))))))
(DO1 (LAMBDA (L) (CONS (CAR L) (CONS (CADR L) (DO2(CAR L)
      (CDDR L))))))
(DO2 (LAMBDA (N L) (COND
      ((NULL L) NIL)
      (T (CONS (LIST (CAAR L) (CADAR L) (ORACLE N (CAAR L)))
      (DO2 N (CDR L)))))))
(FIXLIST (LAMBDA (X Y) (COND
      ((NULL X) (CSAR (DOIT Y)))
      (T (FIXLIST (CDR X) (ENTER (CAR X) Y))))))
(CSAR (LAMBDA (X) (CSETQ FREQDIST X)))
(TEST (LAMBDA (S) (PREDICT S FREQDIST)))
(FORT (LAMBDA (M L) (CDAR (FORQ M L))))
(SCAN (LAMBDA (M L N A) (COND
      ((NULL L) (CONS NIL A))
((SEQ M (CAR L) A) ((LAMBDA (X) (COND
      ((CAR X) X)
      (T (SCAN M (CDR L) N A))))
      (COND ((NULL N) (CONS T (CONS (FORT M (CAR L)) A)))
      (T (SCAN (CAR L) (CAR N) (CDR N) (CONS (FORT M (CAR L))
      (COND ((NULL A) (CONS (FORT (CAR L) M) (LIST (CONS (CAAR
      (FORQ M (CAR L))) M))))
      (T (SCAN M (CDR L) N A)))))
(SEQ (LAMBDA (M L A) ((LAMBDA (X) (COND
      ((NULL X) NIL)
      ((AND (NULL (CDR X)) (NUMBERP (CDAR X))) T)
      (T NIL))) (FORQ M L))))
(FORQ (LAMBDA (M L) (QFORM (FORM M L NIL))))
(QFORM (LAMBDA (X) (COND
      ((CAR X) (QF (CDR X)))
      (T NIL))))
(QF (LAMBDA (X) (COND
      ((NULL X) (QUOTE NIL))
      ((EQ (CAAR X) (CDAR X)) (QF (CDR X)))
      (T (CONS (CAR X) (QF (CDR X)))))))
(FORM (LAMBDA (X Y A) (COND
      ((NULL X) (CONS T A))
```

```
((ATOM X) ((LAMBDA (Z) (COND
      ((NULL Z) (CONS T (CONS (CONS X Y) A)))
      ((EQ Y (CDR Z)) (CONS T A))
      (T (LIST NIL)))) (TLU X A)))
(T ((LAMBDA (Z) (COND
      ((CAR Z) (FORM (CDR X) (CDR Y) (CDR Z)))
      (T Z)))(FORM (CAR X) (CAR Y) A))))))
(LOOP (LAMBDA (S) (COND
      ((NULL (CDR S)) NIL)
      (T (CONS (ORACLE (CAR S) (CADR S)) (LOOP (CDR S)))))))
(TLU (LAMBDA (X L) (COND
      ((NULL L) NIL)
      ((EQ X (CAAR L)) (CAR L))
      (T (TLU X (CDR L))))))
(LAST (LAMBDA (L) (COND ((NULL (CDR L))(CAR L))
      (T (LAST (CDR L))))))
(SBAR (LAMBDA (L) (S1 (CAR L) (CDR L))
(S1 (LAMBDA (M L) (COND
      ((NULL M) NIL)
(T ((LAMBDA (X) (COND
      ((CAR X) (REVERSE (CDR X)))
      (T (S1 (CDR M) L)))) (SCAN (CAR M) (CAR L) (CDR L) NIL))))))
(REVERSE (LAMBDA (L) (R1 NIL L)))
(R1 (LAMBDA (M L) (COND
      ((NULL L) M)
      (T (R1 (CONS (CAR L) M) (CDR L))))))
      FIXLIST(
      (
      (1 2 3 4 5 6 7 8 8 10 11 12 13 14 15 16)
      (1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16)
      (A B C D E F G H I J K L M N O P Q R S T U V W X Y Z)
      )
      NIL
      )
```

The result of FIXLIST is:

```
(((1 2 (2 1 ((TIMES X 2) (PLUS X1))) (1 1 ((TIMES X 1) (PLUS X 0))))
      (2 2 (3 1 ((PRIMEGEN X 1) (PLUS X 1)))
(2 1 ((PRIMEGEN X 0) (POWER X 1) (TIMES X 1) (PLUS X 0))))
      (3 2 (4 1 ((PLUS X 1))) (3 1 ((PRIMEGEN X 0) (POWER
X 1) (TIMES X 1) (PLUS X 0)))) (4 2 (5 1 ((PLUS X 1))) (4 1 ((POWER X 1)
      (TIMES X 1) (PLUS X 0)))) (5 2 (6
```

```
1 ((PLUS X 1))) (5 1 ((PRIMEGEN X O) (POWER X 1) (TIMES X 1)
    (  (PLUS X O)))) (6 2 (7 1 ((PLUS X 1))) (6 1 ((POWER
X 1) (TIMES X 1) (PLUS X O)))) (7 2 (8 1 ((PLUS X 1)))
       (7 1 ((PRIMEGEN X O) (POWER X 1) (TIMES X 1) (PLUS X
0)))) (8 2 (9 1 ((PLUS X 1))) (8 1 ((POWER X 1) (TIMES X 1) (PLUS X 0))))
       (9 2 (10 1 ((PLUS X 1))) (9 1 ((POWER
X 1) (TIMES X 1) (PLUS X 0)))) (10 2 (11 1 ((PLUS X 1)))
       (10 1 ((POWER X 1) (TIMES X 1) (PLUS X 0)))) (11 2
(12 1 ((PLUS X 1))) (11 1 ((PRIMEGEN X 0) (POWER X 1) (TIMES X 1)
       (PLUS X 0)))) (12 2 (13 1 ((PLUS X 1))) (
12 1 ((POWER X 1) (TIMES X 1) (PLUS X 0)))) (13 2 (14 1 ((PLUS X 1)))
       (13 1 ((PRIMEGEN X 0) (POWER X 1) (TIMES
X 1) (PLUS X 0)))) (14 2 (15 1 ((PLUS X 1))) (14 1 ((POWER X 1)
       (TIMES X 1) (PLUS X 0)))) (15 2 (16 1 ((PLUS
X1))) (15 1 ((POWER X 1) (TIMES X 1) (PLUS X 0)))) (A 1 (B 1 NIL))
       (B 1  (C 1 NIL)) (C 1 (D 1 NIL)) (D 1 (E
1 NIL)) (E 1 (F 1 NIL)) (F 1 (G 1 NIL)) (G 1 (H 1 NIL)) (H 1 (I 1 NIL))
       (I 1 (J 1 NIL)) (J 1 (K 1 NIL)) (K
1 (L 1 NIL)) (L 1 (M 1 NIL)) (M 1 (N 1 NIL)) (N 1 (O 1 NIL))
       (O 1 (P 1 NIL)) (P 1 (Q 1 NIL)) (Q 1 (R 1 NIL))
(R 1 (S 1 NIL)) (S 1 (T 1 NIL)) (T 1 (U 1 NIL)) (U 1 (V 1 NIL))
       (V 1 (W 1 NIL)) (W 1 (X 1 NIL)) (X 1 (Y 1 NIL))
(Y 1 (Z 1 NIL))))
```

11. The Prediction Program Getnext

The ALLRELS subroutine finds numerical relations between successive members of a list. LENGHTEST is a parameter signifying that there are no exceptions to the pattern in question. The first two tests are P1 and SUBSTRINGTEST, which admit of no exceptions.

TIMESTEST and SUMTEST find multiplicative and additive relations between two numbers. FINDREL applies these functions to successive members of a list. The most frequently appearing relation is found and the last member of the list substituted for the variable by ALLRELS2. Called by ALLRELS1, this is evaluated by WORKOUT if the relation occurs at least 1/3 of the time.

If the pattern found by the ALLRELS subroutine admits of no exceptions, the parameter LENGTHTEST is substituted for the number of times the pattern actually applies by PARTLIST and PARTLIST1. TESTIT signals that a given prediction is from a pattern that admits of no exceptions.

TRYPART and TRYPART1 take NTHLISs of a list and, applying the ALLRELS subroutine, produce either a prediction whose pattern admits of no exceptions or a list of predicitions and the number of members of the list to which they apply.

TRYDR2, having predicted the next member of first and second first difference and first ratio lists of a given list, predicts the next member of the given list. It produces either a "perfect" prediction or a list of predictions.

The above functions are used for number sequences. The functions GETLET, POSN, POSNLIS, NUMMOD, and RETLET are used to change a letter sequence to a number sequence and a number prediction to a letter prediction. POSN finds the position of a letter in an alphabet. POSNLIS translates a list of letters into a list of number positions. NUMMOD reduces a number modulo alphabetlength. RETLET finds the letter corresponding to a given number position in an alphabet. GETLET gets the letter corresponding to a given prediction.

The functions PATTEST, PARTIAL, PARFCN and PARFCN1 predict the next member of a cyclic or partially cyclic sequence of symbols other than letters or numbers. PARTIAL tests whether various NTHLISs, starting with different members, have members satisfying P1. Using the results of PARTIAL, PARFCN and PARFCN1 produce a LISP program which is applied by PATTEST to the number 1 greater than the length of the original list, i. e., to the position of the next member.

WORTH is an auxiliary function used in returning the prediction.

SETPARMS sets the parameters LENGTHTEST and PARTLENGTH.

GETNEXT1 applies the function ALLRELS to a list of numbers. If no perfect pattern is found, it calls the function TRYPART. If no perfect pattern is found, it then calls the function TRYDR2. The best prediction (that whose pattern covers the most cases), if any, is returned.

GETNEXT sends number, letter, and other sequences to the appropriate subroutines.

```
      CSET (ALPHABET (A B C D E F G H I J K L M N O P Q R S T U
           V W X Y Z))
      CSETQ (LENGTHALPH (LENGTH ALPHABET))
      DEFINE ((
(ALLRELS (LAMBDA (X) (COND ((NULL X) NIL)
      ((PI X) (CONS (CAR X)  LENGTHTEST))
(T ((LAMBDA (Y) (COND
      ((NOT (NULL Y)) (CONS Y LENGTHTEST))
      (T (ALLRELS1 X))))
      (SUBSTRINGTEST X))) )))
(ALLRELS1 (LAMBDA (X) (WORKOUT (ALLRELS2 X))))
(WORKOUT (LAMBDA (X) (COND ((NULL X) NIL)
      ((GREATERP PARTLENGTH (CDR X)) NIL)
      (T (CONS (EVAL (CAR X) NIL) (CDR X))) )))
(ALLRELS2 (LAMBDA (X) (SUBST (LAST X) (QUOTE X) (MODFREQ
      (APPEND
      (DISTAB (FINDREL X (FUNCTION SUMTEST)))
      (DISTAB (FINDREL X (FUNCTION TIMESTEST))) )) )))
(FINDREL (LAMBDA (X FN) (COND ((NULL (CDR X)) NIL)
      (T (RITEFORM (FN (CAR X) (CADR X)) (FINDREL (CDR X)
           FN))) )))
(TIMESTEST (LAMBDA (X Y) (COND ((ZEROP X) NIL)
      ((ZEROP (REMAINDER Y X)) (SUBST (QUOTIENT Y X) (QUOTE Y)
      (QUOTE (TIMES X Y)))) (T NIL) )))
(PARTLIST (LAMBDA (X) (COND ((NULL X) NIL)
      (T (PARTLIST1 (LENGTH X) (ALLRELS X))) )))
(TESTIT (LAMBDA (X) (COND ((EQUAL LENGTHTEST (CDR X)) T)
           (T NIL) )))
(PARTLIST 1 (LAMBDA (X Y) (COND ((NULL Y) NIL)
      ((LESSP X 3) NIL)
      ((EQUAL (SUB1 X) (CDR Y)) (CONS (CAR Y) LENGTHTEST))
           (T Y) )))
(TRYPART (LAMBDA (X N) (COND ((NULL X) NIL)
      ((GREATERP N PARTLENGTH) NIL)
      (T (TRYPART 1 (PARTLIST (REVERSE (NTHLIS (REVERSE X)
           N))) N)) )))
(TRYPART1 (LAMBDA (X N) (COND ((NULL X) (TRYPART X
           (ADD1 N)))
      ((TESTIT X) (LIST X))
      (T (RITEFORM X (TRYPART X (ADD1 N)))) )))
(TRYDR2 (LAMBDA (X N) (COND ((NULL X) NIL)
```

```
      ((EQUAL N 3) NIL)
(T ((LAMBDA (Y) (COND
      ((TESTIT Y) (LIST (CONS (RETPLUS N (CAR Y) X) (CDR Y))))
(T ((LAMBDA (Z) (COND
      ((NULL Z) (COND ((NULL Y) (TRYDR2 X (ADD1 N)))
      (T (CONS (CONS (RETPLUS N (CAR Y) X) (CDR Y))
      (TRYDR2 X (ADD1 N))))))
      ((TESTIT Z) (LIST (CONS (RETTIMES N (CAR Z) X) (CDR Z))))
      ((NULL Y) (RITEFORM (CONS (RETTIMES N (CAR Z) X) (CDR Z))
      (TRYDR2 X (ADD1 N))))
      (T (RITEFORM (CONS (RETPLUS N (CAR Y) X) (CDR Y))
      (RITEFORM (CONS (RETTIMES N (CAR Z) X) (CDR Z))
      (TRYDR2 X (ADD1 N)))))))
      (MODFREQ (TRYPART (NR N X) 1))))))
      (MODFREQ (TRYPART (ND N X) 1)))) )))
(GETLET (LAMBDA (X) (COND ((NULL X) NIL)
      (T (RETLET (NUMMOD X))) )))
(POSN (LAMBDA (X A) (COND((EQUAL X (CAR A))O) (T (PLUS 1
            (POSN X(CDR A)))))))
(POSNLIS (LAMBDA (X) (COND ((NULL X) NIL) (T (CONS (POSN
            (CAR X) ALPHABET) POSNLIS (CDR X)))))))

(NUMMOD (LAMBDA (X) (COND
      ((GREATERP X (SUB1 LENGTHALPH))
            (DIFFERENCE X LENGTHALPH))
      ((LESSP X O) (NUMMOD (PLUS X LENGTHALPH))) (T X) )))
(RETLET (LAMBDA (N) (CAR (COMPOSE (FUNCTION CDR)
            ALPHABET N))))
(PATTEST (LAMBDA (X) (COND ((P1 X) (CAR X))
      (T (APPLY (PARFCN X) (LIST(ADD1 (LENGTH X))) NIL)) )))
(PARTIAL (LAMBDA (X) (PROG (U V M N L)
            (SETQ V NIL)
            (SETQ U X)
            (SETQ L (ADD1 (QUOTIENT (LENGTH U) 2)))
            (SETQ M 1)
            (SETQ N 1)
      A     (COND ((NULL U) (RETURN V))
            ((GREATERP N L) (GO B))
            ((P 1 (NTHLIS U N)) (GO C)))
            (SETQ N (ADD1 N))
            (GO A)
      B     (SETQ U (CDR U))
            (SETQ M (ADD1 M))
```

```
            (SETQ N 1)
            (GO A)
      C     (COND ((GREATERP M N) (GO D)))
            (SETQ V (CONS (LIST (SUB 1 M) N (NTHMEM (SUB1 N)
                U)) V))
      D     (SETQ N (ADD1 N))
            (GO A) )))
(PARFCN (LAMBDA (X) (PARFCN1 (PARTIAL X))))
(PARFCN1 (LAMBDA (X) (PROG (U V W)
            (SETQ U X)
      A     (COND ((NULL U) (GO B)))
            (SETQ V (CAR U))
            (SETQ W (COND ((EQUAL (CAR V) (CONS (LIST (LIST
                (QUOTE ZEROP)
            (LIST (QUOTE REMAINDER) (QUOTE X) (CADR V)))
            (LIST (QUOTE QUOTE) (CADDR V))) W))
            (T (CONS (LIST (LIST (QUOTE ZEROP)(LIST (QUOTE PLUS)
                (LIST (QUOTE
            REMAINDER) (QUOTE X) (CADR V)) (CAR V)))
            (LIST (QUOTE QUOTE) (CADDR V))) W)) ))
            (SETQ U (CDR U))
            (GO A)
      B     (COND ((NULL W) (RETURN NIL)))
(RETURN (LIST (QUOTE LAMBDA) (QUOTE (X)) (CONS (QUOTE COND)
      (APPEND W (QUOTE ((T NIL)))) ))) )))
(WORTH (LAMBDA (X) (COND ((NULL X) NIL) (T (CAAR X)) )))
(SETPARMS (LAMBDA (X) (PROG2 (CSETQ LENGTHTEST (SUB1 X))
      (CSETQ PARTLENGTH (MAX 3 (QUOTIENT X 3))) )))
(GETNEXT1 (LAMBDA (X) ((LAMBDA (Y) (COND
      ((EQUAL LENGTHTEST (CDR Y)) (CAR Y))
(T ((LAMBDA (Z) (COND
      ((NULL Z) (WORTH (BESTONE (TRYDR2 X 1))))
      ((EQUAL (CDAR Z) LENGTHTEST) (CAAR Z))
      (T (WORTH (BESTONE (APPEND (TRYDR2 X 1) Z))))))
      (BESTONE (RITEFORM Y (TRYPART X 2)))))))
      (ALLRELS X))))
(GETNEXT (LAMBDA (X) (COND ((NULL X) NIL)
      ((PROG2 (SETPARMS (LENGTH X)) (NUMBERP (CAR X)))
            (GETNEST1 X))
      ((NOT (LITER (CAR X))) (PATTEST X))
      (T (GETLET (GETNEXT1 (POSNLIS X)))))))
      ))
```

13. Examples of Prediction

ORACLE
(4 16)

((POWER X 2) (TIMES X 4) (PLUS X 12))

ORACLE
(3 27)

((POWER X 3) (TIMES X 9) (PLUS X 24))

ORACLE
(11 13)

((PRIMEGEN X 1) (PLUS X 2))

ORACLE
(11 7)

(PRIMEGEN X -1) (PLUS X -4))

ORACLE
(41 43)

(PRIMEGEN X 1) (PLUS X 2))

PRIMEGEN
(3 5)

17

PRIMEGEN
(41 -3)

29

PRIMEGEN
(2 1)

3

```
TEST
((1 2 3 4 5))
6

TEST
((1 3 5 7))
9

TEST
((1 21 4 8 16 32))
64

TEST
((1 2 6 24 120))
720

TEST
((5 4 3 2 1))
-0

TEST
((1 2 3 1 2 3 1 2 3))
1

TEST
((2 3 5 7 11 13))
17

TEST
((2 5 11 17))
23
```

STUDY IS A MORE GENERAL PROGRAM THAT CALLS NEXTOF

```
STUDY
((37 91 8 17 12 17 16 123 20 1 24
   19 4 5))
32

STUDY
((20 21 23 26 30 35 41 48))
56

STUDY
((4 8 16 24 32 40 48))
56

STUDY
((6 7 9 12 16 21 28 35))
42

STUDY
((3 13 4 15 5 17 6 19 7))
21

STUDY
((20 29 37 44 50 55 59))
62

STUDY
((40 39 43 38 46 37 49 36))
52

STUDY
((10 50 13 54 16 58 19 62))
22

STUDY
((2 90 4 80 6 70 8 60))
10

STUDY
((2 5 46 8 8 14 11 22 15))
32

STUDY
((2 4 5 6 11 10 20 16 32 24))
47

STUDY
((1 2 3 4 4 4 7 6 5 10 8 6))
13
```

SEQUENTIAL PATTERNS

PERTEST
((B A D C F E H G J I L))
K

PERTEST
((A B D C E F H G I J L K M N))
P

PERTEST
((Y Z W X U V S T Q R O P M))
N

PERTEST
((A B C D E F E G H I J L K M))
N

PERTEST
((Z Y X U V W T S R))
O

PERTEST
((A B C B C D C D E E D D E
F E))
F

PERTEST
((Z X V W T R S P N O))

(NO PATTERN)

PERTEST
((X C X D X E X))
F

PERTEST
((A B D C E F H))
G

PERTEST
((B A C A D A E A F A G A))
H

PERTEST
((J I H G F E D C))
B

PERTEST
((Z Y X W V U T S R))
Q

PERTEST
((X X W X V X U X T X S X R))
X

PERTEST
((A C E G I K M O Q))
S

PERTEST
((Z X V T R P N L))
J

PERTEST
((A D G J M P S))
V

PERTEST
((Z W T Q N K H))
E

PERTEST
((A D F I K N P S))
U

PERTEST
((A B C F E D G H I L K J M))
N

PERTEST
((Z Y X U V W T S R))
O

```
PERTEST
((Z Y X U V W T S R O P Q
   N M L))
I

PERTEST
((K L N M O P R Q S T))
V

PERTEST
((A B C D F E G H I J L K M))
N

PERTEST
((A C E D G I H K M L))
U

PERTEST
((A B D C B D D B E E B D F
   B D G B))

NIL

PERTEST
((Z Y W X V U S T R Q O P N M))
K

PERTEST
((Y Z X W U V T S Q R P O
   M N))
L

PERTEST
((B A D C F E H G J I L))
K

PERTEST
((A B D C E F H G I J L K M N))
P

PERTEST
((Y Z W X U V S T Q R O P M))
N

PERTEST
((A B C B C D C D E D E F E))
F

GETNEXT
((5 15 23 29 39 47 53 63))
71

GETNEXT
((AA AB AA AB AA AB AA))
AB

GETNEXT
((19 1 1 2 6 3 2 4 3 5 79))
6

GETNEXT
((7 8 6 7 5 6))
4

GETNEXT
((40 39 43 38 46 37 49 36))
52

GETNEXT
((K L N M O P R Q S T))
V

GETNEXT
((A B C F E D G H I L K J M))
N

GETNEXT
((1 2 3 5 8 13 21 34))
55

GETNEXT
((20 29 37 44 50 55 59))
62
```

```
GETNEXT
((0.5E0 0.25E0 0.125E-1 0.625E-2 0.3125E-2))
0.15625E-3

GETNEXT
((1 4 9 16 25 36))
49
```

Appendix 3

The LISP Listing for the Q-32 Compiler, and Some Samples

Robert A. Saunders

Information International, Inc.

1. THE Q-32 COMPILER

This is the complete deck which is run on an IBM 7090 to produce the compiled part of Q-32 LISP. The listing contains the compiler, which is interpreted to compile itself on the 7090, and as compiled code runs on the Q-32; two versions of LAP, one which runs on the 7090 to produce Q-32 code, and one which runs as compiled code on the Q-32; and miscellaneous functions. In addition to the function compilations, the run generates all of the permanent list structure for the Q-32.

The compiled code uses the following routines which are written in SCAMP, the Q-32 assembly language:

CONS and the garbage collector
ERROR
PRIN1, TERPRI, and PRINOCT
*RATOM and TEREAD
GENSYM
*MKNO
*CALL and *RETRN

A copy of the listing of these is available on request from Information International, Inc.

The compiler listing contains some known errors, including the following (alterations are underlined):

In ATTACH, the second line should read:
((AND (EQ (CAAR A) (QUOTE LDA)) LISTING (EQ (CAAR
LISTING) (QUOTE STA))

In PASS2, the second line in part should read:
(SETQ LEN 2)

In the second listing of LAP, the section starting at EP should read:
EP (COND ((NOT P2) (GO E1)) (EI (GO R)) ((NULL (CAR L))
(GO E2)) ((ATOM (CAR L)) (GO R)))

(CSET (CAAR L) TRW)

E2 (SETQ BPORG LOC)

R (RETURN ST)

```
***  THIS DECK IS INPUT TO THE SHARE DISTRIBUTED VERSION OF LISP 1.5.
IT CONTAINS LAP OCTALS TO QUIET THE GARBAGE COLLECTOR AND TO GET AT
THE PUNCHING FACILITIES. THE OUTPUT IS A) AN ASSEMBLY LISTING OF THE
COMPILED CODE, AND B) OCTAL CARD IMAGES ON THE PUNCH OUTPUT TAPE (B7).
THE PUNCH TAPE IS LOADED INTO THE Q-32 BY A PART OF THE SCAMP-CODED
BASIC FUNCTIONS.  ***


        TAPE    SYSPPT,B7

        SETSET

OPDEFINE (( (PRIN2 5135Q) (TERPUN 5445Q) (MKNO 126700) (PUNACT 5505Q) ))

LAP (( 1417Q (STZ 1 604Q) (STZ 2 604Q) (TRA 3 4) ) NIL)
LAP (( 3301Q (0) ) NIL)
LAP (( 5502Q (43624706) ) NIL)
LAP (( 12773Q (CLA 13011Q) ) NIL)
LAP (( 13033Q (TSX NIL 4) ) NIL)
LAP (( NIL (SLW T) (CLA T) (TRA MKNO) T (0) ) ((SLW . 602Q8)) )

LAP (( (MKVAL SUBR 1)
    (SXA X 4)
    (XCA)
    (CLA AF)
    (TSX CONS 4)
    (PDX 0 4)
    (CLA T5)
    (ORS 0 4)
    (PXD 0 4)
X   (AXT 0 4)
    (TRA 1 4)
AF  (77777Q6)
T5  (5Q5)
    )  ((ORS . 4602Q8)) )

LAP (( (STARTPUN SUBR 0)
    (STL PUNACT)
    (CLA PRIN2)
    (SSM)
    (STO PRIN2)
    (TRA 1 4)
    ) ((SSM . 4760000000003Q) (STL . 4625Q8)) )

LAP (( (TERPUN SUBR 0)
    (STL PUNACT)
    (TRA TERPUN)
    ) ((STL . 4625Q8)) )


LAP (( (OCT SUBR 1)
    (SXA X 4)
    (STO CT)

A   (PXD)
    (LXA CT 4)
    (LGL 3)
    (ALS 3)
    (LGL 3)
    (STQ T)
    (CAS M)
```

```
      (TXI (* 3) 4 1)
      (TRA (* 2))
      (TXI (* 1) 4 1)
      (SXA CT 4)
      (ALS 24)
      (ORA S7)
      (TSX PRIN2 4)
      (LDQ T)
      (TIX A 2 1)
      (CLA CT)
 X    (AXT 0 4)
      (TRA 1 4)
 M    (707Q)
 T    (0)
 CT   (0)
 S7   (77777777Q)
      ) ((ALS . 767Q8) (LGL . 4763Q8) (ORA . 4501Q8) (CAS . 340Q8)) )

 LAP (( (PUNOCT SUBR 2)
      (STQ MQ)
      (SXA X4 4)
      (SXA X2 2)
      (PDX 0 2)
      (LDQ 0 2)
      (RQL 12)
      (AXT 3 2)
      (PXD)
      (TSX OCT 4)
      (AXT 1 2)
      (TSX OCT 4)
      (LXD MQ 4)
      (LDQ 0 4)
      (RQL 12)
      (AXT 3 2)
      (TSX OCT 4)
      (AXT 1 2)
      (TSX OCT 4)
      (LDQ (QUOTE OCTAL))
      (TSX MKNO 4)
 X4   (AXT 0 4)
 X2   (AXT 0 2)
      (TRA 1 4)
 MQ   (0)
      ) ((RQL . 4773Q8)) )

 LAP (( (PUNLOC SUBR 1)
      (SXA X4 4)
      (PDX 0 4)
      (LDQ 0 4)
      (RQL 18)
      (AXT 3 2)
      (PXD)
      (TSX OCT 4)
 X4   (AXT 0 4)
      (TRA 1 4)  )
      ((RQL . 4773Q8)) )

 DEFINE ((
```

```
(LAPEVAL (LAMBDA (X) (PROG (S J K)
   (COND ((NULL X) (RETURN BPORG)) ((ATOM X) (GO L1))
    ((EQ (CAR X) (QUOTE E)) (RETURN (MKOB (CONS (QUOTE SPECIAL)
        (CADR X))))) )
    ((EQUAL X (QUOTE (QUOTE NIL))) (RETURN 777600Q))
    ((EQ (CAR X) (QUOTE QUOTE)) (GO B))
    ((EQ (CAR X) (QUOTE SPECIAL)) (GO A)) )
   (SETQ S 0)
   (SETQ J X)
L  (COND ((NULL J) (RETURN S)) )
   (SETQ S (PLUS S (LAPEVAL (CAR J))) )
   (SETQ J (CDR J))
   (GO L)

L1 (COND ((NUMBERP X) (RETURN X)) ((EQ X DOLLAR) (RETURN LOC)) )
   (SETQ K ST)
L2 (COND ((NULL K) (GO L3)) ((EQ (CAAR K) X) (RETURN (CDAR K))) )
   (SETQ K (CDR K))
   (GO L2)

L3 (SETQ K INISYM)
L5 (COND ((NULL K) (GO L6)) ((EQ (CAAR K) X) (RETURN (CDAR K))) )
   (SETQ K (CDR K))
   (GO L5)

L6 (COND ((GET X (QUOTE TWORD))
     (RETURN (LOGAND (GET X (QUOTE TWORD)) 777777Q))) )

L4 (PRINT (LIST X (QUOTE $$$UNDEFINED -- LAP$) ))
   (RETURN 0)

A  (SETQ IV2 (LOGOR IV2 200Q6))
B  (RETURN (MKOB (CONS (CAR X) (CADR X))))
   )))

(JUST (LAMBDA (A) (COND ((MINUSP A) (PLUS A 777777Q)) (T A) )))

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


(MKOB (LAMBDA (S) (PROG (N L)
   (SETQ N OBORG)
   (SETQ L OBLIS)
A  (COND ((NULL L) (GO I))
    ((EQUAL (CAR L) S) (RETURN N))
    ((NULL (CDR L)) (GO MK)) )
   (SETQ L (CDR L))
   (SETQ N (ADD1 N))
   (GO A)

I  (CSETQ OBLIS (LIST S))
   (RETURN OBORG)
MK (RPLACD L (LIST S))
   (RETURN (ADD1 N)) )))

(PUNOBJ (LAMBDA NIL (PROG (OBC FSC FWC OBL FOO FOO2)
   (SETQ FWC FWORG)
   (SETQ OBC OBORG)
```

```
    (SETQ FSC FSORG)
    (SETQ OBL OBLIS)

A   (COND ((NULL OBL) (RETURN (LIST OBC FSC FWC)))
     ((EQ (CAAR OBL) (QUOTE SPECIAL)) (GO S))
     ((EQ (CAAR OBL) (QUOTE QUOTE)) (GO Q))
     (T (ERROR (LIST (QUOTE PUNOBJ) (CAR OBL) ))) )

S   (SETQ FOO2 (GET (CDAR OBL) (QUOTE TWORD)))
    (PUNWORD (LOGOR 20Q6 FSC) (COND ((NULL FOO2) 0) (T (SUB1 FSC)) ) OBC)
    (BLANKS 3) (PRINT (CDAR OBL))
    (PUNWORD 0Q FWC FSC)
    (TERPRI)
    (SETQ FOO ((LAMBDA (J) (PUNWORD (LEFTSHIFT (MKVAL (CAR J)) -12)
        (LOGOR (LEFTSHIFT (MKVAL (CAR J)) 12) (COND
          ((NULL (CDR J)) 7777Q) (T (LEFTSHIFT (MKVAL (CADR J)) -24)) ))
        (ADD1 FWC) ) ) (GET (CDAR OBL) (QUOTE PNAME)) ))
    (TERPRI)
    (PUNWORD 3000001Q (LOGOR (LEFTSHIFT (LOGAND FOO 7) 18) FSC) FWC)
    (SETQ FWC (PLUS 2 FWC))
    (SETQ FSC (SUB1 FSC))
    (COND ((NULL FOO2) (GO H)) )
    (TERPRI)
    (SETQ FOO FSC)
    (SETQ FSC (SUB1 FOO))
    (PUNWORD 0Q (PUNLIST FOO2) FOO)
    (GO H)

Q   (PUNWORD 0Q (PUNLIST (CDAR OBL)) OBC)

H   (TERPRI)
    (TERPRI)
    (SETQ OBC (ADD1 OBC))
    (SETQ OBL (CDR OBL))
    (GO A)  )))

(PUNLIST (LAMBDA (J) (PROG (N L)
    (COND ((NULL J) (RETURN 0))
     ((NUMBERP J) (GO A))
     ((ATOM J) (RETURN (MKOB (CONS (QUOTE SPECIAL) J))) ) )
    (PUNWORD (PUNLIST (CDR J)) (PUNLIST (CAR J)) FSC)
B   (TERPRI)
    (RETURN (ADD1 (SETQ FSC (SUB1 FSC))))

A   (PUNWORD 20Q6 (LOGOR FWC 75Q6) FSC)
    (BLANKS 3) (PRINT J)
    (COND ((MINUSP J) (PUNWORD 77777777Q (PLUS J 77777777Q) (ADD1 FWC) ))
     (T (PUNWORD (LEFTSHIFT J -24) (LOGAND J 77777777Q) (ADD1 FWC) )) )
    (TERPRI)
    (PUNWORD 2000001Q FSC FWC)
    (TERPRI)
    (SETQ FWC (PLUS 2 FWC))
    (GO B)
    )))

(PUNWORD (LAMBDA (IV1 IV2 LOC) (PROG NIL
    (COND ((NOT PNCH) (GO A)) )
    (STARTPUN)
    (PRINT 0) (PRINT 0)
```

```
    (PUNLOC (CDR LOC))
    (PUNOCT (CDR IV1) (CDR IV2))
    (TERPUN)
A   (PRIN1 BLANK)
    (PUNLOC (CDR LOC))
    (PRIN1 BLANK)
    (RETURN (PUNOCT (CDR IV1) (CDR IV2))) )))

))

DEFLIST ((
    (CONS 6020400020)
    (ERROR 4010400030)
    (PRIN1 4010400040)
    (TERPRI 2000400050)
    (*RATOM 2000400060)
    (GENSYM 2000400070)
    (*CALL 40010Q)
    (*RETRN 40011Q)
    (*MKNO 40012Q)
    (TEREAD 2000400130)
    (PRINOCT 6020400140)
    (*SETFLAG 6020400200)
    (*CLRFLAG 6020400210)
    (PRINLIS (NIL))
    (SCRACH 54400Q)
    (BPORG 55400Q)
    (TBPS 67777Q)
    (BSX 7654321Q)
    (BUC 7654321Q)
    (STA 7654321Q)
    (STZ 7654321Q)
    (BOZ 7654321Q)
    (BNZ 7654321Q)
    (LDA 7654321Q)
    (XOR 7654321Q)
    (CAS 7654321Q)
    (LDB 7654321Q)
    (STB 7654321Q)
    (T T)
    ) TWORD)

  (LAMBDA (J) (MAP J (FUNCTION (LAMBDA (K) (DEFLIST (LIST (LIST
        (CAR K) (EVAL (CAR K) NIL) )) (QUOTE TWORD) ) )) ))
  ((DOLLAR LPAR RPAR SLASH COMMA PERIOD PLUSS DASH STAR BLANK EQSIGN))


  CSET (INISYM ((LDA . 200Q5) (STA . 500Q5) (LDB . 220Q5) (STB . 504Q5)
    (STZ . 051Q5) (BOZ . 600Q5) (BNZ . 601Q5) (BSX . 730Q5)
    (BAX . 740Q5) (BSG . 610Q5) (STP . 514Q5) ($A . 777621Q)
    ($Z . 777600Q)
    (LDX . 420Q5) (ATX . 424Q5) (STX . 520Q5) (BPX . 750Q5) (BXE . 720Q5)
    (LDS . 434Q5) (CON . 430Q5) (ADD . 100Q5) (MUL . 120Q5) (LDC . 210Q5)
    (SFT . 020Q5) (ECH . 524Q5) (FAD . 300Q5) (FMP . 330Q5) (FLT . 320Q5)
    (CAS . 400Q5) (SUB . 110Q5) (BUC . 014Q5) (XOR . 4300303) ))

  CSET (FWORG 100000Q)
  CSET (BPORG 44000Q)
  CSET (OBORG 74000Q)
```

```
CSET (FSORG 167777Q)
CSET (OBLIS NIL)
CSET (PNCH *T*)


SPECIAL ((FSC FWC))
COMMON ((
   LOC ST IV2 INISYM FWORG BPORG OBORG FSORG OBLIS PNCH DOLLAR BLANK
))

COMPILE ((LAPEVAL JUST BLANKS MKOB PUNOBJ PUNLIST PUNWORD))

UNSPECIAL ((FSC FWC))
UNCOMMON ((
   LOC ST IV2 INISYM FWORG BPORG OBORG FSORG OBLIS PNCH DOLLAR BLANK
))


EXCISE (*T*)

STOP ))) ))) ))) ))) ))) ))) ))) )))

       TEST

DEFINE ((

(*SPECIND (LAMBDA (J) (GET J (QUOTE SPECIAL)) ))

(COMPILE (LAMBDA (L) (MAPLIST L (FUNCTION (LAMBDA (J)
     (COM1 (CAR J) (GET (CAR J) (QUOTE EXPR)) ) ))) ))

(COM1 (LAMBDA (N A) (PROG2 (COND
    (A (COM2 (QUOTE SUBR) (LENGTH (CADR A)) A N))
    (T (PRINT (LIST N (QUOTE UNDEFINED)))) ) N)))

(COM2 (LAMBDA (TYPE NARGS EXP NAME) (PROG (LISTING LEN)
   (SETQ LISTING (PASS2 (PASS1 NAME EXP) NAME))
   (LAP (CONS (LIST NAME TYPE NARGS LEN)
     (CAR LISTING)) (CADR LISTING))
   (RETURN NAME) )))

(PASS1 (LAMBDA (NAME FN) (PALAM (PROGITER NAME FN) NIL) ))

(PROGITER (LAMBDA (NAME EXP) (COND
   ((AND (EQ (CAADDR EXP) (QUOTE COND)) (PI1 (CDADDR EXP)))
    ((LAMBDA (G1 G2 VS GS) (LIST (QUOTE LAMBDA) VS (CONS
        (QUOTE PROG) (CONS GS (CONS G1 (PI3 (CDADDR EXP) NIL
           (CONS G2 (PAIRMAP VS GS (LIST (LIST (QUOTE GO) G1)) )) ))))))
     (GENSYM) (GENSYM) (CADR EXP)
     (MAPLIST (CADR EXP) (FUNCTION GENSYM))))
   (T EXP) )))

(PI1 (LAMBDA (L) (PROG NIL
A  (COND ((NULL L) (RETURN NIL))
    ((EQ (CAADAR L) NAME) (RETURN T)) )
   (SETQ L (CDR L))
   (GO A) )))

(PAIRMAP (LAMBDA (L M Z) (PROG (A B)
```

```
   (COND ((NULL L) (RETURN Z)))
   (SETQ A (SETQ B (CONS (LIST (QUOTE SETQ) (CAR L) (CAR M) ) Z)))
A  (SETQ L (CDR L))
   (SETQ M (CDR M))
   (COND ((NULL L) (RETURN A)))
   (SETQ B (CDR (RPLACD B (CONS (LIST
        (QUOTE SETQ) (CAR L) (CAR M) ) Z))))
   (GO A) )))

(PI3 (LAMBDA (L C S) (PROG NIL
A  (COND ((NULL L) (RETURN (CONS (CONS (QUOTE COND) C) S)))
    ((EQ (CAADAR L) NAME) (RETURN ((LAMBDA (G3) (PI3 (CDR L)
          (NCONC C (LIST (LIST (CAAR L) (LIST (QUOTE GO) G3) )))
          (CONS G3 (PAIRMAP GS (CDADAR L)
            (CONS (LIST (QUOTE GO) G2) S) )) )) (GENSYM) )) ) )
   (SETQ C (NCONC C (LIST (LIST (CAAR L) (LIST (QUOTE RETURN) (CADAR L)
       )) ))
   (SETQ L (CDR L)) (GO A) )))


(PALAM (LAMBDA (FN B) (COND
   ((ATOM FN) FN) ((EQ (CAR FN) (QUOTE LAMBDA)) (PA4
     (PA5 (CADR FN)) (GENSYM) (GENSYM)))
   ((EQ (CAR FN) (QUOTE LABEL)) (COMP (CADR FN) (CADDR FN)))
   (T (ERROR (CONS FN (QUOTE (NOT FUNCTION))))) )))

(PA4 (LAMBDA (SPECS G G1) (COND
   ((NULL SPECS) (LIST (QUOTE LAMBDA)
     (CADR FN) (PAFORM (CADDR FN) (APPEND (CADR FN) B) )))
   (T (LIST (QUOTE LAMBDA) (CADR FN) (CONC
      (LIST (QUOTE PROG) (LIST G))
      (PA9 SPECS (QUOTE SPECBIND) G1)
      (LIST (LIST (QUOTE SETQ) G (PAFORM
         (CADDR FN) (APPEND (CADR FN) B) )))
      (PA9 SPECS (QUOTE SPECRSTR) G1) (PA12 G) ))) )))

(PA5 (LAMBDA (VARS) (PROG (M)
A  (COND ((NULL VARS) (RETURN M))
    ((*SPECIND (CAR VARS)) (SETQ M (APPEND M (LIST (CAR VARS))))) )
   (SETQ VARS (CDR VARS))
   (GO A) )))

(COMP (LAMBDA (N E) (COND
   ((ATOM E) E)
   (T (COM2 (QUOTE SUBR) (LENGTH (CADR E)) E N)) )))

(PAFORM (LAMBDA (FORM B) (COND
   ((ATOM FORM) (COND
     ((OR (NUMBERP FORM) (MEMBER FORM (QUOTE (NIL T))))
      (LIST (QUOTE QUOTE) FORM))
     ((EQ FORM (QUOTE F)) (QUOTE (QUOTE NIL)))
     ((*SPECIND FORM) (LIST (QUOTE SPECIAL) FORM))
     ((MEMBER FORM B) FORM)
     (T (PROG NIL (PRINT (CONS FORM (QUOTE (  UNDECLARED  ))))
       (RETURN (LIST (QUOTE SPECIAL) FORM)) )) ))
   ((ATOM (CAR FORM)) (SELECT (CAR FORM)
     ((QUOTE COND) (CONS (QUOTE COND) (MAPLIST (CDR FORM)
        (FUNCTION (LAMBDA (J) (LIST (PAFORM (CAAR J) B)
           (PAFORM (CADAR J) B) ))) )))
```

```
      ((QUOTE QUOTE) FORM)
      ((QUOTE PROG) (PA8 (GENSYM) (PA5 (CADR FORM)) (GENSYM) ))
      ((QUOTE FUNCTION) (LIST (QUOTE SPECIAL) (COMP (GENSYM)
         (CADR FORM) )))
      ((QUOTE GO) FORM)
      ((QUOTE CSETQ) (LIST (QUOTE CSET) (LIST (QUOTE QUOTE)
         (CADR FORM)) (PAFORM (CADDR FORM) B)))
      ((QUOTE SELECT) ((LAMBDA (GS) (LIST (LIST (QUOTE LAMBDA)
           (LIST GS) (CONS (QUOTE COND) (PA3 (CDDR FORM)) ))
          (PAFORM (CADR FORM) B))) (GENSYM) ))
      ((QUOTE NOT) (LIST (QUOTE NULL) (PAFORM (CADR FORM) B)))
      ((QUOTE SET) (PROG NIL (PRINT (QUOTE (SET ILLEGAL)))
        (RETURN (QUOTE (QUOTE NIL))) ))
      ((QUOTE CONC) (PA2 (CDR FORM)))
      (CONS (CAR FORM) (PA1 (CDR FORM)) ) ))
   ((OR (EQ (CAAR FORM) (QUOTE LAMBDA)) (EQ (CAAR FORM)
      (QUOTE LABEL))) (CONS (PALAM (CAR FORM) B) (PA1 (CDR FORM))))
   (T (PROG (G)
     (SETQ G (GENSYM))
     (*SETFLAG G 2)
     (RETURN (LIST (LIST (QUOTE LAMBDA) (LIST (LIST (QUOTE SPECIAL)
          G)) (CONS G (PA1 (CDR FORM)))) (PAFORM (CAR FORM) B))))))))

(PA1 (LAMBDA (L) (MAPCAR L (FUNCTION (LAMBDA (J)
     (PAFORM J B) )))))

(PA2 (LAMBDA (L) (COND
   ((NULL L) (QUOTE (QUOTE NIL)))
   (T (LIST (QUOTE APPEND) (PAFORM (CAR L) B) (PA2 (CDR L)))) )))

(PA3 (LAMBDA (L) (COND ((NULL (CDR L))
    (LIST (LIST (QUOTE (QUOTE  T ) ) (PAFORM (CAR L) B) )))
   (T (CONS (LIST (LIST (QUOTE EQ) GS (PAFORM (CAAR L) B))
      (PAFORM (CADAR L) B)) (PA3 (CDR L)) )) )))

(PA7 (LAMBDA (L B) (COND
   ((NULL L) (QUOTE ((RETURN (QUOTE NIL)))))
   ((AND (NULL (CDR L)) (EQ (CAAR L) (QUOTE GO))) L)
   ((AND (NULL (CDR L)) (EQ (CAAR L) (QUOTE RETURN)))
    (LIST (PAFORM (CAR L) B)))
   ((ATOM (CAR L)) (CONS (CAR L) (PA7 (CDR L) B)))
   (T (CONS (PAFORM (CAR L) B) (PA7 (CDR L) B))))))

(PA8 (LAMBDA (G1 SPECS G)
  (COND ((NULL SPECS) (CONS (QUOTE PROG)
     (CONS (CADR FORM) (PA7 (CDDR FORM) (APPEND (CADR FORM) B)) )))
   (T (CONC (LIST (QUOTE PROG) (CONS G SPECS))
     (PA9 SPECS (QUOTE SPECBIND) G1)
     (LIST (LIST (QUOTE SETQ) G (CONS (QUOTE PROG)
         (CONS (DELETEL SPECS (CADR FORM))
          (PA7 (CDDR FORM) (APPEND (CADR FORM) B)))) ))
     (PA9 SPECS (QUOTE SPECRSTR) G1) (PA12 G) )) )))

(PA9 (LAMBDA (V K G) (COND
   (V (LIST (LIST K (LIST (QUOTE QUOTE) V) G))) (T NIL))))

(PA12 (LAMBDA (G) (LIST (LIST (QUOTE RETURN) G))))

(DELETEL (LAMBDA (B M) (MAPCON M (FUNCTION (LAMBDA (J)
```

```
     (COND ((MEMBER (CAR J) B) NIL) (T (LIST (CAR J))) ) )) )))

(PASS2 (LAMBDA (EXP RENAME) (PROG (AC LISTING STOMAP LOCS)
   (SETQ LEN 0) (SETQ STOMAP (QUOTE ((NIL (1 *N) 1))))
   (MAP (CADR EXP) (FUNCTION (LAMBDA (J) (STORE (CAR J) F) )))
   (SETQ AC (LAST (CADR EXP)))
   (COMVAL (CADDR EXP) STOMAP NIL)
   (COND ((NOT (MEMBER (CAADDR EXP) (QUOTE (PROG COND) )))
     (ATTACH (LIST (LIST (QUOTE BSX) (QUOTE *RETRN) 4 (LIST
         (QUOTE E) RENAME) ))) ))
   (SETQ EXP NIL)
A  (COND ((NULL LISTING) (GO B)) ((ATOM (CAR LISTING)) (GO D)) )
H  (SETQ EXP (CONS (CAR LISTING) EXP))
C  (SETQ LISTING (CDR LISTING))
   (GO A)
D  (SETQ AC LISTING)
E  (COND ((NULL (CDR AC)) (GO K)) ((ATOM (CADR AC)) (GO L))
    ((EQUAL (CADR AC) (LIST (QUOTE BUC) (CAR LISTING)))
     (RPLACD AC (CDDR AC)) ) )

K  (COND ((MEMBER (CAR LISTING) LOCS) (GO C)) )
   (SETQ LOCS (CONS (CAR LISTING) LOCS))
   (GO H)

L  (SETQ AC (CDR AC))
   (GO E)

B  (RETURN (LIST EXP (LIST
      (CONS (QUOTE *N) (MINUS LEN))) )) )))

(COMVAL (LAMBDA (EXP STOMAP NAME) (PROG NIL
   (COND ((OR (ATOM EXP) (MEMBER (CAR EXP) (QUOTE (QUOTE SPECIAL))))
     (LAC EXP))
    ((EQ (CAR EXP) (QUOTE SETQ)) (PROG NIL
      (COMVAL (CADDR EXP) STOMAP NAME)
      (ATTACH (LIST (CONS (QUOTE STA) (LOCATE (CADR EXP))))) ))
    ((EQ (CAR EXP) (QUOTE COND)) (COMCOND (CDR EXP) T))
    ((EQ (CAR EXP) (QUOTE PROG)) (COMPROG (CDDR EXP) (CADR EXP) NAME))
    ((EQ (CAR EXP) (QUOTE OR)) (COMBOOL F F (CDR EXP) NIL))
    ((EQ (CAR EXP) (QUOTE AND)) (COMBOOL T F (CDR EXP) NIL))
    ((MEMBER (CAR EXP) (QUOTE (SPECBIND SPECRSTR))) (GO A))
    ((ATOM (CAR EXP)) (CALL (CAR EXP) (COMLIS (CDR EXP))))
    (T (PROG NIL (COMPLY (CAR EXP) (CDR EXP))
      (COMVAL (CADDAR EXP) STOMAP NAME))) )
   (SETQ AC NAME)
B  (RETURN NAME)
A  (CALL (CAR EXP) (CDR EXP))
   (GO B) )))

(COMPROG (LAMBDA (EXP PROGLIS RETN) (PROG (GOLIST HOLD NAME SETS S)
   (SETQ HOLD EXP)
A  (COND ((NULL HOLD) (GO B)) ((ATOM (CAR HOLD))
     (SETQ GOLIST (CONS (CONS (CAR HOLD) (GENSYM) ) GOLIST)))
    ((NOT SETS) (COND ((EQ (CAAR HOLD) (QUOTE SPECBIND))
       (SETQ S (CADADR HOLD))) (T (SETQ SETS T)) )))
   (SETQ HOLD (CDR HOLD))
   (GO A)
B  (SETQ HOLD PROGLIS)
C  (COND ((NULL HOLD) (GO G)))
```

```
   (STORE (CAR HOLD) NIL)
   (COND ((NOT (EQ (CAR HOLD) S))
     (ATTACH (LIST (CONS (QUOTE STZ) (LOCATE (CAR HOLD)) ))) ))
   (SETQ HOLD (CDR HOLD))
   (GO C)
G  (SETQ HOLD EXP)
D  (SETQ NAME (GENSYM))
   (COND ((NULL HOLD) (GO E))
    ((ATOM (CAR HOLD)) (PROG2 (SETQ AC NIL)
     (ATTACH (LIST (CDR (SASSOC (CAR HOLD) GOLIST NIL)))) ))
    ((EQ (CAAR HOLD) (QUOTE GO))
     (ATTACH (LIST (LIST (QUOTE BUC) (CDR (SASSOC (CADAR HOLD)
          GOLIST (FUNCTION (LAMBDA NIL (ERROR (CONS (CADAR HOLD)
              (QUOTE (NOT A LABEL (COMPROG))) )) )) )) ))))
    ((EQ (CAAR HOLD) (QUOTE COND)) (COMCOND (CDAR HOLD) F))
    (T (COMVAL (CAR HOLD) STOMAP NAME)))
   (SETQ HOLD (CDR HOLD))
   (GO D)
E  (COND (RETN (RETURN (ATTACH (LIST RETN))))) )))

(COMCOND (LAMBDA (EXP MODE)  (PROG (FLAG SWITCH GEN)
   (SETQ FLAG T)
A  (COND ((NULL EXP) (GO B)))
   (SETQ GEN (GENSYM))
   (SETQ SWITCH NIL)
   (COND ((EQ (CAADAR EXP) (QUOTE GO)) (GO C)))
   (COMPACT (CAAR EXP) GEN)
   (SETQ AC (COND (SWITCH (QUOTE (QUOTE NIL))) (T NIL)))
   (COMVAL (CADAR EXP) STOMAP NAME)
   (COND ((OR (AND NAME (NULL (CDR EXP)))
      (MEMBER (CAADAR EXP) (QUOTE (RETURN GO))))
     (GO L)))
   (ATTACH (LIST (COND (NAME (LIST (QUOTE BUC) NAME))
      (T (LIST (QUOTE BSX) (QUOTE *RETRN)
        4 (LIST (QUOTE E) RENAME))) )))
L  (ATTACH (LIST GEN))
D  (SETQ EXP (CDR EXP))
   (SETQ AC (COND (SWITCH (QUOTE NIL)) (T (QUOTE (QUOTE NIL)))))
   (GO A)
B  (COND ((AND  FLAG MODE) (ATTACH (QUOTE ((BUC CONDER))))))
   (COND (NAME (ATTACH (LIST NAME))))
   (RETURN NIL)
C  (COMPACT (LIST (QUOTE NULL) (CAAR EXP))
    (CDR (SASSOC (CADR (CADAR EXP)) GOLIST (FUNCTION
       (LAMBDA NIL (ERROR (CONS (CADR (CADAR EXP))
          (QUOTE (NOT A LABEL (COMCOND))) )) )) ) )
   (GO D)  )))

(COMBOOL (LAMBDA (FN MODE EXP A) (PROG (GEN SWITCH)
   (SETQ GEN (GENSYM))
A  (SETQ SWITCH NIL)
   (COND ((NULL EXP) (GO C))
    ((AND MODE (NULL (CDR EXP))  (EQ A FN) ) (GO B)))
   (COMPACT  (COND (FN (CAR EXP)) (T (LIST (QUOTE NULL) (CAR EXP))))
    (COND ((AND MODE (NOT A     )) (COND (FN NAME) (T GEN)))
     (T (COND ((NOT MODE) GEN)  (FN GEN) (T NAME))) ))
   (SETQ AC (COND ((EQ (CAAR LISTING) (QUOTE BNZ)) (QUOTE (QUOTE NIL))
      ) (T (QUOTE (QUOTE T )))))
   (SETQ EXP (CDR EXP))
```

```
   (GO A)
B  (COMPACT (COND (FN (LIST (QUOTE NULL) (CAR EXP)))
     (T (CAR EXP))  )  NAME)
C  (COND ((NOT MODE) (ATTACH (LIST (QUOTE (BUC ($ 2) )) (LIST (QUOTE
         LDA) (LIST (QUOTE QUOTE) FN)) ))))
   (ATTACH (LIST GEN))
   (COND ((NOT MODE) (ATTACH (LIST (LIST (QUOTE LDA)
        (LIST (QUOTE QUOTE) (NOT FN) ) ))) )) )))

(COMPACT (LAMBDA (EXP NAME) (COND
   ((EQ (CAR EXP) (QUOTE NULL)) (PROG2 (SETQ SWITCH (NOT SWITCH))
     (COMPACT (CADR EXP) NAME)))
   ((EQUAL EXP (QUOTE (QUOTE T))) (COND (SWITCH (ATTACH (LIST
        (LIST (QUOTE BUC) NAME)))) (T (SETQ FLAG F))))
   ((EQ (CAR EXP) (QUOTE OR)) (COMBOOL F T (CDR EXP) SWITCH))
   ((EQ (CAR EXP) (QUOTE AND)) (COMBOOL T T (CDR EXP) SWITCH))
   (T (PROG2
     (COND ((EQ (CAR EXP) (QUOTE EQ))
       (CEQ EXP STOMAP))
      (T (COMVAL EXP STOMAP (GENSYM))))
     (ATTACH (LIST (LIST (COND (SWITCH (QUOTE BNZ))
         (T (QUOTE BOZ))) NAME))) )) )))

(CEQ (LAMBDA (EXP STOMAP) (PROG (A)
   (SETQ A (COMLIS (CDR EXP)))
   (COND ((EQUAL (CAR A) AC) (ATTACH (LIST (CONS (QUOTE XOR)
        (LOCATE (CADR A)) ))))
    (T (PROG2 (LAC (CADR A))
      (ATTACH (LIST (CONS (QUOTE XOR) (LOCATE (CAR A)))) ))))
   (SETQ SWITCH (NOT SWITCH)) )))

(COMPLY (LAMBDA (FN ARGS) (MAP (PAIR (CADR FN) ARGS)
   (FUNCTION (LAMBDA (J) (PROG NIL (COMVAL (CDAR J) STOMAP
       (GENSYM)) (STORE (CAAR J) T))))) ))

(COMLIS (LAMBDA (EXP) (PROG (B)
   (RETURN (MAPCAR EXP (FUNCTION (LAMBDA (J) (COND
        ((OR (EQ (CAR J) (QUOTE QUOTE)) (ATOM J)) J)
        (B (PROG2 (STORE AC T) (COMVAL J STOMAP (GENSYM))))
        (T (PROG2 (SETQ B T) (COMVAL J STOMAP (GENSYM)))) ) ) )))
   )))

(STORE (LAMBDA (X Y) (PROG NIL
   (COND ((OR (NULL X) (EQ (CAR X) (QUOTE QUOTE))) (RETURN NIL)))
   (SETQ STOMAP (CONS (CONS X (LIST (LIST
        (ADD1 (ADD1 (CAADAR STOMAP))) (QUOTE *N)) 1)) STOMAP))
   (COND (Y (ATTACH (LIST (CONS (QUOTE STA) (LOCATE X))) )) )
   (SETQ LEN (*MAX LEN (ADD1 (CAADAR STOMAP)) )) )))

(CALL (LAMBDA (FN ARGS) (PROG (HOLD ITEM NUM S X)
   (COND ((MEMBER FN (QUOTE (  SPECBIND SPECRSTR LIST RETURN GO)))
     (GO E))
    ((NULL ARGS) (GO D)) )
   (SETQ NUM 1)
   (SETQ HOLD ARGS)
F  (COND ((NULL HOLD) (GO G))
    ((NULL (CDR HOLD)) (GO G)) )
   (SETQ NUM (ADD1 (ADD1 NUM)))
   (SETQ HOLD (CDR HOLD))
```

```
    (GO F)
G   (SETQ HOLD (REVERSE ARGS))
    (COND ((NULL HOLD) (GO D)) )
    (SETQ X (CAR HOLD))
    (SETQ HOLD (CDR HOLD))
A   (COND ((NULL HOLD) (GO H)) )
    (SETQ ITEM (CAR HOLD))
    (COND ((EQUAL ITEM (QUOTE (QUOTE NIL))) (ATTACH (LIST (LIST
         (QUOTE STZ) NUM 1))))
     ((EQUAL ITEM AC) (ATTACH (LIST (LIST (QUOTE STA) NUM 1))))
     (T (ATTACH (LIST (LIST (QUOTE STB) NUM 1)
       (CONS (QUOTE LDB) (LOCATE ITEM) )))) )
    (SETQ HOLD (CDR HOLD))
    (SETQ NUM (SUB1 (SUB1 NUM)))
    (GO A)
H   (LAC X)
D   (ATTACH (LIST (LIST (QUOTE BSX) (QUOTE *CALL) 4 (LIST (QUOTE E) FN)
      )))
    (RETURN NIL)
E   (COND ((EQ FN (QUOTE GO)) (ERROR FN))
     ((EQ FN (QUOTE RETURN)) (PROG NIL (LAC (CAR ARGS))
       (ATTACH (LIST (COND (RETN (LIST (QUOTE BUC) RETN))
          (T (LIST (QUOTE BSX) (QUOTE *RETRN) 4 (LIST (QUOTE E) RENAME)
            ) ) ))) ) )
     ((EQ FN (QUOTE LIST)) (PROG NIL
       (SETQ AC (LOCATE AC))
       (ATTACH (LIST (LIST (QUOTE BSX) (QUOTE *LIST) 2 (LENGTH ARGS) )))
       (MAP ARGS (FUNCTION (LAMBDA (J) (ATTACH (LIST (CONS 0
            (LOCATE (CAR J)) ))) ))) ))
     ((EQ FN (QUOTE SPECBIND)) (PROG NIL
       (ATTACH (LIST (LIST (QUOTE BUC) FN 0 4)))
       (ATTACH (CDR ARGS))
       (MAP (CADAR ARGS) (FUNCTION (LAMBDA (J)
          (ATTACH (LIST (LIST (COND ((CDR J) 0) (T (QUOTE CAS)) )
             (LIST (SUB1 (CAAR (LOCATE (CAR J)))) (QUOTE *N)) 1
             (LIST (QUOTE E) (CAR J)) ))) ))) ))
     (T (ATTACH (LIST (LIST (QUOTE BSX) (QUOTE SPECRSTR) 4 (CADR ARGS))
        ))) ) )))

(LAC (LAMBDA (X) (COND ((EQUAL AC X) NIL)
   (T (ATTACH (LIST (CONS (QUOTE LDA) (LOCATE X) )))) )))

(ATTACH (LAMBDA (A) (COND
   ((AND (EQ (CAAR A) (QUOTE LDA)) (EQ (CAAR LISTING) (QUOTE STA))
     (EQUAL (CDAR A) (CDAR LISTING))) NIL)
   (T (SETQ LISTING (APPEND A LISTING))) )))

(LOCATE (LAMBDA (B) (SELECT (CAR B)
   ((QUOTE QUOTE) (LIST B)) ((QUOTE SPECIAL) (LIST B))
   (CDR (SASSOC B STOMAP (FUNCTION (LAMBDA NIL (COND
        ((EQ B AC) (PROG NIL (STORE AC T)
          (RETURN (SASSOC B STOMAP (FUNCTION (LAMBDA NIL NIL))))))
        (T (CONS NIL (LOCATE (LIST (QUOTE SPECIAL) B)))) ))))))))

(LAST (LAMBDA (X) (PROG NIL
A  (COND ((NULL X) (RETURN NIL))
    ((NULL (CDR X)) (RETURN (CAR X))) )
   (SETQ X (CDR X))
   (GO A) )))
```

```
(REVERSE (LAMBDA (X) (PROG (Y)
A  (COND ((NULL X) (RETURN Y) ))
   (SETQ Y (CONS (CAR X) Y))
   (SETQ X (CDR X))
   (GO A) )))

(MAPCAR (LAMBDA (L FN) (COND ((NULL L) NIL)
   (T (CONS (FN (CAR L)) (MAPCAR (CDR L) FN))) )))

(*MAX (LAMBDA (X Y) (MAX X Y) ))

))

DEFINE ((

(LAP (LAMBDA (L ST) (PROG (LIS I ORG LOC IV1 IV2 P2)
   (MKOB (CONS (QUOTE SPECIAL) (CAAR L) ))
   (SETQ ORG BPORG)
   (BLANKS 30)
   (PRINT (CAR L))

NP (SETQ LOC BPORG)
   (SETQ LIS (CDR L))
B  (COND ((NULL LIS) (GO EP)) )
   (SETQ I (CAR LIS))
   (COND (P2 (GO I2)) ((ATOM I) (GO ES)) )
NW (SETQ LOC (ADD1 LOC))
KW (SETQ LIS (CDR LIS))
   (GO B)

I2 (COND ((ATOM I) (GO PS)) )
   (SETQ IV2 0)
   (SETQ IV1 (LAPEVAL (CAR I)))
   (COND ((NULL (CDR I)) (GO IW)))
   (SETQ IV2 (LOGOR (JUST (LAPEVAL (CADR I))) IV2))
   (COND ((NULL (CDDR I)) (GO IW)))
   (SETQ IV2 (LOGOR IV2 (LEFTSHIFT (LAPEVAL (CADDR I)) 18)))
   (COND ((NULL (CDDDR I)) (GO IW)))
   (SETQ IV1 (LOGOR IV1 (JUST (LAPEVAL (CADDDR I))) ))
IW (PUNWORD IV1 IV2 LOC)
   (BLANKS 10)
   (PRINT I)
   (GO NW)

ES (SETQ ST (CONS (CONS I LOC) ST))
   (GO KW)
PS (BLANKS 26)
   (PRINT I)
   (GO KW)


EP (COND ((NOT P2) (GO E1)) )
   (TERPRI) (TERPRI)
   (RPLACD (CAAR L) (CONS (QUOTE TWORD) (CONS
      (LOGOR ORG (LEFTSHIFT (CADDAR L) 18)
      (LEFTSHIFT (CADR (CDDAR L)) 24) ) (CDAAR L) )) ) -
   (CSETQ BPORG LOC)
   (RETURN ST)
```

```
E1 (SETQ P2 T)
   (GO NP)
   )))

(SPECIAL (LAMBDA (X) (MAPLIST X (FUNCTION (LAMBDA (J)
     (DEFLIST (LIST (LIST (CAR J) (LIST NIL))) (QUOTE SPECIAL)))))))

(UNSPECIAL (LAMBDA (L) (MAP L (FUNCTION (LAMBDA (J)
     (REMPROP (CAR J) (QUOTE SPECIAL)) ))) ))

(*SETFLAG (LAMBDA (A B) (SPECIAL (LIST A)) ))

))

(LAMBDA NIL (PROG (J)
A (SETQ J (READ))
  (COND ((EQ J (QUOTE SPECIAL)) (SPECIAL (CAR (READ))))
   ((EQ J (QUOTE UNSPECIAL)) (UNSPECIAL (CAR (READ))))
   ((EQ J (QUOTE LAP)) (PROG2 (SETQ J (READ)) (LAP (CAR J) (CADR J)) ))
   ((NULL J) (RETURN NIL))
   (T (COM2 (QUOTE SUBR) (LENGTH (CADADR J)) (CADR J) (CAR J))) )
  (GO A)
  )) NIL

COMPILE ((
   PASS1 PAIRMAP PA5 COMP PA7 PA9 PA12 LAST REVERSE
))

SPECIAL ((
   NAME GS G2 FN B FORM LEN AC STOMAP LISTING SWITCH FLAG RETN GOLIST
   RENAME HOLD EXP
))

COMPILE ((
   COM2 PROGITER PI1 PI3 PALAM PA4 PAFORM PA1 PA2 PA3 PA8 DELETEL
   PASS2 COMVAL COMPROG COMCOND COMBOOL COMPACT CEQ COMPLY COMLIS
   STORE CALL LAC ATTACH LOCATE
))

COMPILE ((MAPCAR))

UNSPECIAL ((
   NAME GS G2 FN B FORM LEN AC STOMAP LISTING SWITCH FLAG RETN GOLIST
   RENAME HOLD EXP
))

PUNOBJ NIL

STOP))))))))))))

LAP (( (ATOM SUBR 1 4)
   (BOZ A)
   (LDX $A 0 5)
   (LDS 0 5 24Q)
   (BNZ A)
   (LDA (QUOTE NIL))
   (BSX *RETRN 4 (E ATOM))
A  (LDA (QUOTE T))
```

```
    (BSX *RETRN 4 (E ATOM))
    ) NIL)

LAP (( (RPLACA SUBR 2 6)
    (LDA (3 *N) 1)
    (LDB (5 *N) 1)
    (STB 0 17Q 370Q)
    (BSX *RETRN 4 (E RPLACA))
    ) ((*N . -6)) )

LAP (( (RPLACD SUBR 2 6)
    (LDA (3 *N) 1)
    (LDB (5 *N) 1)
    (STB 0 17Q 40217Q)
    (BSX *RETRN 4 (E RPLACD))
    ) ((*N . -6)) )

LAP (( (CAR SUBR 1 4)
    (LDA 0 17Q 370Q)
    (BSX *RETRN 4 (E CAR))
    ) NIL)

LAP (( (CDR SUBR 1 4)
    (LDA 0 17Q 40370Q)
    (BSX *RETRN 4 (E CDR))
    ) NIL)

LAP (( (SPECBIND SUBR 0 2)
    (STA AC)
L   (LDA 0 4 40360Q)
    (LDB 0 17Q)
    (STB 0 24Q 360Q)
    (STA 0 24Q 40217Q)
    (LDX 1 1 100005Q)
    (ATX 0 4 5)
    (STX 0 17Q 5)
    (BSG ($ 2) 0 40367Q)
    (BAX L 4 1)
    (LDA AC)
    (BUC 1 4)
AC  (0)
    ) NIL)

LAP (( (SPECRSTR SUBR 0 2)
    (STA AC)
    (STP X 0 370Q)
L   (LDA 0 4 40360Q)
    (LDB 0 24Q)
    (STB 0 17Q 360Q)
    (STZ 0 24Q)
    (BSG ($ 2) 0 40367Q)
    (BAX L 4 1)
    (LDA AC)
X   (BUC)
AC  (0)
    ) NIL)

LAP (( (*LIST SUBR 0 2)
    (STP G 0 370Q)
```

```
    (STP X 0 370Q)
    (STX GX 0 40002Q)
    (LDA (QUOTE NIL))
    (BXE GX 2 0)
    (BUC LP)

G   (LDB 0 22Q)
    (STB 3 1)
    (BSX *CALL 4 (E CONS))
LP  (BPX G 2 1)
GX  (BSX X 2 0)
X   (BUC 0 2)
    ) NIL)

LAP (( (*EVQ SUBR 2 6)
    (BUC SPECBIND 0 4)
L   (CAS (2 *N) 1 (E *FUNC))
    (LDX ($Z 1) 0 5)

G1  (BOZ G0)
    (LDX $A 0 4)
    (LDA 0 4 370Q)
    (STA 3 5)
    (LDA 0 4 40370Q)
    (BAX G1 5 2)

G0  (LDA 1 5)
    (BSX *CALL 4 (E *FUNC))
    (BSX SPECRSTR 4 L)
    (BSX *RETRN 4 (E *EVQ))
    ) ((*N . -6)) )

LAP (( (*SPECIND SUBR 1 4)
    (LDX $A 0 5)
    (LDS 0 5 51Q)
    (BOZ ($ 2))
    (LDA (QUOTE T))
    (BSX *RETRN 4 (E *SPECIND))
    ) NIL)

LAP (( (NUMBERP SUBR 1 4)
    (STP NX 0 370Q)
    (BOZ NN)
    (LDX $A 0 5)
    (LDS 0 5 24Q)
    (BOZ NN)
    (LDS 0 5 40013Q)
    (BXE N 17Q 7)
NN  (LDA (QUOTE NIL))
NX  (BUC)

N   (LDS 0 5 40040Q)
    (LDX $A 0 2)
    (LDA 0 5 370Q)
    (BUC NX)
    ) NIL)

LAP (( (*NUMVAL SUBR 0 2)
    (STP X 0 370Q)
```

```
    (STA A)
    (BUC NUMBERP)
    (BOZ E)
    (LDA 1 17Q)
X   (BUC)
E   (LDB A)
    (STB 3 1)
    (LDA (QUOTE (NOT A NUMBER)))
    (BSX *CALL 4 (E CONS))
    (BSX *CALL 4 (E ERROR))
A   (0)
    ) NIL)

LAP (( (*FIXVAL SUBR 0 2)
    (STP X 0 370Q)
    (BUC *NUMVAL)
    (BXE C 2 2)
X   (BUC)
C   (FAD MN 0 1Q5)
    (SFT -11 0 12Q4)
    (SFT -36 0 10Q4)
    (BUC X)
MN  (LDC 0 0 1Q5)
    ) NIL)

LAP (( (MINUS SUBR 1 4)
    (BUC *NUMVAL)
    (LDC $A)
    (BUC *MKNO)
    (BSX *RETRN 4 (E MINUS))
    ) NIL)

LAP (( (MINUSP SUBR 1 4)
    (BUC *NUMVAL)
    (BSG M 0 40177Q)
P   (LDA (QUOTE NIL))
    (BSX *RETRN 4 (E MINUSP))
M   (LDA (QUOTE T))
    (BSX *RETRN 4 (E MINUSP))
    ) NIL)

LAP (( (FIXP SUBR 1 4)
    (BUC *NUMVAL)
    (BXE N 2 2)
    (LDA (QUOTE T))
    (BSX *RETRN 4 (E FIXP))
N   (LDA (QUOTE NIL))
    (BSX *RETRN 4 (E FIXP))
    ) NIL)

LAP (( (FLOATP SUBR 1 4)
    (BUC *NUMVAL)
    (BXE N 2 2)
    (LDA (QUOTE NIL))
    (BSX *RETRN 4 (E FLOATP))
N   (LDA (QUOTE T))
    (BSX *RETRN 4 (E FLOATP))
    ) NIL)
```

```
LAP (( (*COMPAT SUBR 0 2)
    (STP X 0 370Q)
    (BUC *NUMVAL)
    (STA CT)
    (LDA (3 *N) 1)
    (BXE F1 2 2)
    (BUC *NUMVAL)
    (BXE F2 2 2)
F3  (LDB CT)
X   (BUC)

F2  (ECH CT)
    (FLT K)
    (ECH CT)
    (BUC F3)

F1  (BUC *NUMVAL)
    (BXE F3 2 2)
    (FLT K)
    (BUC F3)

CT  (0)
K   (1071)
    ) ((*N . -6)) )

LAP (( (*LOGOR SUBR 2 8)
    (BUC *FIXVAL)
    (STA (7 *N) 1)
    (LDA (3 *N) 1)
    (BUC *FIXVAL)
    (CON (7 *N) 1 34Q2)
    (BSX *MKNO 2 5)
    (BSX *RETRN 4 (E *LOGOR))
    ) ((*N . -8)) )

LAP (( (*LOGAND SUBR 2 8)
    (BUC *FIXVAL)
    (STA (7 *N) 1)
    (LDA (3 *N) 1)
    (BUC *FIXVAL)
    (CON (7 *N) 1 4Q2)
    (BSX *MKNO 2 5)
    (BSX *RETRN 4 (E *LOGAND))
    ) ((*N . -8)) )

LAP (( (*LOGXOR SUBR 2 8)
    (BUC *FIXVAL)
    (STA (7 *N) 1)
    (LDA (3 *N) 1)
    (BUC *FIXVAL)
    (CON (7 *N) 1 30Q2)
    (BSX *MKNO 2 5)
    (BSX *RETRN 4 (E *LOGXOR))
    ) ((*N . -8)) )

LAP (( (*PLUS SUBR 2 6)
    (BUC *COMPAT)
    (BXE F 2 2)
    (ADD ($A 1))
```

```
    (BSX *MKNO 2 1)
    (BSX *RETRN 4 (E *PLUS))
F   (FAD ($A 1))
    (BSX *MKNO 2 2)
    (BSX *RETRN 4 (E *PLUS))
    ) NIL)

LAP (( (*TIMES SUBR 2 6)
    (BUC *COMPAT)
    (BXE F 2 2)
    (MUL ($A 1))
    (LDA ($A 1))
    (BSX *MKNO 2 1)
    (BSX *RETRN 4 (E *TIMES))
F   (FMP ($A 1))
    (BSX *MKNO 2 2)
    (BSX *RETRN 4 (E *TIMES))
    ) NIL)

LAP (( (LSHIFT SUBR 2 6)
    (BUC *FIXVAL)
    (LDC $A)
    (STA Q)
    (LDA (3 *N) 1)
    (BUC *FIXVAL)
    (SFT Q 0 1Q4)
    (BSX *MKNO 2 5)
    (BSX *RETRN 4 (E LSHIFT))
Q   (0)
    ) ((*N . -6)) )

LAP (( (*PLANT SUBR 2 6)
    (BUC *FIXVAL)
    (LDX $A 0 3)
    (LDA (3 *N) 1)
    (BUC *NUMVAL)
    (STA 0 3)
    (LDA (3 *N) 1)
    (BSX *RETRN 4 (E *PLANT))
    ) ((*N . -6)) )

LAP (( (*GETNO SUBR 1 4)
    (BUC *FIXVAL)
    (LDA 0 17Q)
    (BSX *MKNO 2 5)
    (BSX *RETRN 4 (E *GETNO))
    ) NIL)

LAP (( (*LOCN SUBR 1 4)
    (BSX *MKNO 2 5)
    (BSX *RETRN 4 (E *LOCN))
    ) NIL)

(CAAR (LAMBDA (X) (CAR (CAR X)) ))
(CADR (LAMBDA (X) (CAR (CDR X)) ))
(CDAR (LAMBDA (X) (CDR (CAR X)) ))
(CDDR (LAMBDA (X) (CDR (CDR X)) ))
(CAAAR (LAMBDA (X) (CAR (CAR (CAR X))) ))
(CAADR (LAMBDA (X) (CAR (CAR (CDR X))) ))
```

```
(CADAR (LAMBDA (X) (CAR (CDR (CAR X))) ))
(CADDR (LAMBDA (X) (CAR (CDR (CDR X))) ))
(CDAAR (LAMBDA (X) (CDR (CAR (CAR X))) ))
(CDADR (LAMBDA (X) (CDR (CAR (CDR X))) ))
(CDDAR (LAMBDA (X) (CDR (CDR (CAR X))) ))
(CDDDR (LAMBDA (X) (CDR (CDR (CDR X))) ))
(CAAAAR (LAMBDA (X) (CAR (CAR (CAR (CAR X)))) ))
(CAAADR (LAMBDA (X) (CAR (CAR (CAR (CDR X)))) ))
(CAADAR (LAMBDA (X) (CAR (CAR (CDR (CAR X)))) ))
(CAADDR (LAMBDA (X) (CAR (CAR (CDR (CDR X)))) ))
(CADAAR (LAMBDA (X) (CAR (CDR (CAR (CAR X)))) ))
(CADADR (LAMBDA (X) (CAR (CDR (CAR (CDR X)))) ))
(CADDAR (LAMBDA (X) (CAR (CDR (CDR (CAR X)))) ))
(CADDDR (LAMBDA (X) (CAR (CDR (CDR (CDR X)))) ))
(CDAAAR (LAMBDA (X) (CDR (CAR (CAR (CAR X)))) ))
(CDAADR (LAMBDA (X) (CDR (CAR (CAR (CDR X)))) ))
(CDADAR (LAMBDA (X) (CDR (CAR (CDR (CAR X)))) ))
(CDADDR (LAMBDA (X) (CDR (CAR (CDR (CDR X)))) ))
(CDDAAR (LAMBDA (X) (CDR (CDR (CAR (CAR X)))) ))
(CDDADR (LAMBDA (X) (CDR (CDR (CAR (CDR X)))) ))
(CDDDAR (LAMBDA (X) (CDR (CDR (CDR (CAR X)))) ))
(CDDDDR (LAMBDA (X) (CDR (CDR (CDR (CDR X)))) ))

SPECIAL ((FN))

(CSET (LAMBDA (X Y) (COND
   ((NOT (ATOM X)) (ERROR (CONS X (QUOTE (NOT AN ATOM (CSET)) ))) )
   (T (RPLACA X (LIST Y))) )))

(DEFINE (LAMBDA (L) (MAPLIST L (FUNCTION (LAMBDA (J) ((LAMBDA (K)
       (COM2 (QUOTE SUBR) (LENGTH (CADADR K))'(CADR K) (CAR K) ))
      (MDEF (CAR J)) ))) )))

(MDEF (LAMBDA (L) (COND ((ATOM L) L)
   ((EQ (CAR L) (QUOTE QUOTE)) L)
   ((MEMBER (CAR L) (QUOTE (LAMBDA LABEL PROG)))
    (CONS (CAR L) (CONS (CADR L) (MDEF (CDDR L)) )) )
   ((GET (CAR L) (QUOTE MACRO)) (MDEF ((GET (CAR L) (QUOTE MACRO)) L)) )
   (T (MAPLIST L (FUNCTION (LAMBDA (J) (MDEF (CAR J)) )) )) )))

(MACRO (LAMBDA (L) (MAPCAR L (FUNCTION (LAMBDA (J) (PROG2
      (COM2 (QUOTE SUBR) (LENGTH (CADAR J)) (CADR J) (CAR J))
      (DEFLIST (LIST (CAR J) (CAAAR J)) (QUOTE MACRO)) ))) )))

SPECIAL ((OB PRO))

(DEFLIST (LAMBDA (L PRO) (MAPLIST L (FUNCTION (LAMBDA (J)
     (DEF1 (CAAR J) (CADAR J)) )) )))

(DEF1 (LAMBDA (OB L) (PROG NIL
   (RPLACA (PROP OB PRO (FUNCTION (LAMBDA NIL
       (CDDR (RPLACD OB (CONS PRO (CONS NIL (CDR OB))) ))))) ) L)
   (RETURN OB) )))

UNSPECIAL ((OB PRO))

(NULL (LAMBDA (A) (COND ((NULL A) T) (T NIL) )))

(EQ (LAMBDA (A B) (COND ((EQ A B) T) (T NIL) )))
```

```
(EQUAL (LAMBDA (A B) (COND
   ((NUMBERP A) (COND ((NUMBERP B) (EQP A B)) (T F) ))
   ((ATOM A) (EQ A B))
   ((ATOM B) F)
   ((EQUAL (CAR A) (CAR B)) (EQUAL (CDR A) (CDR B)))
   (T F) )))

(EQP (LAMBDA (A B) (LESSP (ABSVAL (DIFFER A B)) 3.0E-6)))

(ABSVAL (LAMBDA (X) (COND ((MINUSP X) (MINUS X)) (T X) )))


(MEMBER (LAMBDA (U V) (PROG NIL
A  (COND ((NULL V) (RETURN NIL)) ((EQUAL (CAR V) U) (RETURN T)) )
   (SETQ V (CDR V)) (GO A)  )))

(SASSOC (LAMBDA (X Y FN) (PROG NIL
A  (COND ((NULL Y) (RETURN (FN))) ((EQ (CAAR Y) X) (RETURN (CAR Y))) )
   (SETQ Y (CDR Y)) (GO A)  )))

(PAIR (LAMBDA (X Y) (PROG (M A B)
   (SETQ A X)
   (SETQ B Y)
L  (COND ((NULL A) (COND ((NULL B) (RETURN M))
      (T (ERROR (LIST (QUOTE (PAIR ERROR F2)) X Y))) ))
    ((NULL B) (ERROR (LIST (QUOTE (PAIR ERROR F3)) X Y)) ))
   (SETQ M (CONS (CONS (CAR A) (CAR B)) M))
   (SETQ A (CDR A))
   (SETQ B (CDR B))
   (GO L) )))

(APPEND (LAMBDA (X Y) (COND ((NULL X) Y)
   (T (CONS (CAR X) (APPEND (CDR X) Y))) )))

(NCONC (LAMBDA (X Y) (PROG (M)
   (COND ((NULL X) (RETURN Y)) )
   (SETQ M X)
A  (COND ((NULL (CDR M)) (GO B)) )
   (SETQ M (CDR M))
   (GO A)
B  (RPLACD M Y)
   (RETURN X) )))

(LENGTH(LAMBDA (M) (PROG (N) (SETQ N 0)
A (COND ((NULL M) (RETURN N)) ) (SETQ N (ADD1 N)) (SETQ M (CDR M))
   (GO A) )))

(MAP (LAMBDA (X FN) (PROG (M)
   (SETQ M X)
LP (COND ((NULL M) (RETURN NIL)) )
   (FN M)
   (SETQ M (CDR M))
   (GO LP) )))

(MAPCON (LAMBDA (X FN) (COND ((NULL X) NIL)
   (T (NCONC (FN X) (MAPCON (CDR X) FN))) )))

(MAPLIST (LAMBDA (X FN) (COND ((NULL X) NIL)
```

```
    (T (CONS (FN X) (MAPLIST (CDR X) FN))) )))

(ADD1 (LAMBDA (X) (*PLUS X 1) ))

(SUB1 (LAMBDA (X) (*PLUS X -1) ))

(ZEROP (LAMBDA (X) (EQUAL X 0) ))

(DIFFER (LAMBDA (X Y) (*PLUS X (MINUS Y)) ))

(LESSP (LAMBDA (X Y) (COND ((MINUSP (DIFFER X Y)) T) (T NIL) )))

(GREATERP (LAMBDA (X Y) (COND ((MINUSP (DIFFER Y X)) T) (T NIL) )))

(*MAX (LAMBDA (X Y) (COND ((LESSP X Y) Y) (T X) )))

(*MIN (LAMBDA (X Y) (COND ((LESSP Y X) X) (T Y) )))

(GET (LAMBDA (X Y) (PROG NIL
A   (COND ((NULL X) (RETURN NIL))
     ((EQ (CAR X) Y) (RETURN (CADR X))) )
    (SETQ X (CDR X))
    (GO A) )))

(PROP (LAMBDA (X Y FN) (PROG NIL
A   (COND ((NULL X) (RETURN (FN)))
     ((EQ (CAR X) Y) (RETURN (CDR X))) )
    (SETQ X (CDR X))
    (GO A) )))

UNSPECIAL ((FN))

(PROG2 (LAMBDA (X Y) Y))

(SPECIAL (LAMBDA (L) (MAPCAR L
    (FUNCTION (LAMBDA (J) (*SETFLAG  J  2) )) )))

(UNSPECIAL (LAMBDA (L) (MAPCAR L
    (FUNCTION (LAMBDA (J) (*CLRFLAG  J  2) )) )))

(TRACE (LAMBDA (L) (MAPCAR L
    (FUNCTION (LAMBDA (J) (*SETFLAG  J  1) )) )))

(UNTRACE (LAMBDA (L) (MAPCAR L
    (FUNCTION (LAMBDA (J) (*CLRFLAG  J  1) )) )))

(PRINT (LAMBDA (X) (PROG ()
    (PRIN0 X)
    (TERPRI)
    (RETURN X) )))

(PRIN0 (LAMBDA (X) (PROG (J)
    (COND ((ATOM X) (GO A9)) )
    (SETQ J X)
    (PRIN1 LPAR)
A3  (PRIN0 (CAR J))
    (COND ((NULL (CDR J)) (GO A6)) )
    (PRIN1 BLANK)
    (COND ((ATOM (CDR J)) (GO P1)) )
```

```
   (SETQ J (CDR J))
   (GO A3)

P1 (PRIN1 PERIOD)
   (PRIN1 BLANK)
   (PRIN1 (CDR J))
A6 (PRIN1 RPAR)
   (RETURN X)

A9 (PRIN1 X)
   (RETURN X) )))

(READ (LAMBDA () (PROG (J)
   (SETQ J (*RATOM))
   (COND ((EQ J LPAR) (RETURN (READ1)))
    ((OR (EQ J RPAR) (EQ J PERIOD))
     (ERROR (QUOTE (EXTRA RPAR OR PERIOD) )))
    (T (RETURN J)) ) )))

(READ1 (LAMBDA () (PROG (J K)
   (SETQ J (*RATOM))
   (COND ((EQ J LPAR) (RETURN (CONS (READ1) (READ1))))
    ((EQ J RPAR) (RETURN NIL))
    ((EQ J PERIOD) (GO RD))
    (T (RETURN (CONS J (READ1)))))
RD (SETQ K (*RATOM))
   (COND ((EQ K LPAR) (SETQ K (READ1)))
    ((OR (EQ K RPAR) (EQ K PERIOD)) (ERROR
      (APPEND (QUOTE (BAD RPAR OR DOT AFTER DOT AFTER)) (LIST K) )) ))
   (SETQ J (*RATOM))
   (COND ((EQ J RPAR) (RETURN K))
    (T (ERROR (APPEND (QUOTE (RPAR ROD AFTER)) (LIST J)))) ) )))

(EVALQUOT (LAMBDA NIL (PROG (S1 J SW LWT LOC ROP K)
   (TERPRI)
AA (TEREAD)
   (SETQ ROP NIL)
A  (SETQ SW NIL)
B  (SETQ J (READ))
   (COND ((NOT (ATOM J)) (GO A2)) )
   (SELECT J (SLASH (GO S1)) (EQSIGN (GO E1)) (COLON (GO C1))
    (LARR (GO L1)) (UPARR (GO U1))
    (LSTHAN (GO LI)) (GRTHAN (GO RI)) (DOLLAR (GO D1))
    (STAR (GO STOR)) NIL)

A2 (SETQ LWT J)
   (COND (SW (GO E)) )
   (SETQ S1 J)
   (SETQ SW T)
   (GO B)

E  (COND ((ATOM S1) (GO EV)) )
   (SETQ K BPORG) (SETQ BPORG SCRACH)
   (COM2 (QUOTE SUBR) (LENGTH (CADR S1)) S1 (QUOTE *FUNC))
   (SETQ BPORG K)

EV (SETQ LWT (*EVQ (CAAR (COND ((ATOM S1) S1) (T (QUOTE *FUNC)) )) J))
   (PRINT LWT)
   (GO AA)
```

```
C1 (SETQ LWT (PRINT (*EVALA LWT)))
   (GO A)

D1 (SETQ LWT (PRINT (CONS (CAR LWT) (CDR LWT))))
   (GO AA)

E1 (SETQ LWT (PRINOCT (*EVALA LWT) 16))
   (GO A)

LI (SETQ LWT (PRINT (CAR LWT)))
   (GO AA)

RI (SETQ LWT (PRINT (CDR LWT)))
   (GO AA)

L1 (COND ((AND ROP SW) (*PLANT LWT ROP)) )
   (TERPRI)
   (PRINOCT (SETQ LOC (ADD1 LOC)) 16)
   (GO U2)

U1 (COND ((AND ROP SW) (*PLANT LWT ROP)) )
   (TERPRI)
   (PRINOCT (SETQ LOC (SUB1 LOC)) 16)
U2 (PRIN1 SLASH)
   (BLANKS 3)
   (SETQ SW T)

S1 (SETQ ROP (SETQ LOC (*EVALA LWT)))
   (SETQ LWT (PRINOCT (*GETNO LOC) 16))
   (BLANKS 3)
   (GO A)

STOR (COND ((AND ROP SW) (*PLANT S1 ROP)) )
   (TERPRI)
   (GO AA)
   )))

(*EVALA (LAMBDA (A) (COND ((NUMBERP A) A)
   ((ATOM A) (CAAR A)) (T (*LOCN A)) )))

SPECIAL ((ST IV EI LOC))

(LAP (LAMBDA (L ST) (PROG (LIS I ORG LOC IV P2 MODE TRW EI)
A  (COND ((NULL (CAR L)) (GO A1)) ((ATOM (CAR L)) (GO A2)) )
   (SETQ TRW (*LOGOR BPORG (LSHIFT (LAPEVAL (CADDAR L)) 18)
     (LSHIFT (LAPEVAL (CADR (CDDAR L))) 24) ))
A1 (SETQ ORG BPORG)
   (SETQ MODE T)
   (GO NP)
A2 (SETQ ORG (JUST (LAPEVAL (CAR L))) )

NP (COND (EI (RETURN ST)))
   (SETQ LOC ORG)
   (SETQ LIS (CDR L))
B  (COND ((NULL LIS) (GO EP)) )
   (SETQ I (CAR LIS))
   (COND ((ATOM I) (GO ES)) ((EQ (CAR I) (QUOTE EQU)) (GO D))
    (P2 (GO I2)) )
```

```
NW (SETQ LOC (ADD1 LOC))
KW (SETQ LIS (CDR LIS))
   (GO B)

I2 (SETQ IV (LAPEVAL (CAR I)))
   (COND ((NULL (CDR I)) (GO IW)))
   (SETQ IV (*LOGOR (JUST (LAPEVAL (CADR I))) IV))
   (COND ((NULL (CDDR I)) (GO IW)))
   (SETQ IV (*LOGOR (LSHIFT (LAPEVAL (CADDR I)) 18) IV))
   (COND ((NULL (CDDDR I)) (GO IW)))
   (SETQ IV (*LOGOR (LSHIFT (JUST (LAPEVAL (CADDDR I))) 24) IV))
IW (*PLANT IV LOC)
   (COND ((NOT PRINLIS) (GO NW)) )
   (PRINOCT LOC 6) (PRIN1 BLANK)
   (PRINOCT IV 16)
   (BLANKS 10)
   (PRINT I)
   (GO NW)

D  (COND (P2 (GO D2)) )
   (SETQ IV (LAPEVAL (CADDR I)) )
   (SETQ ST (CONS (CONS (CADR I) IV) ST))
   (GO KW)

D2 (COND ((NOT PRINLIS) (GO KW)) )
   (BLANKS 7)
   (PRINOCT IV 16)
   (PRIN1 BLANK)
   (PRINT I)
   (GO KW)

ES (COND (P2 (GO PS)) )
   (SETQ ST (CONS (CONS I LOC) ST))
   (GO KW)

PS (COND ((NOT PRINLIS) (GO KW)) )
   (BLANKS 26)
   (PRINT I)
   (GO KW)

EP (COND ((NOT P2) (GO E1)) (E1 (RETURN ST)) ((NULL TRW) (RETURN ST)) )
   (CSET (CAAR L) TRW)
   (RETURN ST)

E1 (COND ((AND MODE (GREATERP LOC TBPS)) (GO SE)) )
   (SETQ P2 T)
   (GO NP)

SE (PRINT (LIST (QUOTE (OUT OF BPS AT)) LOC))
   (TERPRI)
   (RETURN ST) )))

(LAPEVAL (LAMBDA (X) (PROG (S J K)
   (COND ((NULL X) (RETURN BPORG)) ((ATOM X) (GO L1))
    ((EQUAL X (QUOTE (QUOTE NIL))) (RETURN 777600Q))
    ((EQ (CAR X) (QUOTE E)) (RETURN (*LOCN (CADR X))) )
    ((EQ (CAR X) (QUOTE QUOTE)) (RETURN (*LOCN
       (PROG (G) (SETQ G (GENSYM)) (RPLACA G (CADR X))
        (RPLACD G NIL) (RETURN G) ))) )
```

```
     ((EQ (CAR X) (QUOTE SPECIAL)) (RETURN (PROG2
        (SETQ IV (*LOGOR IV 20Q6)) (*LOCN (CADR X)) )) ) )

    (SETQ S 0)
    (SETQ J X)
A   (COND ((NULL J) (RETURN S)) )
    (SETQ S (*PLUS S (LAPEVAL (CAR J))) )
    (SETQ J (CDR J))
    (GO A)

L1  (COND ((NUMBERP X) (RETURN X)) ((EQ X DOLLAR) (RETURN LOC)) )
    (SETQ K ST)
L2  (COND ((NULL K) (GO L3)) ((EQ (CAAR K) X) (RETURN (CDAR K))) )
    (SETQ K (CDR K))
    (GO L2)

L3  (COND ((NUMBERP (CAAR X)) (RETURN (CAAR X))) )
U   (PRINT (LIST X (QUOTE NOT) (QUOTE DEFINED) ))
    (SETQ EI T)
    (RETURN 0) )))

(JUST (LAMBDA (A) (*LOGAND A 777777Q) ))

(BLANKS (LAMBDA (A) (PROG NIL
L   (COND ((ZEROP A) (RETURN NIL)) )
    (PRIN1 BLANK)
    (SETQ A (SUB1 A)) (GO L) )))

UNSPECIAL ((ST IV EI LOC))


NIL    ))) ))) ))) ))) ))) ))) ))) )))


        FIN       END SAUNDERS COMPILER
```

1514

TOTAL 1514*

Appendix 4

The LISP Program

for the A-Language

William Henneman

1. The A-Language System Listing

```
DEFINE((
(NOT (LAMBDA (X) (COND ((FQ X T) NIL) (TT))))
(LAST (LAMBDA (X) (COND ((NULL X) NIL) ((NULL (CDR X)) (CAR X))
     (T (LAST (CDR X))))))
(FIFTH (LAMBDA (X) (CAR (CDDDDR X))))
(VAR (LAMBDA (X Y) (COND ((NULL Y) NIL)
     ((OR (EQUAL (CAR Y) (CADR X)) (MEMBER (CAR Y) (CADDR X)))
     (VAR X (CDR Y)))
     (T (CONS (CAR Y) (VAR X (CDR Y)))))))
(UNWIND (LAMBDA (X) (VAR X (FIFTH X))))
(THENS (LAMBDA (X) (COND ((EQUAL (CAR X) (QUOTE THEN)) (THEN
     (CDR X)))
          (T (THENS (CDR X))))))
(DEF (LAMBDA (X) (LIST (QUOTE DEFINE) (LIST (LIST (LIST
(CADR X) (LIST (QUOTE LAMBDA ) (UNWIND X) (DEFN (LAST X)))))))))
(DEFL (LAMBDA (X) (DEFL1 (CONDIT 0 0 X))))
(DEFL1 (LAMBDA (X) (DEFL2 (VERB (IF (CDR X))) (VERB (THENS X)))))
(DEFL2 (LAMBDA (X Y) (LIST)
     (COND ((EQUAL (LENGTH X) 1) (CAR X)) (T X))
     (COND ((EQUAL (LENGTH Y) 1) (CAR Y)) (T Y)) )))
(FF (LAMBDA (X) (COND ((NULL X) NIL)
     ((ATOM X) X)
     (T (FF (CAR X))) )))
```

```
(DEFM1 (LAMBDA (X Y) (COND
     ((NULL X) NIL)
     (T (CONS (DEFL X) (COND
          ((NULL Y) NIL)
          ((EQUAL (CAR Y) (FF Y)) (LIST Y))
          (T Y) ))) )))
(DEFJ (LAMBDA (X N) (COND ((NULL X) NIL)
     ((EQUAL (CAR X) (QUOTE IF)) (DEFJ (CDR X) (ADD1 N)))
     ((EQUAL (CAR X) (QUOTE ELSE)) (COND
          ((EQUAL N 1) (COND
               ((NULL (CDR X)) (PRINT (QUOTE (INC COND))))
               ((EQUAL (CADR X) (QUOTE IF)) (DEFM (CDR X)))
               (T (DEFL (CONS (QUOTE IF) (CONS (QUOTE T)
               (CONS (QUOTE THEN) (CDR X)))))))
          (T (DEFJ (CDR X) (SUB1 N)))) )
     (T (DEFJ (CDR X) N)))))
DEFINE((
(DEFN (LAMBDA (X) (COND ((NULL X) NIL)
     ((EQUAL (CAR X) (QUOTE IF)) (CONS (QUOTE COND) (DEFM X)))
     (T (VERB (CONS (CAR X) (DEFN (CDR X))))) )))
(VERBIOSE (LAMBDA (X Y) (VERBIOSE1 (VERBS X Y Y 0) Y)))
(VERBIOSE1 (LAMBDA (X Y (COND ((EQUAL X Y) T) (T X))))
DEFINE((
(VERB1 (LAMBDA (X Y) (COND ((EQUAL Y T) X) (T (VERB Y)) )))
(SEX (LAMBDA (W X Y Z N) (COND
     ((NULL X) NIL)
     ((EQUAL (DIFFERENCE N (VERBPOS X Y)) 0)
     (CONS Y (SEXY W X (CONS Y Z ) (LENGTH X))))
     (T (UNTOUCH (CDR X) Y Z (SUB1 N))) )))
(BEGINCOUNT (LAMBDA (M N X) (COND ((NULL X) NIL)
     ((EQUAL (CAR X) (QUOTE BEGIN)) (BEGCOUNT M N X))
     (T (BEGINCOUNT M N (CDR X))) )))
(BEGCOUNT (LAMBDA (M N X) (COND ((NULL X) NIL)
     ((EQUAL (CAR X) (QUOTE BEGIN)) (CONS (CAR X) (BEGCOUNT
          (ADD1 M) N (CDR X))))
     ((EQUAL (CAR X) (QUOTE END)) ( COND
          ((EQUAL (SUB1 M) N ) NIL)
          (T (CONS (CAR X) (BEGCOUNT M (ADD1 N) (CDR X))))))
     (T (CONS (CAR X) (BEGCOUNT M N (CDR X))) )))
(VERBDEF1 (LAMBDA (X Y Z) (COND
     ((NULL X) (COND ((NULL X) Y) (T (CONS Y Z))))
     ((NULL Z) (COND ((NULL Y) X) (T (APPEND X (LIST Y)))))
     ((NULL Y) (APPEND X Z))
     (T (APPEND X (CONS Y Z ))) )))
(PROPFORM (LAMBDA (X) (COND ((EQUAL (LENGTH X) L) X)
     (T (CONS (CARX) (LIST (CDR X)))) )))
(CONDIT (LAMBDA (M N X) (COND
     ((NULL X) NIL)
```

```
     ((EQUAL (CAR X) (QUOTE IF))  (CONS (CAR X)
               (CONDIT (ADD1 M) N (CDR X))))
     ((EQUAL (CAR X) (QUOTE ELSE)) (COND
          ((EQUAL (SUB1 M) NL NIL)
          (T (CONS (CAR X) (CONDIT M (ADD1 N) (CDR X))))))
     (T (CONS (CAR X) (CONDIT M N (CDR X)))))))
(H (LAMBDA (X Y) (COND ((NULL Y) (MINUS 1))
     ((NULL X) (H LISTC (CDR Y)))
     ((AND (EQUAL (CAAR X) (QUOTE DEFINE)) (EQUAL (CADAR X)
     (CAR Y)))
          (COND ((GREATERP (CAAR (CDDDAR X)) (H LISTC (CDR Y)))
     (CAAR (CDDDAR X))) (R (H LISTC (CDR Y))) ))
     (T (H (CDR X) Y)) )))
(VERB (LAMBDA (X) (VERB1 X (VERBIOSE LISTC X))))
(VERBS (LAMBDA (X Y Z N) (COND ((NULL X) (VERBS LISTC (CDR Y) Z
     (ADD1 N))) ((NULL Y) T)
     ((MEMBER (QUOTE BEGIN ) Y) (APPEND (PREBEG Y) (CONS
     (BEGINFN (CDR (BEGINCOUNT 0 0 Y))) (ENDCOUNT 0 Y))))
     ((AND (EQUAL (CAAR X) (QUOTE DEFINE)) (EQUAL (CADAR X)
     (CAR Y)))
     (COND ((GREATERP (CAAR (CDDDAR X)) (H LISTC (CDR Y)))
     (VERBDEF (CAR X) Z N))) (T (VERBS LISTC (CDR Y) Z
     (ADD1 N)))))
     (T (VERBS (CDR X) Y Z N)) )))
(TRANSLATE (LAMBDA (X) (COND ((NULL X) NIL)
     ((ATOM (CAR X)) NIL)
     ((EQ (CAAR X) (QUOTE DEFINE)) (APPEND (DEF (CAR X))
          (TRANSLATE (CDR X))))
     (T (APPEND (PROPFORM (VERB (CAR X))) (TRANSLATE
     (CDR X)))) )))
))
     STOP))))))))))STOP
     FIN
     (T (SEX (CDR W) X Y Z (SUB1 N))) )))
(VERBPOS (LAMBDA (X Y) (COND
     ((EQUAL (CAR X) Y) 0)
     (T (ADD1 (VERBPOS (CDR X) Y))) )))
))
DEFINE((
(VERBDEF (LAMBDA (X Z N) (VERBDEF1
     (UNTOUCH (FIFTH X) (CADR X) Z N )
     (SEX Z (FIFTH X) (CADR X) (CADDR X) N)
     (NONTOUCH (FIFTH X) Z N) )))
))
DEFINE((
(NONTOUCH (LAMBDA (X Y N) (COND
     ((NULL Y) NIL)
     ((EQUAL (PLUS N (LENGTH X)) 1) (CDR Y))
```

```
      (T (NONTOUCH X (CDR Y) (SUB1 N))) )))
))
DEFINE((
(IF (LAMBDA (X) (COND ((NULL X) NIL)
      ((MEMBER (QUOTE BEGIN)X) (IF
      (APPEND(PREBEG X) (CONS(BEGIF(BEGINCOUNT 0 0 X))
      (ENDCOUNT 0 X)))))
      ((EQUAL (CAR X) (QUOTE IF)) (LIST (DEFN X)))
      ((EQUAL (CAR X) (QUOTE THAN)) NIL)
      (T (CONS (CAR X) (IF (CDR X)))))))
(THEN (LAMBDA (X) (COND ((NULL X) NIL)
      ((MEMBER (QUOTE BEGIN)X) (THEN
      (APPEND(PREBEG X) (CONS(BEGIF(BEGINCOUNT 0 0 X))
      (ENDCOUNT 0 X)))))
      ((EQUAL (CAR X) (QUOTE IF)) (LIST (DEFN X)))
      ((EQUAL (CAR X) (QUOTE ELSE )) NIL)
      ((EQUAL (CAR X) (QUOTE THEN)) NIL)
      (T (CONS (CAR X) (THEN(CDR X)))))))
(PREBEG (LAMBDA (X) (COND ((NULL X) NIL)
      ((EQUAL (CAR X) (QUOTE BEGIN)) NIL)
      (T (CONS (CAR X) (PREBEG (CDR X)))) )))
(BEGINFN (LAMBDA (X) (COND ((NULL X) NIL) (T (VERB X)))))
(ENDCOUNT (LAMBDA (N X) (COND ((NULL X) NIL)
      ((EQUAL (CAR X) (QUOTE BEGIN)) (ENDCOUNT (ADD1 N) (CDR X)))
      ((EQUAL (CAR X) (QUOTE END)) (COND
            ((EQUAL N 1) (CDR X))
            (T (ENDCOUNT (SUB1 N) (CDR X)))))
      (T (ENDCOUNT N (CDR X))) )))
(BEGIF (LAMBDA (X) (COND ((MEMBER (QUOTE IF) X) (DEFN X))
      (T X))))
(SEXY (LAMBDA (X Y Z N) (COND ((EQUAL N 0) NIL)
      ((NULL X) NIL)
      ((MEMBER (CAR X) Z) (SEXY (CDR X) Y Z (SUB1 N)))
      (T (CONS (CARX) (SEXY (CDR X) Y Z (SUB1 N)))) )))
      (UNTOUCH (LAMBDA (X Y Z N) (COND
      ((NULL X) NIL)
      ((NULL Z) NIL)
      ((EQUAL (CAR X) Y) (COND ((EQUAL N 0) NIL)
      (T (CONS (CAR Z) (UNTOUCH X Y (CDR Z) SUB1 N))))))
```

2. Application of A-Language:

(a) Letter-Sequence Prediction Listing Written in A-Language-Input

```
FUNCTION EVALQUOTE HAS BEEN ENTERED, ARGUMENTS..
CSET
```

```
(LISTB ((PUT (LIST A B C D E F G H I J K L M N O P Q R S T U V
     W X Y Z) INTO ALPHABET)
     (PUT 0.5E0 INTO PARM2)
     (PUT 0.66999999E0 INTO PARM3)
     PUT LENGTH OF ALPHABET INTO PARM4
     (PUT PARM4 MS 1 INTO PARM5
     (DEFINE POSITION (OF IN) (0.105E2) (POSITION OF X IN A)
          (IF X EQUALS FIRST OF A THEN 0 ELSE 1 PS POSITION OF
          X IN REST OF A))
     (POSITION OF C IN ALPHABET)
     (DEFINE POSITIONLIST (OF) (0.105E2) (POSITIONLIST OF X)
          (IF X IS EMPTY THEN NIL ELSE CONNECT POSITIONLIST
          OF REST OF X TO POSITION OF FIRST OF X IN ALPHABET))
     (DEFINE FIRSTDIFFERENCE (OF AND) (0.105E2) (FIRSTDIFFERENCE
          OF X AND Y) (IF X IS EMPTY THEN NIL ELSE CONNECT FIRST-
          DIFFERENCE OF REST OF X AND REST OF Y TO FIRST OF X MS
          FIRST OF Y))
     (DEFINE COMPOSE (THIS OF TIMES) ($) (COMPOSE THIS FC OF X
          N TIMES) (IF N EQUALS 0 THEN X ELSE IF X IS EMPTY
          THEN NIL ELSE COMPOSE THIS FC OF (FC X) N MS 1 TIMES))
     (DEFINE DISCARD (LEADING ELEMENTS OF) (6) (DISCARD N LEADING
          ELEMENTS OF X) (FIRSTDIFFERENCE OF BEGIN COMPOSE THIS
          (FUNCTION CDR) OF X N TIMES END AND X))
     (DEFINE MODLIST (OF) (0.105E2) (MODLIST OF X) (IF X IS
          EMPTY THEN NIL ELSE CONNECT MODLIST OF REST OF X TO
          BEGIN REDUCE FIRST OF X MODULO ALPHABETHLENGTH END))
     (DEFINE DIFFLIST (OF AND) (0.105E2) (DIFFLIST OF A AND X
          (IF X IS EMPTY THEN NIL ELSE IF A EQUALS FIRST OF X
          THEN DIFFLIST OF A AND REST OF X ELSE CONNECT DIFFLIST
          OF A AND REST OF X TO FIRST OF X))
     (DEFINE NEXTMEMBER (TH OF) (0.55E1) (N TH NEXTMEMBER OF X)
          (1 PS BEGIN RMDR WHEN LENGTH OF X IS DIVIDED BY N END
          TH NTHMEM1 OF X)) (DEFINE NTHMEM1 (TH OF) (4) (N TH
          NTHMEM1 OF X) (IF X IS EMPTY THEN NIL ELSE IF N EQUALS
          1 THEN FIRST OF X ELSE N MS 1 TH NTHMEM1 OF REST OF X))
     (DEFINE PATTERNCONSTANTS (OF) (7) (PATTERNCONSTANTS OF X)
          (PROG (P L HLENY N M LENL)
          (SETQ P 1)
          (SETQ HLENY (TIMES PARM2 (LENGTH Y)))
          C (SETQ L (MODLIST (DISCARD P Y)))
          (SETQ LENL (TIMES PARM3 (LENGTH L)))
          (SETQ N 1)
          B (SETQ M (MODLIST (DISCARD N L)))
          (COND ((NULL (DIFFLIST 0 M)) (RETURN (CONS P N))))
          (COND ((GREATERP N LENL) (GO A)))
          (SETQ N (ADD1 N)) (GO B)
          A (COND ((GREATERP P HLENY) (RETURN NIL)))
          (SETQ P (ADD1 P))
          (GP C)))
```

```
(DEFINE PERIODTEST; (OF AND) (6) (PERIODTEST1 OF U AND Y) (IF U
          IS EMPTY THEN NIL ELSE RETURNLETTER CORRESPONDING TO
          REDUCE BEGIN BEGIN REST OF U TH NEXTMEMBER OF DISCARD
          FIRST OF U LEADING ELEMENTS OF Y END PS BEGIN 1 PS
          LENGTH OF Y MS FIRST OF U TH NTHMEM1 OF Y END END
          MODULO ALPHABETLENGTH))
(DEFINE PERIODTEST2 (OF) (9) (PERIODTEST2 OF X) (PERIODTEST1 OF
          PATTERNCONSTANTS OF X AND X))
(DEFINE PERIODTEST (OF) (8) (PERIODTEST OF X) (PERIODTEST2 OF
          POSITIONLIST OF X))
(DEFINE MODULO (REDUCE ALPHABETLENGTH) (4) (REDUCE X MODULO
          ALPHABETLENGTH) (IF X IS GREATER THAN PARM5 THEN
          REDUCE X PS PARM4 MODULO ALPHABETLENGTH ELSE X))
(DEFINE RETURNLETTER (CORRESPONDING TO) (3) (RETURNLETTER
          CORRESPONDING TO N) (FIRST OF BEGIN COMPOSE THIS
          (FUNCTION CDR) OF ALPHABET N TIMES END))
(FIRSTDIFFERENCE OF (2 3 4 5) AND (1 2 3 4 5))
(COMPOSE THIS (FUNCTION CDR) OF (1 2 3 4 5) 2 TIMES
(PERIODTEST OF (J I H G F E D C))
(REDUCE 30 MODULO ALPHABETLENGTH)
(STOP)))
```

(b) Letter-Sequence
Prediction Listing Produced from A-Language Input

```
END OF EVALQUOTE, VALUE IS..
(DEFINE (((INTO (LAMBDA (X Y) (CSETQ Y X)))))
DEFINE (((GREATER (LAMBDA (X Y) (GREATERP X Y)))))
DEFINE (((PS (LAMBDA (X Y) (PLUS X Y)))))
DEFINE (((MS (LAMBDA (X Y) (PLUS X (MINUS Y))))))
DEFINE (((* (LAMBDA (X Y) (TIMES X Y)))))
DEFINE (((/ (LAMBDA (X Y) (QUOTIENT X Y)))))
DEFINE (((FIRST (LAMBDA (X) (CAR X)))))
DEFINE (((REST (LAMBDA (X) (CDR X)))))
DEFINE (((EQUALS (LAMBDA (X Y) (EQUAL X Y)))))
DEFINE (((EMPTY (LAMBDA (X) (NULL X))))))
DEFINE (((CONNECT (LAMBDA (X Y) (CONS Y X)))))
DEFINE (((RMDR (LAMBDA (X Y) (REMAINDER X Y)))))
DEFINE (((LENGTH (LAMBDA (X) (COND ((EMPTY X) 0) (T (PS 1
          (LENGTH (REST X)))))))))
INTO ((LIST A B C D E F G H I J K L M N O P Q R S T U V W X Y Z)
          ALPHABET)
INTO ( 0.5E0 PARM2)
INTO ( 0.66999999E0 PARM3)
INTO ((LENGTH ALPHABET PARM4)
INTO ((MS PARM4 1) PARM5)
DEFINE (((POSITION (LAMBDA (X A) (COND ((EQUALS X (FIRST A)) 0)
          (T (PS 1 (POSITION X (REST A))))))))
```

```
POSITION (C ALPHABET)
DEFINE (((POSITIONLIST (LAMBDA (X) (COND ((EMPTY X) NIL) (T
          (CONNECT (POSITIONLIST (REST X)) (POSITION (FIRST X)
          ALPHABET)))))))))
DEFINE (((FIRSTDIFFERENCE (LAMBDA (X Y) (COND ((EMPTY X) NIL)
          (T (CONNECT (FIRSTDIFFERENCE (REST X) (REST Y)) (MS
          (FIRST X) (FIRST Y)))))))))
DEFINE (((COMPOSE (LAMBDA (FC X N) (COND ((EQUALS N 0) X)
          ((EMPTY X) NIL) (T (COMPOSE FC (FC X) (MS N 1)))))))))
DEFINE (((DISCARD (LAMBDA (N X) (FIRSTDIFFERENCE (COMPOSE
          (FUNCTION CDR) X N) X)))))
DEFINE (((MODLIST (LAMBDA (X) (COND ((EMPTY X) NIL) (T (CONNECT
          (MODLIST (REST X)) ((MODULO (FIRST X)))))))))))
DEFINE (((DIFFLIST (LAMBDA (A X) (COND ((EMPTY X) NIL) ((EQUALS
          A (FIRST X)) (DIFFLIST A (REST X))) (T (CONNECT
          (DIFFLIST A (REST X)) (FIRST X)))))))))
DEFINE (((NEXTMEMBER (LAMBDA (N X) (NTHMEM1 (PS 1 (RMDR (LENGTH
          X) N)) X)))))
DEFINE (((NTHMEM1 (LAMBDA (N X) (COND ((EMPTY X) NIL) ((EQUALS
          N 1) (FIRST X)) (T (NTHMEM1 (MS N 1) (REST X)))))))))
DEFINE (((PATTERNCONSTANTS (LAMBDA (X) (PROG (P L HLENY N M LENL)
     (SETQ P 1)
     (SETQ HLENY (TIMES PARM2 (LENGTH Y)))
     C (SETQ L (MODLIST (DISCARD P Y)))
     (SETQ LENL (TIMES PARM3 (LENGTH L)))
     (SETQ N 1)
     B (SETQ M (MODLIST (DISCARD N L)))
     (COND ((NULL (DIFFLIST 0 M)) (RETURN (CONS P N))))
     (COND ((GREATERP N LENL) (GO A)))
     (SETQ N (ADD1 N))
     (GO B)
     A (COND ((GREATERP P HLENY) (RETURN NIL)))
     (SETQ P (ADD1 P))
     (GO C))))))
DEFINE (((PERIODTEST1 (LAMBDA (U Y) (COND ((EMPTY U) NIL) (T
          (RETURNLETTER (MODULO ((PS (NEXTMEMBER (REST U)
          (DISCARD (FIRST U) Y)) (NTHMEM1 (PS 1 (MS (LENGTH Y)
          (FIRST U))) Y)))))))))))
DEFINE (((PERIODTEST2 (LAMBDA (X) (PERIODTEST1 (PATTERNCONSTANTS
          X) X)))))
DEFINE (((PERIODTEST (LAMBDA (X) (PERIODTEST2 (POSITIONLIST X))))))
DEFINE (((MODULO (LAMBDA (X) (COND ((GREATER X PARM5) (MODULO
          (MS X PARM5) (MODULO (MS X PARM4))) ((GREATER 0 X)
          (MODULO (PS X PARM4))) (T X))))))
DEFINE (((RETURNLETTER (LAMBDA (N) (FIRST (COMPOSE (FUNCTION CDR)
          ALPHABET N))))))
FIRSTDIFFERENCE ((2 3 4 5) (1 2 3 4 5))
COMPOSE ((FUNCTION CDR) (1 2 3 4 5) 2)
```

```
PERIODTEST ((J I H G F E D C))
MODULO (30)
STOP)
```

Appendix 5

The LISP Implementation for the PDP-1 Computer

L. Peter Deutsch and Edmund C. Berkeley

TABLE OF CONTENTS

Part I

Part II

Part I

1. Introduction

In October 1963 a system for implementing LISP on the PDP-1 computer was finished by L. Peter Deutsch. This system was further improved in March 1964 by adding:

-- variable length of push-down list;
-- variable quantity of combined storage;
-- optional machine language subroutines;

and is here called Basic PDP-1 LISP. It uses a minimum of some 2000 (decimal) registers out of 4096 registers in a one-core PDP-1 computer; it may use 16,361 registers in a four-core PDP-1 computer.

Basic PDP-1 LISP is presented in considerable detail in this appendix for the following reasons:

-- the structure of a system for programming LISP on any computer is thereby revealed;
-- if changes are to be implemented, they can be easily linked with the existing system.

In a one-core PDP-1 computer with 4096 registers, as many as 4070 registers may be assigned to regular LISP, and only 23 reserved for the read-in routine (namely, from 7751 to 7777, octal).

With the system described here, additional LISP functions can be defined and included in the system and later used when desired. Or if desired, additional functions can be programmed in machine language and these can be inserted compatibly with the system.

Punched tapes for placing this LISP system on the PDP-1 computer are available through DECUS, the Digital Equipment Corporation Users Organization, Maynard, Mass.

In the following, it is assumed that the reader has a fairly good working knowledge of: (1) LISP (which may be obtained from the "LISP 1.5 Programmer's Manual," 1962); (2) the machine language codes for the PDP-1 computer (which may be obtained from the computer manual supplied by Digital Equipment Corporation); and (3) the program assembly language MACRO, in which the sym-

bolic tapes are written (a description may be obtained in two manuals published by Digital Equipment Corporation).

2. Functions and Properties included in Basic PDP-1 LISP

The functions and properties included in Basic PDP-1 LISP are shown in Table 1. These functions and properties together constitute a basic subset of the functions and properties of the LISP interpreter for the IBM 7090, as stated in the LISP 1.5 Programmer's Manual.

In order to obtain other LISP functions and properties as may be desired for any particular purpose, see Sections 4 and 5 below.

Table 1

FUNCTIONS AND PROPERTIES OF BASIC PDP-1 LISP

A. Functions Identical with the Corresponding IBM 7090 LISP Functions

ATOM	LIST	PROG
CAR	LOGAND	QUOTE
CDR	LOGOR	READ
COND	MINUS	RETURN
CONS	NULL	RPLACA
EVAL	NUMBERP	RPLACD
GENSYM	PLUS	SASSOC
GO	PRINT	SETQ
		TERPRI

B. Functions Somewhat Different from the Corresponding 7090 Functions

EQ	This works both on atoms and on numbers
GREATERP	This tests for X greater than Y, not for X greater than or equal to Y.
STOP	This is equivalent to PAUSE in 7090 LISP. It takes a numerical argument which appears in the accumulator when the computer halts.
PRIN1 X	This prints the atom X without the extra space at the end. Its value is NIL.

C. Functions Which Have No Analog in 7090 Functions

XEQ — This provides for putting into storage a named machine language subroutine, which can be referred to and used by the PDP-1 LISP interpreter. It also provides for executing single specified machine language instructions.

The SUBR (XEQ C A I) executes the machine language instruction C, with A in the accumulator and I in the in-out register; and returns a value in the form of (a i P) where a is the new value of the accumulator after execution, i is the new value of the in-out register after execution, and P is T if the instruction skipped, and NIL if the instruction did not skip.

LOC X — This gives the machine register in which the atom or list X begins; its value is the location.

Of the foregoing functions, COND, LIST, PROG, SETQ, PLUS, TIMES, LOGAND, LOGOR, and QUOTE are FSUBRs and the remainder are SUBRs.

D. The following special form is available and is identical with the corresponding form in 7090 LISP:

LAMBDA

E. The following permanent objects exist in the Basic PDP-1 LISP system:

OBLIST	the current list of atomic symbols
NIL	F has been replaced by NIL
T	
EXPR	
SUBR	
FEXPR	
FSUBR	
APVAL	

F. Miscellaneous

The print names of atomic symbols are not part of property lists. A quick examination of listings of the system will show exactly where the print names are.

Doing a CDR of an atom is permissible and will get the atom's property list. Doing a CAR of an atom may very easily wreck the system.

QUOTE should be used in place of 7090 FUNCTION. This may re-

quire a bit of extra care in defining functions with functional arguments.

It is advisable to use PROG to avoid recursion wherever possible, even though it may take more space.

3. Use of these Functions and Suggested Test Sequences

How to use these functions is briefly explained here.

As soon as the basic PDP-1 LISP system is read into the computer, control stops at register 4. Turn up sense switch 5 for typewriter input; press CONTINUE; and the system enters a waiting loop which causes lamps to light in the program counter, looking like 1335. At this point, the LISP system is ready for manual typewriter input. As soon as the operator types, for example:

(CAR (QUOTE (A B C D)))

together with a final space at the end of the last right parenthesis, the computer takes control of the typewriter, impulses a carriage return, and then types out:

A

which of course is the correct answer. Similarly, for the other suggested test sequences in Table 2 below.

Table 2

SUGGESTED TEST SEQUENCES

Input	Response
(CAR (QUOTE (A B C D)))	A
(CDR (QUOTE (A B C D)))	(B C D)
OBLIST	The interpreter will type out a complete list of the atomic symbols stored within it.
(LIST (QUOTE (A B C D)))	((A B C D))

NIL	NIL
(CDR NIL)	(APVAL NIL)
(CAR (QUOTE (T.NIL)))	T
(CONS (ATOM (CDR T)) (LIST (GENSYM) (GENSYM)))	(NIL G00001 G00002)
(COND (EQ T NIL) (STOP 1)) (T (EQ (PLUS 1 1) 2)))	T
(PROG (U) (PRINT NIL) (TERPRI) (PRINT T) (SETQ U T) (RETURN U))	NIL T T
(RPLACD (QUOTE CAAR) (QUOTE (EXPR (LAMBDA (X) (CAR (CAR X))))))	CAAR
(CAAR (QUOTE ((A))))	A
(STOP 2)	Computer stops and puts 2 in the accumulator.
(PRIN1 (QUOTE CAR))	CAR, with no punctuation before or after; the value of PRIN1 is NIL.
(PRINT X)	Prints out the value of X; the value of (PRINT X) is X.
(TERPRI)	Prints a carriage return; the value of (TERPRI) is NIL.
(LOC NIL)	2651; this is the register where the NIL atom starts.
(LOC (QUOTE COND))	2725; this is the register where the COND atom starts.
(LOGAND 6 7 3)	2
(LOGOR 12 3 15)	17

```
(RPLACA (QUOTE (NIL X Y))          ((A B) X Y)
    (QUOTE (A B)))
```

Suppose the computer contains DDT — DDT is short for "Digital Equipment Corp. Debugging Tape"; its starting register is 6000, and in one of its customary forms it uses registers 5540 to 7750. Then, if the highest storage register of LISP is below 5540, the instruction:

```
(XEQ 606000 0 0)
```

transfers control to DDT, and puts zero in the accumulator and in the in-out register.

If there is the following subroutine stored in the computer:

```
5500    dzm 5507
5501    idx 5507
5502    lac 5507
5503    dpy'
5504    sma
5505    jmp 5501
5506    jmp 2241
5507    (being used for storage)
```

and LISP is below 5500, then:

```
(XEQ 605500 0 0)
```

Will cause a horizontal line to be drawn on the scope from the origin to the x-axis positive limit, and then control will be returned to LISP. NIL will be typed out. 2241 is the register called "prx" in the macro symbolic.

4. Auxiliary Functions Which May Be Defined with LISP Expressions

Any of the functions listed below in Table 3 can be put into the system at will, as follows: Prepare a punched tape listing of it. Insert tape into the reader. Turn on the reader. Turn down Sense Switch 5. Thereupon the computer will read in the

tape. The typewriter, when the reading in is accomplished, will type back the name of the inserted function.

Many other functions besides those listed in Table 3 may be inserted.

Table 3

AUXILIARY LISP FUNCTIONS

ABSOLUTE VALUE

```
(RPLACD (QUOTE ABSVAL) (QUOTE (EXPR (LAMBDA (X) (COND ((GREATERP
    0 X) (MINUS X)) (T X))))))
```

AND

```
(RPLACD (QUOTE AND) (QUOTE (FEXPR (LAMBDA (X A) (PROG NIL N (COND
    ((NULL X) (RETURN T)) ((NULL (EVAL (CAR X) A)) (RETURN NIL)))
    (SETQ X (CDR X)) (GO N))))))
```

ASSOC

```
(RPLACD (QUOTE ASSOC) (QUOTE (EXPR (LAMBDA (X Y) (COND ((EQUAL
    (CAAR Y) X) (CAR Y)) (T (ASSOC X (CDR Y)))))))))
```

CAAR

```
(RPLACD (QUOTE CAAR) (QUOTE (EXPR (LAMBDA (X) (CAR (CAR X))))))
```

CADR

```
(RPLACD (QUOTE CADR) (QUOTE (EXPR (LAMBDA (X) (CAR (CDR X))))))
```

CDAR

```
(RPLACD (QUOTE CDAR) (QUOTE (EXPR (LAMBDA (X) (CDR (CAR X))))))
```

CDDR

```
(RPLACD (QUOTE CDDR) (QUOTE (EXPR (LAMBDA (X) (CDR (CDR X))))))
```

CSET

```
(RPLACD (QUOTE CSET) (QUOTE (EXPR (LAMBDA (X Y) (RPLACD X (LIST
    (QUOTE APVAL) Y))))))
```

CSETQ

```
(RPLACD (QUOTE CSETQ) (QUOTE (FEXPR (LAMBDA (X A) (CSET (CAR X)
     (EVAL (CADR X) A))))))
```

DEX

```
(RPLACD (QUOTE DEX) (QUOTE (FEXPR (LAMBDA (X A) (RPLACD (CAR X)
     (CONS (QUOTE EXPR) (CDR X)))))))
```

DFX

```
(RPLACD (QUOTE DFX) (QUOTE (FEXPR (LAMBDA (X A) (RPLACD (CAR X)
     (CONS (QUOTE FEXPR) (CDR X)))))))
```

DIFFLIST

```
(RPLACD (QUOTE DIFFLIST) (QUOTE (EXPR (LAMBDA (A X) (COND ((NULL
     X) NIL)  ((EQUAL A (CAR X)) (DIFFLIST A (CDR X))) (T (CONS
     (CAR X) (DIFFLIST A (CDR X)))))))))
```

DOUBLE

```
(RPLACD (QUOTE DOUBLE) (QUOTE (EXPR (LAMBDA (X) (PLUS X X)))))
```

EQUAL

```
(RPLACD (QUOTE EQUAL) (QUOTE (EXPR (LAMBDA (X Y) (COND ((ATOM X)
     (EQ X Y)) ((ATOM Y) NIL) ((EQUAL (CAR X) (CAR Y)) (EQUAL
     (CDR X) (CDR Y))) (T NIL))))))
```

GREATEST COMMON DIVISOR

```
(RPLACD (QUOTE GCD) (QUOTE (EXPR (LAMBDA (X Y) (COND ((GREATERP
     X Y) (GCD Y X)) ((ZEROP (REM Y X)) X) (T (GCD (REM Y X)
     X)))))))
```

LAST

```
(RPLACD (QUOTE LAST) (QUOTE (EXPR (LAMBDA (L) (COND ((NULL L)
     NIL) ((NULL (CDR L)) (CAR L)) (T (LAST (CDR L))))))))
```

LENGTH using Program Feature

```
(RPLACD (QUOTE LENGTH) (QUOTE (EXPR (LAMBDA (L) (PROG (U V) (SETQ
     V 0) (SETQ U L) A (COND ((NULL U) (RETURN V))) (SETQ U
     (CDR U)) (SETQ V (PLUS 1 V)) (GO A))))))
```

LENGTH using Recursion

```
(RPLACD (QUOTE LENGTHR) (QUOTE (EXPR (LAMBDA (L) (COND ((NULL L)
     0) (T (PLUS 1 (LENGTHR (CDR L))))))))))
```

MAPLIST using Recursion

```
(RPLACD (QUOTE MAPLIST) (QUOTE (EXPR (LAMBDA (X A) (COND  ((NULL X)
     NIL) (T (CONS (A X) (MAPLIST (CDR X) A))))))))
```

MAPLIST using Program Feature

```
(RPLACD (QUOTE MAPLIST) (QUOTE (FEXPR (LAMBDA (X A) (PROG (V M R)
```

```
(SETQ R (SETQ M (LIST (EVAL (CADR X) A)))) (SETQ V (EVAL
(CAR X) A)) P (COND ((NULL V) (RETURN (CDR R)))) (SETQ M
(CDR (RPLACD M (LIST (EVAL (LIST (CAR R) (LIST (QUOTE QUOTE)
V)) A))))) (SETQ V (CDR V)) (GO P))))))
```

MEMBER

```
(RPLACD (QUOTE MEMBER) (QUOTE (EXPR (LAMBDA (A X) (COND ((NULL X)
    NIL) ((EQ A (CAR X)) T) (T (MEMBER A (CDR X)))))))
```

MINIMUM

```
(RPLACD (QUOTE MIN) (QUOTE (EXPR (LAMBDA (L) (COND ((NULL L) NIL)
    ((NULL (CDR L)) (CAR L)) (T (SMALLER (CAR L) (MIN (CDR L
    ))))))))
```

NOT

```
(RPLACD (QUOTE NOT) (QUOTE (EXPR NULL)))
```

OR

```
(RPLACD (QUOTE OR) (QUOTE (FEXPR (LAMBDA (X A) (PROG NIL N (COND
    ((NULL X) (RETURN NIL)) ((EVAL (CAR X) A) (RETURN T)))
    (SETQ X (CDR X)) (GO N))))))
```

PAIR

```
(RPLACD (QUOTE PAIR) (QUOTE (EXPR (LAMBDA (X Y) (PROG (U V M)
    (SETQ U X) (SETQ V Y) (SETQ M NIL) K (COND ((NULL U) (COND
    ((NULL V) (RETURN M))))) (SETQ M (CONS (CONS (CAR U) (CAR V))
    M)) (SETQ U  (CDR U)) (SETQ V (CDR V)) (GO K))))))
```

PAIRLIS

```
(RPLACD (QUOTE PAIRLIS) (QUOTE (EXPR (LAMBDA (X Y A) (COND ((NULL
    X) A) (T (CONS (CONS (CAR X) (CAR Y)) (PAIRLIS (CDR X)
    (CDR Y) A)))))))
```

PDEF (Print and Punch Definition)

```
(RPLACD (QUOTE PDEF) (QUOTE (FEXPR (LAMBDA (X A) (LIST (QUOTE
    RPLACD) (LIST (QUOTE QUOTE (CAR X)) (LIST (QUOTE QUOTE) (CDR
    (CAR X)))))))))
```

QUOTIENT using Program Feature

```
(RPLACD (QUOTE QUOTIENT) (QUOTE (EXPR (LAMBDA (Q D) (PROG (U V)
    (SETQ V 0) (SETQ U Q) A (COND ((GREATERP D U) (RETURN V)))
    (SETQ U (PLUS U (MINUS D))) (SETQ V (PLUS 1 V)) (GO A))))))
```

QUOTIENT using Recursion

```
(RPLACD (QUOTE QUOTIENTR) (QUOTE (EXPR (LAMBDA (Y X) (COND ((
    GREATERP X Y) 0) ((EQ X Y) 1) ((GREATERP Y X) (PLUS 1
    (QUOTIENTR (PLUS Y (MINUS X)) X)))))))
```

REMAINDER

```
(RPLACD (QUOTE REM) (QUOTE (EXPR (LAMBDA (Y X) (COND ((EQUAL Y
    X) 0) ((GREATERP X Y) Y) (T (REM (PLUS Y (MINUS X)) X)))))))
```

REVERSE (Defined Recursively with Auxiliary Function)

```
(RPLACD (QUOTE R1) (QUOTE (EXPR (LAMBDA (M L) (COND ((NULL L) M)
    (T (R1 (CONS (CAR L) M) (CDR L))))))))
(RPLACD (QUOTE REVERSE) (QUOTE (EXPR (LAMBDA (L) (R1 NIL L)))))
```

REVERSE using Program Feature

```
(RPLACD (QUOTE REVERSE) (QUOTE (EXPR (LAMBDA (M) (PROG (U V)
    (SETQ U M) K (COND ((NULL U) (RETURN V))) (SETQ V (CONS
    (CAR U) V)) (SETQ U (CDR U)) (GO K))))))
```

SEQUENCE

```
(RPLACD (QUOTE SEQUENCE) (QUOTE (EXPR (LAMBDA (L) (PROG (U V W)
    (SETQ U L) (SETQ V (MIN L)) (SETQ W NIL) A (COND ((NULL U)
    (RETURN W))) (SETQ V (MIN U)) (SETQ U (DIFFLIST V U))
    (SETQ W (APPEND W (LIST V))) (GO A))))))
```

SMALLER

```
(RPLACD (QUOTE SMALLER) (QUOTE (EXPR (LAMBDA (X Y) (COND
    ((GREATERP X Y) Y) (T X))))))
```

SUB2

```
(RPLACD (QUOTE SUB2) (QUOTE (EXPR (LAMBDA (A Z) (COND ((NULL A)
    Z) ((EQ (CAAR A) Z) (CDAR A)) (T (SUB2 (CDR A) Z)))))))
```

SUBLIS

```
(RPLACD (QUOTE SUBLIS) (QUOTE (EXPR (LAMBDA (A Y) (COND ((ATOM Y)
    (SUB2 A Y)) (T (CONS (SUBLIS A (CAR Y)) (SUBLIS A (CDR Y)))))
    ))))
```

SUBST

```
(RPLACD (QUOTE SUBST) (QUOTE (EXPR (LAMBDA (X Y Z) (COND ((EQUAL
    Y Z) X) ((ATOM Z) Z) (T (CONS (SUBST X Y (CAR Z)) (SUBST X Y
    (CDR Z)))))))))
```

TIMES using Recursion

```
(RPLACD (QUOTE TIMES) (QUOTE (EXPR (LAMBDA (N M) (COND ((EQUAL
    N 1) M) (T (PLUS M (TIMES M (PLUS N (MINUS 1))))))))))
```

TIMES using Program Feature

```
(RPLACD (QUOTE TIMES) (QUOTE (EXPR (LAMBDA (X N) (PROG (U V)
   (SETQ V 0) (SETQ U 0) A (COND ((EQ V N) (RETURN U))) (SETQ U
     (PLUS X U)) (SETQ V (PLUS V 1)) (GO A))))))
```

UNION

```
(RPLACD (QUOTE UNION) (QUOTE (EXPR (LAMBDA (X Y) (COND ((NULL X)
     Y) ((MEMBER (CAR X) Y) (UNION (CDR X) Y)) (T (CONS (CAR X)
     (UNION (CDR X) Y))))))))
```

ZEROP

```
(RPLACD (QUOTE ZEROP) (QUOTE (EXPR (LAMBDA (X) (COND ((EQUAL X
     0) T) (T NIL))))))
```

5. Some Additional Functions for Basic PDP-1 LISP

In order to remove symbols from the OBLIST, and reuse the storage capacity that they previously occupied, we use:

```
(RPLACD (QUOTE XSY) (QUOTE (EXPR (LAMBDA (X) (PROG (Y) (SETQ Y
     OBLIST) A (COND ((NULL (CDR Y)) (RETURN NIL)) ((EQ X (CAR
     (CDR Y))) (RETURN (RPLACD Y (CDR (CDR Y)))))) (SETQ Y (CDR Y))
     (GO A))))))
```

```
(RPLACD (QUOTE REMOVE) (QUOTE (FEXPR (LAMBDA (X Y) (PROG NIL A
     (COND ((NULL X) (RETURN OBLIST))) (XSY (CAR X)) (SETQ X
     (CDR X)) (GO A))))))
```

XSY stands for "expunge symbol".

REMOVE is used as follows: Suppose we have a case where the OBLIST starts for example with G F OBLITT Y X ATOM CAR CDR COND CONS ..... and we wish to delete F OBLITT Y. We put in: (REMOVE OBLITT F Y), and the computer response is:

```
G X ATOM CAR CDR CONS .....
```

In this way, both accidentally mistyped expressions and symbols no longer needed in the LISP system can be removed from storage, and from any recollection within the LISP system. (Note: REMOVE will not operate on the first expression in the OBLIST, but only on the second and later expressions.)

In order to put in machine-language subroutines, outside of the storage used by LISP, name them, use them, and return from them to LISP, we use:

```
(RPLACD (QUOTE DEPOSIT) (QUOTE (EXPR (LAMBDA (X Y) (PROG NIL A
     (COND ((NULL X) (RETURN Y))) (XEQ (PLUS 240000 Y) (CAR X) 0)
     (SETQ X (CDR X)) (SETQ Y (PLUS 1 Y)) (GO A))))))

(RPLACD (QUOTE PUTSUBR) (QUOTE (EXPR (LAMBDA (N X Y) (PROG NIL
     (RPLACD N (LIST (QUOTE SUBR) (PLUS 160000 Y))) (RETURN
     (DEPOSIT X Y)))))))

(RPLACD (QUOTE DEFSUBR) (QUOTE (EXPR (LAMBDA (N X)
     (RPLACD N (LIST (QUOTE SUBR) (PLUS 160000 X)))))))
```

The EXPR (DEPOSIT X A) deposits the list of numbers X starting at location A; its value is the first register beyond the list.

The EXPR (PUTSUBR N X A) performs (DEPOSIT X A), and then sets up N (name) as a SUBR starting at A.

An example (if LISP storage stops at 5477) is:

```
(PUTSUBR (QUOTE SHOWLINE) (LIST 345507 445507 205507 730007
     640400 605501 602241) 5500)
```

This inserts the line-display program mentioned above into the computer starting at register 5500 and makes it accessible to LISP with the name SHOWLINE.

The EXPR (DEFSUBR N X) accepts an existing, inserted, machine-language subroutine starting at register X, gives it the name N, and makes it accessible to LISP with the name N. For example, the line-display program mentioned above, if already in the computer, could be named and called with:

```
(DEFSUBR (QUOTE SHOWLINE) 5500)
```

The last command in the subroutine, instead of 602241, should be either 600004, if LISP is to return to the starting address 4, or 600005, if LISP is to <u>continue</u> to the waiting loop.

If the A-LIST is wanted, establish GETALIST with:

```
(RPLACD (QUOTE GETALIST) (QUOTE (FEXPR (LAMBDA (X Y) Y))))
```

and then use:

```
(PRINT (GETALIST))
```

6. Input and Output

Input comes from the typewriter if sense switch 5 is up and from the tape reader otherwise. Output is normally on the typewriter; however, SS 3 up causes punching (with correct parity) and SS 6 up independently suppresses typeout.

Each S-expression typed in will be evaluated and its value printed out. Unlike 7090 LISP, arguments of functions are also evaluated on the top level; for example, to evaluate

cons [A;B]

it is necessary to write

(CONS (QUOTE A) (QUOTE B))

In preparing input:

Tab, space, and comma are equivalent;
Carriage return is ignored:
Backspace causes deletion of everything typed since the last control character (parenthesis, space/tab/comma, or period);
An extra space must be typed to terminate the entire expression;
Upper and lower case shifts will be noted but not necessarily inserted into the symbol at that point (for example, the sequence u.c., l.c., u.c., A, space, produces a symbol with print name u.c., A, l.c.);
Alphabetic characters should regularly and generally be in lower case; and basic functions, (such as CAR, CDR, ...) contrary to their representation throughout this report, are in PDP-1 LISP actually stored in lower-case symbols (such as car, cdr ....); and then taken in to the system and put out by the system as lower-case symbols;
It is very advisable to stick to "printout" format for all input since the READ routine is not guaranteed to work on any other form, although it may;
Hyphen, "-", is a letter and does not negate a following number;
All numbers are octal integers; to input the number -1 it is necessary to type 777776;
There is no limit on the length of a print name;
The character overbar "‾" or vertical bar "|" will cause the next character to be inserted in the print name and considered a letter, regardless of what it actually is (the "‾" or "|" itself does not appear

in the print name): thus atoms may be generated for output formatting purposes with names such as "tab" or "space".

In producing the output:

A carriage return is automatically generated after any 100(octal) characters not containing a carriage return;

Unlike the 7090 LISP output, no spaces are provided before and after the "." of concatenation (since there are no floating-point numbers to be concerned with).

7. Operation of the System

First, zero core, to avoid unnecessary difficulties.

Second, put the binary tape in the reader, and press READIN. Do nothing until the tape stops. Almost all of the tape will read in; and the machine will come to a halt. If you wish 7701 to be the highest register of free storage, and 300 to be the length of the push-down list, press READIN once more. The machine will stop at address 4. Turn up Sense Switch 5 (to control from the typewriter). Press CONTINUE.

If you wish to select the highest register of free storage, when the machine stops for the first time, with memory address at 0004, put the number of the highest register of free storage (recommended, 5000 to 7750; possible but not recommended, 4000 to 4777) in the Test Word switches and press CONTINUE. Then put the length of the push down list (suggested 200 to 400) in the Test Word switches, and press CONTINUE. The machine will go to address 4. Turn up Sense Switch 5, and press CONTINUE. The LISP system should be ready for use.

If the tape stops at an improper place, pull the tape back a block, check for missing holes, and CONTINUE. When the tape stops at 4, CONTINUEing begins the READ-EVAL-PRINT cycle. STARTing at 4 at any time and CONTINUEing is safe; indeed, it is the only way to annull most typing errors.

If the system "drops dead", the normal recourse is to start over.

Following is the assignment of the sense switches and the program flags:

SS	1	Idiot trace
	2	-
	3	Punch out
	4	-
	5	Type in
	6	No typeout
PF	1	Used for type-in
	2	Zero suppress in octal print
	3	-
	4	-
	5	Letter in symbol
	6	Off in error printout

8. Error Diagnostics

Error halts cause identification of the error and typing of the error code in red on the typewriter, regardless of the settings of Sense Switches 3 and 6; an error usually sends the system to address 4. The list of error indications follows:

icd	Illegal COND; returns value NIL and continues.
uss	Unbound symbol in SETQ; returns NIL and continues.
tma	Too many arguments for a SUBR (more than 3); ignores extra arguments and proceeds.
uas	Unbound atomic symbol (followed by the form currently being evaluated).
ilp	Illegal parity; halts with character in accumulator. CONTINUE ignores character, but SS 5 may be turned up, and typing used to provide a replacement if desired.
lts	LAMBDA variable list too short.
ats	Argument list (paired with LAMBDA list) too short.
sce	Storage capacity exceeded. CONTINUEing is not advisable, as it will probably call the same error again in short order, unless one promptly deletes several atoms having lengthy definitions from the OBLIST.
pce	Pushdown capacity exceeded.
nna	Non-numeric argument for arithmetic, followed by the argument in question; returns value zero and proceeds.
ana	Argument not atom (for PRIN1); returns NIL as usual and proceeds.
ovf	Division overflow; returns zero and proceeds.

9. Some Remarks

In general, each character in each LISP expression is recognized by the computer as 2 octal digits called concise code. The pairs of octal digits are packed 3 pairs at a time into the 6-octal-digit registers of the PDP-1. If a LISP atom has a number of characters which is not a multiple of three, there will be spaces left over, which are filled arbitrarily with a filler character, 76 (octal). For example, a LISP word with 7 characters such as SMALLER will be packed into three computer registers, S M A in one, L L E in a second, and R along with two filler characters in the third.

These three registers are linked by list structure. An example of a hypothetical list structure which might store SMALLER if introduced as a defined function into the LISP system would be as shown in Table 4:

Table 4

PDP-1 Register	Contents	Meaning	Comments
5763	405765	pointer to 5765	5765 is the start of the print name of the atom SMALLER
5764	005773	pointer to property list	5773 is the start of the property list
5765	224461	S M A	Concise code
5766	005767	pointer	5767 holds continuation of the list
5767	434365	L L E	Concise code
5770	005771	pointer	5771 holds continuation of the list
5771	767651	- - R	Concise code and 2 filler characters
5772	003011	nil	Terminator of list

If SMALLER were defined by the expression:

```
(RPLACD (QUOTE SMALLER) (QUOTE (EXPR (LAMBDA (X Y)
 0      1               1  1     2    3      4 4

     (COND ((GREATERP Y X) X) (T Y))))))
      4    5 6             6  5 5  5 4 3 2 1 0
```

then the property list of SMALLER would be (hypothetically) as shown in Table 5:

Table 5

Register	Contents	Meaning
5773	003271	"EXPR"
5774	005775	pointer
5775	005777	pointer
5776	002651	"NIL"
5777	003255	"LAMBDA"
6000	006001	pointer to forking
6001	006003	pointer to (X Y)
6002	006007	pointer to (COND ....
6003	007701	"X"
6004	006005	pointer
6005	007711	"Y"
6006	002651	"NIL"
6007	002725	"COND"
	etc.	

An accepted LISP expression L is identified within the machine by the address of the list structure in storage which represents L.

The computer evaluates expressions using either machine subroutines (SUBRs and FSUBRs) or LISP subroutines (EXPRs or FEXPRs).

The computer converts the resulting value into concise codes, and presents the value for output to the computer-associated typewriter or the punch.

Basic PDP-1 LISP is very flexible:

1. The number of registers on the push-down list can be reasonably varied between 200 and 400 octal. The number chosen can vary according to the amount of recursion it is desired to provide for.

2. The number of registers of storage (there is only one kind of storage) can be varied from under 1000 octal to over 4000 octal in a one-core machine. In the smallest extreme case, LISP system can occupy only the registers up to about 4000 octal; in the other extreme case LISP can occupy all the registers up to 7750 octal, leaving 7751 to 7777 for the read-in subroutine.

3. Machine subroutines may be located in core, and referred to and used. These machine subroutines should be located above the highest register in free storage.

4. DDT (the Digital Debugging Tape) may be loaded in registers 5500 up and LISP may be loaded below, so that the facilities of DDT are available for modifying LISP.

5. A core dump routine may be loaded into 400 (octal) registers above free storage and used upon LISP.

Part II

1. Macro Symbolic Program for Basic PDP-1 LISP

```
lisp 3-23-64 : 1 field

define                  extend
       termin

define                  lload A,B
       law B
       dac A
       termin

define                  init A,B
       law B
       dap A
       termin

define                  index A,B,C
       idx A
       sas B
       jmp C
       termin

define                  step A,B,C
       index A,(B,C
       termin

define                  setup A,B
       law i B
       dac A
       termin

define                  exit
       jmp R
       termin

define                  move A,B
       lac A
       dac B
       termin

define                  load A,B
       move (B,A
       termin

define                  count A,B
       isp A
       jmp B
       termin

define                  test K,P
       sad (K
       jmp P
       termin

define                  undex A
       law i 1
       add A
       dac A
       termin

define                  swap
       rcl 9s
       rcl 9s
       termin

       smi=spi i
       szm=sza sma-szf
       spq=szm i
       xy=0
       xx=hlt
       clo=spa sma szo i-szf-szf
       mul=540000
       div=560000

start
```

```
Lisp interpreter 3-20-64, part 1

4/

go,         hlt+cla+cli+7-opr-opr
            stf 6
            extend
            dzm 77
            law 77
            dap avx

beg,        law pdo-1
            dac pdl
            lac n
            dac ar2
            cal rin
            cal evo
            cal pnt
            jmp beg

t0,         0
t1,         0
g0,         0
g1,         0
hi,         0
csi,        72
cso,        72
ffi,        0
ga1,        0
0           isi,        isi-1
gst,        repeat 5,20
a0,         0
a1,         0
a2,         0

/append word to pdl

pwl,        0
            dap pwx
            idx pdl
            sad bfw
            jmp qg2
            lac pwl
            dac i pdl

pwx,        exit

/retrieve word from pdl

uw,         0
uwl,        dap uwx
            lio i pdl
            undex pdl

uwx,        exit

buf,
77/         0
            0
            dap rx
            sub (1
            dap .+1
            lac xy
            dap ave+1
            lac rx
            jda pwl
ave,        lac 100
            exit

/create number

crn,        lio (jmp
            rcl 2s
            rar 2s
            dac 100
            jmp cpf

/print or punch character

pc,         and (77
            sad (76
            jmp x
            ior (ral
            dac pcc
            sad (ral 77
            jmp pcc-3
            isp pch
            jmp pcc-1
            law 277
            cal out
            law i 100
            dac pch
            law 252

pcc,        xx
            and (200
            ior pcc
            dac 100
            stf 2
            jmp out

pch,        -100

/get numeric value

vag,        lio i 100
            cla
            rcl 2s
            sas (3
            jmp qi3
            idx 100
```

```
          lac i 100
          rcl 8s
          rcl 8s
          jmp x

/get two values

vad,      dio a1
          cal vag
          dac a0
          lac a1
          cal vag
          dac a1
          jmp x

 /pack character onto end of buffer

 oc,      rar 6s
          lio i isi
          rcl 6s
          sad (76
          jmp oc1
          lac 100
          ior (767600
          cal cf
          lio t0
          idx t0
          idx isi
          dac a1
          dio isi
          lac i a1
          dac i t0
          dio i a1
          jmp x

oc1,      dio i isi
          jmp x

 /output routine

out,      lio 100
          szs 36
          ppa
          szs i 66
          tyo
          jmp x
 /error printout

 err,     clf 6
          dap erx
          lac i erx
          dac ern
          law erm
          cal pra
          stf 6
          idx erx

 erx,     exit

 erm,     357776
          .+1
 ern,     0
          .+1
          347776
 n,fro,   nil

 define   error F
          jsp err
          F
          termin


 /garbage collector, non-compacting

 gc,      dap gcx
          dio ga1
          dio gfr
          lac gfr
          sar 2s
          sza
          jsp gfr+1
          lac ffi
          sza i
          jmp gco
          lac 100
          jda gfr
 gco,     lac i 1ob
          jda gfr
          lac isi
          sas (isi-1
          jmp gc1
          law pdl+1
          dac g1

 gcp,     lac i g1
          jda gfr
          idx g1
          sub (1
          sad pdl
          jmp g2e
          jmp gcp

 /mark one list

 gfr,     0
          dap gfx
          lac gfr
          ral 1s
          spq
          jmp gfx
          lac pdl
          jda pwl

 gfn,     lio i gfr
          idx gfr
          lac i gfr
          spa
          jmp gfu
          ior (add
          dac i gfr
          spi
          jmp gfd
          jda pwl
```

```
        dio gfr
        jmp gfn
gfd,    ril 1s
        spi i
        jmp gfa

gfu,    jsp uwl
        dio gfr
        sas gfr
        jmp gfn
gfx,    exit

gfa,    rir 1s
        dio g0
        dac gfr
gfl,    idx g0
        lac i g0
        spa
        jmp gfn
        ior (add
        dac i g0
        dac g0
        xor (add
        sas n
        jmp gfl
        jmp gfn

/garbage collector, linear sweep phase

g2e,    lac fro
        dac g0

g2n,    idx g0
        lio i g0
        smi
        jmp g2f
        ril 1s
        sir 1s

g2a,    dio i g0
        idx g0
        sas hi
        jmp g2n

g2x,    lio ga1

gcx,    exit

g2f,    lio fre
        sub (1
        dac fre
        jmp g2a

gci,    sad n
        jmp gcp-2
        dac gfr
        dac g0
        lac pdl
        jda pwl
        law gcp-2
        dap gfx
        jmp gfl

/SASSOC

aso,    cal asc
        jmp ase
        jmp x
ase,    lac a2
        cal cns-1
        jmp evo

asr,    lio ar2
asc,    dio a1
        lac a1

as1,    sad n
        jmp x
        lac i a1
        dac t0
        lac i t0
        sad 100
        jmp as2
        idx a1
        lac i a1
        dac a1
        jmp as1

as2,    idx i pdl
        lac t0
        jmp x

/program feature

/PROG

pgm,    lac pa3
        jda pwl
        lac pa4
        jda pwl
        dzm pa4
        dio ar2
        lio i 100
        idx 100
        lac i 100
        dac pa3
        dio ar1

/append program variables

        lac ar1

pg5,    sad n
        jmp pg6
        lac i ar1
        cal cns-1
        lio ar2
        cal cns
        dac ar2
        idx ar1
        lac i ar1
        dac ar1
        jmp pg5
```

```
/expand go-list (on a-list)

pg6,    lac pa3

pg7,    dac ar1
        sad n
        jmp pg0
        lac i ar1
        cal car
        sma
        jmp pg9
        lac ar1
        lio ar2
        cal cns
        dac ar2

pg9,    idx ar1
        lac i ar1
        jmp pg7

/process program

pg0,    lac pa3

pg1,    sad n
        jmp pg2
        lac i pa3
        cal car
        spa
        jmp pg3
        lac ar2
        jda pwl
        lac 100
        cal evo
        jsp uwl
        dio ar2
        cla
        sas pa4
        jmp pg4

pg3,    idx pa3
        lac i pa3
        dac pa3
        jmp pg1

/terminate program

pg4,    lac pa4
pg2,    jda uw
        dio pa4
        jsp uwl
        dio pa3
        lac uw
        jmp x

/RETURN

ret,    dac pa4
        jmp x

/GO

goe,    lio 100
        lac n
        cal cns
        dac pa3
        jmp prx

/SETQ

stq,    dac ar1
        dio t1
        lac i ar1
        cal asc
        jmp qa4
        jda pwl
        lac ar1
        cal cdr
        cal car
        lio t1
        cal evl
        jda uw
        dio t0
        idx t0
        lac uw
        dac i t0
        jmp x

/CDR

cdr,    idx 100

/CAR

car,    lac i 100

x,      jda uw
        dio rx
        lac uw
rx,     exit

/ATOM

atm,    lac i 100
        sma
        jmp fal

tru,    lac tr
        jmp x

/NULL

nul,    lio n

/EQ

eqq,    dio a1
        sad a1
        jmp tru
        lac i a1
        and i 100
        and (jmp
        sas (jmp
        jmp fal
```

```
        lac 100
        cal vad
        sad a0
        jmp tru
        jmp fal

/RPLACD

rdc,    idx 100
        sub (1

/RPLACA

rda,    dio i 100
        jmp x

/create atom

mka,    ior (add
        dac 100
        lio n

/CONS

cns,    idx ffi

cnc,    lac fre
        sad n
        jmp gcs

cna,    dac t0
        lac 100
        dac i fre
        idx fre
        lac i fre
        dio i fre
        dac fre
        lac t0
        jmp x

/PLUS

pls,    cal elc
        lio (add a0
plz,    dzm a0
pl1,    dio plo
pl2,    sad n
        jmp ple
        dac a1
        lac i a1
        cal vag
plo,    0
        dac a0
        lac a1
        cal cdr
        jmp pl2
ple,    lac a0
        jmp crn

/LOGAND, LOGOR, TIMES

lga,    cal elc
        lio (-0
        dio a0
        lio (and a0
        jmp pl1

lgo,    cal elc
        lio (ior a0
        jmp plz

tim,    cal elc
        lio (1
        dio a0
        lio (jmp tic
        jmp pl1

tic,    mul a0
        scr 1s
        dio 100
        add 100
        jmp plo+1

gcs,    jsp gc
        lac fre
        sas n
        jmp cna
        jmp qg1

/TERPRI

tpr,    law 77
        cal pc
        jmp prx

/PRIN1

pr1,    lac i 100
        sma
        jmp qp1
        sub (lac
        spa
        jmp prn
        and (-jmp

pra,    sad n
        jmp x
        dac a0
        lac i a0
        ral 6s
        cal pc
        lac i a0
        rar 6s
        cal pc
        lac i a0
        cal pc
        idx a0
        lac i a0
        jmp pra
```

```
prn,    lac 100
        cal vag
        dac t1
        clf 2
        setup t0,6

prv,    lio t1
        sad (-1
        stf 2
        cla
        rcl 3s
        dio t1
        sza i
        law 20
        sad (20
        szf 2
        cal pc
        isp t0
        jmp prv
        jmp prx

/NUMBERP

nmp,    lac i 100
        and (jmp
        sad (jmp
        jmp tru
        jmp fal


/do a CONS into full word space

cf,     lio n

cpf,    dzm ff1
        jmp cnc

/MINUS

min,    cal vag
        cma
        jmp crn

/XEQ

xeq,    cal vad
        lac tr
        dac t1
        lac a2
        cal vag
        lio a0
        dio xei
        lac a1
        lio uw
xei,    0
        jmp xen
        dio a2
xer,    cal crn
        dac ar1
        lac a2
        cal crn
        dac ar2
        lac t1
        cal cns-1
        lio ar2
        cal efc
        lio ar1
        dac 100
        jmp efc
xen,    dio a2
        lio n
        dio t1
        jmp xer

/GENSYM

gsm,    law gst
        dac t0

gsi,    idx i t0
        sad (12
        jmp gsn
        sad (21
        law 1
        dac i t0

gsp,    lac gst+2
        ral 6s
        ior gst+1
        ral 6s
        ior gst
        cal cf
        law 6700
        ior gst+4
        ral 6s
        ior gst+3
        lio t0
        cal cpf
        cal mka
        jmp x

gsn,    law 20
        dac i t0
        idx t0
        sas (gst+5
        jmp gsi
        jmp gsp

/QUOTIENT

qot,    cal vad
        lio a0
        cla
        spi
        clc
        rcl 1s
        div a1
        jmp q14
        jmp crn

/COND

cnd,    dio ar2
```

```
cd1,    dac ar1
        sad n
        jmp qa3
        jda pwl
        lac ar2
        jda pwl
        lac i ar1
        cal car
        cal evo
        jda uw
        dio ar2
        jsp uwl
        dio ar1
        lac uw
        sas n
        jmp cdy
        idx ar1
        lac i ar1
        jmp cd1

cdy,    lac i ar1
        cal cdr
        cal car
        jmp evo

/STOP

stp,    cal vag
        hlt+cli-opr
        jmp prx


/GREATERP

grp,    cal vad
        clo
        sub a0
        szo
        lac a1
        sma
        jmp fal
        jmp tru

/get a character

ava,    szs 50
        jmp avi
        cli

avx,    lac 77
        sza i
        jmp avr
        rcl 9s
        dio i avx
        ral 2s
        spq
        jmp ava
        ral 7s
        ior (rar
        dac avc
        law 525

avc,    xx
        sma
        jmp qc3

avt,    law 77
        and avc
        sas (72
        sad (74
        dac csi
        sad csi
        jmp ava
        jmp x

avr,    index avx,ave,avx
        init avx,buf
        dap avs

avn,    rpa
        rcr 9s
        rpa
        rcl 9s

avs,    dio xy
        step avs,dio 100,avn
        jmp ava

avi,    szf i 1
        jmp ava
        tyi
        clf 1
        dio avc
        jmp avt

/terminate print name

mkn,    law 72
        sas cso
        cal oc
        idx isi
        dac t0
        lio n
        dio isi
        lac i t0
        dio i t0
        jmp x

/pack character into print name

pak,    dap pk1
        lac csi
        sad cso
        jmp pk1
        dac cso
        cal oc
pk1,    law
        dac 100
        jmp oc

start
```

```
Lisp interpreter 3-20-64, part 2

/PRINT

pnt,    dac a0
        dac a1
        cal tpr

pn1,    lio i a0
        spi
        jmp pn2
        law 57

pn5,    cal pc
        lac a0
        cal cdr
        jda pwl
        lio i a0
        dio a0
        jmp pn1
pn2,    lac a0
        cal pr1

pn6,    jsp uwl
        cla
        dio a0
        spi
        jmp pn7
        lio i a0
        spi i
        jmp pn5
        lac a0
        sad n
        jmp pn3
        law 73
        cal pc
        lac a0
        cal pr1

pn3,    law 55
        cal pc
        jmp pn6

pn7,    cal pc
        lac a1
        jmp a0

/READ

ri8,    0
ri9,    0
rin,    lac rx
        dac ar1
        dzm ri9

ris,    jsp rhe
        sza i
        jmp ric
        sad (57
        jmp ria
        sad (55
        jmp rib

/.

rid,    spi
        jmp ri2

riq,    idx ar1
        lac i ar1
        dio i ar1
        dac ri9
        jsp rhe
        jmp rix

ri3,    dac ri9
        jmp ri3-2

ri2,    lac (jmp ri3
        jda pwl
        law ric


/read symbol and terminator

rhe,    dap rhx
        clf 5
        dzm t1
        law isi-1
        dac isi
        dzm isi-1
        law 72
        dac cso

rhn,    cal ava
        dac 100
        lio csi
        rir 3s
        spi
        jmp rhb
        sad (33
        cla
        sas (57
        sad (55
        jmp rye
        sad (73
        jmp rye
rhb,    sad (56
        jmp ryo
        sad (77
        jmp rhn
        sad (36
        cla
        sza i
        jmp rye
        sad (75
        jmp rhe+1
        law i 7
        and 100
        sza i
        jmp ryn
        lac 100
        sad (20
        jmp ryn
```

```
ryp,    stf 5
        cal pak
        jmp rhn

ryj,    lac t1
        cal crn
        jmp rhr

ryo,    cal ava
        jmp ryp

/symbol lookup

rye,    dac ri8
        cal mkn
        dac a0
        sad n
        jmp ryy
        szf i 5
        jmp ryj
        lac i 1ob

rys,    dac t0
        sad n
        jmp ryc
        lac i t0
        dac t1
        lac i t1
        dac t1
        lac a0

ryw,    dac a1
        sas n
        jmp ryt
        sad t1
        jmp rhh

ryd,    idx t0
        lac i t0
        jmp rys

ryt,    lac t1
        sad n
        jmp ryd
        lac i a1
        sas i t1
        jmp ryd
        idx t1
        lac i t1
        dac t1
        idx a1
        lac i a1
        jmp ryw

ryc,    lac a0
        cal mka
        lio i 1ob
        cal cns
        dac i 1ob
rhh,    lac i t0
        jmp rhr

ryn,    lio 100
        lac t1
        rir 3s
        rcl 3s
        dac t1
        lac 100
        jmp ryp+1

ryy,    clc
        lio (isi-1
        dio isi

rhr,    dac t0
        lac ri9
        lio ri8
        dio ri9
        lio t0

rhx,    exit

/, space tab

ric,    lac ar1
        spi
        jmp ris
        spa
        jmp ri4

rio,    dio t0
        cal cdr
        lio t0

rie,    swap
        cal cns
        idx ar1
        lac t0
        dac i ar1
        dac ar1
        jmp ris

ri4,    lac t0
        jmp ar1

/(

ria,    dio t0
        lac ar1
        jda pwl
        lac t0
        spa
        jmp riz
riy,    cal cns-1
        dac ar1
        lio ar1
        cal rdc
        jmp ris
riz,    dzm ar1
        jmp ris

/)
```

```
rib,    idx ar1
        lac i ar1
        lio n
        dio i ar1

rix,    jda uw
        dio ar1
        ril 1s
        lac uw
        spi
        jmp ar1
        lio uw
        lac ar1
        sza
        jmp rio
        lac uw
        jmp riy

/EVAL

evl,    dio ar2

evo,    dac ar1

/evaluate current expression

ev2,    lac ar1
        szs 10
        cal pnt
        lac i ar1
        spa
        jmp e1
        dac t0
        lac i t0
        spa
        jmp e2

/car[x] not atomic

        sad 11a
        jmp e3
        lac ar2
        jda pwl
        lac ar1
        jda pwl
        lac i ar1
        cal evo
        jsp uwl
        dio ar1
        jsp uwl
        dio ar2
        jmp evc

/evaluate function name and try again

ev3,    lac i ar1
        cal asr
        jmp qa8
        cal cdr

evc,    idx ar1
        lio i ar1
        lac uw
        dzm ar1
        cal cns
        jmp evo

/x is atomic : search a-list,
   then p-list

e1,     ral 1s
        spa
        jmp en1
        lac ar1
        cal asr
        jmp ev5
        cal cdr
        jmp ex

ev5,    lac ar1

ev4,    cal cdr
        sad n
        jmp qa8
        dac t0
        lac i t0
        sad 1ap
        jmp ev6
        idx t0
        lac i t0
        jmp ev4

ev6,    idx t0
        lac i t0
        cal car
        jmp ex

en1,    lac ar1

/exit from EVAL

ex,     szs 10
        jmp pnt
        jmp x

/car[x] is atomic : search
  its p-list

e2,     lac t0

ev8,    cal cdr
        sad n
        jmp ev3
        lac i uw
        sad 1fs
        jmp efs
        sad 1sb
        jmp esb
        sad 1xp
```

```
        jmp exp
        sad 1fx
        jmp efx
        idx t1
        lac i t1
        jmp ev8

/function is FSUBR

efs,    idx uw
        lac i uw
        cal car
        cal vag
        dac exx
        idx ar1
        lac i ar1
        lio ar2

exy,    dac 100
        dzm ar1
exx,    0
        jmp ex

/function is FEXPR

efx,    idx uw
        lac i uw
        cal car
        jda pwl
        lac ar1
        cal cdr
        cal efq
        jda pwl
        lac ar2
        cal efq
        cal cns-1
        jsp uwl
        cal efc
        jsp uwl
        cal efc
        dac ar1
        jmp ev2
efq,    cal cns-1
        lio t0
        lac 1qu
        dac 100
        jmp cns
efc,    dio 100
        lio t0
        jmp cns

/function is EXPR

exp,    idx uw
        lac i uw
        dac a1
        idx ar1
        lio i ar1
        dzm ar1
        lac i a1
        cal cns
        jmp evo

/function is SUBR

esb,    idx uw
        lac i uw
        cal car
        jda pwl
        lac ar1
        cal cdr
        lio ar2
        cal elc
        jmp els

/evaluate argument list : also LIST

elc,    sad n
        jmp x
        dac ar1
        dio ar2
        lac ar2
        jda pwl
        lac ar1
        dzm ar1

ele,    lio i pdl
        dac t0
        jda pwl
        lac ar1
        jda pwl
        lac i t0
        cal evl
        cal cns-1
        jsp uwl
        dio ar1
        lio t0
        lac ar1
        sza i
        dio ar1
        idx ar1
        sub (1
        sas t0
        lio i ar1
        lac t0
        dac i ar1
        dac ar1
        idx t0
        dio i t0
        jsp uwl
        swap
        cal cdr
        sas n
        jmp ele
        jsp uwl
        dio ar2
        idx ar1
        lac i ar1
        lio n
        dio i ar1
        dac ar1
        szs 10
        cal pnt
        lac ar1
        jmp x
```

```
els,    dac ar1
        jsp uwl
        swap
        cal vag
        dac exx
        init esa,a0-1

/store arguments for subroutine

        lac ar1
eda,    sad n
        jmp exs
        idx esa
        sad (dac a2+1
        jmp qa7
        lac i ar1

esa,    dac xy
        idx ar1
        lac i ar1
        dac ar1
        jmp eda

exs,    lac a0
        lio a1
        jmp exy

/caar[x] = LAMBDA

e3,     lac ar1
        jda pwl
        lac ar2
        jda pwl
        lac i ar1
        cal cdr
        cal car
        jda pwl
        lac ar1
        cal cdr
        lio ar2
        cal elc
        dac ar1
        jsp uwl
        dio a0
        jsp uwl
        dio ar2

ep1,    lac a0
        sad n
        jmp ep2
        lac ar1
        sad n
        jmp qf3
        lac i a0
        lio i ar1
        cal cns
        lio ar2
        cal cns
        dac ar2
        idx a0
        lac i a0
        dac a0
        idx ar1
        lac i ar1
        dac ar1
        jmp ep1

ep2,    sas ar1
        jmp qf2
        jsp uwl
        dio ar1
        lac i ar1
        cal cdr
        cal cdr
        cal car
        jmp evo
```

```
/error halt entries

qa3,    lac n
        sas pa3
        jmp x
        error flex icd          /illegal COND
        lac n
        jmp x

qa4,    error flex uss          /undefined atom in SETQ
        jmp prx

qa7,    error flex tma          /too many args
        jmp exs

qa8,    error flex uas          /unbound atomic symbol
        clf 6
        lac ar1
        cal pnt
        cal tpr
        jmp go

qc3,    error flex ilp          /illegal parity
        law 377
        and avc
        hlt+cli-opr+1
        jmp ava

qf2,    error flex lts          /LAMBDA list too short
        jmp go

qf3,    error flex ats          /arglist too short
        jmp go

qg2,    error flex pce          /pushdown cap. exc.
        jmp go

qg1,    error flex sce          /storage cap. exc.
        jmp go

qi3,    lac 100
        dac a2
        error flex nna          /non-numeric arg for arith.
        clf 6
        lac a2
        cal pnt
        cal tpr
        jmp qix

qi4,    error flex ovf          /overflow

qix,    cla 16
        jmp crn

qp1,    error flex ana          /arg non-atom for PRIN1

prx,fal,        lac n
        jmp x

start
```

```
lisp storage 3-23-64

constants

/special symbols

ssy,

1qu,    quo
1la,    lam
1ap,    apv
1ob,    obl
1sb,    sbr
1fs,    fsb
1xp,    xpr
1fx,    fxp
fre,    nil
bfw,    frs-4
tr,     t

pdl,    pdo-1

ar1,    nil
ar2,    nil
pa3,    nil
pa4,    0

pdo,

/load storage parameters

        lio mz
        clc+hlt-opr
        lat+cli-opr
        and ad
        dac hi1
        hlt
        lat
        and ad
        dac lp1

        law i end
        add hi1
        spa
        jmp pdo
        law i frs-pdo
        add lp1
        spa
        jmp pdo
        law i pdo+end-frs
        add hi1
        sub lp1
        spa
        jmp pdo

/set up registers

stu,    law pdo
        add lp1
        dac fro
        lio hi1
        rcr 1s
        ril 1s
        dio hi
        law end
        dac t0

/relocate storage

rrs,    law i 1
        add t0
        dac t0
        law i 4
        add i t0
        sma
        jsp rrl
        jsp mvs
        law i 1
        add t0
        dac t0
        sub frl
        spa
        jsp rrl
        jsp mvs
        lac t0
        sas ofs
        jmp rrs
        law ssy
        dac t0

/relocate special registers

rss,    jsp rrl
        idx t0
        sas esy
        jmp rss
        lac i 1ob
        jda gfr
        law go
        dap gcx
        jmp g2e

/relocate 1 word, move 1 word

rrl,    dap rrx
        lac i t0
        and ad
        sub ofr
        spa
        jmp rrx
        lac i t0
        add fro
        sub ofs
        dac i t0
rrx,    jmp .

mvs,    dap mvx
        lac t0
        add fro
        sub ofs
        dac t1
        lac i t0
        dac i t1
```

```
mvx,    jmp .

/constants etc.

ad,     177777
lp1,    0
hi1,    0
mz,     -0
ofs,    frs
frl,    fws
esy,    pdo
ofr,    pdo

define                  item X
        .+2             .+3
        add X           nil
        termin

define                  next A
        A               .+1
        termin

define                  subr F
        .+2             .+7
        add F+2         .+1
        sbr             .+1
        F               nil
        termin

define                  fsubr F
        .+2             .+7
        add F+2         .+1
        fsb             .+1
        F               nil
        termin

define                  apval A
        apv             .+1
        A               nil
        termin

frs,

nil,    add f38         kz

t,      add f37         kt

kz,     apval nil

kt,     apval t

obj,    add fb0         .+1
        apv             .+1
obl,    ols             nil


/object list

ols,    subr f2
        subr f3
        subr f4
        fsubr f6
        subr f7
        subr f8
        subr f12
        subr f13
        fsubr f14
        subr f18
        subr f21
        subr f24
        subr f26
        subr f27
        subr f32
        subr f33
        subr f34
        fsubr f50
        subr f51
        subr f52
        fsubr f53
        subr f54
        fsubr f60
        fsubr f61
        fsubr f62
        fsubr f63
        subr f00
        subr f01
        subr fa3

quo=. 2                 fsubr fb5
lam=. 2                 item f40
apv=. 2                 item f42
sbr=. 2                 item f43
xpr=. 2                 item f44
fsb=. 2                 item f45
fxp=. 2                 item f46

        next t
        next obj
        subr fb2
        subr fb3
        subr fb4
        nil             nil

fws,

define                  loca A
        opr A           0
        termin

define                  nam1 X
        X               nil
        termin

define                  nam2 X,Y
        X               .+1
        nam1 Y
        termin

define                  nam3 X,Y,Z
        X               .+1
        nam2 Y,Z
        termin
```

```
/SUBRs and FSUBRs
f2,    loca atm    nam2 flex ato,767644
f3,    loca car    nam1 flex car
f4,    loca cdr    nam1 flex cdr
f6,    loca cnd    nam2 flex con,767664
f7,    loca cns    nam2 flex con,767622
f8,    loca eqq    nam1 766550
f12,   loca gsm    nam2 flex gen,flex sym
f13,   loca grp    nam3 flex gre,flex ate,765147
f14,   loca elc    nam2 flex lis,767623
f18,   loca min    nam2 flex min,762422
f21,   loca nmp    nam3 flex num,flex ber,767647
f24,   loca stp    nam2 flex sto,767647
f26,   loca pr1    nam2 flex pri,764501
f27,   loca qot    nam3 flex quo,flex tie,764523
f32,   loca rda    nam2 flex rpl,flex aca
f33,   loca rdc    nam2 flex rpl,flex acd
f00,   loca xeq    nam1 flex xeq
f01,   loca crn    nam1 flex loc
f34,   loca tpr    nam2 flex ter,flex pri
f50,   loca pgm    nam2 flex pro,767667
f51,   loca ret    nam2 flex ret,flex urn
f52,   loca goe    nam1 766746
f53,   loca stq    nam2 flex set,767650
f54,   loca aso    nam2 flex sas,flex soc
fb2,   loca rin    nam2 flex rea,767664
fb3,   loca evl    nam2 flex eva,767643
fb4,   loca pnt    nam2 flex pri,764523
fb5,   loca car    nam2 flex quo,762365
fa3,   loca nul    nam2 flex nul,767643
f60,   loca pls    nam2 flex plu,767622
```

```
f61,   loca tim    nam2 flex tim,766522
f62,   loca lga    nam2 flex log,flex and
f63,   loca lgo    nam2 flex log,764651
/miscellany
f38,   nam1 flex nil
f40,   nam2 flex lam,flex bda
f42,   nam2 flex apv,766143
f43,   nam2 flex sub,767651
f44,   nam2 flex exp,767651
f45,   nam2 flex fsu,766251
f46,   nam2 flex fex,764751
fb0,   nam2 flex obl,flex ist
f37,   nam1 767623
end,
start pdo
```

2. Alphabetic Listing of Defined Macro Symbols

Following is an alphabetic listing of the defined symbols used in the macro symbolic program for Basic PDP-1 LISP. The listing shows either the numeric meaning of the instruction or the numeric register (octal) in which the subroutine commences. For the mnemonic derivation or significance of the symbols, see Section 4 below.

1ap	2333	clo	651600
1fs	2336	cna	620
1fx	2340	cnc	615
1la	2332	cnd	1070
1ob	2334	cns	614
1qu	2331	cpf	762
1sb	2335	crn	112
1xp	2337	csi	26
a1	42	cso	27
a2	43	dba	720061
ad	2470	dcc	720062
apv	3110	dia	720060
ar1	2345	div	560000
ar2	2346	dra	720063
a0	41	e1	1626
as1	405	e2	1661
as2	420	e3	2077
asc	403	eda	2061
ase	377	efc	1743
aso	374	efq	1736
asr	402	efs	1701
atm	562	efx	1715
ava	1133	elc	1770
avc	1152	ele	2000
ave	110	els	2050
avi	1204	en1	1655
avn	1173	end	3530
avr	1165	ep1	2120
avs	1177	ep2	2143
avt	1155	eqq	570
avx	1136	erm	227
beg	11	ern	231
bfw	2342	err	216
buf	63	erx	226
car	555	esa	2067
cd1	1071	esb	1757
cdr	554	esy	2476
cdy	1114	ev2	1565
cf	761	ev3	1614

ev4	1637	fb3	3410
ev5	1636	fb4	3416
ev6	1651	fb5	3424
ev8	1662	fb0	3522
evc	1620	ffi	30
evl	1563	fre	2341
evo	1564	frl	2475
ex	1656	fro	234
exp	1746	frs	2500
exs	2074	f01	3334
exx	1713	f00	3330
exy	1711	fsb	3124
f12	3226	fws	3170
f13	3234	fxp	3130
f14	3244	g1	24
f18	3252	g2a	351
f2	3170	g2e	341
f21	3260	g2f	357
f24	3270	g2n	343
f26	3276	g2x	355
f27	3304	ga1	31
f3	3176	gc	235
f32	3314	gci	363
f33	3322	gco	251
f34	3340	gcp	260
f37	3526	gcs	673
f38	3470	gcx	356
f4	3202	gfa	323
f42	3476	gfd	313
f43	3502	gfl	326
f44	3506	gfn	277
f45	3512	gfr	267
f46	3516	gfu	316
f40	3472	gfx	322
f51	3354	go	4
f52	3362	goe	526
f53	3366	grp	1123
f54	3374	g0	23
f50	3346	gsi	1025
f6	3206	gsm	1023
f61	3446	gsn	1051
f62	3454	gsp	1033
f63	3462	gst	34
f60	3440	hi	25
f7	3214	hi1	2472
f8	3222	ioh	730000
fa3	3432	isi	33
fal	2241	kt	2510
fb2	3402	kz	2504

lai	760040	plz	633
lam	3104	pn1	1240
lga	651	pn2	1253
lgo	656	pn3	1274
lia	760020	pn5	1244
lp1	2471	pn6	1255
min	764	pn7	1277
mka	611	pnt	1235
mkn	1212	pr1	703
mul	540000	pra	712
mvs	2460	prn	730
mvx	2467	prv	736
mz	2473	prx	2241
n	234	pwl	44
nil	2500	pwx	53
nmp	754	qa3	2154
nul	567	qa4	2163
obj	2514	qa7	2166
obl	2520	qa8	2171
oc	165	qc3	2200
oc1	206	qf2	2206
ofr	2477	qf3	2211
ofs	2474	qg1	2217
ols	2522	qg2	2214
out	210	qi3	2222
pa3	2347	qi4	2233
pa4	2350	qix	2235
pak	1224	qot	1057
pc	117	qp1	2237
pcc	135	quo	3074
pch	143	rda	607
pdl	2344	rdc	605
pdo	2351	ret	524
pg1	472	rhb	1360
pg2	516	rhe	1333
pg3	511	rhh	1462
pg4	515	rhn	1343
pg5	437	rhr	1476
pg6	452	rhx	1503
pg7	453	ri2	1330
pg9	466	ri3	1326
pgm	423	ri4	1524
pg0	471	ri8	1302
pk1	1232	ri9	1303
pl1	634	ria	1526
pl2	635	rib	1543
ple	647	ric	1504
plo	642	rid	1316
pls	631	rie	1514

rin	1304	spq	650500
rio	1511	ssy	2331
riq	1320	stp	1120
ris	1307	stq	533
rix	1547	stu	2377
riy	1534	swp	760060
riz	1541	szm	640500
rrl	2445	t	2502
rrs	2410	t1	22
rrx	2457	tic	666
rss	2434	tim	661
rx	561	tpr	700
ryc	1455	tr	2343
ryd	1436	tru	565
rye	1411	t0	21
ryj	1404	uw	54
ryn	1464	uwl	55
ryo	1407	uwx	62
ryp	1401	vad	156
rys	1421	vag	144
ryt	1441	x	556
ryw	1431	xei	1000
ryy	1473	xen	1017
sbr	3114	xeq	767
sft	660000	xer	1003
smi	652000	xpr	3120
sni	644000	xy	0

3. Numeric Listing of the Defined Macro Symbols

Following is a listing in numerical order by register number or other meaning of the defined symbols in the macro symbolic program for Basic PDP-1 LISP.

xy	0	gfr	267
go	4	gfn	277
beg	11	gfd	313
t0	21	gfu	316
t1	22	gfx	322
g0	23	gfa	323
g1	24	gfl	326
hi	25	g2e	341
csi	26	g2n	343
cso	27	g2a	351
ffi	30	g2x	355
ga1	31	gcx	356
isi	33	g2f	357
gst	34	gci	363
a0	41	aso	374
a1	42	ase	377
a2	43	asr	402
pwl	44	asc	403
pwx	53	as1	405
uw	54	as2	420
uwl	55	pgm	423
uwx	62	pg5	437
buf	63	pg6	452
ave	110	pg7	453
crn	112	pg9	466
pc	117	pg0	471
pcc	135	pg1	472
pch	143	pg3	511
vag	144	pg4	515
vad	156	pg2	516
oc	165	ret	524
oc1	206	goe	526
out	210	stq	533
err	216	cdr	554
erx	226	car	555
erm	227	x	556
ern	231	rx	561
fro	234	atm	562
n	234	tru	565
gc	235	nul	567
gco	251	eqq	570
gcp	260	rdc	605

rda	607
mka	611
cns	614
cnc	615
cna	620
pls	631
plz	633
pl1	634
pl2	635
plo	642
ple	647
lga	651
lgo	656
tim	661
tic	666
gcs	673
tpr	700
pr1	703
pra	712
prn	730
prv	736
nmp	754
cf	761
cpf	762
min	764
xeq	767
xei	1000
xer	1003
xen	1017
gsm	1023
gsi	1025
gsp	1033
gsn	1051
qot	1057
cnd	1070
cd1	1071
cdy	1114
stp	1120
grp	1123
ava	1133
avx	1136
avc	1152
avt	1155
avr	1165
avn	1173
avs	1177
avi	1204
mkn	1212
pak	1224
pk1	1232
pnt	1235
pn1	1240
pn5	1244
pn2	1253
pn6	1255
pn3	1274
pn7	1277
ri8	1302
ri9	1303
rin	1304
ris	1307
rid	1316
riq	1320
ri3	1326
ri2	1330
rhe	1333
rhn	1343
rhb	1360
ryp	1401
ryj	1404
ryo	1407
rye	1411
rys	1421
ryw	1431
ryd	1436
ryt	1441
ryc	1455
rhh	1462
ryn	1464
ryy	1473
rhr	1476
rhx	1503
ric	1504
rio	1511
rie	1514
ri4	1524
ria	1526
riy	1534
riz	1541
rib	1543
rix	1547
ev1	1563
evo	1564
ev2	1565
ev3	1614
evc	1620
e1	1626
ev5	1636

f24	3270	fb5	3424
f26	3276	fa3	3432
f27	3304	f60	3440
f32	3314	f61	3446
f33	3322	f62	3454
f00	3330	f63	3462
f01	3334	f38	3470
f34	3340	f40	3472
f50	3346	f42	3476
f51	3354	f43	3502
f52	3362	f44	3506
f53	3366	f45	3512
f54	3374	f46	3516
fb2	3402	fb0	3522
fb3	3410	f37	3526
fb4	3416	end	3530

4. Mnemonic Key or Derivation of Symbols

The purpose of this listing is to state the mnemonic key or derivation, and here and there some comments, for many of the symbols used in the MACRO symbolic listing of Basic PDP-1 LISP. In this way the MACRO symbolic may be more easily read with understanding. For example, to know that "pdl" stands for "push down list" is helpful.

Some of the symbols there included however have been omitted from this listing, particularly those symbols which differ only in subscripts from the symbols in this listing.

a0	argument, sub zero
a1	argument, sub one
ar1	argument one
as2	ASSOC, sub two
ase	ASSOC, entry
aso	ASSOC, origin
atm	atom
ava	get a character, advance
avc	advance, compute parity
ave	advance, end
avi	advance, in from typewriter
avn	advance, next
avr	advance, reader
avs	advance, store
avt	advance, truncate (to 6 bits from reader); also, detecting upper case or lower case in the sign bit
avx	advance, index
beg	beginning
bfw	beginning of full words (end of push down list)
buf	buffer
car	CAR (contents of address register)

cd1	COND, sub one
cdr	CDR (contents of decrement register)
cdy	COND, sub y
cf	cons, full; do a CONS into full word space
cfa	CONS into full word space, sub a
cna	CONS, sub a
cnd	COND
cns	CONS
crn	create number
cpf	CONS pair in full word space
e1	evaluate, sub one
e2	evaluate, sub 2; CAR X is atomic, search its P-list
e3	evaluate, sub 3 CAAR X equals LAMBDA
e4	evaluate, sub 4; CARR X equals LABEL
eda	entry, deposit argument (store arguments for subroutine)
efc	evaluate, function sub c
efq	evaluate, function sub q
efs	evaluate, function is an FSUBR
efw	end of full words
efx	evaluate, function is a FEXPR
elc	evaluate, argument is a list; also LIST
ele	evaluate, list, entry
els	evaluate, list, subroutine
en1	evaluate, number, sub one
ep1	evaluate, pair, sub one
ep2	evaluate, pair, sub two
eqq	entry point of EQ (lengthened)
erm	error message
ern	error name
err	error printout
erx	error exit
esa	evaluate, store argument
esb	evaluate, function is a SUBR
ev2	evaluate current expression, sub 2
ev3	evaluate, sub 3; (evaluate function name and try again)
ev4	evaluate, sub 4
evc	evaluate, construct
evl	EVAL
evo	EVAL, old
ex	exit from EVAL
exp	evaluate, function is an EXPR
exs	eval, exit, subroutine
exx	evaluate, exit, execute
fal	false (value is NIL)
fre	pointer to the beginning of the free storage list (at top)
frs	origin of free storage
fwf	field number where full word storage is kept
fws	full word space
g0	garbage collector, temporary storage, sub zero
g1	garbage collector, temporary storage, sub one
g2a	garbage collector, part 2, advance
g2e	garbage collector, part 2, entry
g2f	garbage collector, part 2, free
g2n	garbage collector, part 2, next
g2r	garbage collector, part 2, return
g2x	garbage collector, part 2, exit
ga1	garbage collector, argument one
gc	garbage collector, non-compacting
gcf	garbage collector, full word

gcp	garbage collector, push-down
gcs	garbage collector, step
gcx	garbage collector, exit
gfn	garbage, free, next
gfr	garbage, free; (not returned to free storage) mark one list
gfu	garbage, free, unsave
gfx	garbage, free, exit
go	go
goe	GO (lengthened)
grp	GREATERP (is greater than)
gsi	gensym, index
gsm	gensym, entry
gsn	gensym, next
gsp	gensym, produce
gst	table for symbol generator
hih	high: first address beyond full word space
hih-i	high minus i (10,000 octal)
i-nfw	i (10,000 octal) minus number of full words, equals left-over space
isi	input string initial (points to a string of characters just read in)
kt	property list for T
kz	property list for NIL
lga	LOGAND
lgo	LOGOR
lpl	length of push-down list
min	MINUS
mka	make atom
mkn	terminate print name, and make name
n	NIL (register containing)
nfw	number of full words
nmp	NUMBERP ("is a number")
nul	NULL
obj	location of atom OBLIST
oc1	output character, sub one
oc3	output character, sub three
oc	output character (pack character on to end of buffer)
ocf	output character into full word space
ocj	output character, jump
ocy	output character, exit
ofw	origin of full words
ols	location of OBLIST
oup	output routine, print
out	output routine
pa3	PROG argument, sub 3
pa4	PROG argument, sub 4
pak	pack character into print name
pc	print or punch character
pcc	punch character, compute correct parity for punching
pch	punch character count, producing carriage return (generated after every 64 (decimal) characters)
pda	push down list, sub a
pdb	push down list, sub b
pdc	push down list, sub c
pdd	push down list, sub d
pde	push down list, sub e
pdf	push down list, sub f
pdg	push down list, sub g
pdi	push down list, sub i
pdj	push down list, sub j
pdk	push down list, sub k
pkl	push down list
pdo	push down list, origin
pg0	program feature, sub zero
pg1	program feature, sub one
pg2	program feature, sub two
pgm	program feature
pk1	pack character into print name, sub one
pk2	pack character into print name, sub two
pl1	plus, sub one
pl2	plus, sub two
ple	plus, exit
plo	plus, operation

pls PLUS

plz plus, zero sum storage register

pn1 print, sub one

pn2 print, sub two

pnt PRINT

pr1 PRIN1

prx print, exit after finishing printing

pwl push word on list (append word to push-down list)

pwx push word exit

qa error halt

qa3 error halt, illegal COND; icd

qa4 error halt, undefined atom in SETQ; uss

qa7 error halt; too many arguments; tma

qa8 error halt, unbound atomic symbol; uas

qc3 error halt, illegal parity; ilp

qf2 error halt, LAMBDA list too short; lts

qf3 error halt, argument list too short; ats

qg1 error halt, storage capacity exceeded; sce

qg2 error halt, pushdown capacity exceeded; pce

qi3 error halt, nonnumeric argument for arithmetic; nna

qi4 error halt, overflow; ovf

qix error halt, exit

qot quotient

qp1 error halt, argument non-atom for PRIN1

rda RPLACA

rdc RPLACD

ret RETURN

rhe read in, entry; read symbol and terminator; (a "terminator" is a left parenthesis, right parenthesis, period, space, comma, or tab) (mnemonic: h precedes i)

rhn read symbol and terminator, next

rhx read symbol and terminator, exit

rhr read symbol and terminator, sub r

ri3 read in, sub 3

ri4 read in, sub 4

ria read in, left parenthesis

rib read in, right parenthesis

ric read in, comma

rid read in, dot

rie read in, sub e

rio read in, sub o

rin READ in

riq read in, sub q

ris read in symbol

rix read in, exit

riy read in, sub y

riz read in, zeroing

rx return to calling sequence of a subroutine

ryc symbol lookup, create (creating what is necessary to put something on the OBLIST)

ryd symbol lookup, index

rye symbol lookup; search for symbol in object list and if not found, put at beginning

ryj symbol lookup, sub j

ryn symbol lookup, number

ryo symbol lookup, sub o

ryp symbol lookup, pack

ryt symbol lookup, test

rys symbol lookup, search for symbol

ryw symbol lookup, search for word

ryy	symbol lookup, exit
stp	STOP
stq	SETQ
t0	temporary storage, sub zero
t1	temporary storage, sub one
tic	times, complete
tim	TIMES
tpr	make a carriage return, TERPRI (terminate printing)
tru	true
undex	decrease (some number) by one
uw	unsave word; retrieve word from push down list
uwl	unsave word from list
uwx	unsave word, exit
vad	get two values
vag	get numeric value
x	exit from machine language LISP functions

Appendix 6

Index for Parts I to VII of the LISP 1.5 Programmer's Manual

Edmund C. Berkeley and Daniel G. Bobrow

The "LISP 1.5 Programmer's Manual", published August 17, 1962, is an important source of information on LISP. It consists mainly of eight parts entitled:

I. The LISP Language
II. The LISP Interpreter System
III. Extension of the LISP Language
IV. Arithmetic in LISP
V. The Program Feature
VI. Running the LISP System
VII. List Structures
VIII. A Complete LISP Program - The Wang Algorithm for the Propositional Calculus (This part discusses an example.)

In addition there are eight appendices, an index to function descriptions, and a glossary. There is no general index.

To make it easier to find the relevant discussion of topics in Parts I to VII of the manual, the following general index for Parts I to VII has been prepared.