An algebraic extension to LISP

by PRENTISS HADLEY KNOWLTON

Harvard University
Cambridge, Massachusetts

INTRODUCTION

An algebraic facility for LISP is quite desirable. Such a capability is motivated by the desire to utilize the primitive LISP arithmetic functions at the algebraic expression level. The requirement for a means of evaluating expressions might very well arise from applications in algebraic manipulation. Thus, the user, having performed some sort of transformation on an algebraic expression, might wish to have the resulting expression evaluated for a specific set of values. This facility, in response to this requirement, has the acronym "LEAF" (LISP Extended Algebraic Facility).

Design considerations and FORTRAN language facilities provided by LEAF include:

1. a list structured organization compatible with existing LISP;
2. an arithmetic assignment statement;
3. a DO statement;
4. a logical IF statement;
5. an unconditional GO TO statement; and
6. an INPUT and OUTPUT statement.

Since LEAF is designed in the "spirit" of LISP, built in functions in a given LISP system which provide for such conveniences as "pretty printing" of functions and editing facilities may also be applied to LEAF programs.

The list structured organization of LEAF

Although the initial motivation in developing LEAF was to extend the LISP language, a number of other motivating properties of the LEAF concept make themselves apparent as one uses the LEAF facility. In order to attain compatibility with the existing LISP language, LEAF is essentially a dialect of FORTRAN in list structure. Hence, a program is a list whose elements are statements. A simple LEAF program to accept two numbers from the teletype, determine their sum, and type out the result might be written as follows:

\[
\text{(INPUT } A \ B) \\
\text{(C} = A + B) \\
\text{(OUTPUT } C)\
\]

In similar manner, a statement is a list whose elements are the components of that statement. In order to execute a statement, the LEAF interpreter typically looks at the keyword (e.g., INPUT), the first element of the statement, to determine how the statement should be processed. This is analogous to the LISP interpreter, in which the first element of a LISP command is a function, and the remaining elements of that command constitute the arguments of the function.

In the "assignment" function, unlike the other LEAF commands, the keyword or "=" is the second element of the list. If the item on the left hand side of the equal sign is an array reference, the subscripting can be thought of as a single list element, a sublist whose elements constitute the subscripts. In SDS 940 LISP as well as in other LISP implementations, commas are perfectly acceptable list element delimiters. Thus, the user is free to use commas for readability in subscript lists if he desires, and he is not constrained to always delimit list elements with blanks. It is important to note in the case of a subscripted variable on the left hand side of the equal sign in the assign-
ment statement that the "=" is in fact the third element of the list. Nevertheless, recognition and processing of the assignment statement is still a relatively straightforward procedure.

In addition to the properties LISP and LEAF share, it is interesting to note that the conveniences which exist for displaying and modifying LISP functions are also applicable to the display and modification of LEAF programs. The nesting of DO loops is readily apparent from the indented listing one obtains from the LISP "pretty printing" facility:

```
( (DO I = 1 TO 10
   (A(I) = B(I))
   (DO J = 1 TO 10
   ( . )
   ( . )
   ( . ) )
   ( . )
   ( . )
   ( . ) )))
```

In like manner, one may utilize the editing facilities available on a given LISP system to modify a LEAF program with equivalent flexibility as modifying a LISP function.

**Justifications for a list structure**

It is worthwhile noting that the list structured approach to the design of an algebraic language lends itself well to the concepts of program block structure, program editing, adaptability to a time sharing environment, and, most important of all, language and data structure compatibility.

Program block structure of the LEAF system is best illustrated by the DO statement, in which a list whose elements are statements constitute the range of the DO specification. This program block structure lends itself well to editing operations, since, armed with an indented listing of his program, one is able to quickly and accurately access and work with his program at any level. An example of program modification using the editing facility of SDS 940 BBN LISP is given in Appendix C.

Like the LISP language, LEAF lends itself well to a time sharing environment, in that LEAF programs are easily interpreted at the source language level. List structured organization of LEAF programs permit several users to work independently with the same reentrant interpreter, even when two separate programs are "intertwined" in the same storage region.

A particularly significant observation one might make of the LEAF language is that it possesses the same basic structure as its data. Hence, there is no reason why one might not wish to devise a program which performs operations upon itself, such as the changing of a "+" to an "*" in an arithmetic expression. In this sense, within the framework of the LEAF language, a statement might be thought of as an alphanumerical vector whose elements are keywords, operators, and operands.

**Fortran language facilities provided by LEAF**

1. The Assignment Statement

The assignment statement of LEAF is identical to that of FORTRAN IV with the additional flexibility of mixed mode arithmetic. Thus, one may work interchangeably with both integer and real data in arithmetic expressions without worrying about problems of mode conversion, since the existing LISP floating point functions are designed to handle such situations automatically.

2. The DO Statement

The DO specification of LEAF is similar to that of PL/I. The remainder of the statement consists of a list whose elements as statements constitute the range of the DO. Any level of nesting is permissible, and the LISP "pretty printing" facility shows the nesting quite clearly as illustrated earlier.

3. The Logical IF Statement

Like PL/I, the logical IF statement consists of an "IF" part followed by a "THEN" part. The "IF" part consists of two arithmetic expressions separated by a relational operator (without periods). The true or false value of the relation determines the execution or non-execution of the "THEN" part. In either event, the next statement in sequence is reached.

4. The Unconditional GO TO Statement

The GO TO statement of LEAF, like that of PL/I, specifies destination by means of a name rather than by means of a statement number as is the case with FORTRAN IV.

5. The INPUT Statement

The INPUT statement consists of the key word "INPUT" followed by the variables to be defined. The "RATOM" (read atom) func-
tion of SDS 940 BBN LISP permits relative free formatting of input data.

6. The OUTPUT Statement

Similarly, the OUTPUT statement consists of the keyword “OUTPUT” followed by the variables to be printed. The “PRINT” function of SDS 940 BBN LISP is utilized in this context.

CONCLUSIONS

The LEAF approach seems to be an answer to certain problems facing users who are dissatisfied with present day LISP and present day FORTRAN. Feasibly, programs already written in FORTRAN IV might be converted to LEAF. The advantages of indented display of program nesting as well as the facilities of the LISP editor would certainly warrant this activity.

Working with an algebraic language at the source language level has many distinct advantages. Among these advantages, this writer suggests that the COMMENT statement should be treated as an executable statement, whose text could be made to be listed by user request during program execution.

The author sincerely hopes that the philosophy of the LEAF system is given some consideration by the implementers of future algebraic compilers.

APPENDIX A

Syntax description of the LEAF system

I. Fundamental Language Components:

- (letter) ::= 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
- (digit) ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
- (identifier) ::= (letter) { (letter) | (digit) } *
- (variable) ::= (identifier)
- (unsigned-integer-constant) ::= (digit) { (digit) } *
- (sign) ::= + | -
- (integer-constant) ::= [ (sign) ] (unsigned-integer-constant)
- (real-constant) ::= [ (sign) ] (unsigned-integer-constant) \\
  ( (unsigned-integer-constant) (exponent-part) )
   [ (sign) ] (unsigned-integer-constant) \\
   (unsigned-integer-constant) (exponent-part) ]
- (exponent-part) ::= [ (sign) ] (digit) ]

II. Basic Language Elements

- (program) ::= [(statement)] *
- (statement) ::= (comment-statement) | (optional-statement-label) (statement-body)
- (comment-statement) ::= COMMENT (commentary) | (commentary) *

ACKNOWLEDGMENTS

The author wishes to extend special thanks to Dr. Daniel G. Bobrow of Harvard University’s Applied Mathematics Department, under whom this work was done as independent study. Dr. Bobrow is also responsible for many of the facilities present in SDS 940 BBN LISP. Special thanks are also due to Aiken Computation Laboratory of Harvard University who graciously provided SDS 940 computer time for the carrying out of this work.

Mr. Cornelius Peterson, manager of the Boston Office of Computer Usage Company, provided the necessary facilities for the writing of this paper. Finally, Mr. Burton Bloom, Senior Staff Analyst of the CUC Boston Office, provided many helpful suggestions during the technical revision of this work.

Finally, the author extends appreciation to Jet Propulsion Laboratory, Pasadena, California, for the use of their facilities in preparing visual aids in the presentation of this paper.

REFERENCE

1 D G BOBROW et al
The BBN 940 LISP system
Bolt Beranek and Newman Inc Cambridge Mass April 1968
APPENDIX B

Some representative functions of the LEAF interpreter

(STATEMENT
 (LAMBDA (COMMAND)
   (COND
    ((COMMENT-STATEMENT COMMAND) NIL)
    ((DO-STATEMENT COMMAND) NIL)
    ((INPUT-STATEMENT COMMAND) NIL)
    ((OUTPUT-STATEMENT COMMAND) NIL)
    ((ASSIGNMENT-STATEMENT COMMAND) NIL)
    ((GO-TO-STATEMENT COMMAND) NIL)
    (T (IF-STATEMENT COMMAND))))

(COMMENT-STATEMENT
 (LAMBDA (COMMAND)
   (EQ (CAR COMMAND) (QUOTE COMMENT)))

(DO-STATEMENT
 (LAMBDA (COMMAND)
   (COND
    ((COMMENT-STATEMENT COMMAND) NIL)
    ((DO-STATEMENT COMMAND) NIL)
    ((INPUT-STATEMENT COMMAND) NIL)
    ((OUTPUT-STATEMENT COMMAND) NIL)
    ((ASSIGNMENT-STATEMENT COMMAND) NIL)
    ((GO-TO-STATEMENT COMMAND) NIL)
    (T (IF-STATEMENT COMMAND))))

APPENDIX B

Some representative functions of the LEAF interpreter

(STATEMENT
 (LAMBDA (COMMAND)
   (COND
    ((COMMENT-STATEMENT COMMAND) NIL)
    ((DO-STATEMENT COMMAND) NIL)
    ((INPUT-STATEMENT COMMAND) NIL)
    ((OUTPUT-STATEMENT COMMAND) NIL)
    ((ASSIGNMENT-STATEMENT COMMAND) NIL)
    ((GO-TO-STATEMENT COMMAND) NIL)
    (T (IF-STATEMENT COMMAND))))

(COMMENT-STATEMENT
 (LAMBDA (COMMAND)
   (EQ (CAR COMMAND) (QUOTE COMMENT)))

(DO-STATEMENT
 (LAMBDA (COMMAND)
   (COND
    ((COMMENT-STATEMENT COMMAND) NIL)
    ((DO-STATEMENT COMMAND) NIL)
    ((INPUT-STATEMENT COMMAND) NIL)
    ((OUTPUT-STATEMENT COMMAND) NIL)
    ((ASSIGNMENT-STATEMENT COMMAND) NIL)
    ((GO-TO-STATEMENT COMMAND) NIL)
    (T (IF-STATEMENT COMMAND))))
(PROG (INDEX FROM TO)
  (COND
    ((NEQ (CAR COMMAND) (QUOTE DO))
     (RETURN NIL))
    (SETQ INDEX (CADR COMMAND))
    (SETQ FROM (CADDDR COMMAND))
    (SETQ INDEX FROM)
    (SETQ TO (CADDDDDDR COMMAND)))
(LOOP (COND
    ((CADDRP INDEX TO)
     (RETURN T))
    (LEAF (CADDDDDAD COMMAND))
    (ADD1 INDEX)
    (GO LOOP)
  )))

(INPUT-STATEMENT
  (LAMBDA (COMMAND)
    (PROG (ARGUMENT-LIST)
      (COND
        ((NEQ (CAR COMMAND) (QUOTE INPUT))
         (RETURN NIL))
        (SETQ ARGUMENT-LIST (CDR COMMAND)))
      (LOOP (COND
        ((NULL (CAR ARGUMENT-LIST))
         (RETURN T))
        (SET (CAR ARGUMENT-LIST) (RATOM NIL))
        (SETQ ARGUMENT-LIST (CDR ARGUMENT-LIST))
        (GO LOOP)
      )))
    )))

(OUTPUT-STATEMENT
  (LAMBDA (COMMAND)
    (PROG (ARGUMENT-LIST)
      (COND
        ((NEQ (CAR COMMAND) (QUOTE OUTPUT))
         (RETURN NIL))
        (SETQ ARGUMENT-LIST (CDR COMMAND)))
      (LOOP (COND
        ((NULL (CAR ARGUMENT-LIST))
         RETURN T))
        (PRINT (CAAR ARGUMENT-LIST))
        (SETQ ARGUMENT-LIST (CDR ARGUMENT-LIST))
        (GO LOOP)
      )))
    )))

(ASSIGNMENT-STATEMENT
  (LAMBDA (COMMAND)
(PROG NIL
  (COND
    ((NEQ (CADR COMMAND)
      (QUOTE =))
      (RETURN NIL)))
    (SET (CAR COMMAND)
      (ARITHMETIC-EXPRESSION (CDDR COMMAND)))
      (RETURN T))
))

(ARITHMETIC-EXPRESSION
 (LAMBDA (LIST)
   (PROG (VALUE)
     (SETQ POINTER LIST)
     (SETQ VALUE (TERM NIL))
     LOOP (COND
       ((NULL (CAR POINTER))
         (RETURN VALUE))
       ((EQ (CAR POINTER)
         (QUOTE +))
         (SETQ POINTER (CDR POINTER))
         (SETQ VALUE (FPLUS VALUE (TERM NIL)))
         (GO LOOP))
       ((EQ (CAR POINTER)
         (QUOTE -))
         (SETQ POINTER (CDR POINTER))
         (SETQ VALUE (FDIFFERENCE VALUE (TERM NIL)))
         (GO LOOP))
       (T (RETURN VALUE)))
    ))

(TERM
 (LAMBDA NIL
   (PROG (VALUE)
     (SETQ VALUE (FACTOR NIL))
     LOOP (COND
       ((NULL (CAR POINTER))
         (RETURN VALUE))
       ((EQ (CAR POINTER)
         (QUOTE *))
         (SETQ POINTER (CDR POINTER))
         (SETQ VALUE (FTIMES VALUE (FACTOR NIL)))
         (GO LOOP))
       ((EQ (CAR POINTER)
         (QUOTE /))
         (SETQ POINTER (CDR POINTER))
         (SETQ VALUE (FQUOTIENT VALUE (FACTOR NIL)))
         (GO LOOP))
       (T (RETURN VALUE)))
    ))

(FACTOR

From the collection of the Computer History Museum (www.computerhistory.org)
(LAMBDA NIL
  (PROG (VALUE POINTER-SAVE)
    COND
      ((NUMBERP (CAR POINTER))
       (SETQ VALUE (CAR POINTER))
       (SETQ POINTER (CDR POINTER))
       (RETURN VALUE))
      ((ATOM (CAR POINTER))
       (SETQ VALUE (CAAR POINTER))
       (SETQ POINTER (CDR POINTER))
       (RETURN VALUE))
      (T (SETQ POINTER-SAVE POINTER)
       (SETQ VALUE (ARITHMETIC-EXPRESSION (CAR POINTER)))
       (SETQ POINTER POINTER-SAVE)
       (SEQ POINTER (CDR POINTER))
       (RETURN VALUE))))

(FDIFFERENCE
  (LAMBDA (A B)
    (FPLUS A (FMINUS B))))

(LEAF
  (LAMBDA (PROGRAM)
    (PROG (LOCATION LABEL
      (SETQ LOCATION PROGRAM)
      LOOP (COND
        ((NULL (CAAR LOCATION))
         NIL)
        ((STOP-STATEMENT (CAR LOCATION))
         (RETURN (QUOTE STOP)))
        (STATEMENT (CAR LOCATION))
        (SETQ LOCATION (CDR LOCATION)
         GO LOOP))
      )))

(STOP-STATEMENT
  (LAMBDA (COMMAND)
    (EQ (CAR COMMAND)
     (QUOTE STOP))))

APPENDIX C

Representative applications of the LEAF system
Examples of input statements, output statements, the assignment statement, and arithmetic expressions

← INPUT-STATEMENT ((INPUT A B C D E F G))
  1.0 2.0 3.0 4.0 5.0 6.0 7.0
  T†

† The "T" indicates that the invoked function succeeded.
A program using input, output, and assignment statements

\[ E(\text{SETQQ PROGRAM} ((\text{INPUT } A B) (C = A + B) (D = A - B) (E = A \times B) (F = A / B) \text{ OUTPUT } A B C D E F) \text{ STOP}) ) \]

\[ E(\text{LEAF PROGRAM}) \]

The ""] causes a sufficient number of right parentheses to be generated.

At this point, the atom "PROGRAM" is bound with the LEAF program as shown. The top-level function "E" merely means "execute the given function (first elements) on its arguments without prior evaluation of those arguments."

The LEAF interpreter is now applied to the designated program. The user satisfies the INPUT statement by typing "2.0 3.0 (CR)," and the LEAF system responds with the desired output, followed by "STOP" as generated by the STOP statement.
A program using the DO statement

\[
\text{← PRETTYPRINT(SUMMATION) \&}
\]

\[
((SUM = 0.0000000000000000)
\quad (COUNT = 0.0000000000000000)
\quad (DO I = 1 \text{ TO } 10 ((COUNT = COUNT + 1.0000000000000000)
\quad (SUM = SUM + COUNT))
\quad (OUTPUT SUM)))
\quad (STOP))
\]

\[
\text{← E(LEAF SUMMATION)}
\]

\[
1.0000000000000000
3.0000000000000000
6.0000000000000000
10.0000000000000000
15.0000000000000000
21.0000000000000000
28.0000000000000000
36.0000000000000000
45.0000000000000000
55.0000000000000000
\]

STOP

Modification of a program using the editing facility

\[
\text{← EDITV(SUMMATION) \&}
\]

EDIT

\[
*(1 \text{ (SUM } = 1.0))
*3
*7
*2
*P
(SUM = SUM + COUNT)
*(4 * )
*↑
*PP
\]

In this instance, we assume that the "SUMMATION" program has already been defined; hence, we need only print it out using the "PRETTYPRINT" of SDS 940 BBN LISP. Note how transparent program block structure becomes via this facility.

† At this point we wish to edit our sample SUMMATION example to no longer produce successive sums, but to produce successive products or factorials. The "*" tells us we are talking to the editor. The command "*(1 (SUM = 1.0))" updates the first statement of our original summation program (1.0 is the identity element for multiplication.). "*3" focuses our attention on the DO statement, "*7" focuses our attention on the range of the DO, and "*2" focuses our attention on the second statement of the range of the DO. "*P" causes that statement to be printed out, the operation "*(4 *)" causes the "*" of that statement to be changed to an "*", "*" returns our attention to the top level, "*PP" "pretty prints" the edited function, and "OK" tells the editor we are all done.
((SUM = 1.000000000)  
(COUNT = 0.000000000)  
(DO I = 1 TO 10 ((COUNT = COUNT + 1.000000000)  
(SUM = SUM * COUNT)  
(OUTPUT SUM))))  
(STOP))  

*OK  
SUMMATION  

← E(LEAF SUMMATION)  
1.000000000  
2.000000000  
6.000000000  
24.000000000  
120.000000000  
720.000000000  
5040.000000000  
40320.000000000  
362880.000000000  
3628800.000000000  
STOP