The research reported in this paper was sponsored by the Advanced Research Projects Agency Information Processing Techniques Office and was monitored by the Electronic Systems Division, Air Force Systems Command under contract F1962867C0004 with the System Development Corporation.

System Development Corporation/2500 Colorado Ave./Santa Monica, California 90406

SP-2604

AUTHOR J. A. Barnett

DATE 20 September 1966

LISP 2 Programming Examples

The LISP 2 Source Language is still in the developmental stages. The programming examples shown reflect one proposed syntax. However, it is expected that numerous changes will be made in the near future.

Definition of S-expressions

<identifier> = a word
ABC is an identifier

<number> = I hope you already know
1.5, 2 are numbers

<array> = by example:
[INTEGER 1 2 3] is an array of three numbers

<functional> = by example:
FUNCTION ADD (A,B) A+B; definition of function ADD that sums its two arguments

<boolean> = TRUE|FALSE

<string> = # a hollerith string #
#ABC# is a string and is different from ABC. Identifiers like ABC may have several properties not attributed to strings. (An explanation of this will not be given here.)

<nil> = NIL|( )
Think of this as the empty list, i.e., a list of no elements. (NIL ≡ ())

<atom> = <identifier>|<number>|<array>|<functional>|<boolean>
<string>|<nil>

<s-expression> = <atom>|(<s-expression> . <s-expression>)

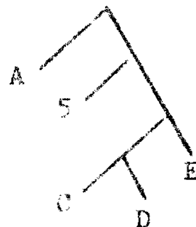
© Copyright System Development Corporation 1966.

Examples of s-expressions:

A , (A . ((B . C) . D))

Note that s-expressions are nice ways of representing binary trees:

(A . (5 . ((C . D) . E)))



A binary tree has the property that either a terminator exists at a point or that point is an ordered pair of "pointers" to more tree structure.

An s-expression has the property that either an atom exists at a point or that point is an ordered pair of other s-expressions.

As is obvious by now, this notation is cumbersome. A few conventions are adapted:

(A . (B --- Z)) → (A B -- Z)

(A . (B . C)) → (A B . C)

(A . ()) → (A)

i.e., if a dot appears to the left of a (, the dot and the matching set (and) are removed. Applying this rule to

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

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

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

(A B C D . ())

and finally (A B C D)

Whenever an s-expression is reducible to a form containing no dots, we call this a pure list. Lists are obviously a subset of s-expressions.

In working with either lists or s-expressions there are three basic and necessary functions:

CAR - the first element of the ordered pair which is an s-expression

```
CAR( (a . b)) = a
CAR( (a b c)) = CAR( (a . (b c))) = a
```

CDR - the second element of the ordered pair which is an s-expression

```
CDR( (a . b)) = b
CDR( (a b c)) = CDR( (a . (b c))) = (b c)
```

· for list structure, the value of CDR is a list of all but the first element

CONS - takes two s-expressions as arguments and produces the ordered pair of them

```
CONS(a, b) = (a . b)
CONS(a, (b c)) = (a . (b c)) = (a b c)
```

CONS is written in LISP 2 Source Language as the infix operator ".", e.g., CONS(a , b) ≡ a . b → (a . b). The "." in the language is compiled as a call to the function CONS.

The functions CAR and CDR are so named for historical reasons and are undefined for an atomic argument.

Another function used in the example programs is APPEND, which has two arguments, both lists.

```
APPEND( (a b) (x y z)) = (a b x y z)
```

APPEND simply combines the two list arguments to form a new list.

CADR(X) = CAR(CDR (X))

CDDAR(X) = CDR(CDR (CAR(X))) etc.

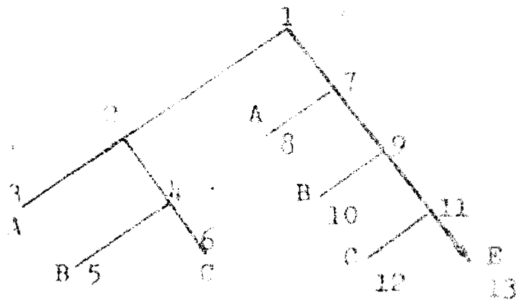
```
%R  VNORM IS A FUNCTION OF ONE ARGUMENT, A ONE DIMENSIONAL
%R  ARRAY OF REAL NUMBERS, TO BE CONSIDERED AS A VECTOR.  THE
%R  VECTOR IS NORMALIZED TO A UNIT VECTOR IN THE SAME DIRECTION.
%R  THE VALUE OF VNORM IS THE NORMALIZED VECTOR.
SYMBOL FUNCTION VNORM (A) REAL ARRAY A:
   BLOCK (N ← ARRAYSIZE (A)) INTEGER N:
    BLOCK (B ← MAKEARRAY (N, 'REAL), L ← VLENGTH(A))
     REAL L; REAL ARRAY B:
     FOR M STEP -1 UNTIL < 1:
          B(M) ← A(M)/L;
     RETURN B
    END
   END,


%R  VLENGTH COMPUTES THE LENGTH OF ITS ARGUMENT.  A
%R  VECTOR REPRESENTED AS AN ARRAY OF REAL NUMBERS.
REAL FUNCTION VLENGTH (A) REAL ARRAY A:
     BLOCK (L, N ← ARRAYSIZE (A)) REAL L; INTEGER N:
        FOR N STEP -1 UNTIL < 1:
           L ← L + A(N) ↑ 2;
        RETURN SQRT(L)
     END,

%R  THE FOLLOWING IS AN EXAMPLE
VNORM ([REAL -2.0 1.0 2.0]);

%R  THE COMPUTER RESPONDS
[REAL -.66666  .33333  .66666]
```

An example of LISP 2 function illustrating the use of functional arguments is the function TESTR. The first argument, X, is an S-expression. The second argument, P, is a function argument which is a predicate of an S-expression. That is, P(X) is either TRUE or FALSE depending on the value of X. The third argument, U, is also a functional argument, and is used to control the flow of the recursion. The function TESTR examines the nodes of the S-expression X one at a time, and at each node asks if the predicate P is TRUE for that node. The value of TESTR (X,P,U) is the first node satisfying P. If all the nodes fail, then the value of TESTR (X,P,U) is NIL. The order of examination of the nodes is illustrated in the following example:



This tree is represented in s-expression notation as

((A . (B . C)) . (A . (B . (C . E))))

or

((A B . C) A B C . E)

Suppose we execute

```
Q ← TESTR ('((A B . C) A B C . E), P, symbol funarg ( ) NIL);
```

If P(X) is TRUE only when car X = 'B then Q will have the value ((B . C)).
If P (X) is TRUE only when X = 'D then Q will have the value NIL.

```
SYMBOL FUNCTION TESTR (X,P,U) SYMBOL X;
              BOOLEAN FUNCTIONAL (SYMBOL) P;
              SYMBOL FUNCTIONAL ( ) U:
    IF P (X) THEN LIST (X)
ELSE IF NOT NODE (X) THEN U ( )
ELSE TESTR (CAR X, P, SYMBOL FUNARG ( ) TESTR (CDR X, P, U));
```

An alternate formulation of the function TESTR is the function TESTR2 of two arguments.

```
SYMBOL FUNCTION TESTR2 (X, P) SYMBOL X;
                BOOLEAN FUNCTIONAL (SYMBOL) P:
     IF P (X) THEN LIST (X) ELSE IF NOT NODE (X) THEN NIL
    ELSE BLOCK (Y ← TESTR2 (CAR X, P)) SYMBOL Y:
               IF NULL Y THEN Y ← TESTR2 (CDR X, P);
               RETURN Y
          END;
```

These two versions are computationally equivalent, in the sense that with the same arguments, they product the same answer.

```
TESTR (X, P, SYMBOL FUNARG ( ) NIL) = TESTR2 (X, P)
```

```
%R  See Page 7 for comments
SYMBOL SECTION TEST:
DECLARE (EXP) FLUID EXP:

FUNCTION COMPILER (EXP) FLUID EXP:
     IF ATOM EXP THEN LIST (LIST ('LOAD, EXP))
       ELSE BLOCK (OP ← CAR EXP)
             RETURN IF OP = 'PLUS THEN COMAD ( ) ELSE
                       IF OP = 'MINUS THEN COMIN () ELSE
                       IF OP = 'TIMES THEN COMPY ( ) ELSE
                       IF OP = 'DIVIDE THEN COMQUO ( ) ELSE ERROR ( )

            END,

FUNCTION COMIN ( ): APPEND (COMPILER (CADR EXP) , ' ((COMPLIMENT))),

FUNCTION COMQUO ( ):
     APPEND ( APPEND (COMPILER (CADDR EXP), '((STORE PUSH.))),
              APPEND (COMPILER (CADR EXP), '((DIV POP . )))),

FUNCTION COMAD ( ): COMAR ('ADD),

FUNCTION COMPY ( ): COMAR ('MPY),

FUNCTION COMAR (INSTRUCTION):
     BLOCK (LISTING ← COMPILER (CADR EXP), X):
         FOR X IN CDDR EXP:
             LISTING ← APPEND (APPEND (LISTING, '((STORE PUSH.))),
                         APPEND (COMPILER(X), LIST (LIST (INSTRUCTION, 'POP . ))));
         RETURN LISTING

     END;
```

COMPILER compiles simple arithmetic expressions. The only relevant point in this program is the handling of the variable EXP, a FLUID variable. Note that many functions reference EXP which is not an argument. For each level of the recursion a new EXP is "created." Work through the code with the following example and see if you understand the use of EXP:

```
(PLUS (TIMES A B) (TIMES C (PLUS D E)))

       A*B + C*(D + E)
```

The list notation for this example is called Polish Prefix notation and is the form of input to COMPILER. The value of COMPILER is:

```
(( LOAD A)
 (STORE PUSH. )
 (LOAD B)
 (MYP POP.)
 (STORE PUSH.)
 (LOAD C)
 (STORE PUSH.)
 (LOAD D)
 (STORE PUSH.)
 (LOAD E)
 (ADD POP.)
 (MPY POP.)
 (ADD POP.))
```

PUSH. and POP. are convenient notations for using a push down stack. PUSH. creates a temporary storage location on the PDS. POP. references the last location created by a PUSH. (and not already popped) and releases it. Many assemblers give such a capability for addressing a PDS.

```
%R LCS FINDS THE LONGEST COMMON SEGMENT OF TWO LISTS
SYMBOL FUNCTION LCS(X, Y) SYMBOL X, Y:
  BLOCK (CS ← ( ), X1, X1, MAXL ← 0) SYMBOL CS, X1, Y1; INTEGER MAXL:
     FOR X1 ON X WHILE LENGTH(X1) > MAXL:
      FOR Y1 ON Y WHILE LENGTH (Y1) > MAXL:
       BLOCK (X2 ← X1, L ← 0) SYMBOL X2; INTEGER L:
        FOR L STEP 1 WHILE NOT NULL X2 AND NOT NULL Y1
                              AND CAR X2 = CAR Y1:
          DO X2 ← CDR X2;
             Y1 ← CDR Y1;
          END;
        IF L > MAXL THEN DO MAXL ← L;
                             CS ← X1;
                          END;
        END;
     RETURN FIRSTN(CS, MAXL)
 END,

SYMBOL FUNCTION FIRSTN(X, L) SYMBOL X; INTEGER L:
  IF L = 0 THEN ( ) ELSE CAR X . FIRSTN (CDR X, L - 1),

%R THE FOLLOWING IS AN EXAMPLE
  LCS ( '(A B C D E F), '(A C D E F A B));

%R THE COMPUTER RESPONDS
  (C D E F)
```

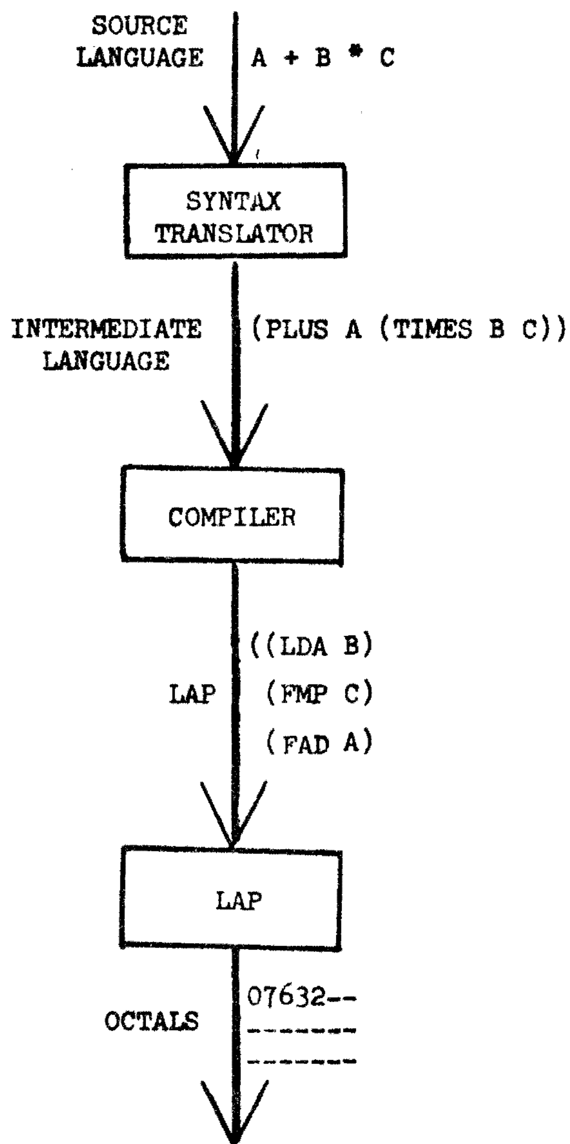
```
FUNCTION NUMLIST(L, N) FLUID INTEGER N; SYMBOL L: WORKER(L),

FUNCTION WORKER(M)
      IF NULL M THEN NIL
         ELSE LIST(CAR M, N ← N+1) . WORKER(CDR M),

%R THE FOLLOWING IS AN EXAMPLE
   NUMLIST ( '(A B C D), 0)

%R THE COMPUTER RESPONDS
   ((A 1) (B 2) (C 3) (D 4))
```

Note the use of the fluid variable N for communication between NUMLIST and WORKER.


SOURCE
LANGUAGE
A + B * C
SYNTAX
TRANSLATOR
INTERMEDIATE
LANGUAGE
(PLUS A (TIMES B C))
COMPILER
((LDA B)
LAP
(FMP C)
(FAD A)
LAP
07632--
OCTALS

S-EXPRESSION SIMPLIFICATION

$(S_1 \ . \ (S_2 \ . \ S_3)) \rightarrow (S_1 \ S_2 \ . \ S_3)$

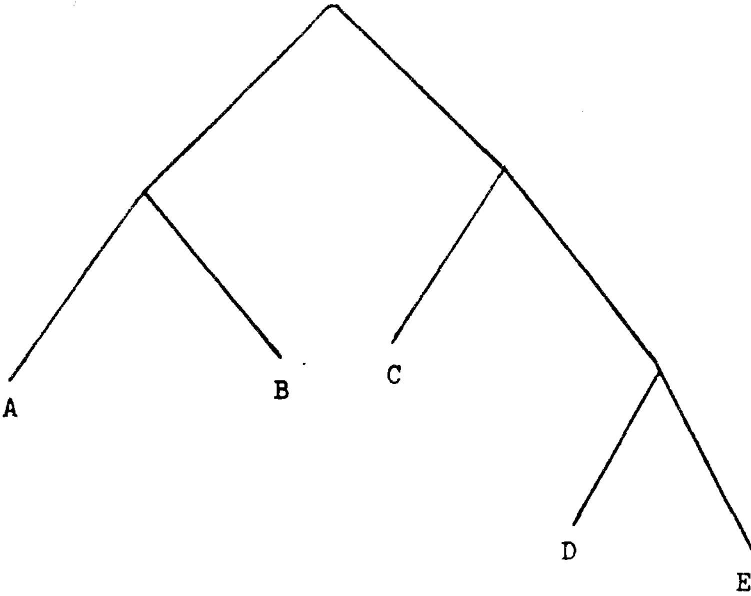
$(S \ . \ ()) \rightarrow (S)$

```
(a  .  (b  .  (c  .  (d  .  ())))))
(a     b   .  (c  .  (d  .  ()))))
(a     b   c  .   (d  . () ))
(a     b   c   d   . () ))
(a     b   c   d   )
```

a, b, c, d are S-expressions

() ≡ NIL is the empty list (<u>also an atom</u>)

BINARY TREE



BINARY TREE