BALM—An extendable list-processing language*

by MALCOLM C. HARRISON

New York University
New York, New York

INTRODUCTION

The LISP 1.5 programming language¹ has emerged as one of the preferred languages for writing complex programs,² as well as an important theoretical tool.³,⁴ Among other things, the ability of LISP to treat programs as data and vice versa has made it a prime choice as a host for a number of experimental languages.⁵,⁶ However, even the most enthusiastic LISP programmers admit that the language is cumbersome in the extreme.

A couple of attempts⁷,⁸ have been made to permit a more natural form of input language for LISP, but these are not widely available. The most ambitious of these, the LISP 2 project, bogged down in the search for efficiency.

The system described here is a less ambitious attempt to bring list-processing to the masses, as well as to create a seductive and extendable language. The name BALM is actually an acronym (Block And List Manipulator) but is also intended to imply that its use should produce a soothing effect on the worried programmer.

The system has the following features:

1. An Algol-like input language, which is translated into an intermediate language prior to execution.
2. Data-objects of type list, vector and string, with a simple external representation for reading and printing and with appropriate operations.
3. The provision for changing or extending the language by the addition of new prefix or infix operators, together with macros for specifying their translation into the intermediate language.
4. Availability of a batch version and a conversational version with basic file editing facilities.

The intermediate language is actually a form of LISP 1.5 which has been extended by the incorporation of new data-types corresponding to vector, string and entry-point. The interpreter is a somewhat smoother and more general version of the LISP 1.5 interpreter, using value-calls rather than an association-list for looking up bindings, and no distinction between functional and other bindings. The system is implemented in a mixture of Fortran (!) and MLISP, a machine-independent macro-language similar to LISP which is translated by a standard macro-assembler. New routines written in Fortran or MLISP can be added by the user, though if Fortran is used a certain amount of implementation knowledge is necessary.

The description given here is of necessity incomplete because of the flexible nature of the system. In practice it is expected that a number of different dialects will evolve, with different sets of statement forms, operators, and procedures. What is described here is a fairly natural implementation of basic features of the intermediate language which will probably form the basis from which other dialects will grow. We will illustrate the facilities by example rather than by giving a formal description, which can hopefully be obtained from the manual.

OVERVIEW OF BALM FEATURES

A BALM program consists of a sequence of commands separated by semi-colons. Each command will be executed before the next one is read. The user can submit his program either as a deck of cards, or type it in directly from a teletype. When submitted as a card deck, any data required by the command should follow the command immediately, and on the output a listing of the cards will appear, interspersed with any printed output resulting from a command. When a teletype is used, just the output requested will appear.

Variables in BALM do not have a type associated with them, so each variable can be assigned any value. The command:

```
A = 1.2;
```
would assign the value 1.2 to A, while:

```plaintext
PRINT(A);
```

would print out:

```
1.2
```

Arithmetic operations are expressed in the usual way, so:

```plaintext
X = 2 * A + 3; PRINT(X);
```

would print:

```
5.4
```

Automatic type conversion is done where necessary. A " symbol is used to allow the input of lists. Thus:

```plaintext
A = "(A(B C)D);
PRINT(HD TL A);
```

would print:

```
(B C)
```

The prefix operators HD and TL have the same effect as the functions CAR and CDR in LISP, giving respectively the first element of a list and the list without its first element. The LISP CONS operator is available either as a procedure, or as an infix colon associating to the right. Thus:

```plaintext
A = "A:"(B C):"D:NIL;
```

would also assign the list "(A(B C)D) to A.

Vectors can be input in a notation similar to that for lists, but using square brackets instead of parentheses. Elements of vectors are accessed by indexing. Thus:

```plaintext
V = "[A[B C]]D; PRINT(V[2]);
```

would print:

```
[B C]
```

Lists can be members of vectors, and vice versa, so:

```plaintext
PRINT(TL"(A(B C)D));
```

would print:

```
((B C) D)
```

while:

```plaintext
PRINT("[A (B C) D][2]);
```

would print:

```
(B C)
```

A non-rectangular matrix can be expressed as a vector of vectors:

```plaintext
W = "[[1][2 3][4 5 6]];
```

and elements can be extracted by indexing. Thus:

```plaintext
PRINT(W[2]);
```

would print:

```
[2 3]
```

Any expression can be indexed so that repeated indexing can be used to extract elements of matrices. Thus:

```plaintext
PRINT(W[2][1]);
```

would print out:

```
2
```

Assignments to vector elements are straightforward:

```plaintext
W[2][1] = "(A B);
```

A whole vector or list can be assigned from one variable to another variable in a single assignment, of course, but then any operation which changes a component of one will change a component of the other. If this is not desired, the vector or list should be copied before the assignment:

```plaintext
Z = COPY (W);
```

subsequent changes to Z will then not affect W.

An arbitrary structure can be broken up into its constituent parts by the procedure BREAKUP. This takes two arguments, a structure whose elements are constants or variables, and a structure to be broken up. Parts of the second structure corresponding to variables in the first structure are assigned as the values of those variables, while constants must match. If the structures cannot be matched, the BREAKUP procedure is terminated and gives the value NIL. Otherwise it has the value TRUE. For example:

```plaintext
BREAKUP ("(A B), "((C C) (D D)));
```

will give the value TRUE and will assign "(C C) to A and "(D D) to B. Either structure can involve vectors, and constants in the first structure are specified by preceding them with the quote mark ("). Thus:

```plaintext
BREAKUP ("[A B C], "[[X X] B [Y Y]]);
```

will have the value TRUE and will assign "[X X] to A and "[Y Y] to B. The converse of BREAKUP is CONSTRUCT, which is given a single structure whose elements are variables, and which will construct the same structure but with variables replaced by their values. Thus:

```plaintext
X = "; Y = [C D];
```

```plaintext
PRINT(CONSTRUCT ("(X Y)));
```

will print ((A B) [C D]). These two procedures allow convenient forms such as:

```plaintext
IF BREAKUP ("A + B), X THEN RETURN
(CONSTRUCT("A B "PLUS"));
```

BREAKUP and CONSTRUCT are quite efficient,
and should be used in preference to the more primitive operations whenever possible.

Character strings of arbitrary length can be specified:

\[ C = \text{EXAMPLE OF A STRING}; \]

They can be concatenated, or have substrings extracted. Thus:

\[ D = C \rightarrow \text{AND ANOTHER}; \]
\[ E = \text{SUBSTR(D,9,4)}; \text{PRINT(E);} \]

would print

\[ \text{OF A} \]

The BALM system allows the user to assign properties to variables. A property consists of a name and a value. For example, the command:

\[ \text{VAR PROP "ABCD = (STR);} \]

assigns the property called ABCD with an associated value of (STR) to the variable VAR. Similarly:

\[ X = \text{VAR PROP "ABCD;} \]

will set the value of X to the value of the property ABCD of variable VAR. A variable can have any number of properties and any number of variables can have the same property.

There is complete garbage collection of all inaccessible objects in the system, so the user does not need to keep track of particular lists or vectors. Procedures are available for creating lists or vectors with values of expressions as their elements, with storage being allocated dynamically:

\[ LL = \text{LIST(Z + Q, ABC, S(XY)}); \]
\[ VV = \text{VECTOR(X + W, ABC, S(XY)}); \]

A procedure in BALM is simply another kind of data-object which can be assigned as the value of a variable. The variable can then be used to invoke the procedure in the usual way. The statement:

\[ \text{SUMSQ = PROC(X, Y), X \uparrow 2 + Y \uparrow 2 END;} \]

assigns a procedure which returns as its value the sum of the squares of its two arguments. The translator translates the PROC . END part into the appropriate internal form, which is assigned to SUMSQ. In fact this is simply a list, which could equally well have been calculated as the value of an expression. The procedure can subsequently be applied in the usual way. For example:

\[ \text{PRINT(5 + SUMSQ(2, 3) + 0.5);} \]

would print:

\[ 18.5 \]

Instead of assigning a procedure as the value of a variable, we can simply apply it, so that:

\[ X = 5 + \text{PROC(X, Y), X \uparrow 2 + Y \uparrow 2 END(2, 3) + 0.5;} \]

would assign \( 5 + 13 + 0.5 = 18.5 \) as the value of X. Note that a procedure can accept any data-object as an argument, and can produce any data-object as its result, including vectors, lists, strings and procedures. Thus it is possible to write:

\[ \text{M = MSUM(M1, MPROD(M2, M3)}); \]

where M1, M2, M3, and M are matrices. Procedures can be recursive, of course.

Analogous to procedures we can also compute with expressions. The statement

\[ E = \text{EXPR A + B END;} \]

would assign the expression \( A + B \), not its value, to E. Subsequently, values could be assigned to A and B, and E evaluated:

\[ A = 1; B = 2.2; V = \text{EVAL(E);} \]

EVAL(E) could also have been written as $E$.

A procedure is simply an expression with certain variables specified as arguments. The most useful expression for procedure definitions is the block, which is similar to that used in ALGOL, but can have a value. The statement:

\[ \text{REVERSE = PROC(L), BEGIN(X),} \]
\[ \text{COMMENT (FIRST TEST FOR ATOMIC ARGUMENT) } \]
\[ \text{IF ATOM(L) THEN RETURN(L),} \]
\[ \text{COMMENT (OTHERWISE ENTER REVERSING LOOP) } \]
\[ X = \text{NIL,} \]
\[ \text{COMMENT (EACH TIME ROUND REMOVE ELEMENT FROM L, REVERSE IT, AND PUT AT BEGINNING OF X) } \]
\[ \text{NXT, IF NULL(L) THEN RETURN(X),} \]
\[ \text{X = REVERSE(HD L) :X,} \]
\[ \text{RETURN L, GO NXT} \]
\[ \text{END END;} \]

shows the use of a block delimited by BEGIN and END in defining a procedure REVERSE which reverses a list at all levels. The COMMENT operator can follow any infix operator, and will cause the following data-item to be ignored.

As well as IF ... THEN ... statement there is an IF ... THEN ... ELSE ... as well as an IF ... THEN ... ELSEIF ... THEN ... etc. Looping statements include a FOR ... REPEAT ... as well as a WHILE ...
REPEAT ... A label should be regarded just as a local variable whose value is the internal representation of the statements following it. Accordingly, assignments to labels, and transfers to variables or expressions are legal, and can give the effect of a switch. A compound statement without local variables or transfers can be written DO ..., ..., END. Of course any of these statements can be used as an expression, giving the appropriate value. Note that a comma is used to separate statements and labels within a block and a compound statement. The semicolon is interpreted as an end-of command by the system (unless it occurs within a string), even if it occurs within parentheses or brackets. Any unpaired parentheses or brackets will be paired automatically, with a warning message being issued.

EXTENDABILITY

The TRANSLATE procedure used by BALM to translate statements into the appropriate internal form is particularly simple, consisting of a precedence analysis pass followed by a macro-expansion pass. Built-in syntax is provided only for parenthesized subexpressions, comments, the quote operator, the NOOP operator, procedure calls, and indexing. All other syntax information is provided in the form of three lists which are the values of the variables UNARYLIST, INFIXLIST, and MACROLIST. The user can manipulate these lists as he wishes, by adding, deleting, or changing operators or macros.

Operators are categorized as unary, bracket, or infix, and have precedence values, and a procedure (or macro) associated with them. Examples of unary operators are -, HD and IF, while infix operators include +, THEN, and =. Bracket operators are similar to unary operators but require a terminating infix operator which is ignored. Examples of bracket operators are BEGIN and PROC, which both can be terminated by the infix operator END.

New operators can be defined by the procedures UNARY, BRACKET, or INFIX. These add appropriate entries onto UNARYLIST or INFIXLIST. For example the statement:

UNARY("PR, 150, "PRINT");

would establish the unary operator PR with priority 150 as being the same as the procedure PRINT. Thus we could subsequently write PR A instead of PRINT(A). Similarly we could define an infix operator by

INFIX("\rightarrow, 49, 50, "APPEND");

to allow an infix append operation. The numbers 49 and 50 are the precedences of the operator when it is considered as a left-hand and right-hand operator respectively, so that an expression such as A → B → C will be analyzed as though it were A → (B → C).

The output of the precedence analysis is a tree expressed as a list in which the first element of each list or sublist is an operator or macro. For example, the statement:

SQ = PROC(X), X * X END;

would be input as the list:

(SQ = PROC (X) , X * X END)

and would be analyzed into:

(SETQ SQ (PROC (COMMA X (TIMES X X) ))))

This would then be expanded by the macro-expander, giving:

(SETQ SQ (QUOTE (LAMBDA (X) (TIMES X X) ))))

the appropriate internal form. This would then be evaluated, having the same effect as the statement:

SQ = "(LAMBDA(X) (TIMES X X) )";

which would in fact be translated into the same thing.

The macro-expander is a function EXPAND which is given the syntax tree as its argument. It is actually defined as:

EXPAND = PROC(TR),
BEGIN(Y),
IF ATOI1(TR) THEN RETURN(TR),
Y = LOOKUP(HD TR, MACROLIST),
IF NULL(Y) THEN RETURN
(MAPCAR(TR, EXPAND)),
RETURN (Y(TR))
END END;

That is, if the top-level operator is a macro, it is applied to the whole tree. Otherwise EXPAND is applied to each of the subtrees recursively. Most operators will not require macros because the output of the precedence analysis is in the correct form. However, operators such as IF, THEN, FOR, PROC ... etc. require their arguments to be put in the correct form for the interpreter. For instance, the IF macro can be defined:

MIF = PROC(TR),
BEGIN(X),
X = HD TL TR,
IF HD X = "THEN THEN RETURN
("COND: LIST(EXPAND(HD TL X),
EXPAND(HD TL TL X) ) :NIL),
RETURN("COND: EXPAND(X) )
END END;
where recursive calls to EXPAND are used to transform subtrees in the appropriate way. The statement:

MACRO("IF, MIF");

would associate the macro MIF with the operator IF.

We can think of this expansion procedure as top-down, in the sense that a higher level macro in the tree is expanded before a lower level macro. In fact the higher level macro can process the tree in any way. This may include not processing the tree at all (as is done by the QUOTE macro), or expanding selected subtrees in a standard or non-standard way. A macro can even act as a translator of a special-purpose sub-language which is quite different from BALM. For example, the expression:

SNOBOL "(X (ABC) ARB Y PP(I) = Y :F(FAIL))"

is perfectly legitimate in BALM, and could be translated into the appropriate internal form by a macro associated with the prefix operator SNOBOL.

One particular outcome of this expansion procedure is the ability to write other than simple variables on the left-hand-side of assignment statements. These are conveniently handled by a macro associated with the assignment operator which checks the expression on the left-hand-side and modifies the syntax tree accordingly. It is this mechanism which permits an element of a vector to appear on the left-hand-side, and also such statements as:

\[ HD \ X = Y; \]

which will be translated as though it had been written:

RPLACA(X, Y);

The assignment macro currently in use looks up the top level operator found on the left-hand-side in a list LMACROLIST, applying any macro associated with the operator to the tree representing the assignment statement. The set of expressions which can be handled on the left-hand-side can easily be extended by adding entries to LMACROLIST. For example:

LMACRO("PROC,MPROC");

could be used to add the left-hand-side macro MPROC to permit assignments such as:

PROC PPP(X,Y) = EXPR ... END;

as an alternative way of defining a procedure.

Note that the essential properties of the system are those of the intermediate language, the most important of which is its ability to treat data as program, and thus to preprocess its program. Even the TRANSLATE procedure described above can be ignored and the user’s own translator substituted. Of course this will require a different level of expertise on the part of the programmer than simply the addition of new operators. However, the translator, which takes about 2000 words on the CDC 6600, is only about 250 cards, and quite straightforward, so this is not an unlikely possibility.

In summary, the BALM system permits extensibility in a number of different ways:

1) By addition of user-defined procedures.
2) By the definition of unary or infix operators.
3) By the definition of macros.
4) By the use of a user-defined translator.

Procedures, macros and translators can be written with the full power of BALM, or in MLISP or assembly language.

ACKNOWLEDGMENTS

Much of the coding of the current version of BALM has been done by Douglas Albert and Jeffrey Rubin, both of whom have made substantial contributions to its development.

REFERENCES

1 J McCARTHY et al
Lisp 1.5 programmers manual
MIT Press 1962
2 M MINSKY
Semantic information processing
MIT Press 1968
3 J PAINTER
Semantic correctness of a compiler for an Algol-like language
A I Memo 44 Stanford University 1967
4 P LANDIN
The mechanical evaluation of expressions
Computer Journal January 1964
5 D BOBROW J WEIZENBAUM
List processing and extension of language facility by embedding
IEEE Trans on Elec Comp EC-13 Aug 1964
6 C ENGLEMAN
Mathlab—A program for on-line machine assistance in symbolic computations
Proc FJCC 1965
7 L P DEUTCH
9.0 LISP reference manual
University of California Berkeley February 1966
8 P ABRAHAMS et al
The Lisp 2 programming language and system
Proc FJCC 1966
9 M HARRISON
BALM users manual
Courant Inst Math Sci New York University