NYO-1480-118

Courant Institute of
Mathematical Sciences

AEC Computing and Applied Mathematics Center

BALM — An Extendable List-processing Language

Malcolm C. Harrison

AEC REsearch and Development Report

Mathematics and Computers
June 1969

New York University

AEC Computing and Applied Mathematics Center
Courant Institute of Mathematical Sciences
New York University

Mathematics and Computers NYO-1480-118

BALM — An Extendable List-processing Language

Malcolm C. Harrison

Contract No. AT(30-1)-1480

ABSTRACT

This paper describes an extendable list-processing system currently implemented on the CDC 6600 which includes the facilities provided by LISP 1.5, and permits the programmer to write in an Algol-like language. Additional data-types include vectors, strings, and entry-points. The language can be extended by adding new operators, and by specifying how expressions involving them should be translated into an intermediate language.

1. Introduction.

The LISP 1.5 programming language[1] 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 attempts[7,8] 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-cells 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 versions 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 languages will grow. We will illustrate the facilities by example rather than by giving a formal description, which can hopefully be obtained from the manual.[9]

2. Overview of BALM features.

Variables in BALM do not have a type associated with them, so each variable can be assigned any value. If we imagine the user sitting at a teletype the following conversation could ensue:

```
-A = 1.2;
-PRINT(A);
1.2
```

Lines starting with a dash are requests for the user to type. The third line is the result of the PRINT command. The usual notation is used for arithmetic operations:

```
-X = 2 * A + 3; PRINT(X);
5.4
```

with a quote symbol being used to allow the input of lists:

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

The operators HD and TL are synonymous with CAR and CDR in LISP. 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:

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

Lists can be members of vectors, and vice versa:

```
-PRINT(TL"(A [B C] D));
([B C] D)
-PRINT(+[A (B C) D][2])
(B C)
```

Arrays can be input as vectors of vectors, so a non-rectangular matrix could be written

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

and elements can be extracted either by the usual type of indexing

```
-PRINT(W[2]);
[2 3]
```

or by repeated indexing

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

Assignments to vector elements are straightforward;

```
-W[2][1] = "(A B); PRINT(W[2]);
[(A B) 3]
```

A whole vector or list can be assigned from one variable to another variable in a single statement, 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:

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

Character strings of arbitrary length can be specified:

```
-C = <EXAMPLE OF A STRING>;
```

They can be concatenated, or have substrings extracted:

```
-D = C CAT <AND ANOTHER>;
-E = SUBSTR(D,9,12); PRINT(E);
<OF A>
```

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 = LIST(X+W, "ABC, S CAT<XY>);
-VV = VECTOR(X+W, "ABC, S CAT<XY>);
```

The equivalent of the LISP CONS operator can be written as an infix colon associating to the right, so that the first of the above statements could also have been written

```
-LL = X+W: "ABC : S CAT<XY>: NIL;
```

Note that NIL is the empty list.

A procedure in BALM is simply another kind of data-object which can be assigned as the value of a variable. Machine-language procedures are specified by a name (as known to the loader) followed by 0> or 1> depending on whether they should have their arguments evaluated or not. Thus

```
-FIRST = CAR0>;
```

will assign the value of the variable FIRST as being the machine-language routine whose name is CAR. In fact, since the value of the variable CAR is also CAR0> this can

also be accomplished by

-FIRST = NOOP CAR;

where we have used NOOP to indicate that the subsequent operator name CAR should be regarded as a variable. Either CAR, CAR(), or FIRST can subsequently be used to invoke this procedure.

A programmer-defined procedure is normally represented within the system in the form of a list, and is executed interpretively when invoked. The usual way of defining a procedure is to assign it as the value of a variable:

-SUMSQ = PROC(X,Y),X↑2+Y↑2 END;

The translator translates the PROC...END part into the appropriate list, which would have the same effect as if we had written

-SUMSQ = "(LAMBDA(X Y)(PLUS (EXPT X 2)(EXPT Y 2)));

Of course we could equally well have produced this list as the value of some expressions.

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

-X = 5 + PROC(X,Y),X↑2+Y↑2 END(2,3) + 0.5;

would assign 5 + 13 + 0.5 = 18.5 as the value of X. Note that procedures 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

-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 = 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 = EVAL(E);
```

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

A procedure is essentially defined as an expression; for more complicated procedures, more complicated expressions can be used. The most important of these is the block, which is similar to that used in Algol, but can have a value. The statement:

```
-REVERSE = PROC(L), COMMENT <REVERSES A LIST>
-          BEGIN(X), COMMENT <X IS LOCAL VARIABLE>
-          COMMENT <FIRST TEST FOR ATOMIC ARGUMENT>
-          IF ATOM(L) THEN RETURN (L),
-          COMMENT <OTHERWISE ENTER REVERSING LOOP>
-          X = NIL,
-          COMMENT <EACH TIME ROUND REMOVE ELEMENT FROM L,
-            REVERSE IT, AND PUT AT BEGINNING OF X>
- NXT,     IF NULL(L) THEN RETURN(X),
-          X=REVERSE(CAR X): COMMENT <: IS INFIX CONS
-            OPERATOR>X,
-          L = CDR L, GO NXT,
-          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.

As well as the 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.

3. 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 aprenthesized 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 - (minus), CAR, 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("→,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 ATOM(TR) THEN RETURN(TR),
-         Y = LOOKUP(CAR TR,MACROLIST),
-         IF NULL(Y) THEN RETURN (MAPCAR(EXPAND,TR)),
-         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 = CAR CDR TR,
-      IF EQ (CAR X, "THEN) THEN RETURN
-         ("COND: LIST(EXPAND(CAR CDR X),
          EXPAND(CAR CDR CDR 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.

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:

```
-CAR(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 the 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 the procedure:

```
-LMACRO("PROP,MPROP);
```

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

```
-PROP("X,"P) = "V;
```

which establishes the value V for property P of atom X.

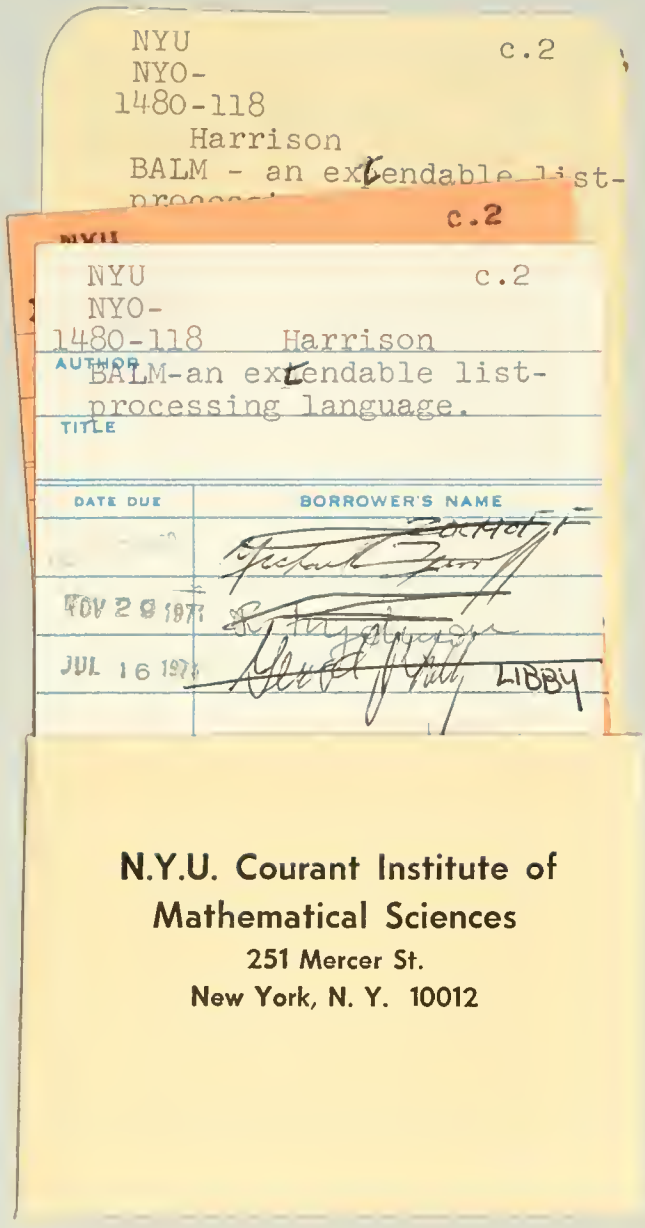
Note that the essential properties of the system are those of the intermediate language, the most important of which is its ability to treat data as program, and thus to preprocess its program. Even the TRANSLATE procedure described above can be ignored and the user's own translator substituted. Of course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

We have not yet had much experience with the extendability features, but anticipate that we will be able to add the equivalent of the PL/1 and Algol 68 structures (as vectors with named components), and at least some of the flavor of the Snobol pattern match and substitution rule. At the very least we will have a very flexible experimental language with powerful list-processing facilities.

The translator currently takes of the order of 2000 words in the CDC 6600, and we do not expect this to increase much, if at all.

References

1. J. McCarthy et al., Lisp 1.5 Programmers Manual, MIT Press, 1962.
2. M. Minsky, Semantic Information Processing, MIT Press, '68.
3. J. Painter, Semantic Correctness of a Compiler for an Algol-like Language, A.I. Memo 44, Stanford Univ., '67.
4. P. Landin, The Mechanical Evaluation of Expressions, Computer Journal, Jan. 1964.
5. D. Bobrow and J. Weizenbaum, List Processing and Extension of Language Facility by Embedding, IEEE Trans. on Elec. Comp., EC-13, Aug. '64.
6. C. Engelman, Mathlab - A Program for On-line Machine Assistance in Symbolic Computations, Proc. FJCC '65.
7. L. P. Deutch, 940 LISP Reference Manual, Univ. Cal. Berkeley, Feb. '66.
8. P. Abrahams et al., The Lisp 2 Programming Language and System, Proc. FJCC '66.
9. M. Harrison, BALM Users Manual, Courant Inst. Math. Sci., New York Univ. (in preparation).



N.Y.U. Courant Institute of Mathematical Sciences
251 Mercer St.
New York, N. Y. 10012