# 19 BALM

In this chapter we will describe in some detail an extendable language developed by the author and his colleagues at the Courant Institute. This is somewhat less ambitious than ALGOL 68, and has somewhat different aims. First of all, it attempts to provide the programmer with more extensive control over the translation of his program than is provided in ALGOL 68. Although the facilities of ALGOL 68 are powerful, in some cases it is necessary to permit the programmer to have access to the translator itself, so that he can make more fundamental modifications to its operation. For this reason we felt that the translator should be made sufficiently simple to permit the experienced user to understand its operation without undue effort. This required sacrifices in efficiency in a number of ways, but we felt that in a number of cases this would not be too much of a restriction. A listing of a version of the system, written in BALM, is given in Appendix B.

## SYSTEM ORGANIZATION

In the BALM system the user is given control over the operation of the translator by making it one of the utility routines which form the basis of the system. The linkage to these routines is flexible, so that the user can replace the standard version of the translator, or any other utility routines, with his own. The system thus consists initially of a set of procedures, including translator and I/O routines and an executive routine, to which the user adds his own by executing commands which define procedures. These new procedures may be part of the users program, or they may be an addition to or a modification of the translator. Thus there is no real distinction between the users' program and the translator—both are written in BALM. The overall organization is illustrated below in Figure 19.1. A BALM program consists of a sequence of commands each one of which is translated and executed before the next one is read, thus permitting the execution to modify the translator.

In order to permit the user strong control over the translation process, while not requiring him to get too deeply into representations of his program in machinelike languages, an intermediate language is used in the BALM system. This is a well-defined simple but powerful language with a very simple syntax, which makes it easy to

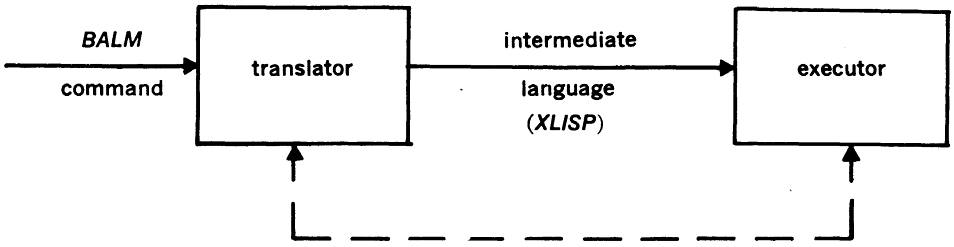

Figure 19.1

process when necessary. Thus the user who wanted to, say, print out the new value assigned to selected variables would be able to add a routine to the compiler to process the intermediate form of the language to make the necessary modifications to the code. The intermediate language is a slightly modified form of LISP, which we call XLISP.

The system has been implemented in several different ways. In the original implementation the executor was a slightly modified version of the standard LISP interpreter. In subsequent implementations a compiler was added to the executor. However, language extensions are normally done to the part of the translator which translates from BALM into XLISP with the executor usually being kept fixed. Below we will concentrate mainly on the extendable part of the translator.

The version of BALM given in Appendix B is a compiler-based implementation with the compiler producing code for a hypothetical machine which we call the MBALM. The MBALM machine can be realized by microprogramming on machines such as the IBM 360 and the Standard 9000, or by simulation. MBALM simulators are available for the CDC 6600, IBM 360, Univac 1108, and other machines.

## DATA-STRUCTURES

Our objective in choosing data-structures was to include the minimum number which would permit the logical implementation of most common data-objects without too much inefficiency. To simplify garbage collection all data-structures are flagged with their type, so can be traced without requiring elaborate mapping information. A uniform format for a data-structure, or *item*, is used to permit handling all types similarly. This has the effect of permitting any variable in the language to have any item as its value, as in SNOBOL and LISP. The format chosen has two fields, a type field, and a pointer field, and in some implementations a third flag field for the user's use is also provided. In the case of some primitive items such as integers or logical quantities the pointer field is used to contain the information directly, but in the case of more elaborate data-structures the pointer field contains a pointer to other information.

The two main data-structures chosen were the *vector* and the *pair*. The first permits the use of arrays and indexing operations, while the second is essentially the basic data-structure of LISP and permits the construction of arbitrary branching structures. The elements of a vector or a pair can be arbitrary items, including other vectors and pairs, of course. From the logical point of view we could dispense with the pair and use a vector of length two, but in practice this would use up a little more space, and we felt that a distinction between vectors and lists (constructed from pairs) would more often than not be useful to the programmer. For the sake of economy of memory space we also provide a *string* of characters as a separate data-structure, stored sequentially in memory. Procedures represented as machine code are stored in a fourth data-structure called a *code block.* Vectors, strings, and code blocks may be of arbitrary size, but once allocated their length is fixed and stored in memory so that the programmer can determine the length if he requires.

To permit the user to manipulate programs, *identifiers* are also provided as a type of item. These can be thought of as represented by a pointer into a symbol table, in which is stored the name of the identifier, its current value, and another entry which we refer to as its property-list. These correspond very closely to the nonnumeric atoms of LISP, and similar operations are provided to manipulate them. In a similar way, there is a type of item called a *label,* which can be manipulated by a program to permit programmed switching of control.

## OUTLINE OF THE BALM LANGUAGE

As we have pointed out, BALM is an extendable language, so it is not possible to give a complete definition of it. However, the system, if not modified by the user, defines a "standard" BALM language which permits a fairly convenient use of most of the basic features of XLISP. It is this "standard" BALM which we will describe below. The reader should note that nearly all features of the "standard" language are translated according to three lists which guide the translator, and which can thus be changed if necessary.

A BALM program consists of a sequence of commands separated by semicolons. Each command is read, translated, and executed before the next one is read. The program can be submitted either as a deck of cards, or typed in directly from a teletype terminal. In the first case, the card images will be printed, and will be followed by any printed output requested by the command. In the case of teletype input, only printed output will be typed. Note that a semicolon will always terminate a command, even if it occurs within parentheses or brackets (though not if it occurs as part of a string), so it acts as a sort of insurance that syntactic errors will not continue into the next command.

Commands in BALM include the usual type of assignment, written with an "equals" sign as in other languages. Thus

```
A = 12.3;
```

will assign to the variable A the value 12.3, while

```
B = C;
```

will assign to B whatever is currently the value of C. The right hand side of an assignment can be any expression, which can be written using infix notation and the usual functional notation for procedure applications. The usual arithmetic operators are available, and will accept numerical arguments of arbitrary type, doing execution-time conversion when necessary.

A command is just an expression which is evaluated, and whose value is ignored. An assignment command is simply an expression whose evaluation has the side effect of assigning the value of the right-hand side to the variable on the left, and as such it has a value, which in this case is the value of the right-hand side. Thus the command

```
A = B = 1.2;
```

will be executed as though it had been written:

```
A = (B = 1.2);
```

Here the right-hand side is an assignment, whose value is 1.2, which will thus be assigned to A as well as B. We will use the term "command" for such expressions merely to indicate this type of usage, rather than to imply some nonexistent distinction. Expressions such as assignments and transfers will usually though not always occur as commands, while those whose evaluation will produce no effect other than to calculate a value will not.

One use of commands will be as main components of a program as described above. However, they will also be used as the main constituent in programmer-defined functions, as we will see below. Expressions often used as commands include assignment, transfer, conditional, procedure invocations, and pattern matching and repetitive or looping operations.

A vector of *n* elements, but whose values are undetermined, can be written

MAKVECTOR(*n*)

This will dynamically allocate space for the vector, which will be collected automatically by the garbage collector when no longer accessible to the program. If the values of the vector are known, VECTOR can be used, so that the expression

```
VECTOR(1,2,3,4)
```

would represent a vector of length 4 whose elements were 1, 2, 3, and 4.

Indexing can be used to refer to the elements of a vector in the usual way, so that

$v[i]$

will refer to the $i$-th element of vector $v$, and

$v[j] = x$

will change the value of the $j$-th element of the vector $v$ to $x$. Here $v$ can be an arbitrary expression whose value is a vector, and $i$ and $j$ can be arbitrary expressions whose values are integers of appropriate size. This means that a two-dimensional array M represented as a vector of vectors can have its elements referred to as

```
M[i][j]
```

and so on. The length of a vector *v* is written SIZE *v*, and the expression VECTQ(*x*) will have the value TRUE if *x* is a vector and NIL otherwise.

The vector which is obtained by concatenating the two vectors *x* and *y* can be written

```
CONCATV(x,y)
```

A vector which has as its elements the *j* elements of the vector *v* starting at the *i*-th can be written

```
SUBV(v,i,j)
```

while these elements can be changed to the first *j* elements of the vector *w* by

```
SUBV(v,i,j)  =  w
```

A string can be converted to a vector whose elements are integers which are the internal representations of the characters in the string, and vice versa, by the operations

```
VFROMS(s)          SFROMV(v)
```

The operations on pairs are essentially those provided in LISP for manipulating lists. A pair whose first component is *x* and whose second component is *y* is written *x*:*y*, and if *p* is a pair the first component is HD *p* and the second component is TL *p*. A list whose elements are the numbers 1, 2, 3, 4, for example, could be written

```
(1:(2:(3:(4:NIL))))
```

or, since the precedence of the colon is arranged appropriately, as

```
1:2:3:4:NIL
```

Here NIL is the logical item used to represent false, and by convention is used to terminate a list. This list could also be written as

```
LIST(1,2,3,4)
```

Components of a pair *p* may be changed by commands of the form

```
HD p = x          TL p = y
```

but these should be used with caution, since the pair could be an element of other data-objects. The expression `PAIRQ(`*x*`)` will have the value `TRUE` if *x* is a pair, and `NIL` otherwise, and is the operation usually used for detecting the end of a list.

An identifier whose name is *n* is written as

"*n*

as long as *n* has the appropriate syntax for a name. In the current system names are sequences of letters and digits starting with a letter, or single special characters (with certain exceptions mentioned below). Note that "*n* does not refer to the value of the identifier named *n*, but to the data-structure which contains a pointer into the symbol table. If *id* is an identifier, the expression

`VALUE(`*id*`)`

gives the value of *id*, and

`VALUE(`*id*`) = `*x*

resets this value to *x*. Similarly

`PROPL(`*id*`)`

refers to the property-list of *id*, and

`PROPL(`*id*`) = `*x*

changes this property-list to *x*. The name of *id* is written

`SFROMID(`*id*`)`

which returns a string. If *s* is a string, then

`IDFROMS(`*s*`)`

refers to the identifier whose name is the characters of the string.

The quote can also be used for specifying constant vectors and lists. The notation used is essentially that of LISP. `VECTOR(1,2,3,4)`, for example, could be written as

```
"[1 2 3 4]
```

while `LIST(1,2,3,4)` could be written

```
"(1 2 3 4)
```

Nesting is permitted, so expressions such as

```
"[[1 2] 3 [ABC $] (ZZZ)]
```

are also permitted. Note that such expressions do not have memory allocated during execution time, but during translation time, so if assigned to a variable and then modified, the "constant" will also be modified.

The above notation is also the one used by the utility I/O routines READ, WRITE, and PRINT. PRINT($x$) will examine $x$ and print it on the standard output medium, without the leading quote. READ($f$) will read from file $f$ the next complete item in the above notation, continuing reading until brackets and parentheses are balanced.

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, and the copy assigned.

A list or vector can be broken up into its constituent parts by the procedure BREAKUP. This takes two arguments, an item whose elements are constants or variables, and an item to be broken up. Parts of the second structure corresponding to variables 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

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

will have the value TRUE and will assign (D D) 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

```
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 C. 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

```
X = "(A B); Y="[C D]; PRINT(CONSTRUCT("(X Y)));
```

will print ((A B) [C D]).

A procedure in BALM is simply another kind of item 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

```
SUMSQ = PROC(X,Y),X*2+Y*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. The procedure can subsequently be applied in the usual way, so

```
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 + 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 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. Procedures can be recursive, of course.

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

```
REVERSE = PROC(L),
          BEGIN(X),
          COMMENT 'FIRST TEST FOR ATOMIC ARGUMENT'
          IF ¬PAIRQ(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(HD L):,
          L = TL 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. The X following BEGIN indicates that X should be considered local to the block, while NXT is a label. The COMMENT operator can follow any infix operator, and will cause the following item to be ignored.

As well as an 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 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.

## USER-DEFINED LANGUAGE EXTENSIONS

The TRANSLATE procedure used by BALM to translate statements into XLISP is particularly simple, consisting of a precedence analysis pass followed by a macroexpansion pass. Built-in syntax is provided only for parenthesized subexpressions, comments, the quote operator, the unary operator NOOP, 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), HD, and IF, while infix operators include +, THEN, and ELSE. 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("AP,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 AP B AP C will be analyzed as though it were A AP (B AP 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. If the top-level operator of the syntax tree has a macro associated with it, the macro 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 execution. For instance, the IF macro, MIF, uses recursive calls to EXPAND to transform subtrees in the appropriate way. The statement

```
    MACRO("IF,MIF);
```

would associate the macro MIF with the operator IF.

One particularly useful outcome of this expansion procedure is the ability to write expressions on the left-hand side of assignment statements. These can be handled by a macro associated with the assignment operator, which tests for particular expressions on the left-hand side and makes appropriate modifications. For instance, the HD operator used on the left allows

```
    HD X = Y;
```

to be written instead of

```
    RPLACA(X,Y);
```

which the SETQ macro will in effect produce. Similarly, we can write

```
    MACRO(IF) = PROC(IF) ... ;
```

as a more concise way of defining the IF macro, as long as the SETQ macro were prepared for this left-hand side form.

To provide this flexibility, the macro for the assignment operator, MSETQ, looks up top-level operators on the left-hand side on the list LMACROLIST, which can be extended by the user. The statement

```
LMACRO(NAME,LMAC);
```

adds macros to LMACROLIST in a way analogous to MACRO.

## THE TRANSLATOR

The BALM-to-XLISP translator is very simple and uses the technique known as precedence analysis. The BALM language is designed in such a way that it consists of "phrases" and "operators," and each phrase is preceded and followed by an operator. In theory this imposes certain limitations on the language, but we feel that in practice these are not serious, and in fact are insignificant compared with the ease of the translation.

To give an example, the expression

```
IF A ≡ B+C THEN GO L
```

contains the operators IF, ≡, +, THEN, GO, and the phrases A, B, C, L. The analysis determines for each triple consisting of an operator, a phrase, and an operator, which operator has precedence. Thus in the expression

```
A = B + C
```

the + has higher precedence than the =, so the expression should be analyzed as

```
A = (B + C)
```

The bracketed version of the above statement would be

```
(IF ((A ≡ (B + C)) THEN (GO L)))
```

from which we see that the operators =, +, THEN require two arguments, and the IF and GO require a single argument. For consistency we will change the order of the bracketed elements whenever necessary so that the operator always comes first. Thus the final version of the above statement would be

```
(IF (THEN (≡ A (+ B C)) (GO L)))
```

We will refer to operators like THEN and + as infix, and to operators like IF and GO as unary.

A slight extension of this scheme is necessary to allow the use of functional notation such as

```
A = B + FF(C, D−E) + G
```

where FF is a function. In this case ( can be regarded as an infix operator, but ) does not fall into either the infix or unary class. Other similar examples are BEGIN and its associated END.

In the BALM version given in Appendix B, the parentheses will already have been removed by the BALM input routine, which will have made a sublist out of the intervening elements. The program will recognize such sublists and treat them accordingly. Note that if a simplified input routine which does not recognize parentheses is used, then only slight modifications need to be made to the program.

The function FNOTN which does this syntax analysis takes as its argument a list which is the program to be translated, and has as its value the tree structure which represents its syntax. The routine recognizes three types of operators, infix, unary, and bracket, and for each element on the list determines if it is such an operator. Associated with each operator is a list which gives appropriate information about it. Thus associated with the operator + is the list

(INFIX PLUS *lpr rpr*)

with PLUS being the atom which will be used in the tree structure, and *lpr* and *rpr* being the precedence of + when used as the left hand and right hand of two operators respectively.

The following input

```
LNGTH = PROC(X),
        BEGIN(U,V),
        U = 0, V = X,
LOOP, IF NULL(V) THEN RETURN(U),
        U = U+1, V = CDR(V), GO LOOP
        END
        END;
```

will be translated into

```
(SETQ LNGTH (LAMBDA (, X
 (PROG (, (, U V)
       (, (SETQ U 0)
       (, (SETQ V X)
       (, (LOOP
       (, (IF (THEN (NULL V) (RETURN U)))
       (, (SETQ U (PLUS U 1))
       (, (SETQ V (CDR V)) (GO LOOP)) ))))))) )))
```

which is easily translated into XLISP during macro-expansion.

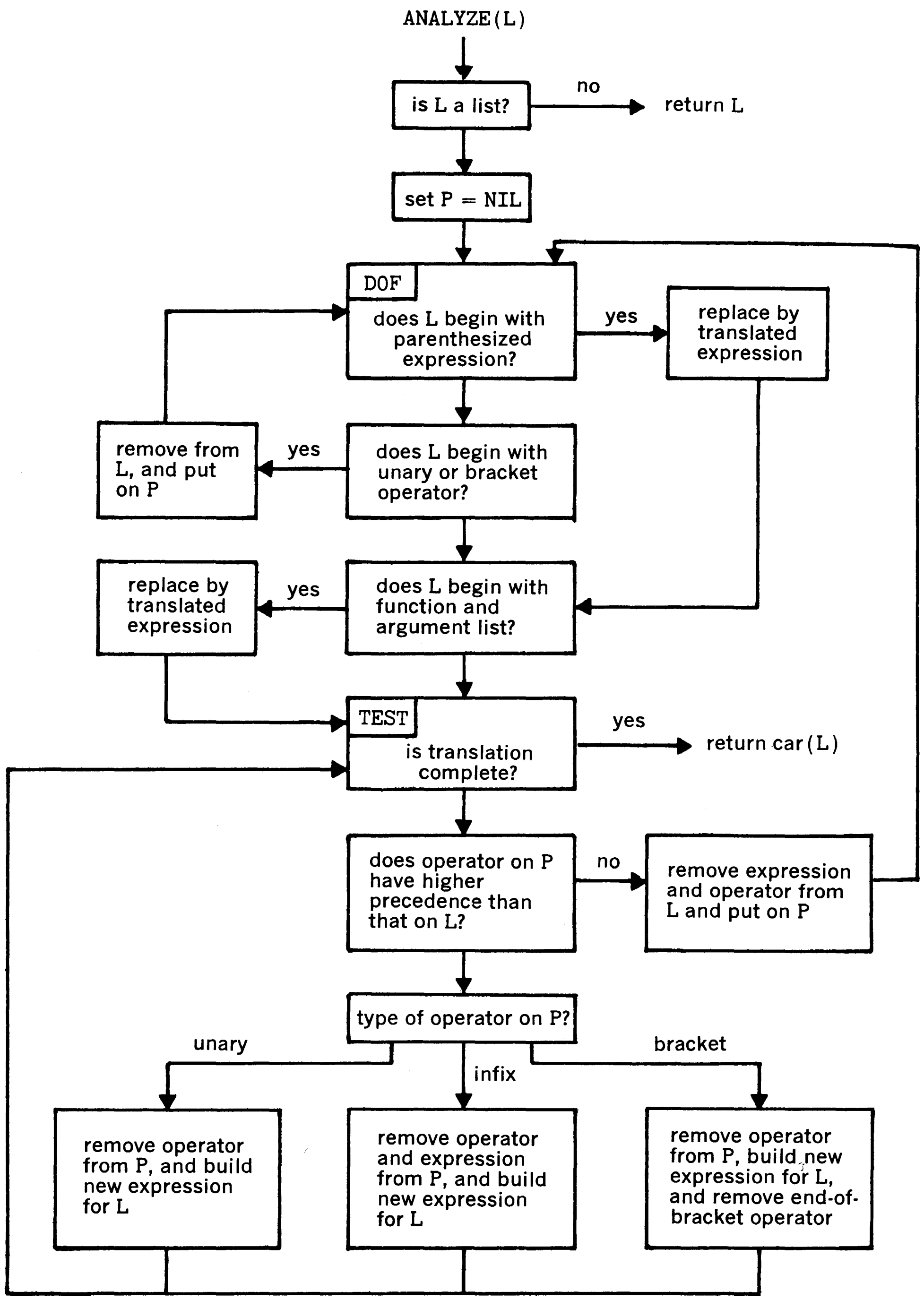
ANALYZE(L)
is L a list?
no
return L
set P = NIL
DOF
does L begin with parenthesized expression?
yes
replace by translated expression
yes
remove from L, and put on P
does L begin with unary or bracket operator?
replace by translated expression
yes
does L begin with function and argument list?
TEST
is translation complete?
yes
return car(L)
does operator on P have higher precedence than that on L?
no
remove expression and operator from L and put on P
type of operator on P?
unary
infix
bracket
remove operator from P, and build new expression for L
remove operator and expression from P, and build new expression for L
remove operator from P, build new expression for L, and remove end-of-bracket operator

Figure 19.2

## AN XLISP INTERPRETER

An XLISP interpreter can easily be written in BALM. It is a little different from the standard LISP interpreter. The main difference arises from the fact that only one form of binding is used. Rather than binding values of variables on an association list, and functions on the property list, all bindings are made in a cell associated with the variable called the *value cell.* This is more efficient than the association-list method, since access to the value cell for removal or insertion of bindings is immediate, and does not require any searching. When a procedure is entered, the current contents of the value-cells of local variables are saved and the new values inserted; the old values are restored on exit from the procedure.

A second difference is the fact that a machine-coded procedure in BALM is referenced by a recognizable data-type. This consists of a pointer to the first instruction of the procedure, together with flags to indicate what types of procedure it is. A function can be bound to a name simply by posting this data-object in the value cell, where it is accessible to the interpreter. Because of this uniform binding, the interpreter EVAL can look up the function bound to a name by simply evaluating it. This is conveniently generalized, so that when EVAL is given an argument of the form:

```
(FN ARG1 ARG2 ...)
```

the expression FN is evaluated, and the result, which should be either an entry point or a list structure representing a procedure such as a LAMBDA-expression applied to the arguments. This allows arbitrary expressions to be used as functions in an expression, but there are no tests for particular names in this position. Thus the form

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

is not legal since the (LAMBDA ...) expression will not evaluate correctly. Instead the form

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

should be used.

After evaluating the functional part of an expression the interpreter expects either a code block, which can be simply transferred to after processing the argument list, or an expression such as (LAMBDA ...) or (LABEL ...). These are conveniently evaluated by assuming them all to be of the form

($l$ $e_1$ $e_2$)

where $l$ evaluates to a function which expects $e_1$, $e_2$ and the arguments list as its arguments. Thus the evaluation of

```
((QUOTE (LL AL EX)) ARG1 ARG2 ...)))
```

will be the same as

```
    (LL AL EX (QUOTE (ARG1 ARG2 ...)))
```

This permits the user to provide his own versions of LAMBDA.

Three types of code block are used. These are called SUBR, FSUBR, and NSUBR, and can be distinguished by the setting of bits in the flag field. A SUBR, as in standard LISP, expects its arguments to be evaluated, and is the preferred type. An FSUBR expects a single argument which is the list of expressions supplied as arguments to the procedure, and, like LISP, is used to implement nonstandard operations such as SETQ, COND, and PROG. An NSUBR expects a single argument which is a list of the evaluated expressions supplied to the procedure as arguments. This is used for procedures which are standard except for the fact that they can have an arbitrary number of arguments, such as PLUS, LIST, VECTOR and PROGN. This last is a generalization of the LISP PROG2 procedure, and is used to implement the

```
    DO ... END
```

type of compound expression. In fact PROGN is simply a procedure which returns the last element of a list.

The EVAL function takes a single argument which is the expression to be evaluated. It could be coded in BALM as follows:

```
EVAL = PROC(X)
       BEGIN(FN,ARGS),
       IF IDQ(X) THEN RETURN VALUE(X),
       IF ¬PAIRQ(X) THEN RETURN(X),
       FN=EVAL(HD X), ARGS=TL X,
       IF SUBRP(FN) THEN RETURN (XEQ(FN,EVLIS(ARGS))),
       IF FSUBRP(FN) THEN RETURN(XEQ(FN,ARGS:NIL))
       IF NSUBRP(FN) THEN RETURN(XEQ(FN,EVLIS(ARGS):NIL)),
       IF ¬PAIRQ(FN) THEN ERROR(FN:"(IS NOT A FUNCTION)),
       RETURN(XEQ(EVAL(HD FN),HD TL FN:HD TL TL FN:ARGS:NIL))
       END END;
```

Here XEQ executes the machine-coded function specified by the code block given as its first argument, with the elements of the list given as its second argument as arguments. Note that the version of EVAL given above can be generalized by changing the last command from:

```
    RETURN(XEQ(EVAL(HD FN):HD TL FN:HD TL TL FN:ARGS:NIL))
```

to:

```
    RETURN(EVAL(HD FN:HD TL FN:HD TL TL FN:ARGS:NIL))
```

which would permit a LAMBDA-like function to be written as a LAMBDA-expression.
The procedure LAMBDA can be written in BALM as follows:

```
LAMBDA = PROC(VL,X,ARGL),
         BEGIN(PREVALS,RESULT),
         PREVALS=EVLIS(VL),
         BREAKUP(VL,EVLIS(ARGS)),
         RESULT=EVAL(X),
         BREAKUP(VL,PREVALS),
         RETURN(RESULT)
         END END;
```

Here we are using BREAKUP to do multiple assignment. The procedure EVLIS is the usual one defined as:

```
EVLIS=PROC(L),MAPX(L,EVAL)END;
```

## EXAMPLES

As an example of the extendibility facilities provided in BALM, we give below a set of routines which permits a convenient form for introducing new expressions or commands. The aim is to permit the user to specify the meaning of these forms without needing to know the characteristics of the intermediate language. This is done by accepting a definition in terms of BALM itself, in the form

*x* MEANS *y*

where *x* and *y* are expressions in BALM. For example, the ALGOL 68 form of conditional could be defined as

```
X1 | X2 | X3 MEANS IF X1 THEN X2 ELSE X3;
```

The method requires that all operators in the new expression be declared as such before the MEANS definition. X1, X2, X3 can be any valid expressions as long as the precedence of the operators is higher than that of the operator. As usual, parentheses can be inserted to ensure the appropriate parsing. Other examples include the following

```
STEP X1 MEANS X1=X1+1;
FOR X1 = X2 STEP X3 UNTIL X4 DO X5 MEANS
    FOR X1 = (X2,X3,X4) REPEAT X5;
```

These would require the previous definition of the precedences of the operators STEP(unary), STEP(infix), UNTIL(infix), and DO(infix).

The technique used to implement MEANS uses the fact that the parse tree of a command of the form *x* MEANS *y*, where MEANS is an infix operator of very low precedence, contains the parse trees which will result from the expressions $x$ and $y$. If, therefore, a macro is associated with MEANS, it can examine these trees and construct a macro which will transform any subtree which matches $x$ into the equivalent tree of the form $y$. This macro will be associated with the top-level operator in the tree of $x$. The code to define MEANS is given below. The procedure SUBST, not given, is assumed to substitute its first argument for occurrences of its second in its third. Note that the code is written to permit multiple macros to be associated with the same operator.

```
MMEANS = PROC(L),                          define procedure with argu-
                                             ment L
   BEGIN(LS,RS,M,OP,PREVM),                define local variables
   LS = HD TL L, RS = HD TL TL L,          extract operands of MEANS
   M=SUBST(LS,"L,TMAC),                    substitute for L and
   M=SUBST(RS,"R,M),                         R in TMAC
   OP = HD LS,                             extract top operator
   PREVM = LOOKUP(OP,MACROLIST),           retrieve any previous macro
   IF PREVM≡NIL THEN                         and substitute it or EXLIS
      M = SUBST("EXLIS,"E,M)                 for E in the modified TMAC
   ELSE M = SUBST(PREVM,"E,M),
   MACRO(OP,TRANSLATE(M)),                 associate new macro with op-
   RETURN NIL                                erator, and return
   END END;

TMAC="(PROC(S),                            define untranslated procedure
   BEGIN(X1,X2,X3,X4,X5,X6),                 to process tree S
   IF MATCH("L,S) THEN                     if tree matches L then rebuild
      RETURN BUILD("R)                       according to R
   ELSE RETURN E(S)                        otherwise process as before
   END END);

MATCH=PROC(L,S),                           define procedure to match
   IF PAIRQ(L) THEN                          trees
     (IF PAIRQ(S) THEN
      (IF MATCH(HD L,HD S) THEN            return false if trees don't
         MATCH(TL L,TL S)                    match
       ELSE FALSE)
     ELSE FALSE)
   ELSEIF L≡"X1 OR L≡"X2 OR L≡"X3          X1...X6 are assigned the
       OR L≡"X4 OR L≡"X5 OR L≡"X6            value of any corresponding
     THEN DO VALUE(L)=S,RETURN TRUE          subtree
                                   END
   ELSEIF L≡S THEN TRUE
   ELSE FALSE
   END;
```

```
BUILD=PROC(R),                            define procedure to rebuild
   IF PAIRQ(R) THEN                         tree
     BUILD(HD R);BUILD(TL R)
   ELSEIF R≡"X1 OR R≡"X2 OR R≡"X3         replace X1...X6 with their
       OR R≡"X4 OR R≡"X5 OR R≡"X6           current values
     THEN VALUE(R)
   ELSE R
   END;

INFIX("MEANS,0,0,"MEANS);                 define MEANS as infix operator
MACRO("MEANS,MMEANS);                     define macro for MEANS
```