MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

Artificial Intelligence
Memo No. 99

Memorandum MAC-M-316
June 1966

CONVERT

Adolfo Guzman and Harold V. McIntosh

A pattern-driven symbolic manipulation language
and its processor (an interpreter) are presented.

Presented at an ACM Symposium on Symbolic and Algebraic Manipulation, held March 29-31, 1966, in Washington, D.C.

To be published in Communications of the ACM, August 1966.

TABLE OF CONTENTS

Introduction 1

RESEMBLE 6

REPLACE 15

CONVERT 24

Conclusion 35

ABSTRACT

A programming language is described which is applicable to problems conveniently described by transformation rules. By this we mean that patterns may be prescribed, each being associated with a skeleton, so that a series of such pairs may be searched until a pattern is found which matches an expression to be transformed. The conditions for a match are governed by a code which also allows sub-expressions to be identified and eventually substituted into the corresponding skeleton. The primitive patterns and primitive skeletons are described, as well as the principles which allow their elaboration into more complicated patterns and skeletons. The advantages of the language are that it allows one to apply transformation rules to lists and arrays as easily as strings, that both patterns and skeletons may be defined recursively, and that as a consequence programs my be stated quite concisely.

INTRODUCTION

LISP (1) is a concise language for describing calculations which are primarily concerned with set theory, especially those which may be defined inductively. However it often requires a complicated expression with a bewildering hierarchy of parentheses to state other kinds of problems. COMIT (2) is better adapted to calculations involving transformations of form because one may indicate that if he sees an expression of a certain type he is to replace it by another. The practical disadvantage of COMIT has been the complicated way it handles lists. To isolate a sublist requires either advance preparation or a program to seek out a string with balanced parentheses. One suspects there would be an advantage to combining the salient features of both LISP and COMIT.

The conversion process which we describe was inspired by a desire to use a transformation language such as COMIT to describe programs for a variety of applications including algebraic simplification and compiler construction. The expressions or patterns to be recognized are constructed from more fundamental patterns in several ways including the system of parenthetical groupings used to form lists. Their analysis may be described by LISP, permitting one to concentrate upon the form of the patterns and not upon the means of their generation according to the rules of list processing.

The language CONVERT has been defined through a series of LISP functions and its implementation centers around the two functions RESEMBLE and REPLACE. The first, (RESEMBLE X L E) is a pattern recognition

function which distinguishes some twenty kinds of fundamental patterns and which analyzes more complex patterns built up out of those fundamental patterns. The second, (REPLACE D S), is used to construct new expressions. These may contain portions of the original expression which have been recognized and identified by RESEMBLE. In addition, REPLACE permits the formation of a new expression by several stages because partial results may be formed and analyzed further. The rules for the analysis are applied by the control function (CONVERT M I E R). Together with its satellites it constitutes the definition of the language CONVERT.

As indicated by writing it in the form (CONVERT M I E R), CONVERT is a function of four arguments. The third of these, E, is the expression which is to be transformed by the given rules. There may be one or more sets of rules all of which together comprise the argument R. A rule itself is a pair, (P S), consisting of a pattern P and a skeleton S. These constituents are generally composite, having been generated from primitive constituents which may either have a fixed meaning or be prescribed for an individual program. Whether a pattern or skeleton is variable or constant, it may be used in two ways depending upon whether it represents just one element of an expression in which it occurs, or whether it could represent a series of several elements. In the former case it is referred to as an expression, but in the latter as a fragment.

The use of patterns is to determine the portions of E which are to be identified and later used in constructing the converted expression,

as well as to ensure that E has a determined form. The variables which are thereby required are introduced into each program by means of the arguments M and I of CONVERT. These two arguments also serve to allow the definition of complex patterns and skeletons for use throughout the rule sets, as well as to introduce the variable skeletons.

The argument R of CONVERT is a collection of rule sets, each of which is paired with an identifying name. In operation, the first rule set is chosen, and the pattern of its first rule is compared against the argument E by means of the function RESEMBLE. In doing so it makes use of a dictionary formed from the arguments M and I which indicate the variables. Should a match occur, the function REPLACE is then used to make appropriate substitutions into the paired skeleton S. Should the match fail, the second rule is investigated, and so on. The search terminates when either a match is found and the corresponding skeletal substitution made or else when the rule set is exhausted and the expression is left unchanged. The other rule sets, aside from the first, may be invoked at a later time because of the provision for forming intermediate results, which may be further analyzed, either by reapplying the original rule set introducing a new one, or calling on another of the sets comprising R.

In the remainder of the paper we shall discuss the details of this process much more minutely, indicate some typical applications, and relate our experiences in using the language. Particularly, we need to catalogue primitive patterns as well as primitive skeletons, and to outline

the flow of control. However we shall first conclude our introduction by giving a glossary of technical terms, the usage of some of which is current in the description of list and string processors, but whose usage in the language CONVERT we wish to make definite.

The following terms are rather general and are concerned with the distinctions between the various parts of a list or string.

CHARACTER One of the recognizable symbols from which programs are constructed.

ALPHABET The collection of admissable characters. For present purposes it is the set of 48 characters available on the IBM 026 keypunch, including the blank.

STRING A linearly ordered sequence of characters.

ATOM A string which contains none of the characters blank, left parenthesis, nor right parenthesis.

DELIMITER One of the three characters blank, left parenthesis or right parenthesis.

LIST A string which commences with a left parenthesis, terminates with a right parenthesis, and otherwise consists of strings, separated by one or more blanks, which are themselves either atoms or lists.

ELEMENT One of those quantities which forms a list; it may be either an atom, or a list, but is used with reference to the list of which it is a member, refers only to a single constituent, does not include one of the delimiters, and does not mean one of the elements of a sublist.

EXPRESSION Either a list or an atom. Unlike the custom in LISP 1.5, the empty list is not regarded as an atom because it is always represented by a pair of parentheses, (), and not by a special atom, such as NIL. An element is an expression but moreover has to be thought of as part of a list and not as an isolated entity.

FRAGMENT A consecutive sequence of elements belonging to an original list, but not enclosed in parentheses.

While the preceding definitions are generally applicable to the LISP language as well as to CONVERT, the following definitions derive their meaning from their intended use in describing the function CONVERT. To be further noted is the fact that some terms have a specific further meaning when applied to a pattern or skeleton.

PATTERN An expression P, intended to be compared to the argument E of CONVERT, in order to discover whether E has a certain form, as well as to identify selected portions of E.

VARIABLE A pattern or skeleton represented by a single atom and which represents a single subexpression of the expression in which it occurs.

FRAGMENT A pattern or skeleton represented by a single atom, but which represents a fragment of the expression in which it occurs.

MODE The specification which accompanies each variable or fragment which governs the circumstances under which it may match an expression, if it is a pattern, or under which substitution is made if it is a skeleton.

VALUE A parameter accompanying each variable or fragment.

SKELETON An expression intended to guide the substitutions made by REPLACE. In general skeletons are formed as lists from primitive skeletons, but it is also possible for skeletons to represent functions together with their arguments written in prefix form.

RULE A pair consisting of a pattern and a skeleton. When CONVERT compares the pattern to its argument E and a resemblance is found, the values of relevant variables and fragments are substituted into the skeleton.

DICTIONARY A list used to establish a correspondence between different quantities.

THE PATTERN RECOGNITION FUNCTION (RESEMBLE X L E)

The pattern recognition function RESEMBLE is responsible in each case for determining whether the pattern half of a rule matches the expression E which it is desired to transform. Every pattern is constructed from primitive patterns so that it suffices to describe the primitive patterns and to call attention to the rules of list formation which say that a sequence of lists or atoms may be separated by blanks and enclosed within parentheses to form a new list. This process may be repeated to form lists of arbitrary complexity.

Actually the principle of list formation is only one of three techniques which may be used to construct compound patterns. To recognize this distinction one calls a pattern primitive if it is list-primitive, but a pattern form if it is primitive in the other senses and not in the list sense. They are built from primitive patterns with the help of identifiers which prevent their dissection by the list analysis mechanism.

The second principle is the principle of substitution, which allows us to define a single symbol as the representative of another pattern of arbitrary complexity which has already been formed and which is subsequently referred to by that name in place of the full pattern. This is a particularly valuable technique, not only for allowing synonymns for otherwise cumbersome expressions, but for the possibility it allows of defining patterns recursively. In order to produce a finite recursion

the pattern must somewhere contain an alternative between a terminal pattern and a repetitive pattern. Subject to the exercise of this precaution, the introduction of a recursive pattern may be a very concise way to define a desired pattern.

The third principle is the boolean composition of patterns. Since patterns are not predicates, they cannot be composed on the level of patterns but rather on the level "has the proper record been made to indicate a match?" Such a question not only transforms a pattern into a predicate, but tacitly recognizes the fact that the primitive patterns are not only recognizers, but are capable of recording data as well. In this way it is possible to check whether the multiple occurrence of a symbol reflects a match always to the same subexpression, as well as making the matched expression available for later substitution into a skeleton.

Our description of REPLACE then consists of a categorical enumeration of the primitive patterns and pattern forms which it recognizes.

First among these are the constants which represent expressions. These include

==	The universal pattern which matches any expression.
=ATO=	a pattern which will match any atom.
=NUM=	a pattern which will match any numeral.
=ORD=	a pattern which will match any ordered set.

The actual list depends upon the CONVERT processor under discussion, and could be extended considerably depending upon the circumstances. For example if there were different kinds of numbers to be distinguished,

such as floating point or integer, one would wish to introduce distinctive primitive patterns. This idea could be followed to more elaborate data types, such as arrays. The characters blank, left parenthesis and right parenthesis are delimiters and cannot be used directly as characters in the language, so that one might wish to introduce pattern synonymns for them, such as =BLA=, =LPR= and =RPR=. Were raw input in the form of BCD character strings available to the processor, it would need the ability to recognize delimiters as well as other characters.

The next category of primitive patterns comprises the variables. These are indicated in each individual CONVERT program by means of its arguments M and I, and eventually appear in the dictionary which is the argument L of RESEMBLE. L is a periodic list of period 3, in which we find the repeated cycle (... variable mode value...). Each variable is a primitive pattern, but the way in which it matches E is governed by its mode and almost always requires a parameter which is its value. Thus rather than describe the variables, which depend upon the program, we describe their possible types by listing the various possible modes.

X VAR G — the variable mode, in which the letter X is used to represent an expression, however complicated. X will match E if the LISP function (EQUAL G E) is true, but produces no change in the dictionary L. The variable mode may be used to introduce a synonymn for the atom X, to fix a constant in the entire CONVERT program, or to avoid repeatedly writing a lengthy expression.

X UAR * the undefined variable mode. X will match anything, but the entry in L is changed to read ... X VAR E ... , so that as a consequence if X appears as part of a more complicated pattern it will have to represent the same quantity each time it occurs, and moreover this common value will be preserved for subsequent use by REPLACE. The value associated with a variable in the UAR mode is of no importance, but some value must be specified to preserve the periodicity of L. The programmer generally need not concern himself with this point since UAR variables are generally listed separately as the argument I of CONVERT, the others with their full mode declarations comprising the argument M.

X PAT P the pattern mode, in which the letter X represents an entire pattern, however complicated. This pattern is the value of the variable, and one computes (RESEMBLE P L E) in place of (RESEMBLE X L E).

X PAV P the pattern variable mode, which is a combination of the modes PAT and UAR. Not only must the pattern P match E, but the dictionary L is altered to read ... X VAR E ... so that the expression E which matched X will not only be available to REPLACE, but so that if X occurs several times it will always match the same expression E.

X BUV (P ...) the bucket variable mode. It is similar to the mode PAV, but rather than requiring that the same expression E match every occurrence of the pattern variable, we simply make a list of these expressions. Thus X in the BUV mode will match any expression, but the dictionary is modified to read ... X BUV (P E ...) ... .

X CUV (P K) the counting variable mode. It is similar to BUV mode, but rather than listing the matching expressions we simply count them. Thus L is modified to read ... X CUV (P K+1) ... .

X STL K the strictly less mode. X will match any number, which is strictly less than K, which must be initially specified as a number, not as another variable.

X STG K the strictly greater mode. It is the same as the mode STL, but with the sign of the inequality reversed so that X matches any number strictly greater than the number K.

X RUL R the rule mode. R is a rule set defining the conditions under which X will match with E. This mode allows the user of CONVERT to define new modes or types of variables.

These are the principal modes which exist for matching variables to expressions. The list could be extended or modified. A diversity of number types or data types could require the introduction of additional arithmetic comparison modes, or modes for lexicographic ordering. Logically some of these might be introduced through the rule mode, but their separate introduction would represent a great convenience.

One type of mode with which we have experimented, but not included in the above list is the subset mode X SUB P, which will match an expression and produce as a value a list of those of its elements which match the pattern P. Should X be repeated several times in a larger pattern, the same list of extracted elements must result, although we will permit them to appear in a different order but not with different multiplicities. A variation of this idea, X MSU P, the maximal subset mode, resulted in a list of all those elements of the matched lists which appeared at each occurrence of the symbol X.

Whenever we encounter an atom in a pattern which does not appear in the dictionary L, that atom will match only itself. Thus all atoms are primitive patterns; those which appear on the dictionary L are to be matched as their mode specifies, while all others match only themselves, except for those such as =ATO= which have fixed meanings in all CONVERT programs.

An empty list may match only itself.

There is a series of primitive pattern forms which match expressions.

(=QUO= P) a quoted pattern. The expression P must be equal to E. This pattern form has two uses. It allows us to use the names of patterns or pattern forms as patterns. For instance if we wish to match the atom =QUO= we would write (=QUO= =QUO=). Also, it allows us to quote expressions which may then be compared by the simpler LISP function EQUAL rather than RESEMBLE.

(=DEF= N1 P1 N2 P2 ... Q) this pattern form allows us to define patterns. It is thus a local version of the PAT mode. Within the pattern Q, if the symbols N1, N2, etc. are used, they represent the patterns P1, P2, etc. The pattern Q is distinct from P1, P2, ..., if =DEF= contains an odd number of arguments, but if there is an even number, it is simply the last pattern in the list. =DEF= permits the recursive definition of patterns. For example, (=DEF= B (=OR= =ATO= (B B))) is the definition of a binary tree, or (=DEF= (E) ((*OR* () (== == E)))) is the definition of a list of even length.

A recursive pattern definition will generally involve a choice between a terminal pattern and a repetitive pattern which choice is permitted by the pattern form =OR= or the form *OR* which refers to fragments. When fragments are defined, their names are enclosed in parentheses. Since the patterns within the OR pattern forms are examined in order, the terminal patterns must be listed first.

(=AND= P1 P2 ... Pn) all the patterns P1, P2, ... Pn must match E. If more than one of the patterns contains the same variable, it must match the same subexpression according to each of the patterns. Since the leftmost fragment variable is given precedence to match the smallest fragment when several fragment variables occur, this should be borne in mind in arranging the arguments of =AND=, since its arguments are also examined in order, from left to right.

(=OR= P1 P2 ... Pn) at least one of the patterns P1, P2, ... Pn must match E. They are considered in order, from left to right, and once a match is obtained, none of the others will be considered. The OR pattern forms are the means by which patterns may be defined recursively. If P_i fails to match E, all variables which may have been tentatively defined by means of P_i are forgotten when P_{i+1} is considered, and only the variables which might have been bound by the successful pattern appear in the value of RESEMBLE.

(=NOT= P) the pattern P must not match E. If it does not match, it cannot bind any variables, while if it does match, the variables it binds do not matter because the overall match fails.

The next group of pattern forms which we have to discuss match fragments, and therefore cannot match atoms, and must always appear as a part of a larger pattern. There is only one constant fragment pattern

=== the indefinite fragment which will match any fragment.

For example, (===) will match any list, (=== == ==) will match any list having at least two elements, and so on.

We may also specify fragment variables. This is done in the arguments M or I of CONVERT by enclosing the variable name in parentheses. This usage of parentheses is a linguistic device motivated by the fact that one needs to have some kind of delimiter to indicate a fragment and the only delimiters readily available in LISP are parentheses. If we enclose both the name and specification of a fragment in parentheses we figure that we are more or less even, and moreover have a notation more or less compatible with LISP. It is important to bear in mind that fragments do not exist independently of a larger expression which contains them.

The way in which a fragment variable is to be regarded as representing a fragment of its pattern and thus to be eventually matched with a fragment of some expression is also specified by a mode declaration, which is accompanied by a value which gives necessary parametric information to make the match. The possible modes are the following.

(... (XXX) VAR EE ...) the variable mode. EE is an expression which is to be taken as a fragment of E. Thus (XXX ...) requires that the first element of EE match (CAR E), the second match (CADR E) and so on, until finally whatever portion of X follows XXX must match the resulting (CD...DR E). The elements of EE must match the corresponding portion of E by equality, not by RESEMBLE. For example, with (XXX) VAR (0 1), (XXX 2) will match the list (0 1 2) but not the list ((0 1) 2).

(XXX) UAR EE the undefined variable mode. In general the value EE is the empty list (). The fragment XXX will match with whatever fragment which is larger than EE.

However, if XXX appears several times in a larger pattern it must always match with the same fragment. If upon seeing a subsequent occurrence of the fragment XXX we find that the second fragment needed does not correspond to the fragment first found, we may revise our original estimate by trying a larger fragment for the initial occurrence of XXX. Hence we always take EE as our estimate of the fragment, usually the null fragment, and try successively larger fragments by adding elements to the end of EE until if possible a satisfactory candidate is found.

(XXX) PAT P the pattern fragment mode. The fragment chosen to match the fragment XXX must match the pattern P as well.

(XXX) PAV P the fragment pattern variable mode. Not only must the chosen fragment satisfy the pattern P, but for repeated occurrences of XXX the same pattern must appear. The final dictionary will contain the common fragment.

(XXX) BUV P The fragment bucket variable mode. It differs from the the PAV mode in the respect that any suitable fragment satisfying the pattern P will be accepted, and that the dictionary L will eventually contain a list of the fragments which matched each instance of the pattern XXX.

(XXX) CUV P the fragment count variable mode. It differs from the mode BUV simply in the respect that the instances of XXX are simply counted. The principal use of such a mode is in recursively defined patterns for which this number is not known in advance and we wish to determine it.

(XXX) REP (P K) the repeat mode. We expect that XXX is a fragment in which the pattern P is repeated K times. For example, a list of ten elements could be defined by (XXX) in which we define (XXX) REP (== 10).

Our remaining pattern forms consist of quoted and boolean combinations of fragments. They are always part of a larger expression.

(*QUO* P) a quoted fragment. P is a list, which is suposed to appear as a fragment of a matching expression; again it must match through equality.

(AND P1 P2 ... Pn) All the patterns P1, P2, ..., Pn must be fragments of a larger pattern P. However, they must all match exactly the same fragment of E. For example, (=== (AND (X ==) (== Y)) ===) will match (A (X Y) B) but not ((X A) (Y B)), whereas (=== (*AND* (X ==) (== Y)) ===) will match both.

(*AND* P1 P2 ... Pn) When this combination appears as a fragment of a larger pattern P, we must regard P1, P2, ... Pn in turn as fragments of P occupying the same position as (*AND* ...), and all these patterns must match E. However the same fragment of E which corresponds to P1 need not correspond to P2, and so on. For example, ((*AND* (X ===) (=== Y ===)) X), when X is in UAR mode, matches any list which begins and ends with the same element and which contains at least one Y, not the last element.

(*OR* P1 P2 ... Pn) At least one of the fragments P1 P2 ... Pn, when used in place of (*OR* ...) must result in a pattern which matches E. For example, (=DEF= (E) ((*OR* () (== == E)))) matches a list of even lenght; (X (*OR* (* *) (X X X)) X) will match a list of four elements of which the first and last are equal when X is in UAR mode and the middle two are stars, or else a list of five identical elements.

(*NOT* P) When P is inserted as part of the larger expression containint (*NOT* P), the resulting expression must not match E. Thus, ((*NOT* (1 2 3)) 4 5) will match (3 2 1 4 5) but not (1 2 3 4 5).

With this we have completed our survey of the primitive patterns and pattern forms. Once a match is obtained to a given pattern, a dictionary is produced which we may then use to make substitutions into a given skeleton.

Accordingly we now investigate the various possibilities for skeletons.

THE SUBSTITUTION FUNCTION (REPLACE D S).

The arguments of the function REPLACE are a dictionary D and a skeleton S. The dictionary is an edited version of the value of RESEMBLE when a successful match is obtained. The skeleton is an expression in which these variables are to be replaced by their equivalents which they matched in the original expression E. For example, if the pattern (cos x cos y - sin x sin y) were to be matched to the expression (cos 23° cos 19° - sin 23° cos 19°) in which x and y were declared as variables, we would perhaps like to substitute into the skeleton cos(x+y) to obtain cos(23° + 19°), or perhaps

even better, cos 42°.

In addition to making simple substitutions, we would like to make calculations as this example indicates, as well as permitting the generation of still other skeletons. Moreover, it is a very important aspect of the CONVERT language that we are able to permit intermediate results to be formed and analyzed further. Thus a portion of REPLACE is devoted to some of the control functions of CONVERT.

For these reasons we may also enumerate several categories of skeletons and skeleton forms. Again, by a skeleton form, we mean a pattern which becomes a skeleton after appropriate substitution of its constituents. One of these categories is comprised by the atomic skeletons, of which there are constants and variables. The constants are generally synonymns for quantities which cannot be written directly, such as delimiting parentheses. The variables take their meaning from the dictionary D, and depend upon the particular CONVERT program under consideration. There is then a category of skeleton forms which are comprised of functions and their arguments, which permits us to use a LISP-like functional notation within a skeleton. Another category allows us to introduce single symbols to stand for other quantities, be they expressions or skeletons. Finally there are those skeleton forms which allow the formation of intermediate

results and to continue their analysis.

All primitive skeletons and skeleton forms may be also regarded as fragments, which means that they are to be inserted into a larger skeleton without their delimiting parentheses.

In order to preserve some systematic notation, we have found it convenient to introduce a number of informal conventions, which are meaningless to the CONVERT processor, but which greatly improve the legibility of a program and facilitate its interpretation. For example, we choose three letter combinations as the names of the primitive patterns and pattern forms (with the exception of the two-lettered OR and ==), while we choose uniformly four letter combinations for primitive skeletons and skeleton forms. This choice is also reflected in our choice of mode names, which are again of three letters for patterns and four letters for skeletons. Dealing with patterns, we have found it useful to use single characters as atoms, but triple characters such as XXX for fragments, unless some more mnemonic combination presents itself in a particular case. Again, it is useful to distinguish fragments from expressions, and for this reason we surround the names with equal signs, as =QUO=, when they refer to expressions, or with stars, as *AND*, when they refer to fragments.

Considering the possible skeletons in detail, we commence with the constant skeletons.

=TRUE= a skeleton used by the RUL mode, which causes RESEMBLE to continue with whatever dictionary exists when it is encountered.

=FAIL= the corresponding skeleton which produces a false answer in the execution of the RUL mode.

=SAME= A skeleton is generally part of a CONVERT rule, whose pattern is being compared to some expression E. =SAME= refers to this expression, whatever it was.

=LPAR= a synonym skeleton, which stands for a left parenthesis.

=RPAR= synonym for a right parenthesis

=BLNK= synonym for a blank.

As in the case of the atomic patterns, this list could be presumably considerably extended should the particular application warrant. For instance, a skeleton such as =READ= could be used to obtain one expression from some input apparatus, and its value would be the resulting expression. We see in the present list two types of skeletons --synonyms for delimiters which cannot be quoted, and referents to objects which exist within a CONVERT program.

For the variable atomic skeletons, we have only two modes.

X EXPR S the expression mode. Every quantitiy which was in the VAR mode, or was changed into the VAR mode, such as those previously in the UAR or PAV modes, is transferred to the EXPR mode when the value of RESEMBLE is edited to become the dictionary of REPLACE. In addition, quantities may be specified initially in the argument M of CONVERT to be in the EXPR mode. When X, in the EXPR mode, is encountered, it is substituted by its value, S, without any modification.

X SKEL S the skeleton mode. The single atomic sybol X stands for the entire skeleton S, which is placed instead of X, and then modified (replaced) using again the function REPLACE.

Any atom which is not listed in the dictionary D in one of these two modes and is not an atomic constant --such as =SAME= -- is taken to be itself, in other words it is copied without change.

For example, if the dictionary D is (A EXPR (M N) N SKEL (G A))
and the skeleton S is (A A () (1 B A) *),
then (REPLACE D S) is ((M N) (M N) () (1 B (M N)) *);
with S_1 = (C A N (I) C A S), (REPLACE D S_1) will be
(C (M N) (G (M N)) (I) C (M N) S)
Note that instead of A [expr] we put its value (M N) exactly as it appears in the dictionary, whereas N [skel] is substituted by (G A) and then we REPLACE this, to obtain (G (M N)).

Among the skeleton forms, we have a large number which are in reality functions specified in the LISP prefix form, (F A1 A2 ... An), in which F is the name of the function and A1, A2, ... An are its arguments. In every case, all the arguments of such a function are replaced before it is executed.

(=PRNT= S) prints its argument, which is its value.

(=RAND= S1 S2) makes a random choice, probability 1/2, between S1 and S2, the chosen argument then being replaced.

(=COMP= A B) treats A and B as sets and calculates their relative complement A - B, consisting of those elements of A not in B. Repeated elements of A appear in the complements with the same multiplicity, if they appear at all.

(=INTS= SSS) forms the intersection of the skeletons comprising the fragment SSS, treating them as sets. If an element is repeated n times in the intersection, it occurred at least n times in each argument, and no more than n times in at least one of them. SSS should not be empty.

(=UNON= SSS) forms the union of the skeletons forming the argument list SSS. However, no element appears more than once in the union.

(=CONC= SSS) concatinates the argument skeletons. It is an alternative to =UNON= when one wishes to preserve multiplicity and order.

(=CART= SSS) forms the cartesian product of the argument skeletons, once they are replaced.

(=PLUS= SSS) sums the argument skeletons after replacing them.

(=MINS= S1 S2) computes the difference S1 - S2.

(=TIMS= SSS) multiplies its replaced arguments.

(=DIVD= S1 S2) forms the integer part of the quotient $S1/S2$.

(=REMN= S1 S2) forms the remainder of S1 after division by S2.

(=INCR= S) adds 1 to its argument.

(=DECR= S) substracts 1 from S; that is, its value is S - 1.

(=ARRY= I N S) forms an array of dimension N whose Ith element is computed according to the skeleton S in which I may appear as a variable. As is true of all these function skeletons, I, N, S are first replaced before any of this construction is attempted. If N is a list and not a number, the dimension of the array is the lenght of N, whose elements depend on the corresponding list elements. If S is missing, zeros fill the array.

(=ENTR= S1 S2 S3) stores S1 as the S2nd element of the array S3. S1, S2 and S3 are first replaced.

(=EXTR= S1 S2) produces the S1st element of the array S2.

All these function skeletons follow the LISP convention that their arguments are to be evaluated first, in this case by the function REPLACE which treats them also as skeletons, before the function is to be executed. If D_1 is (T EXPR 2 U SKEL (=PLUS= T 3))
S_1 = (T U) will be transformed to (2 5);
S_2 = ((B) (=DECR= (=TIMS= T U U)) (U)) will become ((B) 49 (5)).
When D = (M SKEL (A B C) A EXPR AA N SKEL (=CONC= (3 4)(5 6)))
and S is (Y (=UNON= N M (1 3 3)) (S (=INTS= M N) S) Z)
then (REPLACE D S) will be (Y (3 4 5 6 AA B C 1) (S () S) Z)
Note that the intersection has the value () in this example, and the union supresses repeated elements.

It is clear that some of these functions are primitive, in the sense that they could be written in no other way, while the others are composite. However, they have been listend among the primitive skeletons as a matter of convenience.

Another skeleton form closely related to this idea contains =ITER=.

(=ITER= I1 N1 I2 N2 ... S) In this skeleton form, Ii are variables which serve as indices while Ni are their corresponding ranges. For each combination of possible values the skeleton S, which may contain them as variables, is evaluated, and a list made of the resulting values. N2 may depend upon I1, so that this skeleton form is equivalent to (=ITER= I1 N1 (*ITER* I2 N2 (*ITER* ... (*ITER* In Nn S) ...))). When the range Ni is a numeral, Ii takes the values 1, 2, 3, ..., Ni inclusive, while if Ni is a set, Ii takes successively the values (CAR Ni), (CADR Ni) , ... For example, we could write (=CART= S1 S2) in the alternative form (=ITER= I1 S1 I2 S2 (I1 I2)). If a variable is enclosed in parentheses, as (Ii), it is taken as representing a fragment.

The next group of three primitive skeleton forms allows us to introduce temporary definitions of skeletons or expressions. They accomplish locally exactly the same thing which is done by the modes EXPR and SKEL globally. Again it is clear that they have considerable utility in allowing us to make recursive definitions of skeletons.

(=QUOT= N1 S1 N2 S2 ... S) makes a replacement of the skeleton S, but after adjoining to the dictionary D the information N1 EXPR S1 N2 EXPR S2 ..., ;in which S1, S2, etc., are <u>not</u> replaced. Rather, they are effectively quoted. A variation of this skeleton form is (=QUOT= S), in which S is copied without replacement, so that in this manner primitive skeleton names and form names may be referred to as themselves.

(=EXPR= N1 S1 N2 S2 ... S) makes a replacement of the skeleton S, after adjoining to D the information N1 EXPR S1' N2 EXPR S2' ... where S1', S2', ... denote the values of these skeletons after replacement.

(=SKEL= N1 S1 N2 S2 ... S) makes a replacement of the skeleton S but after adjoining to D the information N1 SKEL S1' N2 SKEL S2' ... where by S1', S2', etc., we mean the value of S1, S2, ... after replacement.

For both the skeleton form (=EXPR= ...) and (=SKEL= ...) the evaluation of the arguments S1, S2, ... is carried out independently, so that in evaluating S2, N1 does not yet stand for the evaluated S1, and thus will retain its previous meaning.

For example, if we use the dictionary (A EXPR AA B EXPR BB) to replace the skeleton (A B C (=EXPR= A (K A) C ((A B C)) (1 A B C 2)) A B C), we will obtain (AA BB C (1 (K AA) BB ((AA BB C)) 2) AA BB C), where we note that only inside (1 A B C 2) A has the value (K AA) and C is ((AA BB C)).

There are three skeleton forms containing =CONT=, =REPT= and =BEGN=, which are control skeletons governing the formation and subsequent analysis of intermediate results by another rule set, whose description we shall defer to the next section which explains the general flow of control.

With this we exhaust the skeletons and skeleton forms which refer to expressions, and arrive to those which refer to fragments. In general there is a series of skeleton fragments entilery analogous to the expression fragments. Of course there is no function *PLUS* since the value of =PLUS= is not a list, but there are functions *INTS*, *UNON* and so on. Then,

there are a few skeletons which make sense as fragments for which no expression analogue exists, *ANUL* being the prototype example.

First, we have the analogue of =SAME=.

SAME is the expression which the currente rule set is examining, and presuming that this is a list, is inserted into the proper place in our skeleton as a fragment.

Then, we have the fragment variables, for which there are four modes.

(XXX) EXPR (EEE) The expression (EEE), assumed to be a list, is inserted in place of XXX as a fragment, but none of its elements are replaced.

(XXX) SKEL (SSS) The fragment SSS is substituted for the symbol XXX in any skeleton in which XXX appears, before replacement is made.

XXX CONT R This is essentially the mechanism by which we define functions in CONVERT. R is a rule set. If our skeleton contains the skeleton form (XXX A1 A2 ... ... An), we construct the list resulting from replacing (A1 A2 ... An) and use it as a new expression E, and evaluate the skeleton form (=CONT= E * R), whose explanation we have deferred to the next section. However we may simply think of XXX as the name of a function. The difference between the CONT and REPT mode is that we use respectively =CONT= or =REPT=, whose distinction is in their treatment of free variables. CONT preserves previously existing variable definitions, REPT erases them. If the function's name is not enclosed in parentheses, its value is an expression, but if it is enclosed in parentheses the value is treated as a fragment.

XXX REPT R Another mode which permits function definitions, which is the same as CONT except that it may not use the values of any variables which have become defined in the course of the program, with certain exceptions covered in the discussion of =CONT=, =REPT= and =BEGN= in the next senction. Rather, all such variables are restored to their state when the CONVERT program was first entered.

We now describe those skeleton forms which correspond to functions whose value is a fragment rather than an expression. They are identified by a * which forms part of their name.

(*ANUL* S) After replacement of the skeleton S, nothing is done to the main skeleton, which is effectively equivalent to inserting an empty fragment in place of (*ANUL* S). Clearly such a skeleton as this makes sense only if S is an "operator skeleton". By this we mean that in the course of replacement of S, some permanent or auxiliary changes are effected. For example, S might involve =PRNT=, and in this way we could write on the output tape without retaining a copy of what we have written in the expression which we are developing.

The following are precise fragment analogues of the corresponding expression valued skeletons, and consequently need no further explanation.

QUOT *EXPR* *SKEL* *COMP* *INTS* *UNON* *CONC* *CART* *ITER* *CONT* *REPT* *BEGN*

This completes our analysis of the possible skeletons, and we proceed to the main function CONVERT, which is the control function together with its satellites, for a CONVERT program.

THE CONTROL FUNCTION (CONVERT M I E R)

Of the four arguments of CONVERT, the one which essentially specifies the program to be executed is R, the collection of rule sets. The expression which is to be transformed comprises the argument E, while the arguments M and I specify the variables which

are to be used in the program. I, the argument of initially undefined variables, consists of a list of those variables which are going to belong to the mode UAR, as well as those fragments which are initially empty and which belong to the mode UAR. The variables are indicated as individual atoms, while the fragments are indicated by enclosing the atomic name within parentheses. Thus a program which uses the variable X and the fragment XXX would have an argument I of the form (X (XXX)). The argument M, which contains the mode definitions, may be regarded as a sort of annotation to the main program, in which it is explained that various symbols have various special meanings. Again the same general system applies; variables which are to represent symbols are indicated by atoms, while those which represent fragments are indicated by atoms enclosed within parentheses. One further convenience has been that if the mode name itself is enclosed in parentheses, one assumes that sufficient information is already available in the initial argument M to calculate its value using CONVERT, so that the value may be given as a skeleton to be evaluated. In this way much tedious re-evaluation of constant values may be avoided, with a corresponding increase in operating efficiency.

The argument R, the collection of rule sets, is an alternating list; in which the names of the rule sets alternate with the sets themselves. Each rule set is a list of pattern-skeleton pairs. In operation, the first rule of the first set is taken and its pattern

compared to the argument E. Should it match, the corresponding skeleton is evaluated; but if not the second rule is examined and so on. Whenever one arrives to the end of a rule set without a match, the original expression is left unchanged.

The previous two sections have contained a detailed description of the fragments and how their match is effected, as well as of the possible skeletons and how substitutions are made with them. Thus it remains only to discuss the skeletons =CONT=, =REPT=, and =BEGN=, as well as their fragment valued analogues *CONT*, *REPT*, and *BEGN*. These skeletons permit the formation of intermediate results and direct the continuation of the analysis with respect to a specified rule set, so that they are the agencies responsible for the flow of control in a CONVERT program.

There is only one skeleton form (=BEGN= S), and its replacement is effected by first replacing the skeleton S, then using this result as the argument E, and starting the entire CONVERT program over again from the beginning. This implies in particular that all variables must be restored to their original status of UAR or whatever other mode which they had originally and which might have been altered during the course of execution of teh program.

There are three skeleton forms, (=CONT= S), (=CONT= S K), and (=CONT= S K1 R1 K2 R2 ...). In each of these, the skeleton S is first

replaced, and then taken to be the argument E of CONVERT. However, the rule set to be applied depends upon which of the three skeleton forms we are using. For the first, similar to the way we use =BEGN=, we return to the current rule set. In the second, (=CONT= S K), we continue with the rule set whose name is K. In the third form, we continue with the rule set K1 which is defined to be R1. In addition, we are privileged to introduce other named rule sets at the same time; these are R2 with the name K2, and so on. These names take precedence over any similar names used in the argument R, because the original list of rule sets is appended to the end of this new list. Names not usurped in this way refer to the older list.

The difference between the skeleton forms involving =CONT= and those involving =BEGN= is that in continuing to the new rule set, all variables which may have previously arisen are retained. Thus, in the rule (X (=CONT= S)), if X was originally in the mode UAR, it will have changed to the mode VAR and eventually to EXPR when S is evaluated. However, the second time we apply the rules, X will no longer be in the UAR mode, but will retain its identificaiton in the VAR and EXPR modes.

The third category of skeleton forms, (=REPT= S), (=REPT= S K), and (=REPT= S K1 R1 K2 R2 ...) is the precise analogue of =BEGN=, in that its members undertake to forget variable definitions and restore the dictionary to its original state. Otherwise the distinction between

these three forms is the same as the corresponding distinction for the =CONT= forms; they differ in the rule set they choose for the continuation. At this point a conflict makes itself apparent. With =CONT= we have no real problem because we retain all previously defined variables. However, =BEGN= undertakes to restore all variables to their original condition.

The conflict exists because the skeleton form (=EXPR= N1 S1 ... S) undertakes to guarantee that at all times within the expression S, the symbol N1 means S1', the replaced S1. This is presumably true no matter what rule sets and revisions of variables are encountered on account of the skeleton forms using =BEGN=, =REPT= or =CONT=. A similar situation holds with respect to =SKEL= and =QUOT=, and presumably also for =ITER= and =ARRY=. On the other hand, =BEGN= provides a convenient synonymn whereby we may re-enter our program recursively, and this demands that the program be in the same state every time it is re-entered, and that it cannot depend upon a previous history of having bound variables.

The resolution of the dilemma consists simply in establishing a hierarchy of precedence. Thus, in decreasing order of precedence, we have

=BEGN=

=QUOT=, =EXPR=, =SKEL=, =ITER=, =ARRY=

=REPT=

=CONT=

In this way, =BEGN=, which has only one form since it is the recursive re-entry to the program, causes all variables to be restored to their original status. The skeletons such as =EXPR= of the second level cause their definitions to be retained at all times within their argument skeleton S except when =BEGN= or *BEGN* are encountered. Then, =REPT= causes all variables to be restored to their original condition when proceeding to a new rule set, except those which have been bound by one of the skeletons of the second level. Finally, =CONT= retains all variable definitions, whatever their origin. The fragment valued skeletons are equivalent to the expression valued versions, in this hierarchy.

In order to illustrate these ideas, we might consider a few very simple programs. For example, the LISP function (MERGE L M) merges two lists, and is defined by

```
(MERGE (LAMBDA (L M) (IF (NULL L) L (CONS (CAR L) (CONS (CAR M)
(MERGE (CDR L) (CDR M)))))))
```

If L = (0 1 2) and M = (A B C), (MERGE L M) = (0 A 1 B 2 C). The same function written in CONVERT takes the form

```
(MERGE (LAMBDA (L M) (CONVERT

       (LIST)

       (QUOTE (P Q (PPP) (QQQ)))
       (LIST L M)
       (QUOTE (* (
                 (((P PPP) (Q QQQ)) (P Q (*BEGN* ((PPP) (QQQ)))))
                 ((() ()) ())
                 )))

       ))).
```

This second definition is a bit more space consuming because of the necessity to define the CONVERT program as a LISP function, but its essence is conveyed by the two rules comprising the rule set. The argument M is an empty list because there are no mode declarations to make, while the second argument I indicates that we are going to use the variables P and Q and the fragments PPP and QQQ. The third argument is a list of L and M, which is our expression to be transformed. Effectively, CONVERT functions are functions of one variable, and to deal with functions which we would ordinarily regard as having several variables we must combine all these arguments into a single list. Finally the argument R, which is a collection of rule sets, contains only one rule set, which has been given the noncommital name *. It consists of two rules, the first of which is the terminal condition and tells us to quit with an empty list when both the lists L and M are empty. Otherwise they are separated into their CAR's and CDR's by the second rule, which lists (CAR L), followed by (CAR M), followed by whatever fragment results from recursively applying the same process to the list of (CDR L) and (CDR M). Were we to have written =BEGN= in place of *BEGN* we would have obtained a list of three elements: (CAR L), (CAR M), and the merged CDR's.

A second example is the inverse function, UNMERGE, which takes a list of even length and separates its odd and even elements into two separate lists. Defined in LISP it takes the form:

```
(UNMERGE (LAMBDA (L) (IF (NULL L) (LIST L L)
      ((LAMBDA (X) (LIST (CONS (CAR L) (CAR X))
            (CONS (CADR L) (CADR X)))) (UNMERGE (CDDR L))))))
```

while in CONVERT we find

```
(UNMERGE (LAMBDA (L) (CONVERT
      (LIST)
      (QUOTE (P Q (PPP) (QQQ) (RRR)))
      L
      (QUOTE ( *(
            ((P Q RRR) (=CONT= (=BEGN= (RRR)) * (
                  (((PPP) (QQQ)) ((P PPP) (Q QQQ)))
                  )))
                  (() (()()))
            )))
      ))).
```

Again the four arguments of CONVERT are mostly trivial. However, we may see how the skeleton (=BEGN= (RRR)) corresponds to the ((LAMBDA (X) ...) (UNMERGE (CDDR L))) portion of the analogous LISP function, since we are dealing with a common subexpression which we wish to compute in advance. Since we are sure of always obtaining a list with two sublists, the solitary rule of the skeleton =CONT= simply serves to give these sublists names and to identify them as fragments. The variables P and Q were defined in the outer pattern, which is why =CONT= rather than =REPT= was used in proceeding to the inner rule set.

For our final example we choose an elementary but non-trivial example, a program to calculate derivatives of algebraic expressions and to simplify moderately the resulting expression.

```
(DERIVATIVE (LAMBDA (X E) (CONVERT
(CONS (QUOTE T) (CONS (QUOTE VAR) (CONS X (QUOTE (
    K       PAV     =NUM=
    L       PAV     =NUM=
    LL      SKEL    (UNLS LLL)
    RR      SKEL    (UNLS RRR)
    DL      SKEL    (=BEGN= (UNLS LLL))
    DR      SKEL    (=BEGN= (UNLS RRR))
    UNLS    REPT    (((X) X))
    .+.     REPT    (
                     ((K L) (=PLUS= K L))
                     ((X 0) X)
                     ((0 X) X)
                     ((X X) (2 * X))
                     ((X Y) (X + Y))
                    )
    .-.     REPT    (
                     ((X 0) X)
                     ((0 X) (- X))
                     ((K L) (=MINS= K L))
                     ((X X) 0)
                     ((X (- Y)) (.+. X Y))
                     (((- X) Y) (- (.+. X Y)))
                     (((- X) (- Y)) (.+. Y (- X)))
                     ((X Y) (.+. X (- Y)))
                    )
    .*.     REPT    (
                     (((=== 0 ===) 0)
                     ((X 1) X)
                     ((1 X) X)
                     ((K L) (=TIMS= K L))
                     ((X (- Y)) (- (X * Y)))
                     (((- X) Y) (- (X * Y)))
                     (((- X) (- Y)) (X * Y))
                     ((X X) (X ** 2))
                     ((X (X ** K)) (X ** (=INCR= K)))
                     (((X ** K) X) (X ** (=INCR= K)))
                     (((X ** K) (X ** L)) (X ** (=PLUS= K L)))
                     ((X Y) (X * Y))
                    )
    ./.     REPT    (
                     ((X 1) X)
                     ((0 ==) 0)
                     ((X X) 1)
                     (((X ** K) (X ** L)) (X ** (=MINS= K L)))
                     ((X (X ** 2)) (1 / X))
                     ((X Y) (X / Y))
                    )
```

```
    .**.   REPT    (
                    ((X 1) X)
                    ((== 0) 1)
                    ((1 ==) 1)
                    ((X Y) (X ** Y))
                   )
    )))))
(QUOTE (
    X Y (LLL) (RRR)
       ))
E
(QUOTE (* (
    (T 1)
    (=ATO = 0)
    ((LLL + RRR) (.+. DL DR))
    ((LLL - RRR) (.-. DL DR))
    ((LLL * RRR) (.+. (.*. LL DR) (.*. DL RR)))
    ((LLL / RRR) (./. (.-. (.*. RR DL) (.*. LL DR)) (.**. RR 2)))
    ((LLL ** RRR) (.*. RR (.*. (.**. LL (.-. RR 1)) DL)))
    )))
)))
```

The actual rule set R is quite modest. After noting that the derivative of the variable of differentiation is 1 and that the derivative of other atoms is zero, there follow the rules for the four algebraic operations, as well as for powers. By arranging the rules in just this order the hierarchy of precedence of the algebraic connectives is established. For example, multiplication is selected as a connector only if there are no sums or differences on the top level. Moreover, since the shortest leftmost fragment satisfying a pattern is always chosen association to the left is always made.

The right hand parts of the rules contain rather cryptic notation which is the result of a certain amount of revision of a more straightforward rule set. For example we could write ((LLL + RRR) ((=BEGN= (LLL)) + (=BEGN= (RRR)))). However, when LLL is a fragment containing only one element, we would prefer not to write the extra set of parentheses, which is accomplished by the skeleton form (UNLS XXX) defined in the REPT mode. Moreover we judge DX to be a more suggestive name than (=BEGN= (UNLS X)), in accordance with which we introduce the appropriate skeleton definitions.

The fact that the right hand sides of the rules use prefix rather than infix notation permits us to define the algebraic connectives as functions which effect the obvious simplifications of the crude results. For example to differentiate $2*x^2$ we obtain as a result $0*x^2 + 2*2*x^{1}*1$,

and the resulting superfluous factors of unity or zero summands must be eliminated, as well as making adjustments of zero or unit powers, and so on. The alternatives to this mode of operation are to use an auxiliary simplification routine which requires a second pass through a rather long expression, and for which it is somewhat unclear when to terminate a given simplification process or in what order to combine them. Another alternative would be to recognize more complicated patterns such as cx for constant c, and prescribe a more appropriate derivative which would not need so much simplification.

The actual rules for the simplification may be as elaborate as desired, but those which we display here suffice to remove the obvious redundancies produced in the differentiation, and to clean the result up slightly beyond this. For example, the pattern variables K and L, which recognize numbers, are employed several times to avoid indicated operations on numbers which could just as well be executed.

CONCLUSIONS AND EXPERIENCES

The CONVERT language has been implemented for the IBM 709 computer of the Centro Nacional de Calculo (CENAC) of the National Polytechnic Institute in Mexico City, as well as for the Q-32 computer of the Systems Development Corporation in Santa Monica, California. However, since it is basically written in LISP it is by implication available for computers

for which a LISP system exists, and a version is also in operation in the IBM 7094 time-sharing system of Project MAC at MIT. There have been several complications which have arisen on account of the LISP substrate, and efforts are underway to provide a direct machine language version for various computers.

The first version was written in MBLISP, a LISP dialect which differs from LISP in a number of technical details, but which had a sufficient amount of pushdown list and free storage space available to be able to execute reasonably complicated examples. However, as an interpreter, interpreting a CONVERT interpreter for a slow machine, it was decidedly show. The Q-32 version using a faster machine with a LISP compiler gave a very much better performance. One of the programs tested involved analyzing a group of order 16, defined in a moderately complicated manner as a semi-direct product C8:C2 of cyclic groups of order 8 and 2 respectively. It was possible to obtain the group table in about 3 minutes at times when the time-sharing competition was not intense which meant about a second per group product, a figure several hundred times as fast as for MBLISP in the 709.

However, the Q-32 LISP has a limited pushdown list available, which prevented the execution of quite a number of programs since CONVERT as it stands is highly recursive. However, the latest experience has been with yet another LISP processor, a compiler, constructed for the CENAC by Lowell Hawkinson and Robert Yates, and which they are presumably

continuing to develop. It is unique among LISP dialects in having an excellent array and floating point numerical capability, as well as being very carefully organized in all its other aspects. As a result it is one of our current vehicles for CONVERT programs.

Our principal applications so far have fallen in two areas. One is involved with the improvement of the CONVERT processor itself. The other area has been concerned with group analysis, both for the determination of irreducible representations and analyzing the properties of finite groups, and for calculations involving the non-commutative generators of Lie algebras. Here CONVERT is applied not only to effect the calculations, but again to provide a notation to improve the ease with which the desired calculation can be indicated.

The substance of our experience to date has been that it is indeed quite easy to describe programs and adduce novel notations. For a certain class of problems whose results are of high importance and for which this means of description is especially convenient, the application of CONVERT has been quite satisfactory.

Further extension depends upon a number of factors. Various changes in the processor in use at Project MAC, which consist for the most part of introducing iterative rather than recursive analysis wherever possible, including a CONVERT program feature, have increased its speed by a factor between five and ten. The prospect of a compiler to replace the present interpreter without working through LISP as a substrate has already been partially realized in the Hawkinson-Yates LISP, which admits

a compiled function as a data type. This has permitted the compilation of special functions to perform arithmetic operations directly on the arrays which we use to represent group elements. This in turn has revealed the possibility of introducing whole new hierarchies of notation adapted especially for particular problems, as well as to alleviate awkward programming situations of a more general nature.

In summary, although we doubt that our system of notation will persist unmodified, the general scheme seems fairly serviceable. The transformation rule format, the ability to define patterns and operations recursively, the implicit search strategies and construction rules, as well as versatility for introducing still other types of notation, all combine to produce a very concise programming language.

ACKNOWLEDGEMENTS

We have to express our gratitude and appreciation to Dr. Arturo Rosenblueth, Director of the Centro de Investigacion y de Estudios Avanzados of the Instituto Politecnico Nacional for his hospitality during the period of this investigation and for his interest in the development of programming languages and computer sciences. The machine time necessary for the development of this work has been made available by the Centro Nacional de Calculo of the National Polytechnic Institute, and by the Systems Development Corporation, Santa Monica, California. We greatly appreciate the cooperation of both institutions, as well as the Red de Communicacion y Experimentacion de los Centros de Ensenanza del Instituto Politecnico Nacional for providing the telex communication to the Q-32 time sharing system. Finally we acknowledge our indebtedness to Lowell Hawkinson and Robert Yates for the LISP processor which we currently use.

Work reported herein was also supported in part by Project MAC, an M.I.T. research program sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract Number Nonr-4102(01). Reproduction in whole or in part is permitted for any purpose of the United States Government.

REFERENCES

1. McCarthy, J. Recursive functions of symbolic expressions and their computation by machine. Communications of the ACM, 3 184-195 (1960).

   McCarthy, J., et al., LISP 1.5 Programmer's Manual. Massachusetts Institute of Technology, Cambridge, Mass. (1962).

   Berkeley, E. C., and Bobrow, D. G. (Eds.) The Programming Language LISP, its Operation and Application. Information International Inc., Cambridge, Mass. (1964).

2. Yngve, V. H., et al., An Introduction to COMIT Programming. The MIT Press, Cambridge, Mass., (1963).

3. Guzmán, A. CONVERT. Professional Thesis. Instituto Politécnico Nacional, Mexico City, (1965). (spanish).