PATTERNS AND SKELETONS IN CONVERT

by

Adolfo Guzman

Project MAC
Massachusetts Institute of Technology

and

Harold V. McIntosh

Escuela Superior de Fisica y Matematicas
Instituto Politecnico Nacional de Mexico
and
Centro de Calculo Electronico
Universidad Nacional Autonoma de Mexico

April 1, 1967

A B S T R A C T

CONVERT is a programming language based on the transformation rule format, but nevertheless making full use of program specification and recursive function definition. A transformation rule consists of a pattern-skeleton pair, such that when the pattern matches a given expression and selected constituents are isolated, they are substituted into the skeleton to effect the transformation. Skeletons contain the flow of control of a program, acting in one of three ways: either an intermediate result may be formed and analysed by further rules, a series of program steps may be specified, or a function may be applied to a series of skeletal arguments. The paper describes the matching and substitutional processes in terms of equivalent LISP functions, and then enumerates the primitive patterns and skeletons which are available in the CONVERT language.

INTRODUCTION

The programming experience which has evolved during the past two decades has produced substantially three different styles of programming. Perhaps the most widespread is the iterative style in which a series of program steps is described, and these are interspersed with labels allowing one to pass from one part of the program to another, including the repetition of some of its parts, until it is finally terminated. The recursive style permits one to define a process inductively, and in the symbolic realm LISP has been the foremost example of such a language. Finally, a third style consists in the transformation rule format, in which one gives an example of some quantity, and an example of the way in which it is to be modified. COMIT and SNOBOL have long typified this style.

Each style which has been realized in practice has had its theoretical counterpart, even if the implementation has often arisen from some practical need and independently of the theoretical formulation. Thus one may see in the Turing machine a prototype of both the iterative and recursive processes, while recursive function theory and the lambda calculus of Church was quite explicitly the formulative background of LISP. Markov algorithms may be seen as the organizational framework of COMIT-like languages even though the development has been more independent and primarily motivated by intended applications.

Moreover, each style has had its unique advantages and disadvantages. Iterative programming corresponds more closely to the electronic structure of computers, and is very simple and direct. Recursive programming is much more elegant and in many areas permits much more concise programming. Its disadvantage has been that the necessity to continually preserve temporary results requires a vast amount of memory, as well as prolonging the time of execution. Finally, transformation rules find their application in situations where it is easier to specify a complex structure by form or example than by detailling its assembly from primitive constituents, and where it is equally more convenient to specify a transformation conditioned on the presence of such form.

It is hardly surprising that there have been various efforts toward combining three such fundamental and attractive styles in a single language. For example one of the early additions to LISP was the program feature, and FORMULA ALGOL has endeavoured to incorporate both recursive procedures and Markov algorithms. A recent effort involving LISP was the METEOR program of Bobrow, which was a COMIT interpreter written in LISP. Its advantage was that it made use of LISP's ability to handle lists, which has always been very cumbersome in COMIT. A more extensive attempt at the synthesis has been the program FLIP of Teitelman.

CONVERT was devised as a transformation language which would incorporate the desirable features of all these schemes. Rather than being an extension of LISP it undertakes to be an independent language, which has been defined by means of LISP and which for the moment is interpreted by a LISP processor. Nevertheless it embodies the LISP techniques of list processing and recursive function definition, permits an iterative program format, but is based on the use of transformation rules as its primary format. In this way all three programming styles are homogeneously available, the utility and convenience of the arrangement chosen being claimed as the principal advantage of the new language.

CONVERT is defined through a series of LISP functions, and its implementation centers around the two functions RESEMBLE and REPLACE. The first, (RESEMBLE P 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 patterns. The second, (REPLACE D S), is used to construct new expressions, which may contain portions of the original expression which have been recognized and collected 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 this analysis and the subsequent synthesis are applied by a control function (CONVERT M I E R), from which the language derives its name and which, together with its satellites, constitutes the definition of the language.

As indicated by writing it in the form (CONVERT M I E R), CONVERT is a function of four variables. The third, E, is an expression which is to be transformed. The transformation is effected by consulting rules, which are grouped into sets which comprise the argument R. A rule itself is a pair (P S) consisting of a pattern P and a skeleton S. Patterns are expressions to be matched against the argument E, while skeletons are forms into which E will be transformed. Patterns and skeletons are generally composite; some have a fixed meaning and are constant, while others are variable, having meaning prescribed in the individual program.

The use of patterns is twofold; to determine the portions of E which are to be identified and used in constructing the converted expression, as well as to ensure that E has a specified form. The required variables are introduced into each program by means of the arguments M and I of CONVERT. These two arguments also allow the definition of complex patterns and skeletons for use throughout the rule sets and to introduce the variable skeletons.

As we noted, 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 so doing use is made of a dictionary formed from the arguments M and I which indicate the variables. M contains variables which need mode definitions, this being the mechanism by which the precise manner of matching is specified. I contains those variables whose mode is the general undefined variable mode UAR, since it is more convenient to group them separately. 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 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 a provision for forming intermediate results which may be further analyzed either by reapplying the original rule set, by calling on one of the others in the set R, or by introducing a new set locally.

As the foregoing introduction suggests, there is considerable detail involved in defining the matching and the substitutional processes, as well as a considerable amount of technical vocabulary associated with the use of CONVERT. Although we do not wish to give the listing of the entire CONVERT program, nevertheless in its place we shall include a detailed discussion of some of the historical predecessors of RESEMBLE and REPLACE in order to show the essential features of the process of conversion. The processor actually in use at present differs from one based on these preliminary functions in having a wider selection of alternatives available, as well as being written using the LISP program feature to make it as iterative as possible, and other details of a similar nature.

LISP PATTERN RECOGNITION FUNCTIONS

One of the operations which one is frequently called upon to perform with LISP is to compare two expressions. The simplest function which accomplishes this purpose is the primitive function EQ, which tests two atoms for identity. It will also distinguish an atom from a list, and may be used to construct the LISP function EQUAL, which compares two expressions. Two expressions are equal if they are both the same atom, both empty lists, or else are both lists made up of the same atoms grouped together in the same way.

(EQUAL X Y) is accordingly a predicate of two variables, designed to compare two expressions to see if they are equal, irrespective of whether they are atoms or lists. The formal definition of EQUAL is

```
(EQUAL (LAMBDA (X Y) (OR (EQ X Y)
                         (AND (NULL X) (NULL Y))
                         (AND (NOT (ATOM X))
                              (NOT (ATOM Y))
                              (NOT (NULL X))
                              (NOT (NULL Y))
                              (EQUAL (CAR X) (CAR Y))
                              (EQUAL (CDR X) (CDR Y))))))
```

In explaining this definition we parenthetically mention that our programs are written in MBLISP, a LISP dialect one of whose technical features is that an empty list is considered to be a list, not an atom: NIL is not an exceptional atom in MBLISP, and thus NULL is a primitive function. Nevertheless, it is well known that a test for the combination of either null or atom will simplify the definition of many lisp functions and that this is the intended role served by the predicate ATOM in LISP. Another variation between LISP and MBLISP is that T and F are not exceptional atoms. Thus, (AND) of no variables is a constant function assuming the value T. In like manner (OR) always takes the constant value F, and (LIST) will produce an empty list.

We often want to describe more general properties of a list than to simply say it is equal to another. A certain category of properties has to do with the form of a list, independently of the actual atoms which the list contains. For instance one might define a LISP predicate SINGLET, whose purpose is to recognize a list having exactly one element.

```
(SINGLET (LAMBDA (E) (AND (NOT (ATOM E)) (NOT (NULL E) (NULL (CDR E))))))
```

Similar predicates might be used to detect a list of at least one element, exactly two elements, an odd number of elements, and so on. As the conditions which the list has to fulfill become more and more complicated, the construction of these predicates becomes correspondingly more complicated, and the utility of a general function which will test the form of a list is seen.

Accordingly, we shall describe a series of functions which will recognize more and more general specifications for a list. The first of these, (SIMILAR X E), resembles (EQUAL X Y) but with the difference that certain special symbols are allowed to appear in the argument X, which modify the interpretation of the corresponding subexpressions in E. A double equal, ==, may match anything, corresponding to a blank in the expression X which may be filled in any manner whatsoever in the expression E. A triple equal, ===, much like the triple dot of conventional mathematical usage, may be matched by a fragment of a list, arbitrarily long.

The definition of SIMILAR is

```
(SIMILAR (LAMBDA (X E)
                 (OR (EQ X (QUOTE ==))
                     (IF (ATOM X)
                         (EQ X E)
                         (IF (NULL X)
                             (NULL E)
                             (AND (NOT (ATOM E))
                                  (IF (EQ (CAR X) (QUOTE ===))
                                      (OR (SIMILAR (CDR X) E)
                                          (AND (NOT (NULL E))
                                               (SIMILAR X (CDR E))))
                                      (AND (NOT (NULL E))
                                           (SIMILAR (CAR X) (CAR E))
                                           (SIMILAR (CDR X) (CDR E))))))))))
```

The two variables can be SIMILAR in several ways. A true response occurs, meaning that the two variables are SIMILAR, if one of the following is true.

1. The variable X is a double equal.

2. Each variable consists of only one atom, these atoms being identical.

3. Both variables are empty lists.

4. X is a list commencing with a triple equal, ===, which stands for a fragment of a list of arbitrary length, possibly null. The expression (===) will match any list, including the empty list, but not an atom. If the triple equal is followed by further expressions, they must be found on the list E but may be preceded by arbitrarily many other elements.

5. (CAR X) is SIMILAR to (CAR E) and (CDR X) is SIMILAR to (CDR E).

SIMILAR allows us to easily specify lists having a fixed form; for example we could now define SINGLET as

(SINGLET (LAMBDA (L) (SIMILAR (QUOTE (==)) L)))

To specify a list of at least two elements, we would use the pattern (== == ===); a list containing two X's in one of its subexpressions would be designated by (=== (=== X === X ===) ===), and so on.

The predicate SIMILAR is restricted to the pure recognition of form, and cannot require anything more than the definite positioning of those atoms whose presence is forced; for example we cannot even demand the presence of the atom == in E, since it has been preempted as a space filler. To meet such demands, we might allow predicates to be included among the elements of X, which would then be applied to the corresponding elements of E.

A function which meets this requirement, (ANALOGOUS Q E) imagines that the pattern Q is composed entirely of predicates, rather than of atoms to be tested by EQ, and requires that each predicate, when matched to its corresponding subexpression, take the value T. Since the elements of ANALOGOUS are predicates, we also expect to find their Boolean combinations, AND, OR, and NOT, which will allow the construction of more complicated patterns and predicates. Another provision is to allow the naming of patterns, allowing recursive pattern definition.

ANALOGOUS is a predicate of two variables, Q, which is a pattern, and E, which is the comparand. The list structure of Q and E is intended to be the same, in that they are both formed of lists of the same length having corresponding subexpressions. However the atoms to be found in Q are predicates which are to be applied to the corresponding subexpressions in E, in the expectation that the value of the predicate in every case will be T. It is convenient to use the Boolean operations to build up composite predicates, as well as to be able to name patterns to define them recursively. Consequently Q may be constructed in the following ways:

1. An atomic predicate is ANALOGOUS to any expression for which it takes the value true.

2. An empty list is ANALOGOUS to an empty list.

3. (AND Q1 Q2 ... Qn) means that E is simultaneously ANALOGOUS to Q1, Q2, ..., and Qn.

4. (OR Q1 Q2 ... Qn) means that E is ANALOGOUS to at least one of the patterns Q1, Q2, ..., or Qn.

5. (NOT Q) means that E is not ANALOGOUS to Q.

6. (=== Q1 Q2 ... Qn) means that any number of subexpressions, possibly none, may precede those ANALOGOUS to Q1, Q2, ..., Qn.

7. (DEF P Q1 Q2) means that we compare the pattern Q2 to E, while using the definition that P means analogy with respect to the pattern Q1. Should it happen that Q1 and Q2 are the same, we may abbreviate this pattern to (DEF P Q1).

8. If Q is not an empty list, nor (CAR Q) one of the symbols NOT, AND, OR, ===, DEF, then (CAR E) must be ANALOGOUS to (CAR Q) and (CDR E) must be ANALOGOUS to (CDR Q), which assures that Q and E will possess the same list structure.

By incorporating OR within the definition of a pattern, it is possible to define patterns recursively. OR tests its arguments one at a time, reading from left to right, until one is found which matches the expression E. To form such a recursive definition we need only list some well defined terminal patterns, followed by one or more repetitive patterns. For example if * is written to represent the predicate ATOM, a binary tree is defined as (DEF B (OR * (B B))), or a tree which branches only to the left is (DEF L (OR * (L *))).

ANALOGOUS is defined, by the aid of several satellites, as follows.

```
(ANALOGOUS (LAMBDA (Q E) (ANALOG Q E (LIST))))

(ANALOG (LAMBDA (Q E ALIST)
                (COND ((NULL Q) (NULL E))
                      ((ATOM Q) ((ASSOC Q ALIST) E))
                      ((EQ (CAR Q) (QUOTE NOT)) (NOT (ANALOG (CADR Q) E ALIST)
                       ((EQ (CAR Q) (QUOTE OR)) (ORANALOG (CDR Q)))
                      ((EQ (CAR Q) (QUOTE AND)) (ANDANALOG (CDR Q)))
                      ((EQ (CAR Q) (QUOTE ===))
                       (AND (NOT (ATOM E))
                            (OR (ANALOG (CDR Q) E ALIST)
                                (AND (NOT (NULL E)) (ANALOG Q (CDR E) ALIST)))
                       ((EQ (CAR Q) (QUOTE DEF))
                       (ANALOG (IF (NULL (CDDDR Q)) (CADDR Q) (CADDDR Q))
                               E
                               (CONS (CADR Q) (CONS (BIND (CADDR Q)) ALIST))))
                      ((AND) (AND (NOT (ATOM E))
                                  (NOT (NULL E))
                                  (ANALOG (CAR Q) (CAR E) ALIST)
                                  (ANALOG (CDR Q) (CDR E) ALIST))))))

(ORANALOG (LAMBDA (L)
                  (AND (NOT (NULL L)))
                  (ORANALOG (CAR L) E ALIST)
                  (ORANALOG (CDR L))))

(ANDANALOG (LAMBDA (L) (OR (NULL L)
                           (AND (ANALOG (CAR L) E ALIST) (ANDANALOG (CDR L))))
```

```
(BIND (LAMBDA (E) (PERSUBS E
                           (QUOTE E%)
                           (QUOTE (LAMBDA (E) (ANALOG (QUOTE E%) E ALIST))))))

(PERSUBS (LAMBDA (X Y L)
                 (COND ((NULL L) L)
                       ((EQ L Y) X)
                       ((ATOM L) L)
                       ((AND)
                        (CONS (PERSUBS X Y (CAR L)) (PERSUBS X Y (CDR L)))))))

(ASSOC (LAMBDA (X L) (COND ((NULL L) X)
                           ((EQ X (CAR L)) (CADR L))
                           ((AND (ASSOC) X (CDDR L)))))))
```

ANALOGOUS, as it is written, will not allow the incorporation of LAMBDA-statements to define predicates, and insists that all LISP predicates enter on the atomic level.

The pattern recognition functions which we have defined so far all suffer one failing, that they cannot compare two different parts of the same expression. Suppose that in studying the transformation of trigonometric expressions we want to study the identity sin%%2 X + cos%%2 X = 1. SIMILAR might be used to recognize the prefix form of the left hand side of this equation by using the pattern (+ (%+% SIN ==) 2) (%% (COS ==) 2)), but unfortunately SIMILAR, in such a case, would also accept sin%%2 X + sin%%2 Y, since the double equal will match anything at all. The problem is that X must be recognized as a bound variable, to take the same value every time that it is matched to some subexpression of E.

For this reason we now introduce the function (SIMILAR% X L E) which contains an additional argument, a list of free variables, which are to be bound during the search through E. In a majority of applications, one will want also to know the values of the bound variables which make the two expressions X and E SIMILAR, in the event such values exist. For this reason SIMILAR% is a "semipredicate" taking the value F in the event of lack of similarity, but its value is an alternating dictionary of values rather than T in the contrary case.

In summary, the arguments of SIMILAR% are

X a standard pattern which may contain designated bound variables, V1, V2, ..., Vn.

L a list of these variables in the form (() V1 V2 ... Vn).

E the comparand, an expression which is to be matched to X.

The value of SIMILAR% is F if no choice of values for the bound variables V1, V2, ..., Vn will make X similar to E. Otherwise its value is a list A,

A = ((Vk Ek ... Vi Ei) Vl Vm ...).

(CAR A) is an alternating dictionary, in which the bound variables are interspersed with their values, while (CDR A) contains any variables specified but not appearing in X. The order in which the variables appear in the dictionary depends on the order in which they were encountered in searching X.

The definition of SIMILAR% is

```
(SIMILAR% (LAMBDA (X L E)
                  (IF (ATOM L)
                      L
                      (IF (EQ X (QUOTE ==))
                          L
                          (IF (ATOM X)
                              (IF (ELEM X (CDR L))
                                  (CONS (CONS X (CONS E (CAR L)))
                                        (REMOVE X (CDR L)))
                                  (IF (EQUAL (ASSOC X (CAR L)) E) L (OR)))
                              (IF (NULL X)
                                  (IF (NULL E) L (OR))
                                  (IF (ATOM E)
                                      (OR)
                                      (IF (NULL E)
                                          (OR)
                                          (SIMILAR% (CDR X) (SIMILAR% (CAR X)
                                                                      L
                                                                      (CAR E))
                                                    (CDR E)))))))))

(REMOVE (LAMBDA (X L)
                (IF (EQ (CAR L) X) (CDR L) (CONS (CAR L) (REMOVE X (CDR L))))))
```

The rules for preparing the pattern X are almost the same as for the function SIMILAR.

1. A double equal matches any expression.

2. An atom which is a free variable matches E and is then bound. The value of an atom which is a bound variable must match the expression E. Otherwise an atom must match the same atom.

3. An empty list matches an empty list.

4. (CAR X) is compared to (CAR E) and (CDR X) is compared to (CDR E).

PRELIMINARY VERSION OF RESEMBLE

In the last section we have seen several stages of complexity in pattern matching functions, commencing with EQ and EQUAL, and including SIMILAR, SIMILAR*, and ANALOGOUS. SIMILAR served for nothing more than assigning a definite list structure to the expression with which it was compared. Although it would have been possible to have included predicates within the framework of SIMILAR to verify particular attributes of the subexpressions to which they were matched. Instead this mode of operation was used wholesale by ANALOGOUS. The list structure of a pattern used by ANALOGOUS still determined the structure of its comparand. The pattern elements being predicates, it was not only natural to form Boolean combinations of patterns while still taking advantage of the information provided by their list structure, but in addition the opportunity arose for defining patterns recursively. We begin to see two principles at work in the generation of patterns for ANALOGOUS, the formation of complex patterns from simpler ones by their arrangement into lists, and their generation by means of the Boolean connectives.

Another theme made its appearance in the function SIMILAR* on account of the desire to compare different parts of the comparand with one another. For instance, using SIMILAR we could demand a list of three elements by the use of the pattern (== == ==), but we could not say all three had to be equal until we had SIMILAR* available, whereupon pattern (X X X), with X declared to be a variable served the purpose.

The two functions ANALOGOUS and SIMILAR* both use a dictionary, although for different purposes; in the first case to remember pattern definitions, and in the second to retain variable bindings to permit comparison of different parts of an expression. Not only does this suggest that the two modes of operation be combined by a common function, but further that the dictionary be allowed to carry synonyms, and also that the circumstances under which matches be obtained be generalized.

Yet another consideration is that fragments of a list be comparable; for instance we might write (XXX == XXX) to specify a list of odd length, whose central element does not interest us, but which terminates with precisely the same series of elements with which it began. Once fragments are taken into consideration, one would also wish to combine their specifications by the Boolean connectives, as well as to introduce definitions and synonyms for fragments as well as expressions.

There is probably no end to the length to which a list of such desiderata might be extended. For example, we might think that our previous example, (XXX == XXX) should specify a list whose left half contains the same elements as its right half, without regard to their order, or even multiplicity. Or, we might think of an implicit match, such as using the pattern ((X+Y+Z) (X-Y) (X+Y-2Z)) to match a list of three numbers, (A B C), in which the implicit algebraic equations for X Y and Z would have to be solved to see if a numerical rather than symbolic match occurs.

In order to see how the features of EQUAL, SIMILAR, SIMILAR%, and ANALOGOUS may be combined into a single function, we now present a simple preliminary version of RESEMBLE. Further diversity and complexity than this is achieved by including more alternatives in the definition of RESEMBLE than may be found in the preliminary version, but its basic structure is not further altered.

```
(RESEMBLE (LAMBDA (X L E)
                  (RESEMBLE% (LIST) L (LIST) (LIST X (LIST)) (LIST E (LIST)))))

(RESEMBLE% (LAMBDA (X L E XX EE)
                   (COND ((NULL XX) (IF (NULL EE) L (FALSE)))
                         ((EQ (CAR XX) (QUOTE =DEF=))
                          (RESEMBLE% X (CDDDR L) E (CDR XX) EE))
                         ((NULL EE) (FALSE))
                         ((EQ X (QUOTE ==)) (TRUE))
                         ((EQ X (QUOTE =ATO=)) (IF (ATOM E) (TRUE) (FALSE)))
                         ((ATOM X)
                          ((LAMBDA (Y)
                                   (COND ((EQ (CAR Y) (QUOTE VAR))
                                          (IF (EQUAL (CADR Y) E)
                                              (TRUE)
                                              (FALSE)))
                                         ((EQ (CAR Y) (QUOTE UAR))
                                          (RESEMBLE% (CAR XX)
                                                     (ENTER (LIST X
                                                                  (QUOTE VAR)
                                                                  E)
                                                            L)
                                                     (CAR EE)
                                                     (CDR XX)
                                                     (CDR EE)))
                                         ((EQ (CAR Y) (QUOTE PAT))
                                          (RESEMBLE% (CADR Y) L E XX EE))
                                         ((AND) (IF (EQ X E) (TRUE) (FALSE)))))
                           (ASSOC X L)))
                         ((NULL X) (IF (NULL E) (TRUE) (FALSE)))
                         ((EQ (CAR X) (QUOTE =QUO=))
                          (IF (EQUAL (CADR X) E) (TRUE) (FALSE)))
```

```
((EQ (CAR X) (QUOTE =DEF=))
 (IF (NULL (CDDR X))
     (RESEMBLE* (CADR X) L E XX EE)
     (RESEMBLE* (IF (NULL (CDDDR X))
                    (CADDR X)
                    (IF (NULL (CDDDR X))
                        (CADDDR X)
                        (CONS (CAR X) (CDDR X))))
                (CONS (CADR X)
                      (CONS (QUOTE PAT)
                            (CONS (CADDR X) L)))
                E
                (CONS (LIST) (CONS (QUOTE =DEF=) XX))
                (CONS (LIST) EE))))
((EQ (CAR X) (QUOTE =AND=))
 (IF (NULL (CDR X)) (TRUE) (RESEMBLE* (CADR X)
                                      L
                                      E
                                      (CONS (CAR X)
                                            (CDDR X))
                                      XX
                                      (CONS E EE))))
((EQ (CAR X) (QUOTE =OR=))
 (IF (NULL (CDR X))
     (FALSE)
     ((LAMBDA (Y)
              (IF (ATOM Y)
                  (RESEMBLE* (CONS (CAR X) (CDDR X))
                             L
                             E
                             XX
                             EE)
                  Y))
      (RESEMBLE* (CADR X) L E XX EE))))
((EQ (CAR X) (QUOTE =NOT=))
 (IF (ATOM (RESEMBLE* (CADR X) L E XX EE))
     (TRUE)
     (FALSE)))
((ATOM E) (FALSE))
((EQ (CAR X) (QUOTE ===))
 (IF (NULL (CDR X))
     (TRUE)
     ((LAMBDA (Y)
              (IF (ATOM Y)
                  (IF (NULL E)
                      (FALSE)
                      (RESEMBLE* X L (CDR E) XX EE))
                  Y))
      (RESEMBLE* (CDR X) L E XX EE))))
```

```
((ATOM (CAR X))
 ((LAMBDA (Y)
          (COND ((EQ (CAR Y) (QUOTE VAR))
                 (FLAG (CAADR Y) E))
                ((EQ (CAR Y) (QUOTE VAR))
                 (RED (CADR Y) E))
                ((EQ (CAR Y) (QUOTE PAT))
                 (RESEMBLE* (APPEND (CADR Y) (CDR X))
                            L
                            E
                            XX
                            EE))
                ((NULL E) (FALSE))
                ((AND) (RESEMBLE* (CAR X)
                                  L
                                  (CAR E)
                                  (CONS (CDR X) XX)
                                  (CONS (CDR E) EE)))
   (ASSSOC* (CAR X) L)))
((NULL (CAR X))
 (IF (NULL E)
     (FALSE)
     (IF (NULL (CAR E))
         (RESEMBLE* (CDR X) L (CDR E) XX EE)
         (FALSE))))
((EQ (CAAR X) (QUOTE *CR*))
 (IF (NULL (CDAR X))
     (FALSE)
     ((LAMBDA (Y) (IF (ATOM Y)
                      (RESEMBLE* (CONS (CONS (CAAR X)
                                             (CDDAR X
                                                  (CDR X))
                                       L
                                       E
                                       XX
                                       EE)
                                 Y))
        (RESEMBLE* (APPEND (CADAR X) (CDR X) (CDR X))
                   L
                   E
                   XX
                   EE))))
 ((NULL E) (FALSE))
 ((AND) (RESEMBLE* (CAR X)
                   L
                   (CAR E)
                   (CONS (CDR X) XX)
                   (CONS (CDR E) EE))))))
```

The function (TRUE) is defined by
(TRUE (LAMBDA () (RESEMBLE% (CAR XX) L (CAR EE) (CDR XX) (CDR EE))))

and is used to examine the right part of the string as the left part is finished. Should the match fail, (FALSE) is defined by

(FALSE (LAMBDA () (QUOTE =FALSE=))).

A number of other auxiliary functions are used in the natural way, and are not explained in detail.

(RESEMBLE X L E) is a function of three variables, respectively a pattern, a dictionary, and a comparand. Its dictionary is more complex than the one used by SIMILAR%, since we now intend to identify the elements listed by the dictionary according to their intended use, which is called their mode. The dictionary has the cyclical form (... variable mode value ...). The value of a variable is a parameter associated with it, its equivalent upon being bound, for example.

Since neither SIMILAR nor ANALOGOUS bind variables, and thus do not need information gained in one part of a list structure for use in another part, the repetitive condition in their definitions was quite simple, being essentially (AND (SIMILAR (CAR X) (CAR E)) (SIMILAR (CDR X) (CDR E))). In the case of SIMILAR%, a variable bound in the one part of a list still had to be compared with the same variable in another part, and thus the repetitive condition took the form (SIMILAR% (CDR X) (SIMILAR% (CAR X) L (CAR E)) (CDR E)), advantage being taken of the fact that the value of SIMILAR% would be a dictionary of the same form as its argument L. All variable bindings made in analyzing (CAR X) are available for use in the analysis of (CDR E).

However, once the analysis of (CAR E) is completed, the dictionary so obtained is final, and cannot be revised if a match fails in (CDR E). This is of no importance in dealing with fixed expressions, but if one expects to continually adjust fragments until all possibility of finding a match is exhausted, it is necessary to convert the list into a string in a reversible manner. It is for this reason that RESEMBLE is defined in terms of the auxiliary function (RESEMBLE% X L E XX EE), whose additional arguments XX and EE are used just for unstringing the arguments X and E. The repetitive condition for RESEMBLE% takes the form (RESEMBLE% (CAR X) L (CAR E) (CONS (CDR X) XX) (CONS (CDR E) EE))), while L is modified as needed when RESEMBLE is applied recursively.

In this preliminary version of RESEMBLE, matches may be formed under the following conditions, in whose enumeration we shall use X to mean the pattern argument of RESEMBLE, and E to denote its comparand.

1. X is the atom ==, which matches any expression.

2. X is the atom =ATO=, in which case X will match any atom. =ATO= is the primitive pattern corresponding to the primitive LISP predicate ATOM.

3. X is an atom declared by the dictionary to be in the variable mode VAR, in which case its value must be EQUAL to E.

4. X is an atom declared to be in mode UAR, an undefined variable. It will match any expression, but its dictionary entry is changed to VAR, requiring that any subsequent occurrences be EQUAL to the first occurrence.

5. X is an atom declared to be a pattern by the mode PAT. Its value must RESEMBLE E.

6. Both X and E are empty lists.

7. X has the form (=QUO= P), and P is EQUAL to E.

8. X has the form (=DEF= P Q1 Q2), in which case Q2 must RESEMBLE E, with the dictionary supplemented by the entry (... P PAT Q1 ...). Or, X may have the form (=DEF= P Q), in which case Q1 and Q2 were equal.

9. X has the form (=AND= P1 P2 ... Pn), in which case all the patterns P1, P2, ..., Pn must RESEMBLE E.

10. X has the form (=OR= P1 P2 ... Pn), in which case at least one of the patterns P1, P2 ..., Pn must RESEMBLE E. In determining whether one of the patterns matches, they are examined in order from left to right and the search stops when either the first matching pattern is found, or none is found to match. In defining patterns recursively, the terminal patterns must be listed first, before the repetitive patterns.

11. X has the form (=== ...), in which case (CDR X) must RESEMBLE either E, (CDR E), (CDDR E), (CDDDR E), or in general some (CD...DR E), if such exists. The shortest CDR chain for which a match is possible will be accepted.

12. X has the form (XXX ...), where XXX is an atom appearing in the dictionary with mode VAR, in which case its value is appended to (CDR X), and the resulting expression must RESEMBLE E. Hence XXX is a synonym for an initial fragment of X.

13. X has the form (XXX ...), where XXX is an atom appearing in the dictionary with the mode VAR, in which case (CDR X) is required to RESEMBLE some terminal fragment of E, and the corresponding initial fragment of E is entered as the value of XXX in the dictionary and its mode is changed to a conditional form of VAR.

14. X has the form (XXX ...), where XXX is an atom appearing in the dictionary with the mode PAT, in which case its value is appended to (CDR X), and the resulting patern must RESEMBLE E.

15. X has the form ((MOPS (XXX) (YYY) ... (ZZZ)) WWW), in which case at least one of the fragments formed by appending (XXX) to (WWW), (YYY) to (WWW), ..., (ZZZ) to (WWW) must RESEMBLE E. The fragments from which we choose are tried from left to right, and the first match made is accepted.

The preliminary version of RESEMBLE thus combines the ability to define patterns, analyze list structures, compare different pats of an expression, and test for specific constituents which were possessed to some degree by EQUAL, SIMILAR, SIMILAR* and ANALOGOUS. It employs a search strategy which will guarentee, if no match was found, that there exists absolutely no combination of fragments or variable definitions for which a match is possible. When a match is found, the leftmost fragments satisfying the match will be as small as possible, and the first possible alternative specified by each OR will be taken.

The final version of RESEMBLE differs from this preliminary version only in the respect that more alternatives are listed, both in the possible modes available, and the types of patterns which may be admitted.

THE PATTERN RECOGNITION FUNCTION (RESEMBLE P L E)

Having now seen a variety of pattern matching schemes which allow the specification of the form and content of a list as well as the LISP functions implementing these schemes, we now turn our attention to the systematic arrangement of these schemes into a formal system.

Every pattern is constructed from primitive patterns so that it suffices to describe the primitive patterns and draw attention to the rules by which they are compounded into more complex patterns. The principal mechanism is list formation, since a series of patterns may be separated by blanks and enclosed within parenthesis to form a list and hence a pattern corresponding to the indicated list structure. However there are two other principles of pattern composition, substitution and Boolean combination. All three principles may be used, to form patterns of arbitrary complexity.

By the principle of substitution we mean that atomic patterns may represent other, likely more complicated patterns. A method for introducing definitions is then necessary as well as for quoting such patterns when their substitution is not intended. Especially, a definition may make use of the defined term, and the existence of recursively defined patterns is thereby possible.

By the principle of Boolean combination, we mean Boolean composites of statements of the form "the pattern P has matched E" together with supplementary statements about the circumstances of variable definition.

The term "primitive pattern form" is reserved for the list structure necesary to describe patterns generated according to the latter two principles because such patterns are not to be dissected into their atoms in order to make a match; rather they contain characteristic identifiers such as =OR= which indicate to the processor the necesity of appropiate treatment.

More accurately, a pattern form is the representative of a class of patterns, since only certain elements of the form are specified and the remaining components may be substituted by any valid pattern. A pattern form is strictly a skeleton, requiring such substitution before becoming a pattern.

As we have noted, recursive pattern definition is afforded by the ability to name entire patterns and to indicate alternatives between different patterns. In order to obtain a finite recursion, one must allow a choice between a terminal pattern and a repetitive pattern, the later being one which refers to the entire pattern as one of its constituents.

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

First among the primitive patterns are the constants which test the type of expressions. These include:

== ——————— the universal pattern which matches any expression.

=ATO= —— a pattern which we will match any atom.

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 can not be used directly as characters in the language, so that one might wish to introduce pattern synonymns for them.

=NUM= —— a pattern which will match any numeral.

=ORD= —— a pattern which will match any ordered set.

=BLA= —— a pattern which will match a blank space.

=LPR= —— a pattern which will match a left parenthesis.

=RPR= —— a pattern which will match a right parenthesis.

If raw input is available in the form of BCD character strings, the processor must have 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 cyclic list of period 3, in which we find the repeated configuration (... 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 called 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 C —— 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 C 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 M --- 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 be 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 programer generally need not concern himself with this point since UAR variables are usually listed separately as the argument I of CONVERT, the other variables 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).

Of the foregoing modes, VAR allows synonyms for expressions, and PAT allows synonyms for patterns; in the former case matching takes place by EQUAL and is terminal, while in the latter matching takes place by RESEMBLE and is repetitive. The mode UAR allows the comparison of different subexpressions of a comparand, again by EQUAL. However it is clearly possible to treat variables in other manners, as regards their selection, retention, and comparisons.

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 comparand will be available to REPLACE, and furthermore if X occurs several times all its comparands will be required to be equal. Strictly speaking, the use of the configuaration ... X PAV == ... in the dictionary renders the mode UAR redundant.

X BUV (P ...) --- the bucket variable mode. It is similar to the mode PAV, but rather than requiring that the same comparand 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 UNO (X1 X2 ... Xn) --- the unordered variable mode. X will match E if any one of the patterns X1, X2, ..., Xn matches E. That being the case, the successful pattern is removed from the value. If it is desired to record E, the X's themselves may be placed in the PAV mode. If the value is an empty list, the match fails. For example, if we have a dictionary (... X UNO (A B) ...), (X + X) will match either (A + B) or (B + A), which would be equivalent expressions on account of the commutativity of addition.

If one admits numbers, it is necessary to make provision for the usual arithmetic comparisons. One might introduce patterns which would match only strictly positive numbers, or negative numbers or any of a number of combinations possible. Here we introduce a mode for the comparison of relative magnitude.

Finally, we allow the possibility of introducing new modes defined through a rule set as a CONVERT program.

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 RUL R --- the rule mode. R is a rule set defining the condition 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 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 they are permitted to appear in a different order but not with different multiplicities. A variation of this idea, X MSU P, the maximal subset mode, results in a list of all those elements of the matched lists which appeared at each occurrence of the symbol X.

Whenever an atom is encountered in a pattern [illegible] it does not appear in the dictionary L, that atom will [illegible] only itself. Thus all atoms are primitive patterns; those which appear in the dictionary L are to be matched as their mode specifies, while all others match only themselves, unless they have fixed meaning, such as the atoms == or =ATO=.

() —— An empty list will RESEMBLE only itself.

There is a series of primitive pattern forms which match expressions, and which allow the formation of complex patterns by means of the substitution principle.

(=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 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 by binary tree, or (=DEF= (E) ((%OR% () (== == E)))) is the definition of a list of even length. The usual convention is followed, that expression names are atoms, fragment names are enclosed in parentheses.

A recursive pattern definition usually involves a choice between a terminal pattern and a repetitive pattern, a decision which is made by one of the pattern forms =OR= or %OR%. 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. However, the other Boolean connectives may be used as well, and permit the formation of complex patterns by means of Boolean connectives.

(=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 br 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 Pi fails to match E, all variables which may have been tentatively defined by means of Pi are forgotten when Pi+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 not match the variables which it binds do not matter because the overall match fails.

The next group of pattern forms match fragments, and therefore cannot match atoms; they must always appear as part of a larger pattern. There is only one fixed 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, (=== (===) ===) matches any list containing at least one sublist, and so on.

We may also specify fragment variables. This is done through the arguments N 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. By enclosing the name of a fragment in parentheses we remind ourselves that fragments do not exist independently of a list of which they form a part.

The way in which a fragment variable is to be regarded as representing the fragment of a pattern in which it appears and thus to be eventually matched with a fragment of some expression is also specified by the 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 EQUAL, 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) VAR EE --- the undefined variable mode. In general, the value EE is the empty list (). The fragment XXX will match whatever fragment is larger than EE. Moreover, if XXX appears several times in a larger pattern it must always match the same fragment. If upon seeing a subsequent occurrence of the fragment XXX one finds that the second fragment needed does not correspond to the fragment first found, he may revise the original estimate by trying a larger fragment for the initial occurrence of XXX. Hence, we always take EE as the estimate of the fragment, usually initially the null fragment, and try successively larger fragments by adding elements to the end of EE until a satisfactory candidate is found, if it is at all possible.

(XXX) PAT P --- the pattern fragment mode. P must be a list; it is substituted in place of the atom XXX as a fragment, resulting in an augmentation of the pattern in which XXX occurs by several new expressions, each of which is a pattern.

(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.

As is the case for atomic symbols which represent expressions, those representing fragments may initiate various kinds of concomitant modification of the dictionary.

(XXX) BUV P --- the fragment bucket variable mode. It differs from 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 its determination is desired.

(XXX) REP (P K) --- the repeat mode. It is expected 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).

The 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 supposed 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. However, they must all match exactly the same fragment E. For example, (=== (AND (X ==) (== Y)) ===) will match (A (X Y) B) but not ((X A) (B Y)), whereas (=== (A (X ==) (== Y)) ===) will match both.

(*AND* P1 P2 ... Pn) —— when this combination appears as a fragment of a larger pattern P, one must regard P1, P2, ..., Pn in turn as fragments (*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= (O) (*OR* () (== == O)))) matches a list of odd length; (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 containing (*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).

This completes the survey of the primitive patterns and pattern forms. Composites of the primitive patterns or pattern forms may be formed either in the list, Boolean, or substitutional sense. Once a match is obtained to a given pattern, a dictionary is produced which may then be used to make substitutions into a given skeleton.

PRELIMINARY DISCUSSION OF REPLACE

Having completed the matching process by means of the function RESEMBLE, one obtains as its value the final dictionary on which appears, among other things, those initially undefined variables which have been bound. They are now available for use, which in the context of the transformation rule format consists of substituting them into another expression called a skeleton.

As was true of the matching process, it is possible to carry out the substitutional process with varying degrees of complexity. For example, one of the simplest functions of this nature is (SUBS X Y L) which substitutes X for the atom Y wherever it occurs on the list L.

```
(SUBS (LAMBDA (X Y L)
              (IF (NULL L) L (CONS (IF (EQ Y (CAR L)) X (CAR L))
                                   (SUBS X Y (CDR L))))))
```

On the other hand, SUBS will only effect its substitution in the top level of the list L, and if one wishes substitution on all levels, he needs a more powerful function such as (PERSUBS X Y L) defined by

```
(PERSUBS (LAMBDA (X Y L)
                 (COND ((NULL L) L)
                       ((EQ Y L) X)
                       ((ATOM L) L)
                       ((AND)
                        (CONS (PERSUBS X Y (CAR L)) (PERSUBS X Y (CDR L))))))))
```

Should one wish to make a simultaneous subsitution of several atoms, he could replace the arguments X and Y of these functions by an alternating dictionary and define PERSUBST by

```
(PERSUBST (LAMBDA (D L)
                  (COND ((NULL L) L)
                        ((ATOM L) (ASSOC L D))
                        ((AND)
                         (CONS (PERSUBST D (CAR L)) (PERSUBST D (CDR L))))))))
```

where he has defined (ASSOC X D) in order to read the dictionary, replacing X by its equivalent if such exists, and leaving it unchanged otherwise.

```
(ASSOC (LAMBDA (X D) (COND ((NULL L) X)
                           ((EQ X (CAR L)) (CADR L))
                           ((AND) (ASSOC X (CDDR L)))))))
```

From these functions it is but a simple step to more complicated substitutions. For example, more than simply substituting an expression for an atom found on the dictionary, one might regard the new expression as one in which further substitution should be made. In this way a single expression could be expanded recursively into a very complicated structure. Moreover, the atom being substituted might be taken as a fragment rather than an expression, so that a single atom would be replaced by a series of several expressions, producing an expression which in turn might have been left as it was or subjected to a further replacement process.

With these considerations in mind, we introducee a preliminary version of the function (REPLACE D S), expecting to see mode definitions in its dictionary L; either EXPR to indicate a simple replacement, or SKEL indicating that after replacement the resultant expression will be subjected to another cycle of replacement. The dictionary entries are moreover distinguished as expressions or fragments, according to whether their name in enclosed in parentheses or not.

If we take for the dictionary

```
D = (X EXPR (* * *)
     Y SKEL (X X)
     (XXX) EXPR (* *)
     (YYY) SKEL (X X)),
```

we may illustrate the differing values of (REPLACE D S) according to differnet choices of S.

(X X) will produce ((* * *) (* * *)),
(y y) will produce (((* * *) (* * *)) ((* * *) (* * *))),
(XXX X) will produce (* * * *), while
(YYY X) will produce ((* * *) (* * *) (* * *)).

The definition of this preliminary version is

```
(REPLACE (LAMBDA (D S)
                 (COND ((ATOM S)
                        ((LAMBDA (Y) (COND ((EQ (CAR Y) (QUOTE EXPR)) (CADR Y)
                                           ((EQ (CAR Y) (QUOTE SKEL))
                                            (REPLACE D (CADR Y)))
                                           ((AND) S)))
                          (ASSSOC S D)))
                       ((NULL S) S)
                       ((EQ (CAR S) (QUOTE =QUOT=)) (CADR S))
                       ((ATOM (CAR S))
                        ((LAMBDA (Y)
                                 (COND ((EQ (CAR Y) (QUOTE EXPR))
                                        (APPEND (CADR Y) (REPLACE D (CDR S))))
                                       ((EQ (CAR Y) (QUOTE SKEL))
                                        (REPLACE D (APPEND (CADR Y) (CDR S))))
                                       ((AND)
                                        (REPLACE (CONS (REPLACE D (CAR S))
                                                       (REPLACE D (CDR S))))))
                          (ASSSOC* (CAR S) D)))
                       ((AND) (CONS (REPLACE D (CAR S)) (REPLACE D (CDR S)))))))
```

More complicated functions are needed to search the dictionary argument of REPLACE. ASSSOC searches for expression names and returns a list consisting of the associated mode and value when the name is found, while ASSSOC* searches for fragment names.

```
(ASSSOC (LAMBDA (X L) (COND ((NULL L) (QUOTE (NIL NIL)))
                            ((EQ X (CAR L)) (LIST (CADR L) (CADDR L)))
                            ((AND) (ASSSOC X (CDDDR L))))))

(ASSSOC* (LAMBDA (X L) (COND ((NULL L) (QUOTE (NIL NIL)))
                             ((ATOM (CAR L)) (ASSSOC* X (CDDDR L)))
                             ((EQ X (CAAR L)) (LIST (CADR L) (CADDR L)))
                             ((AND) (ASSSOC* X (CDDDR L))))))
```

The dictionary used by REPLACE may originate from any source, but in its intended use this source is the collection of variables identified by RESEMBLE, and accordingly one has use of a function to coordinate the operation of RESEMBLE and REPLACE. First an expression is tested to see if it corresponds to a given pattern, and if so, parts of it are collected for use in the construction of the transformed expression. This process may be applied by consulting but a single pattern-skeleton pair, or it may be amplified to include the testing of a series of such pairs. In the event the first transformation is not applicable, perhaps the second will be, and if not then the third, and so on. If no transformation rule is applicable, we leave the original expression unchanged.

This search is carried out by the function CONVERT*, which is a satellite of CONVERT, whose argument is a rule set. Intermediary functions between CONVERT and CONV* serve to prepare the dictionary L from the arguments M and I of CONVERT, bind the expression E being transformed, and some other similar housekeeping chores.

```
(CONVERT* (LAMBDA (T L E) (CONV* T)))

(CONV* (LAMBDA (U) (COND ((NULL U) E)
                         ((ATOM (RESEMBLE (CAAR U) L E)) (CONV* (CDR U)))
                         ((AND) ((LAMBDA (L*) (REPLACE (EDIT L*) (CADAR U)))
                                 (RESEMBLE (CAAR U) L E))))))
```

The replacement process is quite straightforward, but in CONVERT it is complicated by the fact that it also involves the flow of control in the CONVERT program. These complications fall in three groups, depending upon the programming style involved. First, one may form an intermediate result and specify that the transformation process is to be applied once again to this new argument. During the reapplication one may possibly wish to bind some variables ... either as patterns or as skeletons ..., or he may wish to start anew with all variables in their original states. Thus there is both a binding and a repetition process to be considered, as well as their interaction one with the other.

Second, one may wish to avail himself of functional notation, especially in its prefix form which is already closely adapted to the left to right scanning process which REPLACE employs. At the sime time that we check CAR of a skeleton to see if it is an atom which should be replaced by a fragment, we could as well check to see if it were a function name. Were this so, its arguments -- CDR of the skeleton -- could be replaced and that list used as the expression to which a set of transformation rules should be applied. Thanks to the ability to reference this function within its defining rule set, by first replacing its arguments and in turn taking them as a new expression to be transformed, one has precisely the same capability for recursive function definition that he had in LISP itself. By looking two levels deep, at CAAR of a skeleton, we can even recognize functions taking fragments as values, and thus functions, recursively defined or otherwise, enjoy a perfectly natural existence in any part of a CONVERT program.

A third alternative satisfies the desire to construct an expression through a series of steps, not all of which produce elements of the expression, and which produce the elements in their order of execution, rather than their written order. Such nonproductive steps might consist in the storage or retrieval of information, or the decision as to which series of steps to follow next; in short one might like to write a program to perform some task, rather than using functional notation or expressing it as a series of successive transformations of form. Thus one may have skeletons which are programs, describing in iterative manner how to transform an expression, which require prmitive skeletons and skeleton forms corresponding to the iterative process.

THE SUBSTITUTION FUNCTION (REPLACE D S)

The arguments of the complete function REPLACE are still 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.

There are several categories of skeletons and skeleton forms, by which, we recall, is meant an expression which becomes a skeleton after appropriate substitution of its constituents. One of these categories comprises 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. Then there is a category of skeleton forms comprising functions and their arguments, which permits the use of a LISP-like functional

notation within a skeleton. Another category allows the introduction of single symbols to stand for other quantities, be they expressions or skeletons. There are skeleton forms which allow the formation of intermediate results and their analysis. Finally, other skeletons are used for iterative programming.

All primitive skeletons and skeleton forms may also be 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 the ==) and four letter combinations for primitive skeletons and skeleton forms. This choice is also reflected in the choice of mode names, which are again of three letters for patterns and four letters for skeletons. In dealing with patterns, it was found useful to use single characters as atoms, but triple characters such as XXX for fragments, unless some more mnemonic compination occurred in a particular application. Again, it was useful to distinguish fragments from expressions, and for this reason we enclose their names between equal signs when they refer to expressions, as for example =QUO=, or with asterisks when they refer to fragments, as *AND*.

We now list the possible skeletons in detail, commencing with the constant skeletons.

=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 synonymn skeleton, which stands for a left parenthesis.

=RPAR= --- synonymn for a right parenthesis.

=BLNK= --- synonymn for a blank.

As in the case of the atomic patterns, this list could (presumably) be 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. Shown in the present list are two types of skeletons; synonymns for delimiters which cannot be quoted, and referents to objects which exist within a CONVERT program.

For the variable atomic skeletons, there are only two modes.

X EXPR S —— the expression mode. Every quantity which was in the VAR mode or was changed to the VAR mode, such as those previously in the UAR or PAV modes, is transformed to the EXPR mode when the value of RESEMBLE. Is edited to become the value 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 symbol X stands for the entire skeleton S, which is placed instead of X, and then modified (replaced) again using 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.

Among the skeleton forms, there is a large number which are in reality functions specified in the LISP prefix form, (F A1 A2 ... An), where F is the name of a 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 with 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 in A but not in B. Repeated elements of A appear in the compliment with their original multiplicity.

(=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) —— subtracts 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 an not a number, the dimension of the array is the length of N, and whose elements depend upon the corresponding list elements of nn. If S is missing, zeroes fill the array.

All these function skeletons follow the LISP convention in 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.

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

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

(=ITER= I1 N1 I2 N2 ... S) —— in this skeleton form, the I1 are variables which serve as indices, while Ni are their corresponding ranges. For each combination of possible values of the skeleton S, which may contain them as variables, S is evaluated and a list is 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 of N1 is a numeral, I1 takes the values 1, 2, ... ,Ni inclusive, while if N1 is a set, I1 takes successively the values (CAR N1), (CADR N1), ... 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 the introduction of temporary definitions of skeletons or expressions. They accomplish locally the same results that the modes EXPR and SKEL do 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 not replaced. Rather they are effectively quoted. A variant of this skeleton form is (=QUOT= S) in which S is copied without replacement, so that in this manner primitive skeleton names and primitive forms 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', etc 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', ..., we mean the values of S1 S2, etc. After replacment.

The three skeleton forms =QUOT=, =EXPR=, =SKEL=, differ with respect to the times at which the skeletons S1 are evaluated. There are actually four possible combinations, accoriding to whether S1 is evaluated before the dictionary is altered, and according to whether it is evaluated again after it is encountered in the evaluation of the skeleton S. =QUOT= evaluates the skeletons S1 on neither occasion, while =SKEL= evaluates them on both, and =EXPR= evaluates only at the time of dictionary formation. The fourth possibility is missing, but can be realized by using =SKEL= together with (=QUOT= S1).

For both the skeleton forms (=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.

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.

(=BEGN= S) —— the recursive reentry to the entire program. The replacement of this skeleton form 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 UAR status or to whatever other mode they originally had and which might have been altered during the course of execution of the program.

(=CONT= S K1 R1 K2 R2 ...) --- continue to a new rule set. Again the skeleton S is to be replaced and taken as the new argument E of CONVERT. However the rule set to be applied depends upon the arguments K1, R1, etc. If these arguments are missing and we see only (=CONT= S), we reapply the rule set currently in use. The difference with respect to (=BEGN= S) is that all variables are retained in their current form. However, if (=CONT= S K) is written, we apply rule set K in the continuation.

In the general form, we continue with the rule set K1 which is defined as R1. In addition, there is available the convenience of introducing 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 XXX) (=CONT= S)), if X and XXX were originally in the mode UAR they will have been changed to the mode VAR and eventualy to EXPR when S is evaluated. However, the second time the rules are applied, they will no longer be in the UAR mode, but will retain their new identification in the VAR and EXPR modes.

(=REPT= S K1 R1 K2 R2 ...) --- repeat the designated rule set. The skeleton S is replaced and taken as the new argument E of CONVERT, the rule set R1 being applied undert the name K1. At the same time the additional definitions K2 for R2 and so on are placed in effect. If R1 is missing, and our skeleton has the form (=REPT= S K1) the already defined rule set K1 is applied in the case we abbreviate the skeleton to (=REPT= S), the current rule set is reapplied. =REPT= differs from the corresponding form with =CONT= in that the new rule set is applied after the variables have been restored to their original modes and values. At this point a conflict makes itself apparent. With =CONT= all previously defined variables are retained. However, =BEGN= undertakes to restore all variables to their original condition.

The conflict exists because the skeleton form (=EXPR= N1 S1 ... S) guarantees 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 due to the skeleton forms using =BEGN=, =REPT= or =CONT=. A similar situation holds with respect to =SKEL= and =QUOT= as well as for =ITER= and =ARRY=. On the other hand, =BEGN= provides a convenient synonym whereby the program may be reentered recursively, so this demands that the program be in the same state every time it is reentered, 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=

There are a number of skeleton forms which operate the CONVERT program feature. A program is itself a skeleton form, containing the identifier =PROG=, as well as a series of skeletons which are to be replaced one by one. Such skeletons are generally operator skeletons, effecting some input-output operation, modification of the program variables, or other permanent changes. Others direct the flow of control, or terminate the execution of the program.

(=PROG= (XXX) S1 S2 ... Sn) --- the skeleton form which introduces a program. (XXX) is a list of program variables, which are atomic skeletons declared to be in the EXPR mode during the execution of the program. They may be indicated to represent fragments by enclosure in parentheses, according to the prevailing convention. Initially their values are respectively an empty list or an empty fragment, but their values may be modified by the operators =SETQ= or %SETQ%.

If a program variable is already in use as a skeleton, its old value is pushed down upon entrance to the proram and automatically restored upon the completion of the program. S1, S2, ..., Sn are statements in the program which consist of skeletons which are to be replaced in order, starting with the first. In addition to skeletons to be replaced, program elements may also consist labels, used as heading or location markers, "GO TO" statements, and "RETURN" statements.

Since a (=PROG= ...) skeleton form may contain any permisible skeleton, =PROG=s inside =PROG=s are permissible to any depth.

(=GOTO S) —— a "GO TO" statement which causes the skeletons following the label S to be replaced in sequence, rather than those following the GO TO statement itself. The label S is replaced before the transfer of control is made, meaning that we have allowed a "computed GO TO" of arbitrary complexity. Should the computed label not belong to the =PROG= skeleton form in which it occurs, automatic regression is made to the next higher level if it exists, and the label is sought on that level. If it is still not found yet another regression is made until either it is found or else the program will be automatically terminated if no such label exists on any higher level.

(=RETN= S) —— the program is terminated, the skeleton S is replaced, and returned as the value of the program. If no =RETN= skeleton is found and we arrive at the end of the program, the value of the (=PROG= ...) is the value of the last skeleton found.

It is possible to write (=RETN= (=GOTO= S)), a return whose argument is a GO TO statement. S is replaced in the current program level, the resulting GO TO being returned to the higher level, where it causes the appropriate transfer. This is another admissable type of computed GO TO.

(=SETQ= A S1) makes a REPLACEment of the skeleton S1, and its value is assigned to A; A is considered =QUOT=ed, and it is not REPLACEd. The value of (=SETQ= A S) is the value of its argument S, suitably replaced. If A is an atom it represents a variable, whereas if it is an atom enclosed in parentheses it represents a fragemnt and is so treated when it occurs in a skeleton.

Generally a skeleton requires a certain amount of decision making, in order to know whether to repeat a previously executed portion of the program, or to go on to replace the next skeleton. Such decisions can be made by forming the quantity upon which the decision will be based and studying it by means of a rule set containing appropriate GO TO's among its skeletons. This decision would normally be made with the help of a =REPT= skeleton form, but there are some simple configurations of such frequent occurrence that it is convenient to introduce their corresponding skeleton forms directly.

(=WHEN= S P S1 S2) —— the skeleton S is replaced and compared to the pattern P. If it matches, the skeleton S1 is replaced; if not S2 is replaced if S2 is missing, S is retained unaltered. The =WHEN= skeleton form is an abbreviation for

(=CONT= S * ((P S1) (== S2)))

(=COND= S P S1 S2) —— the skeleton S is replaced, then the variables are restored to their original values and modes, following which S is compared to the pattern P. If it matches S1 is replaced, otherwise S2 is replaced unless it is missing, in which case S is preserved intact. The =COND= skeleton form is an abbreviation for

(=REPT= S * ((P S1) (== S2))).

=COND= differs from =WHEN= in this way: when comparing S against P, =WHEN= uses the most recent dictionary, as =CONT= does, and =COND= uses the original dictionary, as =REPT= does. It should be emphasized that =BEGN=, =REPT= or =CONT= can be used directly rather than the short forms =WHEN= and =COND=.

Having exhausted the skeletons and skeleton forms which refer to expressions, let us now consider those which refer to fragments. In general there is a series of skeleton fragments entirely 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 an example.

First we have the analogue of =SAME=.

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

Atoms may be used as the names of fragments as well as function names, in both of which cases they occur as CAR of a list, and are used in the manner specified by their mode declarations when they appear in the dictionary.

(XXX) EXPR (EEE) —— the expression mode. 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 skeleton mode. The fragment SSS is substituted for the symbol XXX in any skeleton in which XXX appears, before replacement is made.

(XXX) CONT R —— the continue mode. This is the mechanism by which functions are defined in CONVERT. R is a rule set. If the skeleton contains the skeleton form (XXX A1 A2 ... An), a list is constructed by replacing (A1 A2 ... An) and using it as a new expression E, and evaluating the skeleton form (=CONT= E * R). The difference between the CONT mode

and the REPT mode lies in their treatment of free variables; CONT preserves previously existing variable definitions; REPT erases them. If the functions 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 --- the repeat mode. Another mode which permits function definitions, which is the same as CONT except that the values of any variables which have become defined in in the course of the program, are lost. Rather, all such variables are restored to their original state when the CONVERT program was first entered, with the exception of those bound by =EXPR= or =SKEL=.

Those skeleton forms corresponding to functions whose values are fragments rather than exprrssions are identified by *'s which form part of their names.

(*ANUL* S) --- after replacement of the skeleton S, nothing is done to the main skeleton, which is equivalent to inserting an empty fragment in place of (*ANUL* S). Clearly such a skeleton only makes sense if S is an "operator skeleton". By this is meant that in the course of replacement of S some permanent changes or auxiliary changes are effected. For example, S might involve =PRINT= and in this way one could write on the output tape without retaining a copy of what was written in the expression being developed.

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* *PROG* *WHEN*
SETQ *COND*

References

CONVERT — Adolfo Guzman and Harold V. McIntosh. Comm. ACM 9, 604-616 (1966).

Adolfo Guzman. CONVERT. Professional Thesis (Spanish), Instituto Politecnico Nacional, Mexico City, 1965.

LISP — J. McCarthy. Recursive functions of symbolic expressions and their computation by machine. Comm. ACM 3, 184-195 (1960).

COMIT — V. H. Yngve et al. An introduction to COMIT programming. The MIT Press, Cambridge, Mass. 1963.

SNOBOL — D. J. Farber, R. E. Griswold, and I. P. Polonsky. Journal ACM 11, 21-30 (1964 SNOBOL, A String Manipulation Language.

Formula ALGOL — A. J. Perlis, R. Iturriaga, T. A. Standish. A preliminary sketch of Formula Algol. Carnegie Inst. of Technology, Pittsburgh, Pa., April 1965, 52 pp.

PANON-1B — A. Caracciolo di Forino, L. Spanedda and N. Wolkenstein. A programming language for Symbol Manipulation. Comm ACM 9 (1966).

AXLE — K. Cohen and J. H. Wegstein. An axiomatic language for string transformations. Comm ACM 8, 657-661 (1965).

MARKOV ALGORITHMS — Markov, A. A. Theory of Algorithms. Academy of Sciences of U.S.S.R. (English translation by NSF and US Dept. of Commerce), Vol. 42, Moscow, 1954.

M. I. T. REPORTS

Adolfo Guzman and Harold V. McIntosh. A program feature for CONVERT.
Project Mac-Memorandum M-305 (A.I. 95); April 1966.

Adolfo Guzman. POLYBRICK: Adventures in the domain of Parallelepipeds.
Project Mac-Memorandum M-308 (A.I. 96); May 1966.

INSTITUTO POLITECNICO NACIONAL (Mexico) REPORTS

Raymundo Segovia and Harold V. McIntosh.
Computer Analysis of Finite Groups. Esc. Sup. de Fisica y Mat. (1966)

Daniel Conrad. CONVERT functions for generating figures.
Ciencias de la Informacion y la Computacion 1, 1 (April-June 1966).
U. N. A. M. (Mexico)

Daniel Conrad. LISP functions for generating and plotting figures.
Program Note No. 2, Centro Nacional de Calculo (I. P. N.) August 1964.

John Williams. LISP pattern recognition functions.
Program Note No. 1, Centro Nacional de Calculo (I. P. N.) August 1964.

A P P E N D I X I

Some examples of CONVERT programs.

(REVERS L)	reverses the order of elements on a list.
(SUBSETS S)	produces all the subsets of S.
(FORMUL L)	transforms n-ary infix algebraic expressions to the binary prefix form.
(PATHS A B M)	lists all paths between the points A and B in the network whose primitive links are given by M.

The function (REVERS L) reverses the top level elements of its argument L; i. e., if L is (A B C D), then (REVERS L) is (D C B A).

In this example, the first argument of CONVERT is (): the dictionary M is empty. In the dictionary I, second argument of the function CONVERT, we declare X and (XXX) to be variables in the UAR mode.

The third argument is L, the expression we want to reverse. Finally, the fourth argument contains one rule-set named C1, which contains a single rule:

```
( (X XXX)  ((*BEGN* (XXX)) X) )
```

The left half of this rule is the pattern (X XXX), which dissects the list we want to reverse into CAR and CDR. If this dissection is possible, that is, if (X XXX) matches, we proceed to replace the skeleton ((*BEGN* (XXX)) X), which is composed of two pieces, (*BEGN* (XXX)) and X.

The value of (*BEGN* (XXX)) is computed recursively, applying the entire original CONVERT program to (XXX) ---we are reversing the list (XXX)--- and then taking the contents of this result as its value.

The value of X is the expression which matched with it during the pattern comparisson, that is, the CAR of the list to be reversed.

Having computed the values of the skeletons (*BEGN* (XXX)) and X, we simply put them together, as the skeleton ((*BEGN* (XXX)) X) orders, and this is the result of our transformation.

```
(REVERS (LAMBDA (L) (CONVERT
(LIST)
(QUOTE (  X (XXX)  ))
L
(QUOTE ( C1 (
     ((X XXX) ((*BEGN* (XXX)) X))
      )))
      )))
```

```
r convrt
W 208.4
load ((revers))
 NIL

revers ( () )
 NIL

revers ((1 2))
 (2 1)

revers ((a b c e f g h I j))
 (J I H G F E C B A)

revers(( a b c d (1 2) e f g (3 4) h I j))
 (J I H (3 4) G F E (1 2) D C B A)

stop
R 2.200+6.700
```

(SUBSETS S) produces all the subsets of the set S.

C1 is a set of two rules, the first of which says that the only subset of an empty set is an empty set.

The second rule identifies X with the CAR of S, and XXX with its CDR; it then computes (=BEGN= (XXX)) --the subsets of (XXX)-- and binds such result to the variable (AAA); replacement is then made on the skeleton

(AAA (*ITER* (J) (AAA) (X J)))

This skeleton contains two parts or halves; the first is simply AAA; that is, the subsets of the CDR are also subsets of the whole list; the oter part (*ITER* (J) (AAA) (X J)) adds X to each element J of the subsets of the CDR.

```
(SUBSETS (LAMBDA (S) (CONVERT
(LIST)
(QUOTE (
     X (XXX)
     ))
S
(QUOTE ( C1 (
( () (()) )
( (X XXX) (=SKEL= (AAA) EXPR (=BEGN= (XXX))
                  (AAA (*ITER* (J) (AAA) (X J))) ))
)))
) ))
```

```
r convrt
W 212.1
load ((subset))
 NIL

subsets (())
 (NIL)

subsets ((1))
 (NIL (1))

subsets ((1 2))
 (NIL (2) (1) (1 2))

subsets ((1 2 3 4))
 (NIL (4) (3) (3 4) (2) (2 4) (2 3) (2 3 4) (1) (1 4) (1 3)
 (1 3 4) (1 2) (1 2 4) (1 2 3) (1 2 3 4))

subsets ((a b c d e))
 (NIL (E) (D) (D E) (C) (C E) (C D) (C D E) (B) (B E) (B D)
 (B D E) (B C) (B C E) (B C D) (B C D E) (A) (A E) (A D) (A
 D E) (A C) (A C E) (A C D) (A C D E) (A B) (A B E) (A B D)
 (A B D E) (A B C) (A B C E) (A B C D) (A B C D E))

stop
R 3.283+9.633
```

(FORMUL L) is a LISP function defined through CONVERT which will transform an algebraic formula written in infix notation to the binary prefix form. The argument L, which is the formula to be transformed, may use any variables, connected by the symbols PL (plus), MI (minus), TI (times), DI (divide) or PO (power), whose binding strength increases in the order mentioned. Thus the usual conventions for avoiding parentheses in associative or distributive configurations apply.

The CONVERT function implementing FORMUL is

```
(FORMUL (LAMBDA (L) (CONVERT
 (QUOTE (
      LL      SKEL      (=REPT= (=WHEN= (LLL) (L) L))
      RR      SKEL      (=REPT= (=WHEN= (RRR) (R) R))
      ))
 (QUOTE (
      L R (LLL) (RRR)
      ))
L
 (QUOTE (*0 (
      ((LLL PL RRR) (PLU LL RR))
      ((LLL MI RRR) (MIN LL RR))
      ((LLL TI RRR) (TIM LL RR))
      ((LLL DI RRR) (DIV LL RR))
      ((LLL PO RRR) (POW LL RR))
      )))
      )))
```

The binding strength hierarchy is determined by the order in which the rules are written; thus no attempt will be made to locate a product in any expression which contains a sum, and so on. Since CONVERT always makes leftmost matching fragments as small as possible, association is always made to the right; (A PL B PL C) would be transformed to (PLU A (PLU B C)). Any expression not connected by one of the five admissable algebraic connectors is left unchanged. The choice of the names of the connectors is occasioned by LISP 1.5's aversion for the pure algebraic signs.

The skeletons LL and RR serve to analyze the arguments of the algebraic connectives further as formulas, so that formulas may be formed recursively with formulas as subexpressions. Moreover, when the fragments LLL or RRR contain only one expression, it is necessary to avoid endowing them with a spurious pair of parentheses, wherefore the configuration (=WHEN= (LLL) (L) L).

```
r convrt
W 214.6
load ((formul))
 NIL


formul ((x pl y pl z pl 7 ti x po 2))
 (PLU X (PLU Y (PLU Z (TIM 7 (POW X 2)))))


formul ((x ti (y pl z pl 28) po 2))
 (TIM X (POW (PLU Y (PLU Z 28)) 2))

formul (((x pl 2) ti (z pl y po 3) pl 3 ti t))
 (PLU (TIM (PLU X 2) (PLU Z (POW Y 3))) (TIM 3 T))


formul (((x pl 2) ti (x pl z po 3) pl 3 ti t mi 5))
 (PLU (TIM (PLU X 2) (PLU X (POW Z 3))) (MIN (TIM 3 T) 5))


formul (( y pl 3))
 (PLU Y 3)


stop
R 4.100+7.933
```

A rat, upon being introduced to a new maze, is immediately interested in learning all the paths leading between various pairs of points. The interest of a mathematician is more modest and yet more systematic, since once a program is given for finding all the paths between one pair of points, that pair being arbitrary, there exist known methods for solving the general case. To find all paths between a pair of points, we may suppose that the points are in fact identical, and that moreover, we are not interested in loops which lead from a point to itself. Such not being the case, it would be possible that we have a pair of points which are directly lin ked, through a primitive path. Neither being the case, we might consider the image set of our initial point, by which we mean the set of all points linked to the initial point by a non-null, primitive path, and the counterimage set of the final point, meaning the set of points linked to the final point by a single primitive path. If we form the cartesian product of the image set and the counterimage set, and suppose known all paths linking those pairs of points ---a point in the image set one step away from the initial point to a point in the counterimage set, likewise one step away from the final point--- we then have a recursive solution to the original problem.

For mathematical ---or computer--- consideration we must have a representation of the maze which is to be studied, and this is conveniently given by a list of the primitive links, the pair of points (X Y) belonging to the link list L if a path joins X to Y. We may suppose this is a directed path, and insist that (Y X) also appear in the list if the path is bidirectional.

A pair of points is then connected if there exist links (A X1) (X1 X2) ... ... (Xi Xi+1) ... (Xn B) all belonging to the list L, A and B being the initial and final point respectively. Such a link may exist in one direction but not the other.

An effective way to avoid loops in enumerating the paths is to remove the initial and final points from consideration in the inductive step, since any path arriving eventually at the initial point must form a loop, as well as a path to the final point initiating from the final point.

In the program (PATHS A B M), A is the initial point, B is the final point, and M is the link list. A and B are variables within the program enjoying a similar significance. (LLL) and (RRR) are fragments retaining the left and right halves of a list which we analyzed. The bucket variables are respectively A* which collects the elements of the image set, B* which colects the elements of the counterimage set, and X which collects links not originating or terminating from A or B. The fragment pattern (UU)

```
((*OR* ((A A*) UU) ((== A) UU) ((B* B) UU) ((B ==) UU) (X UU) ()))
```

is used to see the link list decomposed by the bucket variables; X will become the new link list in the recursive subproblem.

The key rule is

```
((A B (UU)) (=ITER= I A* J B* (K) (=REPT= (I J X)) (A K B)))
```

which sees the initial and final point together with the link list decomposed into the image set, counterimage set, and link list from which the initial and final points have been eliminated. For each pair of elements from A* and B*, the process is repeated, and to the path list -- a list of points forming the joining path --- is appended the otuer points A and B.

This rule defines the repetitive condition of the CONVERT program; the terminal conditions arise when we encounter a pair of points which turn out to be in fact the same, or a pair which are directly linked. In the latter case we search for any additional indirect links.

Should we eventually exhaust the link list without the ends joining, we produce a null fragment which, by producing a vacuous index set for =ITER=, causes the tentatively formed chain to be discarded.

```
print paths lisp
W 245.4

 PATHS   LISP      05/03  0245.5


     DEFINE ((

(PATHS (LAMBDA (A B M) (CONVERT
(QUOTE (
(UU) PAT ((*OR* ((A A*)UU) ((== A)UU) ((B* B)UU) ((B ==)UU) (X UU) ()))
 A*  BUV ==
 B*  BUV ==
 X   BUV ==
))
(QUOTE ( A B (LLL) (RRR) ))
(LIST A B M)
(QUOTE ( *0 (
( (A A ==) ((A)) )
( (A B (LLL (A B) RRR)) ((A B) (*REPT* (A B (LLL RRR)))) )
( (A B (UU)) (=ITER= I A* J B* (K) (*REPT= (I J X)) (A K B)) )
( == () )
)))
)))
))

CSET (TRIAL ((0 1) (0 2) (0 3) (0 4) (1 0) (2 0)
             (3 0) (4 0) (1 2) (1 4) (2 3) (4 3)))

CSET (NET ((A E) (A F) (E A) (E K) (E C) (E F) (K E)
           (K P) (K F) (C E) (C M) (C D) (D C) (D P)
(D F) (P K) (P D) (P L) (L P) (L B) (L M) (M L) (M Y)
(M C) (Y M) (Y B) (F A) (F E) (F K) (F D) (B L) (B Y)))

CSET (MESH  ((A C) (C B) (B C) (D E) (D H) (D F)
             (E F) (E G) (E H) (F H) (F G) (F D)
      (G D) (G H) (G E) (H D) (H E) (H G) (H F)
       I () ))
R 1.633+.616
```

```
r convrt
W 238.1
load ((paths))
 NIL

clock (())   e  (paths 1 3 trial)          clock (t)
 0
 ((1 0 3) (1 0 2 3) (1 0 4 3) (1 2 0 3) (1 2 3) (1 2 0 4 3)
 (1 4 0 3) (1 4 0 2 3) (1 4 3))
 32

clock (())  e (paths 3 1 trial)      clock (t)
 0
 ((3 0 1))
 11

clock (())   e (paths 2 4 trial)      clock (t)
 0
 ((2 0 4) (2 0 1 4) (2 3 0 4) (2 3 0 1 4))
 17


clock (())   e (paths (quote a) (quote b) mesh)       clock (t)
 0
 ((A C B))
 17

clock (())  e (paths (quote b) (quote a) mesh)      clock (t)
 0
 NIL
 23

clock (())  e (paths (quote c) (quote h) mesh)   clock (t)
 0
 NIL
 41

clock (())   e (paths (quote I) (quote g)  mesh)   clock (t)
 0
 NIL
 16

clock (())   e (paths (quote d) (quote f)  mesh)   clock (t)
 0
 ((D F) (D E F) (D E H F) (D E G H F) (D H E F) (D H G E F)
 (D H F))
 34

clock (())   e (paths (quote f) (quote d) mesh)     clock (t)
 0
 ((F D) (F H G D) (F H E G D) (F H D) (F G D) (F G H D) (F G
 E H D))
 34

clock (())  e (paths (quote h) (quoe"te f) mesh)   clock (t)
 0
 ((H F) (H D F) (H D E F) (H E G D F) (H E F) (H G D F) (H G
 E F) (H G D E F))
 44
```

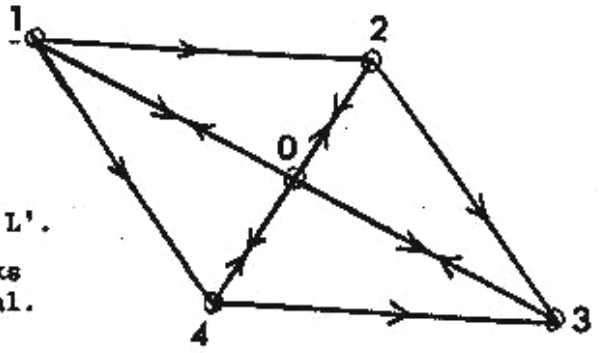

Fig. 'T R I A L'.

Not all the links are bidirectional.

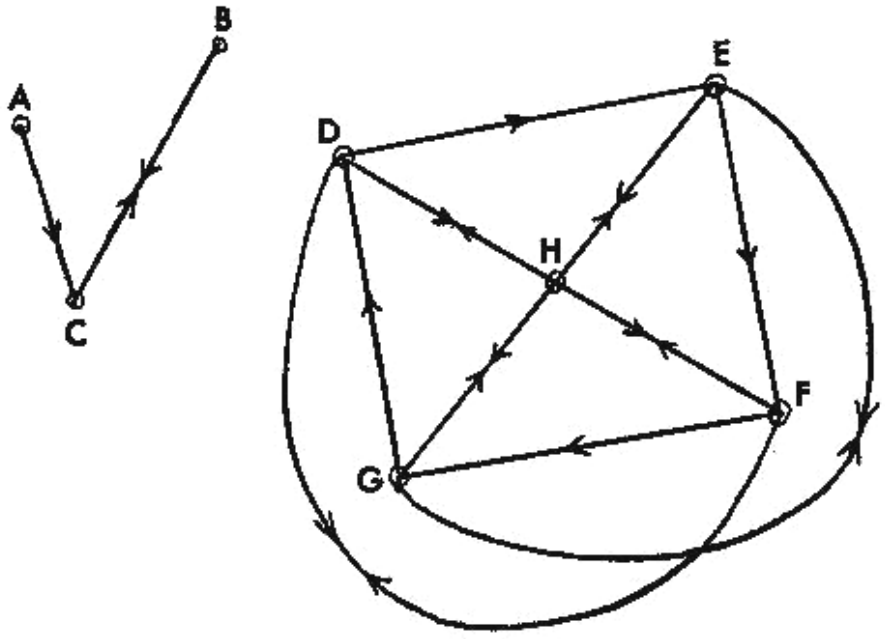

Fig. 'M E S H' .

A P P E N D I X I I

Listing of the CONVERT processor.