MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

Artificial Intelligence
Memo. No. 150.

January 1968.

CGRU AND CONG

CONVERT AND LISP PROGRAMS TO FIND THE

CONGRUENCE RELATIONS OF A FINITE STATE MACHINE

Harold V. McIntosh*

*ESCUELA SUPERIOR DE FISICA Y MATEMATICAS
INSTITUTO POLITECNICO NACIONAL
MEXICO 14 D.F., MEXICO.

ABSTRACT

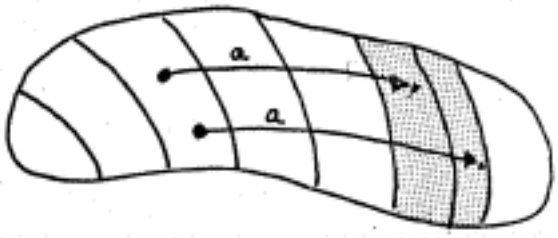
CGRU is a CONVERT program, CONG its literal transcription into LISP, realized in the CTSS LISP of Project MAC, for finding all the congruence relations of a finite state machine whose transition table is given as an argument. Central to both programs is the hull construction, which forms the smallest congruence relation containing a given equivalence relation. This is done by examining all pairs of equivalent elements to see if their images are equivalent. Otherwise the image classes are joined and the calculation repeated. With the hull program, one starts with the identity relation and procedes by joining pairs of congruence classes in previously found partitions, and forming the hull in order to see if he may produce a new partition. The process terminates when all such extensions have been tried without producing any new relations.

A congruence relation in a finite state machine is an equivalence relation which satisfies the substitution property, which requires that the images of equivalent states be equivalent. If M(s,w) is the transition function, this requirement is stated formally by requiring that, for all w, whenever states s and t are equivalent, so also are M(s,w) and M(t,w). Congruence relations play an important role in the theory of finite state machines; for example the minimal machine accepting a certain class of words may be found from any other such machine by forming the factor machine with respect to the largest congruence relation smaller than the equivalence relation of equal outputs. If a machine has two complementary congruence relations, it may be decomposed into the direct product of the two factor machines. If there is one non-trivial congruence relation, the machine may be represented as a serial combination of the factor machine and a fibre machine.

In order to have congruence relations available for such considerations, we discuss two constructions. The first is the determination of the smallest congruence relation smaller than a given congruence relation, which we may call its hull. The second is to systematically enumerate all the congruence relations for a given machine. This latter is done by first listing all the minimal equivalence relations. These are the equivalence relations in which there is only one point in each equivalence class, except for a certain pair of equivalent points. There is one minimal relation for each pair of distinct points. One then forms the hull of each of these $n(n+1)/2$ equivalence relations. Among the hulls will be included the minimal congruence relations, but there may well be included some larger congruence relations as well. One then continues by trying to join the equivalence classes found on the first round in all possible pairs, and computing their hulls in turn. The process is then repeated until no new congruence relations are found, and an attempt has been made to extend those already found in all possible ways.

Two versions of the program are described. CGRU is a CONVERT program, while CONG is an equivalent LISP program. The existence of the two programs has permitted a comparison of the speed of execution of CONVERT, LISP, and compiled LISP programs.

The hull construction is quite straightforward. One examines one by one all pairs of equivalent elements and tests their images according to the various possible input letters. If a pair of nonequivalent images is found, their equivalence classes are joined, and the procedure repeated. It terminates when all pairs of equivalent elements and all letters have been tested and no non-equivalent image pairs have been found.



If a pair of equivalent points have inequivalent images, join the image classes.

CONVERT can be caused to carry out the searching which we have described automatically, if we present the partition of the state set and the transition table properly. A partition is readily described by a list of its equivalence classes which in turn are lists of mutually equivalent elements; thus if the integers 1 to 10 are partitioned into their residue classes modulo 3 we would write ((1 4 7 10) (2 5 8) (3 6 9)) to represent the partition. The transition table takes the form

```
(=== (X === (I Y) ===) ===)
```

to indicate that M(X,I) = Y. In other words, for each state we form a list headed by that state and followed by pairs of letters and the corresponding image.

We make the following definitions.

=T= is the transition table, in the above form, a VAR.

```
 T       PAT     (=AND= L M)
 L       PAT     (=== (X === (I U) ===) ===)
 M       PAT     (=== (Y === (I V) ===) ===)
(UU)     PAV     (=== U ===)
(VV)     PAV     (=== V ===)
=I=      PAT     (=== (=AND= (=== X ===) (=== Y ===)) ===)
=O=      PAT     (XXX (UU) YYY (VV) ZZZ)
```

The pattern =I= is used to see whether there is an equivalence class containing the two equivalent elements X and Y. One might avoid the form =AND= by using unordered variables, but the present form is more transparent. Thus =I= matches any partition containing two equivalent elements (not necessarily distinct), and hence will match any partition. The pattern T is designed to match a transition table but particularly such a transition table that U appears as the successor of X by the letter I and simultaneously V appears as the successor of Y by the same letter I. Since U and V are not previously bound, T will match any well-formed transition table, but in the process will bind I, U, and V. Next, we see that UU and VV are fragments, namely those partition cells which contain U and V respectively. Consequently, when we expect =O= to match the partition of the state set, we are expecting to find that U and V are in <u>different</u> cells of the partition.

If we then present CONVERT with the list (X T X) in which X is the partition of the state set and T is the transition table, and expect it to match (X T X) to (=I= =T= =O=), we are expecting it to find two equivalent elements X and Y and an input letter I such that M(X,I) = U and M(Y,I) = V, and U and V are not equivalent. The fail procedure of the pattern matching apparatus will exhaust all possibilities before finally giving a negative response.

The hull construction is then effected by the function HU,

```
HU     REPT    (((X) (=REPT=  (X =T= X) *1 (
                  ((=I= T =O=) (=REPT= ((XXX(UU VV) YYY ZZZ)
                                       =T=
                                       (XXX (UU VV) YYY ZZZ))))
                    (I == ==) I)
                ))))
```

Writing (HU X), we see that HU has one argument, which is listed when we evaluate the REPT skeleton. We thus extract the argument from this list in the first line, and make the comparison which we have described. If there are elements with inequivalent images, their classes are joined in the form (UU VV), and the entire process is repeated. Otherwise we accept the final partition as the hull.

The hull function can be used to find all the congruence relations for a given machine. The construction depends on the fact that we can readily construct all the equivalence relations for a given set since it is only necessary to enumerate all the disjoint subsets. However, it is

only necessary to initiate the process by an enumeration of the minimal equivalence relations. Those are the ones for which only a single pair of points are equivalent, all other equivalence classes containing only single points. If we now find the hulls of all these partitions, we are certain to find the minimal congruence relations, but we may well find larger ones also. Actually each hull has the significance that it is the minimum congruence relation for which the given pair of points are equivalent, but the equivalence of one pair of points may well force the equivalence of another pair and not conversely, so that the hull of the forced pair could be smaller than the hull of the forcing pair.

Given a minimal congruence relation, there are at least two equivalent points, as the term "minimal" excludes the identity relation. The hull of those points must be the relation with which we commenced. Thus a minimal congruence relation is the common hull of all pairs of points which it identifies. If some other pairs of equivalent points generate a non-minimal congruence relation, that is another matter. Some pairs of points are more congruent than others, depending on whether the least congruence relation which considers them equivalent is absolutely minimal or not.

Our initial round then produces at least the minimal congruence relations, and perhaps some others. We know that any larger congruence relation will have to join at least one pair of classes of a minimal relation. Simply joining two congruence classes need not produce a congruence class, so the hull construction is to be applied once more. We therefore make all extensions possible in this manner, and are thereby ensured of finding all congruence relations for which the possible triples of points are equivalent. It is more efficient to extend the first round of congruence relations than to find the hulls of all possible triples, in general.

The algorithm then procedes as follows. First we note the identity relation, which is always a congruence relation. Then we form the hulls which consider the possible pairs of points equivalent. As each new hull is found, it is compared to see if the same as a previously found hull, and if not is placed on a waiting list, as well as on a list of relations to be extended. In this way we have our initial selection of congruence relations.

From this point on, we will take the partitions one by one from the waiting list and adjoin them to the finished list. In addition, we shall join its equivalence classes in all possible pairs and form the hull of the resulting equivalence relation. Each new hull so formed is compared against the waiting list and the finished list to see if it is new. When new, it is adjoined to the waiting list. Eventually no new partitions will be found and the process will terminate when the last waiting partition has been extended.

The comparison of two partitions is not automatic, neither in LISP nor in CONVERT, no more so than the comparison of sets, because sets may be equal without being represented by equal lists. One must either write a special comparison function for sets which will pass through one of them to see if all its elements are members of the other and conversely, or as we shall do in the present case, always reduce the list describing a set to a standard form.

To create the standard form, we maintain a list of the state set in some arbitrary order. In our present program this state set is (*S*). We shall require that the elements of a subset of S, such as an equivalence class, appear in this same order. The ordering is realized by a double complementation, and is effected by the function (N1 X).

```
N1    REPT    (((X) (=COMP= (*S*) (=COMP= (*S*) X))))
```

Next, in any set of subsets which are so normalized, we require that the subsets be listed so that their first element preserves the order of (*S*). The function N2 achieves this ordering.

```
N2    REPT    (((X) (=ITER= I (*S*) (*SKEL* I VAR I
                    (=WHEN= X (=== (I XXX) ===) ((I XXX)) ()))))))
```

Here we pass through (*S*) and if there is any subset which begins with the selected letter it is extracted and placed in order.

Finally there is a function NO which will take an arbitrary list of partitions, normalize them and eliminate duplications.

```
NO    REPT    ((X (=UNON= (=ITER= I X (N2 (=ITER= J I (N1 J)))))))
```

We have now seen the essential structure of the program CGRU. It is organized as a program, whose program variables have the following usage.

(*A*) is used to store the letters of the input alphabet

(*G*) is the finished list

(*S*) is the list of internal states

(*P*) is the waiting list, and a temporary workspace.

in addition,

=T= is the transition table.

As a convenience, but also as a check against errors, the program initially computes the alphabet and the state set. These could be input as data, but are also implicitly present in the transition table. Bucket variables are used in the collection;

AA for the alphabet

SS for the state set

while the collecting patterns are

```
(=S=)   PAT     ((*OR* (=== (SS =R=) =S=) (===)))
(=R=)   PAT     ((*OR* ((AA SS) =R=) ()))
```

First, then, the alphabet and state set are found, and the results are printed, as well as being stored in the program variables (*A*) and (*S*).

The phrase (CONGRUENCE RELATIONS) is written.

The identity relation is printed and stored in (*G*).

(*P*) is then set to a list of the equivalence relations of single pairs, done in two stages; pairs are formed, and the remaining unit classes adjoined.

1 is the label for the extension loop

The hulls of each of the partitions in (*P*) are formed.

These are normalized and duplicates on the finished list are rejected.

== is written to separate partitions found in different cycles.

If no new partitions were formed, we finish, going to head 2.

The newly found partitions are printed,

and adjoined to the finished list.

Otherwise the pair-joining extension is made for each of the newly found partitions,

and the cycle is repeated.

2 is the label for the exit,

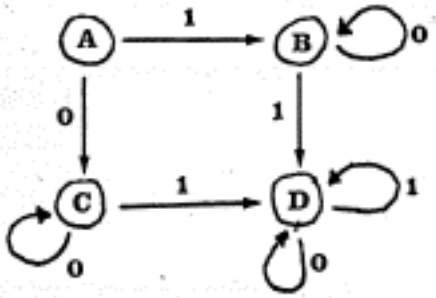
which consists simply in writing (===== GOODBYE =====).

It should be noted that the program speaks of extending the waiting list in one step while our description of the algorithm spoke of one partition at a time. This is because the individual steps are all encompassed in one =ITER=, and hence the order of some of the steps is slightly different.

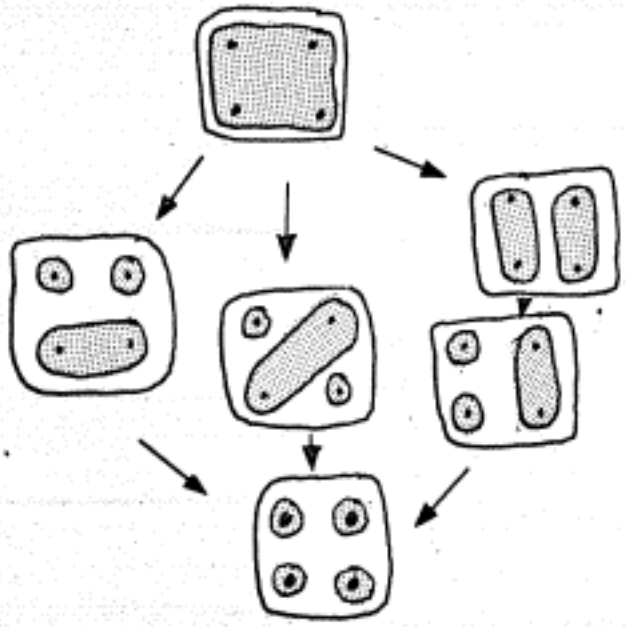
To illustrate the program we now consider an illustrative example.

```
cgru (((a (1 b) (0 c)) (b (0 b) (1 d)) (c (0 c) (1 d)) (d (0 d) (1 d))))
(ALPHABET)
(0 1)
(STATE SET)
(A B C D)
(CONGRUENCE RELATIONS)
((A) (B) (C) (D))
==
((A C) (B D))
((A B C D))
((A) (B C) (D))
((A) (B D) (C))
((A) (B) (C D))
(***** GOODBYE *****)
```

The machine is:



and its congruence relations are:



The algorithm, although simple, calls for a considerable amount of calculation. Moreover, the amount rises rather sharply with the size of the machine since there are many processes which depend on the square of the machine order --- the search to see if equivalent elements have equivalent images, the starting step which considers individual pairs of equivalent elements, and so on. It already required about one minute or actual running time in CTSS LISP to analyze a four state machine, which made this about the limit of practical size, but hardly larger than could be readily analyzed by hand. Accordingly, the program was rewritten almost verbatim in LISP and compiled. The principal function, (CONG T), ran about twice as fast as the CONVERT program CGRU when interpreted, and 38 times faster when compiled. Depending one's perspective, he would think that CONVERT was not inordinately slow, unless he considered the CTSS LISP interpreter inordinately slow also. However the LISP program was about twice as long, in the volume of letters which its statement consumed, and probably rather less transparent than CGRU, depending upon one's experience in reading LISP versus CONVERT.

Since the two programs are identical in specification we shall not describe CONG separately nor show an example of its operation. However, in order to make the listing more understandable, we briefly describe the auxiliary functions upon which it calls.

(CONG T) is the main function, which computes all the congruence relations for the machine specified by the transition table T. This table has the form (=== (X === (I Y) ===) ... ===) where X is a state, I an input letter, and Y the image state M(X,I).

It is a program whose program variables have the following significance

- A is the input alphabet
- G is the finished list
- S is the state set
- P is the workspace holding the extensions of one relation
- W is the waiting list

The steps of the program are nearly identical to the corresponding steps of CGRU.

(ELEMENT X L) is a predicate, true if the expression X is found on the list L.

(COMPLEMENT A B) is a list of all those members of A which do not belong to B. A and B are both lists. Retained elements occur in the original order with the same multiplicity.

(PAIRSET A) has as its argument a partition of the state set. Its value is a list of the various partitions which arise by joining together different pairs of equivalence classes in A. For example, if A = ((A) (B) (C)), (PAIRSET A) = (((A B) (C)) ((A C) (B)) ((B C) (A))). The joined class appears first, followed by these remaining. A partition of no or one class is treated correctly in the initialization. The significance of the program variables is

I ranges through A
J ranges through (CDR I)
P retains the joined partitions already found.

(UNITIZE L) produces a list whose elements are the elements of L, listed. Thus if L = (0 1 2), (UNITIZE L) = ((0) (1) (2)).

(NONREDUNDANT L) produces a list whose elements are the elements of L in the same order, with the exception that if there is a multiple occurrence of an element, all but its last instance is deleted. Thus, if L = (1 0 1 1 0), (NONREDUNDANT L) = (1 0).

(REMOVE I J L) produces a list whose elements are the elements of L, except that both I and J are deleted; otherwise the order of the elements is not changed. If L= (0 1 2 2 1 0), I = 0, J = 1, then (REMOVE I J L) = (2 2).

(JOIN I J L) assumes that its three arguments are lists. In the present context, I and J are equivalence classes, while L is a partition. I and J are appended, and removed from L. The value is a list of first the joined I and J, then the list L from which they have been removed.

(IMAGE S L T) assumes as arguments a state S, a letter L and a transition table T. Its value is the image of S by L according to the table T. The function is a program which searches T until it finds a sublist beginning with S. It then searches the CDR of this sublist to find a sublist beginning with L, whose CADR is the value.

(CONTAINING X L) finds the sublist of L which contains X, if any, otherwise the empty list.

(HULL P A T) is the analogue of the CONVERT function HU, which finds the hull of the partition P according to the machine which has alphabet A and transition table T. These latter might well have been left as free variables.

The program variables signify:

AA which ranges through the alphabet A as we test images by various letters
I which ranges through a class of the partition
II which is the class containing the image of I
J which ranges through CDR of the class of I, forming the second element of a pair
JJ which is the class containing the image of J
K which ranges through the classes of P, I and J both belong to K

The operation of HULL follows exactly our earlier description.

(BEGINNING X L) finds the sublist of L beginning with the expression X, otherwise the empty list.

(NOR1 L S) normalizes the sublists of L with respect to the set (state set) S. In other words, the elements in each sublist are made to appear in the same order in which they appear in S.

(NOR2 L S) normalizes the list L according to the list S in the sense that the sublists of L are made to appear with their first elements in the same order in which they appear in S.

(JOINNONNULL A B) CONS's A to B unless A is null, in which case B remains unchanged.

(NOR3 P S) causes each element of the list P to be normalized according to the functions NOR2 and NOR1.

(HULLS P A T) causes HULL of each element of P to be computed.

(ADJOIN X L) places X on the list L if it is not already a member.

(GATHER T), where T is the transition table, gathers all the letters of the alphabet and all the state symbols, making a list of these two lists. Its action is exactly analogous to the bucket variables which perform a similar function in CGRU.

Usage of the program variables is

A to collect the letters of the alphabet
S to collect the state set
U to retain (CDR T) while we search (CAR T).

After compilation, CONG required 20 seconds for a 6-state machine while CGRU required 70 seconds for a series of 4-state machines which indicates that the practical limit may occur for 8 or 10 state machines, although no effort was made to make accurate timing evaluations. The method we have presented is completely straightforward, but to handle machines of moderate size, further consideration is necessary to see how calculations may be simplified or avoided. For example, each time we see that two classes need joining in calculating the hull, we start completely anew to test the extended relation, yet one can see that he will then uselessly have to repeat a great deal of calculation. Perhaps it would be worthwhile to test the resulting relation against the known congruence relations each time a juncture was made, since such a comparison would be much faster than testing all the pairs.

```
(CONG (LAMBDA (TT) (PROG T A G S P W)
      H0
      (CLOCK T)
      (SETQ A (GATHER TT))
      (SETQ S (CADR A))
      (SETQ A (CAR A))
      (PRINT (QUOTE (=== ALPHABET ===)))
      (PRINT A)
      (PRINT (QUOTE (=== STATE SET ===)))
      (PRINT S)
      (PRINT (QUOTE (CONGRUENCE RELATIONS)))
      (PRINT (QUOTE ==))
      (SETQ W (LIST (UNITIZE S)))
      H1
      (COND   ((NULL W) (GO H2)))
      (SETQ G (CONS (PRINT (CAR W)) G))
      (SETQ P (NONREDUNDANT (NOR3 (HULLS (PAIRSET (CAR W)) A TT) S)))
      (SETQ P (COMPLEMENT (COMPLEMENT P W) G))
      (SETQ W (CDR W))
      (COND   ((NULL P) (GO H1)))
      (SETQ W  (APPEND P W))
      (GO H1)
      H2
      (PRINT (CLOCK T))
      )))
```

```
(ELEMENT (LAMBDA (X L) (AND (NOT (NULL L)) (OR (EQUAL X (CAR L)) (ELEMENT X (CDR L))))))

(COMPLEMENT (LAMBDA (A B) (COND
     ((NULL A) A)
     ((ELEMENT (CAR A) B) (COMPLEMENT (CDR A) B))
     (T (CONS (CAR A) (COMPLEMENT (CDR A) B)))
     )))

(PAIRSET (LAMBDA (A) (PROG ( I J P )
     (COND   ((NULL A) (RETURN A)))
     (SETQ I A)
     (COND   ((NULL (CDR A)) (RETURN (LIST))))
     (SETQ J (CDR A))
     H1
     (COND   ((NULL (CDR I)) (RETURN P))
             ((NULL J) (GO H2)))
     (SETQ P (CONS (JOIN (CAR I) (CAR J) A) P))
     (SETQ J (CDR J))
     (GO H1)
     H2
     (SETQ I (CDR I))
     (SETQ J (CDR I))
     (GO H1)
     )))

(UNITIZE (LAMBDA (L) (COND ((NULL L) L) (T (CONS (LIST (CAR L)) (UNITIZE (CDR L)))))))

(NONREDUNDANT (LAMBDA (L) (COND
     ((NULL L) L)
```

```
      ((ELEMENT (CAR L) (CDR L)) (NONREDUNDANT (CDR L)))
      (T (CONS (CAR L) (NONREDUNDANT (CDR L))))
      )))

(REMOVE (LAMBDA (I J L) (COND
      ((NULL L) L)
      ((EQUAL I (CAR L)) (REMOVE I J (CDR L)))
      ((EQUAL J (CAR L)) (REMOVE I J (CDR L)))
      (T (CONS (CAR L) (REMOVE I J (CDR L))))
      )))

(JOIN (LAMBDA (I J L) (CONS (APPEND I J) (REMOVE I J L))))

(IMAGE (LAMBDA (S L TT) (PROG ()
      H1
      (COND   ((NULL TT) (RETURN (LIST))))
      (COND   ((EQ S (CAAR TT)) (GO H2)))
      (SETQ TT (CDR TT))
      (GO H1)
      H2
      (SETQ TT (CDAR TT))
      H3
      (COND   ((NULL TT)  (RETURN (LIST))))
      (COND   ((EQ L (CAAR TT)) (RETURN (CADAR TT))))
      (SETQ TT (CDR TT))
      (GO H3)
      )))

(CONTAINING (LAMBDA (X L) (COND
      ((NULL L) (LIST))
      ((ELEMENT X (CAR L)) (CAR L))
      (T (CONTAINING X (CDR L)))
      )))

(HULL (LAMBDA (P A TT) (PROG  ( AA I II J JJ K )
      (COND   ((NULL P) (RETURN P)))
      H0
      (COND ((NULL (CDR P)) (RETURN P)))
      (SETQ K P)
      H1
      (COND   ((NULL K) (RETURN P)))
      H2
      (SETQ I (CAR K))
      H3
      (COND   ((NOT (NULL (CDR I))) (GO H4)))
      (SETQ K (CDR K))
      (GO H1)
      H4
      (SETQ J (CDR I))
      H5
```

```
    (SETQ JJ (CONTAINING (IMAGE (CAR J) (CAR AA) TT) P))
    (COND   ((EQUAL II JJ) (GO H9)))
    (SETQ P (JOIN II JJ P))
    (GO H0)
    H9
    (SETQ AA (CDR AA))
    (GO H7)
    )))

(BEGINNING (LAMBDA (X L) (COND
    ((NULL L) L)
    ((EQUAL X (CAAR L)) (CAR L))
    (T (BEGINNING X (CDR L)))
    )))

(NOR1 (LAMBDA (L S) (COND
    ((NULL L) L)
    (T (CONS (COMPLEMENT S (COMPLEMENT S (CAR L))) (NOR1 (CDR L) S)))
    )))

(NOR2 (LAMBDA (L S) (COND
    ((NULL S) S)
    (T (JOINNONNULL (BEGINNING (CAR S) L) (NOR2 L (CDR S))))
    )))

(JOINNONNULL (LAMBDA (A B) (COND ((NULL A) B) (T (CONS A B)))))

(NOR3 (LAMBDA (P S) (COND
    ((NULL P) P)
    (T (CONS (NOR2 (NOR1 (CAR P) S) S) (NOR3 (CDR P) S)))
    )))
```

```
      (SETQ JJ (CONTAINING (IMAGE (CAR J) (C..AA) TT) P))
      (COND   ((EQUAL II JJ) (GO H9)))
      (SETQ P (JOIN II JJ P))
      (GO H0)
      H9
      (SETQ AA (CDR AA))
      (GO H7)
      )))

(BEGINNING (LAMBDA (X L) (COND
      ((NULL L) L)
      ((EQUAL X (CAAR L)) (CAR L))
      (T (BEGINNING X (CDR L)))
      )))

(NOR1 (LAMBDA (L S) (COND
      ((NULL L) L)
      (T (CONS (COMPLEMENT S (COMPLEMENT S (CAR L))) (NOR1 (CDR L) S)))
      )))

(NOR2 (LAMBDA (L S) (COND
      ((NULL S) S)
      (T (JOINNONNULL (BEGINNING (CAR S) L) (NOR2 L (CDR S))))
      )))
```

```
)                                                                    )
(HULL (LAMBDA (P A TT) (PROG  ( AA I II J JJ K )
      (COND   ((NULL P) (RETURN P)))
      H0
      (COND ((NULL (CDR P)) (RETURN P)))
      (SETQ K P)
      H1
      (COND   ((NULL K) (RETURN P)))
      H2
      (SETQ I (CAR K))
      H3
      (COND   ((NOT (NULL (CDR I))) (GO H4)))
      (SETQ K (CDR K))
      (GO H1)
      H4
      (SETQ J (CDR I))
      H5
      (COND   ((NOT (NULL J)) (GO H6)))
      (SETQ I (CDR I))
      (GO H3)
      H6
      (SETQ AA A)
      H7
      (COND   ((NOT (NULL AA)) (GO H8)))
      (SETQ J (CDR J))
      (GO H5)
      H8
      (SETQ II (CONTAINING (IMAGE (CAR I) (CAR AA) TT) P))
```

```
(HULLS (LAMBDA (P A TT) (COND
     ((NULL P) P)
     (T (CONS (HULL (CAR P) A TT) (HULLS (CDR P) A TT)))
     )))

(ADJOIN (LAMBDA (X L) (COND ((ELEMENT X L) L) (T (CONS X L)))))

(GATHER (LAMBDA (TT) (PROG ( A S U )
     H1
     (COND   ((NULL TT) (RETURN (LIST A S))))
     (SETQ S (ADJOIN (CAAR TT) S))
     (SETQ U (CDAR TT))
     H2
     (COND   ((NULL U) (GO H3)))
     (SETQ A (ADJOIN (CAAR U) A))
     (SETQ S (ADJOIN (CADAR U) S))
     (SETQ U (CDR U))
     (GO H2)
     H3
     (SETQ TT (CDR TT))
     (GO H1)
     )))

     ))
```

```
     DEFINE ((
(CGRU (LAMBDA (TT) (CONVERT
(CONS (QUOTE =T=) (CONS (QUOTE VAR) (CONS TT (QUOTE (
     AA        BUV       =ATO=
     SS        BUV       =ATO=
    (=S=)      PAT       ((*OR* (=== (SS =R=) =S=) (===)))
    (=R=)      PAT       ((*OR* ((AA SS) =R=) ()))
     =I=       PAT       (=== (=AND= (=== X ===) (=== Y ===)) ===)
     =O=       PAT       (XXX (UU) YYY (VV) ZZZ)
     T         PAT       (=AND= L M)
     L         PAT       (=== (X === (I U) ===) ===)
     M         PAT       (=== (Y === (I V) ===) ===)
    (UU)       PAV       (=== U ===)
    (VV)       PAV       (=== V ===)
     HU        REPT      (((X) (=REPT= (X =T= X) *1 (
                          ((=I= T =O=) (=REPT= ((XXX (UU VV) YYY ZZZ ) =T= (XXX (UU VV) YYY ZZZ ))))
                          (((I == ==) I)
                         ))))
     N1        REPT      (((X) (=COMP= (*S*) (=COMP= (*S*) X))))
     N2        REPT      (((X) (=ITER= I (*S*) (*SKEL* I VAR I (=WHEN= X (=== (I XXX) ===) ((I XXX)) ())))))
     NO        REPT      ((X (=UNON= (=ITER= I X (N2 (=ITER= J I (N1 J)))))))
     )))))
(QUOTE (
     I N S U V X Y (XXX) (YYY) (ZZZ)
     ))
(LIST)
(QUOTE (*O (
(== (=PROG= ( (*A*) (*G*) (*S*) (*P*) )
     (=WHEN= =T= (=S=) (=PROG= ()
          (=PRNT= (=QUOT= (ALPHABET)))
          (=SETQ= (*A*) (=PRNT= (=UNON= AA)))
          (=PRNT= (=QUOT= (STATE SET)))
          (=SETQ= (*S*) (=PRNT= (=UNON= SS)))
           ))
     (=PRNT= (=QUOT= (CONGRUENCE RELATIONS)))
     (=SETQ= (*G*) ((=PRNT= (=ITER= I (*S*) (I)))))
     (=SETQ= (*P*) (=ITER= I (*S*) J (=SKEL= I VAR I (=WHEN= (*S*) (=== I XXX) (XXX))) (I J)))
     (=SETQ= (*P*) (=ITER= I (*P*) (I (*ITER* J (=COMP= (*S*) I) (J)))))
     1
     (=SETQ= (*P*) (=ITER= I (*P*) (HU I)))
     (=SETQ= (*P*) (=COMP= (NO *P*) (*G*)))
     (=PRNT= ==)
     (=WHEN= (*P*) () (=GOTO= 2))
     (=ITER= I (*P*) (=PRNT= I))
     (=SETQ= (*G*) (*P* *G*))
     (=SETQ= (*P*) (=COMP= (NO (*ITER* I (*P*) (J) I (K) (=SKEL= (J) VAR (J) (=WHEN= I (=== (J) XXX) (XXX)))
               ((J K) (*COMP* I ((J) (K))))))) (*G*)))
     (=GOTO= L)
     2
     (=RETN= (===== GOODBYE =====))
     ))
     )))
     )))
     ))
```

REFERENCES

Congruence Relations:

Michael A. Harrison, INTRODUCTION TO SWITCHING AND AUTOMATA THEORY, New York: McGraw-Hill Book Company, 1965

Harold V. McIntosh, "THE MATHEMATICAL THEORY OF MACHINES," Lecture Notes for Mathematical Logic I, ESFM, 1967.

CONVERT:

Adolfo Guzman and Harold V. McIntosh, "CONVERT," Communications of the Association for Computing Machinery 9 604-615 (1966).

Harold V. McIntosh and Adolfo Guzman, "A MISCELLANEY OF CONVERT PROGRAMMING," Project MAC Artificial Intelligence Group Memo 130 (April 1967).

LISP:

John McCarthy et al, LISP 1.5 PROGRAMMER'S MANUAL, Cambridge: The M.I.T. Press, 2d Ed. 1965.

Edmund C. Berkeley and Daniel G. Bobrow, Editors, THE PROGRAMMING LANGUAGE LISP: ITS OPERATION AND APPLICATIONS, Cambridge: The M.I.T. Press 1964.