MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

Artificial Intelligence
Memo. No. 153.

January 1968.

REEX
A CONVERT PROGRAM TO
REALIZE THE McNAUGHTON-YAMADA ANALYSIS ALGORITHM

Harold V. McIntosh*

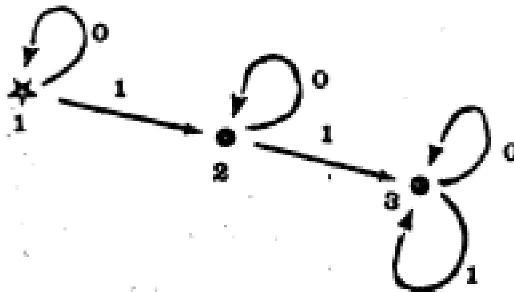
* ESCUELA SUPERIOR DE FISICA Y MATEMATICAS
INSTITUTO POLITECNICO NACIONAL
MEXICO 14 D.F., MEXICO.

ABSTRACT

REEX is a CONVERT program, realized in the CTSS LISP of Project MAC, for carrying out the McNaughton-Yamada analysis algorithm, whereby a regular expression is found describing the words accepted by a finite state machine whose transition table is given. Unmodified the algorithm will produce 4^n terms representing an n-state machine. This number could be reduced by eliminating duplicate calculations and rejecting on a high level expressions corresponding to no possible path in the state diagram. The remaining expressions present a serious simplification problem, since empty expressions and null words are generated liberally by the algorithm. REEX treats only the third of these problems, and at that makes simplifications mainly oriented toward removing null words, empty expressions, and expressions of the form XuX*, AuB*A, and others closely similar. REEX is primarily useful to understand the algorithm, but hardly useable for machines with six or more states.

Since regular expressions form such a convenient characterization of the words accepted by a finite state machine it is desirable to have a means of deducing the descriptive regular expression from the transition table of the machine. The first such algorithm was described by McNaughton and Yamada, and for some time was apparently the only such algorithm known. While it is conceptually quite simple, its application in practice can lead to grossly cumbersome expressions. Nevertheless its mechanization is instructive, and is the object of the CONVERT program REEX.

We will use the following example to illustrate our discussion.



The states of the machine to be analyzed are supposed to be numbered, 1 through n. We then define regular expressions α_{ij}^{k} recursively as follows.

$$\alpha_{ii}^{0} = \{\sigma\varepsilon\Sigma | M(i,\sigma) = i\} \cup \lambda$$

$$\alpha_{ij}^{0} = \{\sigma\varepsilon\Sigma | M(i,\sigma) = j\}$$

$$\alpha_{ij}^{k} = \alpha_{ij}^{k-1} \cup \alpha_{ik}^{k-1}\cdot(\alpha_{kk}^{k-1})^{*}\cdot\alpha_{kj}^{k-1}$$

By examining these definitions it can be seen that α_{ij}^{k} is a regular expression representing all the words corresponding to transitions from state i to state j without passing through states numbered greater than k. Transitions from state i to state j without any such restriction are then given by the expressions α_{ij}^{n}, and the regular expression representing the machine is the union of such expressions where i is the initial state and j belongs to the accepting set.

The transcription of this recursive definition into CONVERT presents no complications. In fact, if we let the index k be the state set, and the

indices i and j be actual states, we may avoid the necessity of numbering the states. The pattern we are to recognize is then the list (I J K), and it will be seen in the three forms

```
(I I ())
(I F ())
(I F (X XXX))
```

The skeletons which will be substituted in the three cases will be respectively a set of letters union the null letter, a set of letters, and the CONVERT form of the recursive formula. We need a means of extracting the letters causing a given transition, for which we introduce the patterns (U*) or (T*).

```
(U*)    PAT    ((*OR* (=== (I L I) U*) (===)))
(T*)    PAT    ((*OR* (=== (I L F) T*) (===)))
```

where

```
L       BUV    =ATO=
```

These definitions assume that the transition table TT is presented in the form (=== (I L F) ===) where I is the state from which the letter L causes a transition to F; ie M(I,L) = F. (U*) and (T*) are then collection patterns using the bucket variable L to find all the letters causing transitions respectively from I to I, or I to F.

If we use the symbol $, available in the character set of CTSS LISP, for the null word, we can now write the McNaughton-Yamada algorithm in CONVERT form. The rules are

```
((I I ())       (=WHEN= TT (U*) (UNO $ (*UNON* L))))
((I F ())       (=WHEN= TT (T*) (UNO (*UNON* L))))
((I F (X XXX))  (UNO (=REPT= (I F (XXX))
                     (CON (=REPT= (I X (XXX)))
                          (ITR (=REPT= (X X (XXX))))
                          (=REPT= (X F (XXX))))))
```

The symbols UNO, CON, and ITR stand respectively for union, concatination, and iteration, the "polish" form of the connectors which form regular expressions. In actual practice, the state set is not given and must be deduced from the transition table. This is done with a bucket variable and collecting skeleton.

```
S       BUV    =ATO=
(S*)    PAT    ((*OR* (=== (S == S) S*) (===)))
```

which is applied to the transition table. Hence our actual CONVERT program contains the rule

((I F (S*)) (UNO (*ITER* J F (=REPT= (I J (=UNON= S)) *1 (###)))))
wherein ### is the rule set displayed above. In this way we take account of all the states in the accepting set, and obtain the state set implicitly from the bucket variable S. It follows that (REEX I F T) has as arguments the initial state, the set of accepting states, and the transition table as a list of triplets of the form (I L F).

If we now set out to calculate some examples we begin to find that the algorithm is not really very satisfactory, mathematically correct that it may be. The basic problem is that the terminal condition very often leads to an empty set. Consequently if it occurs as a term in the concatination, the concatinated expression will likewise be an empty set. However, this is not obvious from the expression which the algorithm produces, and some simplification must be made. We run the risk that we may calculs e all three terms of the concatination before discovering that one of them renders the result trivial.

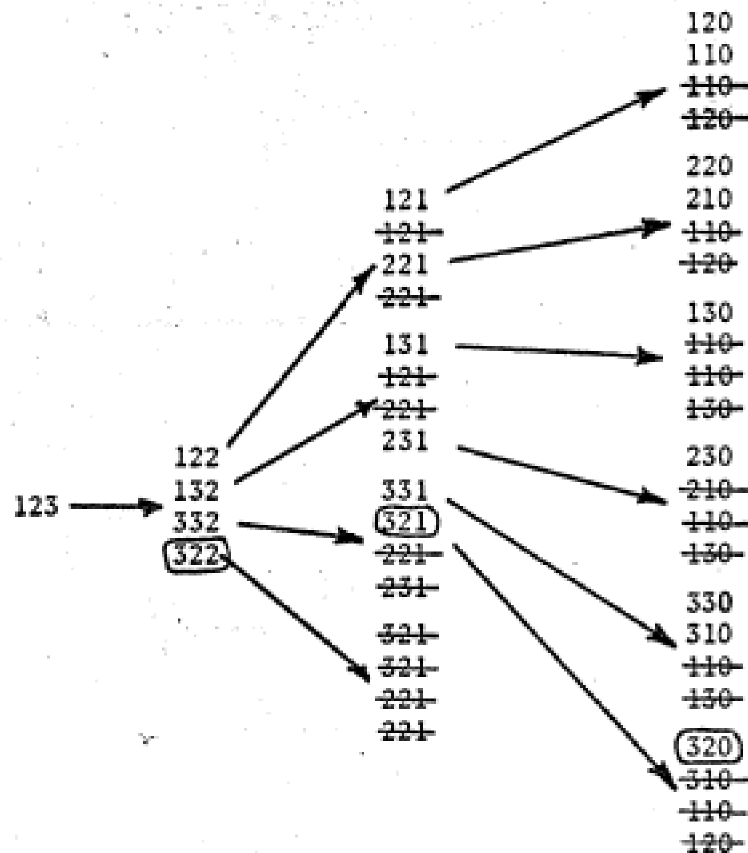
A second problem is that a very great deal of duplicate calculation can occur. In other words, the same subexpression may arise from a variety of histories, and be calculated anew each time. This is the more time consuming, the higher the level on which it occurs.

A third flaw lies in the fact that the method is prone to produce redundant expressions; that is such things as the union of X and $ concatinated with X* in the simple case, or 110* union 0*110*, to cite a slightly more complicated example. If all such redundancies were of a simple nature, they could be edited away, but unfortunately they become more and more subtle as the number of states of the machine increases. They arise from otherwise identical paths which do or do not include certain loops or branches.

The order in which the states are listed not only may lead to alternate regular expressions representing the machine, but sometimes can lead to vastly different amounts of calculation even when simplifying and precautionary techniques are included. McNaughton-Yamada reccommend giving high numbers to heavily trafficked states, to increase the number of null subexpressions which can be recognized on sight. In this regard it is certainly profitable to find which pairs of points have no path whatsoever connecting them, for if there is no such path there will be none passing through designeted intermediate states either, and one can write the empty expression at once without proceeding through the recursion.

To get an idea of how impressively expansive the algorithm is, we have to see that at each step in the recursion we generate four new regular expressions, bound together by various operators. Hence after n steps, when the recursion terminates, we have 4^n expressions; a truly exponential growth. Moreover this number is to be multiplied by the number of accepting states.

In the example we have cited, the regular expression corresponding to initial state 1 and accepting state 2 is clearly 0*10*. If we list only subscripts, the expression we need is (i j k) = (1 2 3). Expanding, we find



The indices which have been lined out are those which are repeated, so that their calculation an additional time is redundant. Moreover, the circled index 322 is ∅ on the grounds that no arrow of any sort runs from state 3 to state 2. Hence the concatination of the last three terms will also be ∅, and only the first expression need be pursued, an observation which would immediately reduce the calculation to 1/4.

Even if that simplification were not made, half the terms on the third level and over half those on the fourth level are redundant, reducing the calculation to 1/4. Both simplifications are reasonably typical of more complex expressions. For the moment let us use the simplification afforded by 322 = ∅, to write

123 = 122
  = 121 u 121·221*·221

now,

121 = 120 u 110·110*·120
  = 1 u (0 u $)·(0 u $)*1

and

221 = 220 u 210·(110)*·120
  = (0 u $) u ∅·(0 u $)*1

Here we see on the lowest level quite simple expressions written in a very cumbersome way. For example, 121 simplifies to 0*1, while 221 is (0 u $). Thus

123 = 122 = 0*1 u 0*1(0 u $)*(0 u $)
  = 0*10*

Thus the higher levels continue to contribute clumsiness, even though we finally obtain the obvious result.

Our present program makes no attempt to avoid duplicate calculations nor to exclude those which are destined to produce ∅ for lack of any possible paths. One would think it a small sacrifice to reserve an array of n^3 elements to retain this information, since otherwise the calculation will be far too time-consuming to treat machines with even half a dozen elements. Even so, such measures seem to be destined to lower the rate of growth only to 2^n rather than 4^4.

However, we have studied to a slight extent the third problem, of simplifying the expressions produced by the algorithm. It was clear from examining results that some very simple redundancies were accounting for a substantial fraction of the complexity in the final result. The technique is to make the regular expression operators UNO, CON, and ITR into functions responsible for simplifying their arguments.

Let us review them one by one.

```
CON   REPT   (
              ((=== () ===) ())
              ((X) X)
              ((XXX $ YYY) (=REPT= (XXX YYY)))
              ((XXX (CN YYY) ZZZ) (=REPT= (XXX YYY ZZZ)))
              ((XXX (IT X) (UN $ X) YYY) (=REPT= (XXX (IT X) YYY)))
              ((XXX (UN $ X) (IT X) YYY) (=REPT= (XXX (IT X) YYY)))
              (== (CN *SAME*))
             )
```

The significance of these simplifications are, line by line

A concatination involving the empty set is empty
No sign of operation is written to concatinate one element
The null word need not be written explicitly
Concatination is associative
$(\$ \cup X)\cdot X^* = X^*\cdot(\$ \cup X) = X^*$
Otherwise prefix notation is used with the symbol CN.

There is already apparent in the simplification ($ u X)·X* the fact that there are a great many equivalent forms which it is a nuisance to have to list separately; here we have used two rules for ($ u X)·X* and X*·($ u X), however there should be two more for (X u $)·X* and X*(X u $). The unordered variables mode of CONVERT is helpful in such situations, and is used in the simplification of the union. We have

```
   I     UNO   (X =I=)
   J     UNO   (X (IT X))
   K     UNO   ((CN WWW) AB*)
   L     UNO   (B*A (CN WWW))
  AB*    PAV   (CN (IT ==) WWW)
  B*A    PAV   (CN WWW (IT ==))
  =I=    PAV   (=OR= (CN X (IT ==)) (CN (IT ==) X))
```

With these constituents, we have

```
UNO   REPT   ((== (=REPT= (=UNON= =SAME=) *2 (
              ((XXX () YYY)              (=REPT= (XXX YYY)))
              ((XXX (UN YYY) ZZZ)        (=REPT= (XXX YYY ZZZ)))
              ((XXX  J  YYY  J  ZZZ)     (=REPT= (XXX YYY (IT X) ZZZ)))
              ((XXX  I  YYY  I  ZZZ)     (=REPT= (XXX YYY =I= ZZZ)))
              ((XXX  K  YYY  K  ZZZ)     (=REPT= (XXX YYY AB* ZZZ)))
              ((XXX  L  YYY  L  ZZZ)     (=REPT= (XXX YYY B*A ZZZ)))
              ((X XXX $ YYY)             (=REPT= ($ X XXX YYY)))
              ((X) X)
              (== (UN *SAME*))
             ))))
```

Again we may make a line-by-line analysis. On entry to the function UNO, we eliminate obviously repeated arguments. Then

The null set is deleted from a union
Union is associative
If X and X* appear in the union, we retain only X*
If both X and XY* appear then we retain only XY*
Likewise if WWW X* and WWW appear
Or X* WWW and WWW
In a list of at least one element we place the null

word first, an assumption made in the simplification of the concatination.

We write no operator for the union of one element Otherwise the prefix UN is written in prefix notation.

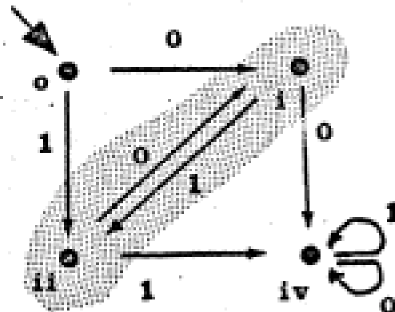
Finally, the simplifications of an iteration are the following.

```
ITR   REPT   (((X) (=REPT= X *3 (
              ((UN XXX $ YYY) (=REPT= (UNO XXX YYY)))
              ($ $)
              (() $)
              (== (IT =SAME=))
             ))))
```

There is very little simplification that we have seen fit to do directly on an iteration. The initial transformation is used to overcome the fact that CONVERT functions list their arguments, even when there is only one. We note that $* = ()* = $, otherwise the expression is left intact.

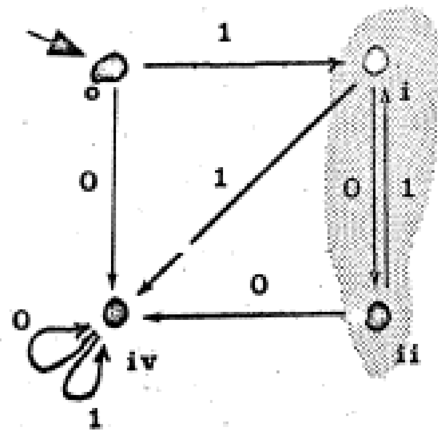
To see how effective these rules are we could consider some examples.

```
reex (o (i ii) ((o 1 ii) (o 0 i) (i 1 ii) (i 0 iv)
               (ii 0 i) (ii 1 iv) (iv 0 iv) (iv 1 iv)))

(UN 0 (CN 1 0) (CN (UN 0 (CN 1 0) (IT (CN 1 0))) 1 (CN (UN
0 (CN 1 0)) (IT (CN 1 0)) 1))
```



$0 \cup 10 \cup \left[(0 \cup 10)(10)^*\right] \cup 1 \cup \left[(0 \cup 10)(10)^* 1\right]$

```
reex (o (i ii) ((o 0 iv) (o 1 i) (i 0 ii) (i 1 iv)
               ((ii 0 iv) (ii 1 i) (iv 0 iv) (iv 1 iv)))

(UN 1 (CN 1 0 (IT (CN 1 0)) 1) (CN 1 0 (IT (CN 1 0))))
```



$$1 \cup (10(10)^{*}1) \cup (10(10)^{*})$$

As may be seen from the examples, the rules given succeed in eliminating almost all of the complexity due to redundant empty expressions and null words. Nevertheless, they are reasonably ad hoc, and do not eliminate more subtle types of redundancy. The subject might be worth pursuing further to test ones understanding of the simplification process, but a basic fault of the method is that it generates such cumbersome and so numerous expressions initially. Fortunately there are more amenable techniques available to form the regular expression which corresponds to a machine or transition system, principally the method of writing a series of regular expression equations for the states and solving them simultaneously.

REFERENCES

The McNaughton-Yamada Algorithm:

R. McNaughton and H. Yamada, "REGULAR EXPRESSIONS AND STATE GRAPHS FOR AUTOMATA," reprinted in Edward F. Moore (Editor), SEQUENTIAL MACHINES: SELECTED PAPERS, Reading, Massachusetts: Addison-Wesley Publishing Company, 1964.

Seymour Ginsburg, AN INTRODUCTION TO MATHEMATICAL MACHINE THEORY, Reading, Massachusetts: Addison-Wesley Publishing Company, 1962.

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

Canonical Equations:

Janusz A. Brzozowski, "DERIVATIVES OF REGULAR EXPRESSIONS," Journal of The Association for Computing Machinery 11 481-494 (1965).

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).

```
      DEFINE ((

(REEX (LAMBDA (I FF TT) (CONVERT
(CONS (QUOTE TT) (CONS (QUOTE EXPR) (CONS TT (QUOTE (
      L        BUV     ==
      S        BUV     =ATO=
      =I=      PAV     (=OR= (CN X (IT ==)) (CN (IT ==) X))
       I        UNO     (X =I=)
       J        UNO     (X (IT X))
       K        UNO     ((CN WWW) AB*)
       L        UNO     (B*A (CN WWW))
      AB*       PAV     (CN (IT ==) WWW)
      B*A       PAV     (CN WWW (IT ==))
     (U*)      PAT     ((*OR* (=== (I L I) U*) (===)))
     (T*)      PAT     ((*OR* (=== (I L F) T*) (===)))
     (S*)      PAT     ((*OR* (=== (S == S) S*) (===)))
      CON      REPT    (
                        ((=== () ===) ())
                        ((X) X)
                        ((XXX $ YYY) (=REPT= (XXX YYY)))
                        ((XXX (CN YYY) ZZZ) (=REPT= (XXX YYY ZZZ)))
                        ((XXX (IT X) (UN $ X) YYY) (=REPT= (XXX (IT X) YYY)))
                        ((XXX (UN $ X) (IT X) YYY) (=REPT= (XXX (IT X) YYY)))
                        (== (CN *SAME*))
                       )
      UNO      REPT    ((== (=REPT= (=UNON= =SAME=) *2 (
                        ((XXX () YYY)            (=REPT= (XXX YYY)))
                        ((XXX (UN YYY) ZZZ)      (=REPT= (XXX YYY ZZZ)))
                        ((XXX  J  YYY  J  ZZZ) (=REPT= (XXX YYY (IT X) ZZZ)))
                        ((XXX  I  YYY  I  ZZZ)    (=REPT= (XXX YYY =I= ZZZ)))
                        ((XXX  K  YYY  K  ZZZ)    (=REPT= (XXX YYY AB* ZZZ)))
                        ((XXX  L  YYY  L  ZZZ)    (=REPT= (XXX YYY B*A ZZZ)))
                        ((X XXX $ YYY) (=REPT= ($ X XXX YYY)))
                        ((X) X)
                        (() ())
                        (== (UN *SAME*))
                       ))))
      ITR      REPT    (((X) (=REPT= X *3 (
                        ((UN XXX $ YYY) (=REPT= (UNO XXX YYY)))
                        ($ $)
                        (() $)
                        (== (IT =SAME=))
                       ))))
      )))))
(QUOTE (
      I F X (WWW) (XXX) (YYY) (ZZZ)
      ))
(LIST I FF TT)
(QUOTE (*0 (
(((I F (S*)) (UNO (*ITER* J F (=REPT= (I J (=UNON= S)) *1 (
      ((I I ())         (=WHEN= TT (U*) (UNO $ (*UNON* L))))
      ((I F ())         (=WHEN= TT (T*)  (UNO (*UNON* L))))
      ((I F (X XXX)) (UNO (=REPT= (I F (XXX)))
                             (CON (=REPT= (I X (XXX)))
                                   (ITR (=REPT= (X X (XXX))))
                                   (=REPT= (X F (XXX)))))))
 )))))
 )))
```