JOB 5646-#175 RGX   PRINTED AT N. C. S. U. COMPUTING CENTER ON THE ITEL AS/4 AT 04:42:08 ON 09/04/80

```
SEPARATE   PUNCHED    PLOTTED    CARDS
 PRINT      CARDS     OUTPUT      IN
              0          0        732        732 LINES     12 PAGES ON PRINTER1
```

TUCC LABOR DAY SCHEDULE:HOLIDAY MODE(DOC.NO. IM-029-8) FROM MIDNITE SUN AUG 31
UNTIL 0800 TUES SEPT 2.

```
PROGRAMMER=#RATFOR                ACCOUNT=NCS.SAC.A4202

     JOB TURN-AROUND                     JOB PARAMETERS                    TIME USED INCLUDES          MISC. JOB VALUES

            DATE        TIME                 SPECIFIED   USED                           NO.  EQUIVALENT
ENTERED    9/03/80  22:15:39.6       TIME      0:02.0   0:02.9    CPU                        0:00.0     LINES IN        732
EXECUTED   0/00/00   0:00:00.0       PAGES        0       12      UR EXCPS            732    0:02.9     LINES OUT       732
RETURNED   9/03/80  22:15:13.9       CARDS      400        0      DISK EXCPS            0    0:00.0     MEMORY TIME       0 K-SEC
JOB ENTERED ON WEDNESDAY             PLOTS        0        0      TAPE READ-WRITE            0:00.0     PRIORITY         02
                                                                  TAPE FILE SEARCH           0:00.0     APPROX. COST  $1.25
```

```
@NOLIST
@COPY DB
@COPY PDS
   APPLY('DEF' 'INE',"RGXSOLVINIT()")
   PARTITION.OPEN('EXPRLIB')
   RGXSOLVINIT()
start:
#
#  Read each equation in and store it in the array eqn
   eqn = ARRAY(40)
   no.of.eqns = 0
   out = bl
   out = bl
   out = 'enter options'
   optionstring = TRIM(INPUT)
   out = "enter eqn's"
   while(eline = TRIM(INPUT)) {
      eline 11 $ ech rem $ member
      if(ident(ech,'E'))
         :(END)
      if(ident(ech,'$')) {
         PARTITION.POSITION(member)
         while(eline = TRIM(PARTITION.READ())) {
            no.of.eqns = no.of.eqns + 1
            eqn<no.of.eqns> = eline
            }
         break
         }
      eline br(' ') $ eqno 11 rem $ *eqn<eqno>
      if(ident(ech,'.'))
         break
      no.of.eqns = no.of.eqns + 1
   }
#
#  The set of equations if now formed, call RGXSOLV to solve the
# set and print the results.
   sttime = TIME()
   RGXSOLV(eqn,no.of.eqns,optionstring)
   stoptime = TIME()
   out = 'Solution time ' (stoptime - sttime) / 1000. ' secs.'
   :(start)


#-h-intro
##############################################################################
#
#  This is the subsystem of the REGS system that controls the solution of
# a set of equations for a given machine.  Rgxsolv is composed of several
# functions:
#
#  RGXSOLVINIT      Initialization of rgxsolv system
#  RGXSOLV          Control program that does initial reduction of system
#                    of equations passed to it, and then calls solve to
#                    actually produce the solution.
#  solve            Subroutine that actually does solution of equation system
#  options          Processes options passed to RGXSOLV
#  wait             Causes program to wait if desired at points in processing
#  arden,comsym,
#  distrib,parens,
#  rfmt             Functions used to manipulate equations during the solution
#                    process.
#
##############################################################################
#
#
#  General Information
```

```
#
#  RGXSOLV is called by the REGS mainline program to produce a set of
# regular expressions that denote the language accepted by a particular
# machine.  the mainline program passes RGXSOLV a set of equations and
# RGXSOLV cranks out the regular expressions.  More information on the
# actual operation of RGXSOLV can be found in its header.
#
#  If the reader desires a complete description of the solution process,
# the header section for "solve" explains the solution process in detail.
#
#  The key concept to the program's operation is the concept of
# Standard Form.  During the solution process, the equations are
# maintained in Standard Form.  Several things characterize an equation
# in standard form.  An equation is assumed to be of the form:
#   {state variable}=coefficient{st. var.}+coef{st. var} . . . .
# The state variable on the left of the equals sign can be any string
# except LAMBDA or EMPTY, which have special meanings, and it must
# be enclosed in brackets.  The right half of the equation is composed
# of one or more coefficient-state variable pair terms, or the value
# {EMPTY} standing alone.  The coefficient part of a term can contain
# any symbols except brackets and it is assumed to be well formed.  The
# coefficient is always maintained in a form so that for coefficients
# A and B, the operations A*, and A+B produce no ambiguities.  This
# form is maintained by adding parentheses when the order of symbols
# in an expression evaluated from left to right would produce a value
# other than the one desired.  Coefficients are also always enclosed
# by an all-surrounding set of parentheses.
#  The state variable part of a term is composed of one state name
# surrounded by parentheses.  If an operation causes a coefficient
# to be left with out a state, as often occurs in substitution in
# solve, a term of coefficient{LAMBDA} is used to make processing more
# uniform.  If the equation represents a final state, a +{LAMBDA}
# term is used which has a null coefficient.  Note that the user has
# the option to display the lambda's in a term if desired, but by
# default they do not appear.
#
######################################################################
#-%-intro
#-h-RGXSOLVINIT
#
#  RGXSOLVINIT is a function that will perform the necessary initiliazations
# for RGXSOLV and the functions it calls.  This function must be called
# before RGXSOLV.
RGXSOLVINIT:
#
#  First, some handy OPSYN's, and also some handy constant and
# pattern assignments.
   OPSYN(.L,.LEN) OPSYN(.BR,.BREAK)
   OPSYN(.l,.LEN) OPSYN(.br,.BREAK)
   rem = REM; l1 = LEN(1)
   OPSYN(.def,'DEF' 'INE')
   OUTPUT(.OUT,.OUT)
   OPSYN(.differ,.DIFFER)  OPSYN(.ident,.IDENT)
   no = 'n'; yes = 'y'
   OUTPUT(.out,.OUT)
   eqnsplit = br('=') $ lh l1 rem $ rh
   bl = ' '
   symbsize = 1
#
#  The optab table is used to define the various options that are
# available.  It is used by "options"
   optab = TABLE(10)
   optab<'SA'> = null;   optab<'WAIT'> = 'set.wait'
   optab<'SHL'> = 'sh.lam';  optab<'SHP'> = 'sh.parens'
   optab<'LA'> = 'listall';  optab<'LI'> = 'l.input'
#
#  Function definitions
   def("arden(expr)nexpr,lh,rh")
```

```
    def("comsym(expr)nexpr,lh,rh")
    def("distrib(expr)nexpr,lh,rh")
    def("options(optstring)")
    def("parens(yesno)")
    def("rfmt(expr)")
    def("solve(neqns)nexpr,lh,rh")
    def("RGXSOLV(eqn,eqcount,optstg)")
    def("wait()")
    return

#-%-RGXSOLVINIT
#-h-RGXSOLV
#
#  RGXSOLV is the driving subroutine called from the mainline program
# that controls the process of solving the set of equations.
#    Use:
#        RGXSOLV(eqn,eqcount,optstg)
#  Where "eqn" is an array of equations, of size "eqcount".  Optstg is
# a string consisting of processing options, with each option separated
# by a comma.  The options are not assumed to be valid.
#  RGXSOLV assumes that the equations being passed to it are well-formed
# and are of the form:
#  {State}=symbol{State}+symbol{State} . . .
# with no embedded blanks.  "symbol{State}" may be replaced with {LAMBDA}
# to indicate a final state.  It is not advisable to include more than
# one {LAMBDA} in an equation.
#
RGXSOLV:
#
#  Call options to process the options passed to RGXSOLV
options(optstg)

#
#  Print the initial (i.e. passed set of equations if the LI or LA
# option was specified
  if(ident(l.input,yes)){
      out = 'Initial set of equations:'
      for(eqno = 1;DIFFER(eqn<eqno>);eqno = eqno + 1)
          out = '[' eqno ']' rfmt(eqn<eqno>)
      wait()
      }

#
#  Now, do an initial reduction of the equations to standard form by
# successively calling comsym, arden and distrib for each of the
# equations in "eqn".
  for(eqno = 1;DIFFER(eqn<eqno>);eqno = eqno + 1){
      line = eqn<eqno>
      comsym = comsym(line)
      arden = arden(comsym)
      arden br('=') $ lh l1 rem $ expr
      #
      #  Before calling distrib, get rid of the left side of the
      # equation momentarily
      distrib = distrib(expr)
      distrib = lh '=' distrib
      #
      #  Replace the new equation in standard form in the array
      eqn<eqno> = distrib
  }

#
#  The equations have now all been reduced to standard form,
# if the user so specified, print resulting equations before solve
# goes to work.
  if(ident(l.all,yes)){
      out = 'Resulting equations after reduction to standard form'
      for(en = 1;LE(en,eqcount);en = en + 1)
```

```
        out = '[' en ']' ' rfmt(eqn<en>)
        wait()
        out = bl
        }

#
#  Now we are ready to call solve, the routine that plug-n-chugs the
# solution for the set out.  (i.e. The regular expression)
  out = 'Ready to solve equations . . .'
  solve(eqcount)
  out = 'done . . .'
  out = bl; out = bl

#
#  The set of equations has been solved, print the final set of
# equations, showing the regular expression that starting in each of
# the states represents.
  out = 'Final set of equations:'
  for(eqno = 1;LE(eqno,eqcount);eqno = eqno + 1)
     out = '[' eqno ']' ' rfmt(eqn<eqno>)
  return

#-%-RGXSOLV
#-h-arden
#
#  This subroutine tries to apply Arden's lemma to the passed expression
# and returns a resulting expression as the function value.
#    Use:
#     arden(expr)
#  Where "expr" is an expression to attempt to apply Arden's Lemma on.
# The function examines the expression to see if it can be modified to
# produce the general form of X=EX+G.  The G term might be null in which
# case the identity X=EX ==> X={EMPTY} is applied.  The EX term does not
# need to be the first term in the expression.  When the EX term is
# located, the term is removed from the body of the expression and the
# remainder of the expression is taken as the G term.  If X=EX is
# determined to be the form of the expression, then the
# expression is assigned a value of {EMPTY}.  The G term is enclosed in
# brackets to facilitate easy identification by distrib which is always
# called immediately after arden in case X=E*[G] is an equation that
# is not in standard form.
#  Arden assumes that the expression passed to it is in standard form,
# which in this case essentially implies that X can only occur once on
# the right hand side of the equals sign.
#
#  A possible problem arises concerning when and when to not enclose
# the E term in parentheses.  Obviously, if the E term is only one symbol
# there is no need to surround it with parentheses.  However, if the E
# term is longer than one symbol, it may or may not need enclosure to
# ensure the correctness of the resulting expression.  The key to this
# problem is remembering that the equations are maintained in standard
# form.  As a rule, arden is called immediately after comsym.  Because
# of this, we can assume that the E term is suitable for concatenation
# with an arbitrary term since it was concatenated with the X term.
# However, when the Kleene star operator is added, the E term must be
# surrounded by an all-enclosing set of parentheses.  This condition is
# indicated by the presence of left and right parentheses as the first
# and last characters of the E term.  If parentheses are not present
# in those positions, the E term must be enclosed before the Kleene star
# can be applied.
#
arden:
  #
  #  The equation is broken into a left and a right half, lh and rh
  # respectively.  A '+' is added on rh to avoid a special case for the
  # last term in the expression
  expr br("=") $ lh 11 rem $ rh
  rh = rh '+'
```

```
oldrh = rh        # rh is saved for later use
#
#  The following loop extracts each each term from the expression.
# If the term is found to be of the form EX, control passes out of
# the loop to "xfound".  If the loop falls through, it indicates that
# the equation can't be modified to produce a X=EX+G form and the
# original passed equation is returned as the function value.
while(rh ((br('}') 11) $ trans 11 ) = ){
   trans br('{') $ val (br('}') 11) $ st
   if(ident(st,lh))
      :(xfound)
}

#  EX term not found, return original expression
arden = expr
return

#
#  An EX term has been found, take the original right half of the
# and extract the EX term, leaving the G term.  If the G term is null,
# we have X=EX, so return a value of {EMPTY} for the expression.
xfound:
   g = oldrh
   g (trans '+') = null
   g RTAB(1) $ g     #  remove trailing '+'
   if(ident(g)) {  #   X=EX ==> X={EMPTY}
      arden = lh '={EMPTY}'
      return}

   #
   #  If val (E term) has a length of one symbol, or if it has a totally
   # enclosing set of parentheses, don't enclose it with parentheses before
   # applying Kleene star.  Otherwise, enclose it.
   if(val POS(0) '(' RTAB(1) ')')
      parens(no)
   else
      if(EQ(SIZE(val),symbsize))
         parens(no)
      else
         parens(yes)
   #
   #  Rebuild the new expression X=E*G, enclosing G in square brackets
   # to facilitate operation by distrib.
   nexpr = lparen val rparen '*[' g ']'
   arden = lh '=' nexpr   #  stick the left hand side back on the equation
   return
#-%-arden
#-h-comsym
#
#  The function "comsym" is used to identify terms that have common state
# variables, and combine the coefficients for all such terms to produce a
# new coefficient for each common state.
# For example, an input of X=AX+BY+CX+DY+EY would produce:
#  X=(A+C)X+(B+D+E)Y
#    Use:
#     comsym(expr)
#  Comsym is a conceptually very simple routine.  Each term, CX, in the
# expression is broken off in turn.  A table, sttab, based on the state X
# in each term holds all the coefficients, C that occur with state X.
# If a state X appears more than once in an expression, and occurs with
# coefficients A, B, and C for example, an entry of A+B+C would be
# recorded in the X entry of sttab.  Another table, modtab, is used to
# denote whether or not the expression in sttab for the corresponding
# state needs to be enclosed in parentheses before further operations
# can be done on it.  Modtab<X> will have a value of 'yes' if sttab<X>
# has a value that is the union of more than one coefficient.
#  The entire expression is broken into terms and each term into state
# variable and coefficient pairs.  Each pair is entered in sttab with
```

```
# the values in modtab possibly being changed.
#  When the entire expression has been processed, the table sttab is
# converted into an array.  The elements of the array are sequentially
# produced in coefficient-state variable pairs.  If modtab for a particular
# state is 'yes', the coefficient will be enclosed in parentheses.  The
# coefficient is then concatenated onto the state variable and the new
# term is added to the expression.  (The expression is rebuilt from null.)
# as soon as all of the pairs have been produced, the resulting expression
# is returned as the function value.
#  One important thing to note is that the sequence of state variables
# produced in the new expression may not be the same as the sequence
# they originally had.
comsym:
    sttab = TABLE(eqcount)
    #
    #  The expression is broken into a left and right half based on
    # the equals sign.  The '+' is added to avoid a special case.
    modtab = TABLE(eqcount)
    expr br('=') $ lh 11 rem $ rh
    rh = rh '+'

    #
    #  Break off the next coefficient-variable pair, placing the
    # coefficient in symb and the variable in st.  The coefficient
    # and variable are then entered in sttab and modtab.
    while(rh (br('{') $ symb br('+') $ st 11) = ) {
       #
       #  Treat a {LAMBDA} term standing by itself as a special case.
       if(ident(st,'{LAMBDA}')  ident(symb,null)) {
          symb = '.'; st = 'LAMBDA'}

       #
       #  Make entry in sttab for state variable.  If this is not
       # the first entry for this state variable, union the new
       # coefficient with the previous one(s).
       if(ident(sttab<st>))
          sttab<st> = symb
       else{
          sttab<st> = sttab<st> '+' symb
          modtab<st> = yes}
       #
       #  If we get an empty state, assume that {EMPTY} is the value
       # of the entire expression, and just return the entire expression.
       if(ident(st,'{EMPTY}')) {
          comsym = expr
          return}
       }

    #
    #  All the states have been processed, convert the sttab table to
    # an array for easy sequential recall.
    sttab = CONVERT(sttab,'ARRAY')
    #
    #  Recall each state-coefficient pair in turn to  rebuild the expr.
    for(i = 1;DIFFER(sttab<i,1>);i = i + 1) {
       #
       #  If a complex coefficient was formed, turn on the parentheses.
       if(ident(modtab<sttab<i,1>>,yes))
          parens(yes)
       else
          parens(no)
       #
       #  If we have do not have a free-standing lambda term, rebuild
       # the next term of the expression and add it to the new expression.
       #  If we do have a lambda term, just add it by itself.
       if(differ(sttab<i,1>,'LAMBDA')){
          comsym =  '+' lparen sttab<i,2> rparen sttab<i,1> comsym
       else
```

```
        comsym = '+{LAMBDA}' comsym
   }
   comsym 11 = null  # remove leading '+' from new expression
   comsym = lh '=' comsym  # put left-hand side back on and return
   return
#-%-comsym
#-h-distrib
#
#   Distrib is used to distribute a coefficient with a group of variables.
# The passed expression should be in the form:
#    Coefficient[term1+term2+term3 . . .] and distrib will return:
#    Coefficient term1 + Coef. term2 + Coef. term3 . . .
#   Use:
#      distrib(expr)
#   Distrib is the conceptually easiest of the three reducing functions.
# The expression is broken into coefficient and terms.  Each term is
# broken off in turn and concatenated to the coefficient until all
# the terms have been processed.  If the passed expression was of
# the form: coeff.term, ie. no [ ]'s, the argument will be returned as
# the function value.
distrib:
   #  put coeff. in comval, and [term+term. . .] in cfactor
   if(expr br('[') $ comval rem $ cfactor)
      ;
   else {  # no [ ]'s, return the passed expression
      distrib = expr
      return}
   # remove [ ]' surrounding cfactor
   cfactor 11 (RTAB(1) $ cfactor)
   # append '+' for uniform processing
   cfactor = cfactor '+'
   nexpr = null
   #
   #  Break out each term in turn and append it to the
   # common coefficient.
   while(cfactor (br('}') 11) $ factr 11 =)
      nexpr = nexpr '+' comval factr

   #
   #  Remove the leading '+' and return the new expression.
   nexpr 11 = null
   distrib = nexpr
   return

#-%-distrib
#-h-options
#
#   The options function is used to set the options taken by the program
# while processing the equations.  The table optab is used to establish
# an indirect branch location for each specifiable option.  The control
# variables are all null by default.  (Option not taken if null)
#   The following control variable-option associations are used:
#     LA-l.all        LI-l.input     SHL-sh.lam
#     WAIT-set.wait SHP-sh.parens
#   The options in the option string passed as an argument should be
# separated by commas.  If the option string is null, all options are
# turned off and the function returns.
#
options:
#
#   Set default tracing values
 sh.lam = null;   wait.fl = null;  sh.parens = null;
 s.all = null;  l.all = null;  l.input = null;
#
# See if we have any options
if(ident(optstring,null))
   return
```

```
#
#  Break off each option specified and process using indirect branch.
# If a option is unknown, tell the user and ignore it.
#
  optstring = TRIM(optstring) ','
  while(optstring br(',') $ opt l1 = ) {
     if(differ(optab<opt>,null))
        {branch = optab<opt>; :($branch)}
     else
        out = '"' opt '" is invalid and also ignored.'
nextopt: ;
  }
return

#
#  Indirect branch table to set options
solveall:  s.all = yes     :(nextopt)
listall:   l.all = yes
l.input:   l.input = yes   :(nextopt)
sh.lam:    sh.lam = yes    :(nextopt)
set.wait:  wait.fl = yes   :(nextopt)
sh.parens: sh.parens = yes  :(nextopt)
#-%-options
#-h-parens
#
#  The function parens is used to set lparen and rparen to '(',')'
# or null depending on the value of yesno.  If sh.parens is set, always
# set the parens.
parens:
  if(ident(sh.parens,yes)) {
     lparen = '('
     rparen = ')'
     return}
  if(ident(yesno,yes)) {
     lparen = '('
     rparen = ')'}
  else{
     lparen = null
     rparen = null}
  return
#-%-parens
#-h-rfmt
#
#  Rfmt is used to remove unnecessary {lambda}'s from a term for
# printing.  A lambda is deemed unnecessary if it occurs in the form:
# coefficient{LAMBDA}, or anotherwords not is the form . . .+{LAMBDA}
# or . . .={LAMBDA}
rfmt:
#
#  If sh.lam is set, don't remove the lambda's
  if(ident(sh.lam,yes)) {
     rfmt = expr
     return}
  rfmt = expr
  #
  # loop in while stmt. until all unnecessary lambda's are gone.
  while(rfmt NOTANY('+=') $ lpfx '{LAMBDA}' = lpfx)
     ;
  return

#-%-rfmt
#-h-solve
#
#  This is solve, the heart of the program.  This subroutine does the
# acutal work of solving the equations using the functions: arden,
# comsym, distrib, and rfmt.
#  Use:
```

```
#     solve(neqns)
#   Solve is called by RGXSOLV.  When solve is called, the array eqn
#  contains the set of equations to solve.  There are "neqns" equations
#  in the array, and they are all in standard form.
#
#   Solve starts with the Nth (last) equation in the list, and
#  substitutes the value for the regular expression represented by
#  the equation in each place the state variable for the Nth equation
#  appears throughout the set of equations.  The replacement is made
#  in the set of equations in order from the first to the last.
#   Each equation is broken up into terms of the standard CX variety
#  where C is the coefficient and X is the state variable for the
#  particular term.  If X happens to be the same state variable as the
#  one whose value is represented by the expression in the last equation,
#  X is replaced with the value of the last expression, i.e., a
#  substitution of two equivalent things has been made.  The resulting
#  term is in the form C[V], where V can be anything that can occur on
#  the right hand side of an equation in standard form.  In order to
#  keep things orderly, distrib is called with C[V] as an argument and
#  the resulting distributed expression is substituted in the equation
#  where CX originally was.  Because we know that a particular state
#  can only occur once in an equation in standard form, if and when
#  this substitution is made, we can move on to the next equation.
#   If the value for X had been {EMPTY} in the Nth equation, the CX
#  term would have been removed from the equation being processed.
#   A problem arises at this point, because after the substitution has
#  been made and distrib has been called, the resulting equation might
#  not be in standard form.  So, after each equation is processed for
#  a possible substitution, comsym and arden are called to reduce the
#  new equation to standard form.
#
#   The substition process continues for each equation in the array.
#  When the Nth equation has been processed, the state variable value
#  for the expression represented by the Nth equation no longer exists,
#  but instead its value in terms of a regular expression in all of the
#  equations where it used to be.
#
#   The same process will be repeated with the N-1th equation.  (Note that
#  at this time, the state variable for the Nth equation no longer is
#  present anywhere in the set of equations.)  The global substitution for
#  the value of state variable represented by the N-1th equation is done
#  just as it was for the Nth equation.  After the substitution process
#  is complete, the state variable for the N-1th equation is no longer
#  present in the set of equations, but instead its value.
#
#   The process repeats until all of the state variables have had their
#  values substituted throughout the set of N equations. AND at this
#  time, there will be NO state variables left in the equations, but
#  only their corresponding values.
#   In this manner, a solution is been obtained for each state of the
#  input machine, i.e. for each state as a hypothetical start state.
#
solve:
    #
    #  Use reqno to index the Nth, N-1th, . . . equations until all
    #  of the equations have been done.
    for(reqno = neqns - 1;GE(reqno,0);reqno = reqno - 1){
        #
        #  Separate the equation whose state variable we are going
        #  to look for into left and right side parts.  Rvalue is
        #  the value we will substitute for rstate if we find it.
        eqn<reqno + 1> eqnsplit
        rstate = lh; rvalue = rh
        anymatch = no   #  Turns to yes if we get a match for rstate
        #
        #  Eqno is used to point to each equation in the list in turn.
        #  Each equation will be checked for rstate and a substitution
        #  made if it is found.
```

```
for(eqno = 1;LE(eqno,neqns);eqno = eqno + 1) {
    eqn<eqno> eqnsplit  # get lh and rh
    nexpr = null;  rh = rh '+'
    #
    #  Break out each state in the equation and see if it matches
    # rstate.  Continue this process until a match is found or
    # we run out of equation.
    repeat{  # break off each term in equation
        if(rh br('{') $ value br('+') $ state l1 = null)
            ;
        else  # false if stmt. indicates all of equation is done
            break
        nullst = no;  # non-null state by default
        #
        #  See if the just extracted state matches
        if(ident(state,rstate)) {
            #
            #  Got a match, set match and anymatch to indicate it.
            match = yes
            anymatch = yes
            if(differ(rvalue,'{EMPTY}')) {
                #
                #  We have a non-null state to replace, construct a term
                # to place in the equation which has rstate replaced by
                # rvalue.
                ntrans = value '[' rvalue ']'
                ntrans = distrib(ntrans);  nullst = no}
            else
                #
                #  We have a null value for a state variable, use
                # nullst to indicate later removale of term from equation.
                nullst = yes
            }
        else
            match = no  # gives definite match or lack of it

        #
        #  If a match was made, part of eqn<eqno> has to be modified.
        # If the value for rstate is not null, the previous coeff.-state
        # pair is replaced by the old coeff and the value for rstate,
        # namely, rvalue.
        #  If rvalue is {EMPTY}, the coeff-state term being processed
        # can be removed from the equation.
        #  If no match was found for the particular state, the term
        # currently being worked on is replaced in the equation.
        #  Note that if a match was made, control will pass out of the
        # repeat-until, however if a match was not made, the next term
        # in eqn<eqno> will be examined for an occurance of rstate.
        if(ident(match,yes))  #  got a match
            if(differ(nullst,yes))  # see if null value
                nexpr = nexpr ntrans '+'   #  not a null
            else  #  have a null state
                ;
        else  # no match, replace involved part
            nexpr = nexpr value state '+'

    } until(ident(match,yes))

    #
    #  eqn<eqno> has been updated to reflect the value of rstate,
    # any occurance of rstate has been replaced by rvalue.
    #
    #  rh contains the part of the equation to the left of the
    # term that contained rstate if there was one.  If none was
    # found, it will be null.  In either case, append the leftover
    # part of the expression in rh to the new expression in nexpr.
    nexpr = nexpr rh
    nexpr RTAB(1) $ nexpr  # zap trailing '+'
```

```
        #
        #  If nexpr doesn't contain anything, we have an empty state
        if(ident(nexpr,null))
              nexpr = '{EMPTY}'
        #
        #  Put left-hand side of eqn back on to give new equation
        nexpr = lh '=' nexpr
        #
        #  The equation from eqn<eqno> has had rstate replaced with
        # rvalue, (if possible), but the new equation might not be
        # in standard form.  Call comsym, arden and distrib to make
        # sure we replace eqn<eqno> with an equation in standard form.
        c.expr = comsym(nexpr)
        a.expr = arden(c.expr)
        a.expr eqnsplit
        d.expr = lh '=' distrib(rh)
        #
        # stick sparkling new equation back in list
        eqn<eqno> = d.expr
      }  # for(eqno=1. . .

      #
      #  If the user specified the LA option, print the resulting set of
      # equations after global replacement of rstate with rvalue.
      if(ident(l.all,yes) ident(anymatch,yes) NE(reqno,0)) {
        out = bl
        out = 'After replacing ' rstate ' with ' rfmt(rvalue) ','
        out = ' the resulting set is:'
        for(en = 1;LE(en,neqns);en = en + 1)
          out = '[' en '] ' rfmt(eqn<en>)
        wait()
        }
    }  #for(reqno . . .
  return
  #-%-solve
  #-h-wait
  #
  #  If the WAIT option was specified, this function will print a line of
  # three dots and wait for a carriage-return before continuing execution,
  # thus allowing the user to have time to observe each operation.
  #  If the WAIT option was not specified, the function returns immediately.
  wait:
    if(ident(wait.fl,yes)){
      out = '. . .'
      junk = INPUT
      return
      }
    else
      return
  #-%-wait
  end
```

```
-#175--#175--#175--#175--#175--#175--#175--#175--#175--#175--#175--#175--#175--#175--#175--#175--#175--#175--#175--#175--#175--#175-

JOB 5646-#175 RGX        PRINTED AT N. C. S. U. COMPUTING CENTER ON THE ITEL AS/4  AT 04:43:23  ON 09/04/80
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
RGX         #175        12  PAGES                                                         RGX         #175        12  PAGES
RGX         #175        12  PAGES                                                         RGX         #175        12  PAGES
RGX         #175        12  PAGES                                                         RGX         #175        12  PAGES
RGX         #175        12  PAGES                                                         RGX         #175        12  PAGES
RGX         #175        12  PAGES                                                         RGX         #175        12  PAGES
RGX         #175        12  PAGES                                                         RGX         #175        12  PAGES
RGX         #175        12  PAGES                                                         RGX         #175        12  PAGES
RGX         #175        12  PAGES                                                         RGX         #175        12  PAGES
```


```
-#175--#175--#175--#175--#175--#175--#175--#175--#175--#175--#175--#175--#175--#175--#175--#175--#175--#175--#175--#175--#175--#175-

JOB 5646-#175 RGX        PRINTED AT N. C. S. U. COMPUTING CENTER ON THE ITEL AS/4  AT 04:43:23  ON 09/04/80
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
RGX         #175        12  PAGES                                                         RGX         #175        12  PAGES
RGX         #175        12  PAGES                                                         RGX         #175        12  PAGES
RGX         #175        12  PAGES                                                         RGX         #175        12  PAGES
RGX         #175        12  PAGES                                                         RGX         #175        12  PAGES
RGX         #175        12  PAGES                                                         RGX         #175        12  PAGES
RGX         #175        12  PAGES                                                         RGX         #175        12  PAGES
RGX         #175        12  PAGES                                                         RGX         #175        12  PAGES
RGX         #175        12  PAGES                                                         RGX         #175        12  PAGES
```