@MOLST
@COPY DB
@COPY PDS
APPLY 'DEF', 'INE', 'RGXSVINIT();'
PARTITION-OPEN('EXPLIB')
RGXSVINIT()

start:

# Read each equation in and store it in the array eqn
eqn = ARRAY[40]
no.of.eqns = 0
out = bl
out = dl
out = 'enter options'
optionstring = trim(input)
out = 'enter eqn'
while (eline = trim(input)) {
  eline l1 $ ech rem $ member
  if (!ident(\"ech,"\") { ; (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
    } else br (' '), $ eqn l1 rem $ *eqn<eqn>
    if (!ident(\"ech,"\") { ; (PARTITION-READ())
      no.of.eqns = no.of.eqns + 1
      break

# The set of equations if now formed, call RGXSVOLV to solve the
# set and print the results.

statime = time()
RGXSVOLV(eqns, no.of.eqns, optionstring)
STOPTIME = TIME()
out = 'Solution time' (STOPTIME - STATIME) / 1000. ' secs.' ; (start)

#-intro
# This is the subsystem of the REGS system that controls the solution of
# a set of equations for a given machine. RGXSVOLV is composed of several
# functions:
# RGXSVINIT  Initialization of rgxsvolv system
# RGXSVOLV  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 RGXSVOLV
# wait  Causes program to wait if desired at points in processing
# arden, consys, distrib, parens,
# rmat  Functions used to manipulate equations during the solution
# process.
#
# General Information
RGSOLV 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 RGSOLV a set of equations and RGSOLV cranks out the regular expressions.

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 [state var.] coefficient [state 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 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 use of the option to display the lambda's in a term if desired, but by default they do not appear.

RGSOLV is a function that will perform the necessary initializations for RGSOLV and the functions it calls. This function must be called before RGSOLV.

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; il = LEN(1)
OPSYN (L, LEN) OPSYN (L, LEN)
OUTPUT [.OUT, OUT]
OPSYN (DIFF, DIFF) OPSYN (IDENT, IDENT)
do = 'n'; yes = 'y'
OUTPUT [.OUT, OUT]
eqnsplit = br('n') $ ln l1 rem $ rh
p1 = 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.shem'; optab[SHM] = 'sh.shem'
optab[LA] = '1install'; optab['LI'] = '1input'

Function definitions
def* arden (expri nexpr.ih.rh)
def("comsys<expt> nepx,1,rb")
def("distflb<expt> nepx,1,rb")
def("options(optstring)")
def("paren(yesno)")
def("int<expt>")
def("solve<eqno> nepx,1,rb")
def("RGX$SOLV(eqno,eqcount,opttag)")
def("wait()")
return

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

# Print the initial (i.e., passed set of equations if the LI or LA
# option was specified
if(ident(...,input,ren))
  out = '+Initial set of equations:'
  for(eqno = 1;DIFFER(eqno<eqno>);eqno = eqno + 1)
    out = '+ eqno 'j + diff(eqno<eqno>)
  wait()

# Now, do an initial reduction of the equations to standard form by
# successively calling comsys, arden and distrib for each of the
# equations in "eqn".
for(eqno = 1;DIFFER(eqno<eqno>);eqno = eqno + 1)
  line = eqn<eqno>
  comsys = comsys(line)
  arden = arden(comsys)
  arden br('="") $ ln $ rem $ expr
  # Before calling distrib, get rid of the left side of the
  # equation momentarily
  distrib = distrib<expr>
  distrib = ln := 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
if(ident(L-all,yes))
  out = 'Resulting equations after reduction to standard form'
  for(en = 1;L(en,eqcount);en = en + 1)
out = '{' \n\nwait()
\nout = bl
\n}

# 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(ekcoun)
\nout = '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,eqcoun); eqno = eqno + 1)
out = '{' eqno '}' \nrfat(eqncoun)
return

#-RGISOLV
#-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 first term 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=EMPTY 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 consym. 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 $ \n
expr = rh '*'

oldrh = rh  # rh is saved for later use
#
# The following loop extracts 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 fails through, it indicates that
# the equation can't be modified to produce a EX=G form and the
# original passed equation is returned as the function value.
while rh {
  br (++) = |
  trans br (++) = val (br (++) = |
  if (ident (stat rh))
    : (xfound)
}
#
# EX term not found, return original expression
ar den = 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 = Old rh
  y (trans ++ = ) = null
  g STAB (1) $ y  # remove trailing ++
  if (ident (g) )  
    x = EX =⇒ X = [EMPT Y]
  ar den = y (EMPT Y)
  return
#
# If val (k 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 PLS (1) " " $ STAB (1) " ")
  paren (no)
else
  if (EQ (STIZE (val), syntize))
    paren (no)
  else
    paren (yes)
#
# Rebuild the new expression X=E*G, enclosing G in square brackets
# to facilitate operation by distrib.
  nexpr = jparen val rparen (ref (g) )
  ar den = ln " * " nexpr  # stick the left hand side back on the equation
return
#-ar den
# h=cosys
#
# The function "cosys" 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)*Y
# Use:
# cosys (expr)
# cosys is a conceptually very simple routine. Each term, CX, in the
# expression is broken off in turn. A table, STAB, based on the state X
# is 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 STAB. Another table, MDTAB, is used to
# determine whether or not the expression in STAB for the corresponding
# state needs to be enclosed in parentheses before further operation.
# can be done on it. MDTAB<IX> will have a value of 'yes' if STAB<IX>
# 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 STAB 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.

consym = TABLE(egcount)

# 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(egcount)

exp = br('**')$ if it then $ 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 i]) =

# 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(sttabcst))

sttabcst = symb
else

sttabcst = sttabcst '++' 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]'))

consym = exp

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(j = 1; DIFFER(sttab<1,1>); j = j + 1)

# If a complex coefficient was formed, turn on the parentheses.
if(ident(modtab<sttab<1,1>,yes))

paren(yes)
else

paren(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<1,1>,LAMBDA))

comsym = +++a[paren sttab<1,2>] paren sttab<1,1> consym
else

# If we have a lambda term, just add it by itself.
if(differ(sttab<1,1>,LAMBDA))

comsym = +++a[paren sttab<1,2>] paren sttab<1,1> consym
else

# 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<1,1>,LAMBDA))

comsym = +++a[paren sttab<1,2>] paren sttab<1,1> consym
else

# 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<1,1>,LAMBDA))

comsym = +++a[paren sttab<1,2>] paren sttab<1,1> consym
else

# 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<1,1>,LAMBDA))

comsym = +++a[paren sttab<1,2>] paren sttab<1,1> consym
else
comsys = '+'[LAMEDA]' comsys

comsys 1+ = null  # remove leading '+' from new expression
comsys 1h ="" comsys  # put left-hand side back on and return

# X-comsys
# X-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:
# Coeff 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.

# put coeff in convl and (term+term - - ) in cfactor
if(expr br('['') $ convl rem $ cfactor)
  else ( # no ['s, return the passed expression
    distrib = expr
    # put ['] surrounding cfactor
    cfactor l1(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('['') l1) $ factr l1 =)
      nexpr = nexpr ', ' convl factr
    # remove the leading '+' and return the new expression.
    nexpr l1 = null
    distrib = nexpr
    return

# X-distrib
# X-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 it null)
# The following control variable-option associations are used:
# LA- L.all LI-l.input SHL-sh.lam
# WAIT-set.wait SHP-sh.paren
# 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.li = null; sh.paren = 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')(') $opt'11 = ){
  if(differ(optabOpt, null))
    (branch = optabOpt):($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.paren: sh.paren = yes : (nextopt)
#-a- options
#-h-paren

# The function paren is used to set lparen and rparen to '(, ')'.
# or null depending on the value of yes/no. If sh.paren is set, always
# set the paren.
paren: if(iden(sh.paren, yes)) {
  # lparen = '('
  lparen = '('
  return
}
if(iden(yes, yes)){
  lparen = '('
  rparen = ')'}
else{
  lparen = null
  rparen = null
return

#-a-paren
#-h-rfat

# Rfat 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 \(-= [ [lambda] ]\)
rfat: 
if sh.lam is set, don't remove the lambda's

if(iden(sh.lam, yes)) {
  rfat = expr
  return
}
rfat = expr
# loop while in brackets until all unnecessary lambda's are gone.
while(rfat NOTANY('='+) $lpfx '{ [lambda] }' = lpfx)

return

#-a-rfat
#-h-solve

# This is solve, the heart of the program. This subroutine does the
# actual work of solving the equations using the functions: arden,
# comax, distrib, and rfat.
# Use!
# Solve(negs)

Solve is called by RGXSOLV. When solve is called, the array `negs`
contains the set of equations to solve. There are `negs` 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 form
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, distr is called with (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 I has 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 distr has been called, the resulting equation might
not be in standard form. So, after each equation is processed for
a possible substitution, consya and arden are called to reduce the
new equation to standard form.

The substitution 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
the equations where it used to be.

The same process will be repeated with the (N-1)th 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 Nth equation is done
just as it was for the Nth equation. After the substitution process
is complete, the state variable for the (N-1)th 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 that
time, there will be NO state variables left in the equations, but
only their corresponding values.

In this manner, a solution is obtained for each state of the
input machine, i.e. for each state as a hypothetical start state.

solve:

Use regno to index the Nth, N-1th, ... equations until all
of the equations have been done.
for(regno = negs - 1;GE(regno,0);regno = regno - 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.
eqns = regno + 1; eqnsplit;
next = 1h; rvalue = rth
anymatch = no
# Turns to yes if we get a match for rstate
# Each equation 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; eqno < eqno+offset; eqno = eqno + 1) {
  eqn<eqno> = eqnsplit; // get lb 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 equations.
  repeat # Break off each term in equation
    if (! (rbr[1] && value(rbr[2]) && state lb = null)) {
      # false if stat. indicates all of equation is done
      break
    } else {
      null list = 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]; null list = no
        } else {
          # # We have a null value for a state variable, use
          # null list to indicate later removal of term from equation.
          null list = 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 occurrence of rstate.
      if (ident(match, yes)) # got a match
        nexpr = nexpr btrans "*" # not a null
      else # have a null state
        nexpr = nexpr value state "*"
    }
  until (ident(match, yes))
  # eqn<eqno> has been updated to reflect the value of rstate,
  # any occurrence 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

```plaintext
if (ident(nexpr) = 'null')
    nexpr = 'empty'
```

Put left-hand side of `eqn` back on to give new equation

```plaintext
nexpr = lh = nexpr
```

The equation from `eqn` has had rstate replaced with
```plaintext
rvalue, if possible, but the new equation might not be
```
```plaintext
in standard form. Call consym, arden and distrib to make
```
```plaintext
sure we replace `eqn` with an equation in standard form.
```
```plaintext
c.expr = consym(nexpr)
a.expr = arden(c.expr)
a.expr eqn.split
```
```plaintext
d.expr = lh = distrib(rh)
```
```plaintext
# stick sparking new equation back in list
```
```plaintext
eqn(eqn) = d.expr
```
```plaintext
# for(eqn = 1 .. n)
```

If the user specified the LA option, print the resulting set of
```plaintext
equations after global replacement of rstate with rvalue.
```
```plaintext
if (ident(l.all, yes) ident(ahmatch, yes) NE(regno, 0)) {
    out = being
    out = 'After replacing ' rstate ' with ' rfmt(rvalue) ' ',
    out = 'the resulting set is:
    for(en = 1; len[en, negns]; en = en + 1)
        out = ['en '] rfmt(eqn(en))
    wait()
}
```
```plaintext
# for(regno = 1 .. n)
```
```plaintext
return
```
```plaintext
#-x-solve
```
```plaintext
#-h-wait
```
```plaintext
#-x-wait
```
```plaintext
```
```plaintext
# 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.
```
```plaintext
wait:
if (ident(wait.fl, yes)) {
    out = '...
    junk = INPUT
    return
}
else
    return
```
```plaintext
end
```