APPENDIX G

PROGRAM LISTING

```
# Public Domain (P) 1991 by Doug Witmer.  No Rights Reserved.
# This is version 1.2 of TRAN1.  11/04/91.

# This is an early version of Tran1.  It is being circulated in
# hope that some kind souls will review it, and send their comme
# the author.  Of particular interest are questions like:
#
#    1) Can anything be done more efficiently?
#    2) Can anything be done in a more Iconish manner?
#    3) Have any standards of good programming practice been vio
#    4) Does the reviewer see any obvious flaws or opportunities
#       improvement?
#
# Please send any comments to the author at:
#
#                                Doug Witmer
#                          1102 Enterprise Drive #149
#                          Grand Prairie, Texas  75051
#
#   A more recent address for the author can be obtained by writ
#   the Icon Project at the address below:
#
#                                Icon Project
#                       Department of Computer Science
#                           Gould-Simpson Building
#                         The University of Arizona
#                         Tucson, Arizona  85721  USA
#
#                     e-mail:  icon-project@cs.arizona.edu
#
#                                      or
#
#                 ...{uunet,allegra,noao}!arizona!icon-project


# This program reads a semantic representation of a text in Engl
# and translates it into another language.  It is meant to be us
# process texts which are going to be translated into a wide var
# target languages.  The extra effort needed to produce the sema
# representation of the text cannot be justified unless multiple
# languages are involved.

# It is hoped that by using this program the majority of the man
# involved in the translation process can be performed by people
# mono-lingual.  That is, the semantic representation of the tex
# produced by a native speaker of English, and the final editing
# the translated text can be done by a native speaker of the tar
# language.

# The program is meant to run on a machine with substantial memo
# However, it could be run on a lesser machine if semread() and
```

```
# were rewritten.  It will also run on a lesser machine if the
# size of the semanticon is kept small.  (This is the case for t
# included with the pre-release version.)  The program was devel
# MS-DOS, but there is no reason why it shouldn't run under othe
# systems.

# Lines to be translated are kept in a list structure.
# Each list element is itself a list consisting of a semantic ta
# followed by another list which is the semanticon entry
# corresponding to the semantic tag.  Here is a sample list
# element:
#
#                              |---- Morphological tag
#                              |
#                              |      |----- Target language sense
# Semantic tag -----|          |      |
#                   |          |      |
#               ["feast1",["n","la fiesta"]].
#
# A list of these elements constitutes a line to be translated.

# Morphological tags (ie. parts of speech).
global adj, adj1, adj1n, adj2, adj2n, adv
global art_def, art_ind, com, conj, conj1
global dem_fs, dem_fp
global dem_mp, dem_ms, noun, neg, np, num, n_mass, neg, np, num
global prep, prep_phra, pro_dio_3p
global pro, pro_dio_3p_pol, pro_dio_3s_pol
global pro_dio_3sf, pro_dio_3sm
global pro_do_1p, pro_do_1s
global pro_do_2p, pro_do_2pm_pol
global pro_do_2s, pro_do_3pf, pro_do_3pf_pol
global pro_do_3pm, pro_do_3pm_pol, pro_do_3sf
global pro_do_3sm, pro_do_3sn, pro_io_1p, pro_io_1s
global pro_io_2p, pro_io_2pm, pro_io_2p_pol, pro_io_2s, pro_io_2
global pro_io_3p, pro_io_3p_pol, pro_io_3s_pol, pro_io_3sf, pro_
global pro_op_1pf, pro_op_1pm, pro_op_1s, pro_op_2p_pol, pro_op_
global pro_op_2pm, pro_op_2s, pro_op_2s_pol, pro_op_3p, pro_op_3
global pro_op_3pf, pro_op_3pm, pro_op_3s, pro_op_3s_refl, pro_op
global pro_op_3sf_refl, pro_op_3sm, pro_op_3sm_refl
global pro_poss_1s, pro_poss_2s
global pro_poss_3ms, pro_poss_3p
global pro_refl_3p, pro_rel
global pro_su_1pf, pro_su_1pm, pro_su_1s, pro_su_2p, pro_su_2p_p
global pro_su_2s_pol, pro_su_3pf, pro_su_3pf_pol, pro_su_3pm, pr
global pro_ref_3p, pro_rel, sent_adv
global v_fut, v_fut_perf, v_impf, v_impf_subj, v_inf
global v_past, v_past_part, v_past_perf, v_past_perf_subj, v_pas
global v_perf, v_pres, v_pres_impv, v_pres_perf, v_pres_prog, v_
global v_subj, ve_fut, ve_past, ve_past_perf, ve_pres, ve_pres_p
global vs_past, vs_past_perf, vs_pres
```

```
# Miscellaneous globals.
global a_el2, al2, capitalize, citation, dash, del2, de_el2
global diags1, el, equalsign
global la, la2, las, las2, le2, les2, le_la2, le_las2, les_la2,
global se2, se_la2, se_las2, pf, pm, sf, sm
global los, lb, lcs
global letters, nul, pun, punb, rb, rparen
global semtable, sentpunct, sp, spc, spaces
global the, The, ucp, underscore, yes

procedure main(main_args)
# Arguments: main_args[1]  == "\ICON\DATA\" to read and write al
#                                          files from this dire
#                                          This argument is req
#            main_args[2]  == "y" to receive  diagnostics in tex
#            main_args[2] ~== "y" to suppress diagnostics in tex
#            main_args[3]  == "null" to send the diags1 diagnost
#                                    null device (ie. suppress t
#            main_args[3]  == "NAME.EXT" to send the diags1 diag
#                                  the file NAME.EXT.
#            main_args[3]  == "" to send diags1 diagnostics to t
#                               file DIAGS1.OUT.

# Perform initialization of morphological tags (ie. parts of spe
   adj               := "adj"
   adj1              := "adj1"
   adj1n             := "adj1n"
   adj2              := "adj2"
   adj2n             := "adj2n"
   adv               := "adv"
   art_def           := "art-def"
   art_ind           := "art-ind"
   com               := "com"
   conj              := "conj"
   conj1             := "conj1"
   dem_fs            := "dem-fs"
   dem_fp            := "dem-fp"
   dem_mp            := "dem-mp"
   dem_ms            := "dem-ms"
   el                := "el"
   la                := "la"
   las               := "las"
   los               := "los"
   n_mass            := "n-mass"
   noun              := "n"
   neg               := "neg"
   np                := "np"
   num               := "num"
   prep              := "prep"
   prep_phra         := "prep-phra"
   pro               := "pro"
   pro_dio_3p        := "pro-dio-3p"
```

```
pro_dio_3p_pol    := "pro-dio-3p-pol"
pro_dio_3s_pol    := "pro-dio-3s-pol"
pro_dio_3sf       := "pro-dio-3sf"
pro_dio_3sm       := "pro-dio-3sm"
pro_do_1p         := "pro-do-1p"
pro_do_1s         := "pro-do-1s"
pro_do_2p         := "pro-do-2p"
pro_do_2pm_pol    := "pro-do-2pm-pol"
pro_do_2s         := "pro-do-2s"
pro_do_3pf        := "pro-do-3pf"
pro_do_3pf_pol    := "pro-do-3pf-pol"
pro_do_3pm        := "pro-do-3pm"
pro_do_3pm_pol    := "pro-do-3pm-pol"
pro_do_3sf        := "pro-do-3sf"
pro_do_3sm        := "pro-do-3sm"
pro_do_3sn        := "pro-do-3sn"
pro_io_1p         := "pro-io-1p"
pro_io_1s         := "pro-io-1s"
pro_io_2p         := "pro-io-2p"
pro_io_2pm        := "pro-io-2pm"
pro_io_2p_pol     := "pro-io-2p-pol"
pro_io_2s         := "pro-io-2s"
pro_io_2s_pol     := "pro-io-2s-pol"
pro_io_3p         := "pro-io-3p"
pro_io_3p_pol     := "pro-io-3p-pol"
pro_io_3s_pol     := "pro-io-3s-pol"
pro_io_3pm        := "pro-io-3pm"
pro_io_3s_pol     := "pro-io-3s-pol"
pro_io_3sf        := "pro-io-3sf"
pro_io_3sm        := "pro-io-3sm"
pro_op_1pf        := "pro-op-1pf"
pro_op_1pm        := "pro-op-1pm"
pro_op_1s         := "pro-op-1s"
pro_op_2p_pol     := "pro-op-2p-pol"
pro_op_2pf        := "pro-op-2pf"
pro_op_2pm        := "pro-op-2pm"
pro_op_2s         := "pro-op-2s"
pro_op_2s_pol     := "pro-op-2s-pol"
pro_op_3p         := "pro-op-3p"
pro_op_3p_refl    := "pro-op-3p-refl"
pro_op_3pf        := "pro-op-3pf"
pro_op_3pm        := "pro-op-3pm"
pro_op_3s         := "pro-op-3s"
pro_op_3s_refl    := "pro-op-3s-refl"
pro_op_3sf        := "pro-op-3sf"
pro_op_3sf_refl   := "pro-op-3sf-refl"
pro_op_3sm        := "pro-op-3sm"
pro_op_3sm_refl   := "pro-op-3sm-refl"
pro_poss_1s       := "pro-poss-1s"
pro_poss_2s       := "pro-poss-2s"
pro_poss_3ms      := "pro-poss-3ms"
pro_poss_3p       := "pro-poss-3p"
```

```
   pro_refl_3p       := "pro-refl-3p"
   pro_rel           := "pro-rel"
   pro_su_1pf        := "pro-su-1pf"
   pro_su_1pm        := "pro-su-1pm"
   pro_su_1s         := "pro-su-1s"
   pro_su_2p         := "pro-su-2p"
   pro_su_2p_pol     := "pro-su-2p-pol"
   pro_su_2s         := "pro-su-2s"
   pro_su_2s_pol     := "pro-su-2s-pol"
   pro_su_3pf        := "pro-su-3pf"
   pro_su_3pf_pol    := "pro-su-3pf-pol"
   pro_su_3pm        := "pro-su-3pm"
   pro_su_3s         := "pro-su-3s"
   sent_adv          := "sent-adv"
   v_fut             := "v-fut"
   v_fut_perf        := "v-fut-perf"
   v_impf            := "v-impf"
   v_impf_subj       := "v-impf-subj"
   v_inf             := "v-inf"
   v_past            := "v-past"
   v_past_part       := "v-past-part"
   v_past_perf       := "v-past-perf"
   v_past_perf_subj  := "v-past-perf-subj"
   v_past_prog       := "v-past-prog"
   v_perf            := "v-perf"
   v_pres            := "v-pres"
   v_pres_impv       := "v-pres-impv"
   v_pres_perf       := "v-pres-perf"
   v_pres_prog       := "v-pres-prog"
   v_pres_subj       := "v-pres-subj"
   v_subj            := "v-subj"
   ve_fut            := "ve-fut"
   ve_past           := "ve-past"
   ve_past_perf      := "ve-past-perf"
   ve_pres           := "ve-pres"
   ve_pres_perf      := "ve-pres-perf"
   ve_pres_perf_subj := "ve-pres-perf-subj"
   vs_past           := "vs-past"
   vs_past_perf      := "vs-past-perf"
   vs_pres           := "vs-pres"

# Perform miscellaneous initializations.
# Note: Strings are surrounded by double quotes.  Csets have sin
   a_el2 := " a el "
   al2 := " al "
   capitalize := ""
   dash := '-'
   de_el2 := " de el "
   del2 := " del "
   equalsign := "="
   la2 := " la "
   las2 := " las "
```

```
   le2 := " le "
   les2 := " les "
   le_la2 := " le la "
   le_las2 := " le las "
   les_la2 := " les la "
   les_las2 := " les las "
   lb := "{"
   letters := &ucase ++ &lcase       # Icon version 7.1 lacks &l
   nul := ""                         # Not to be confused with &
   pf := "pf"                        # Plural Feminine
   pm := "pm"                        # Plural Masculine
   rb := "}"
   rparen := ")"
   se2  := " se "
   se_la2 := " se la "
   se_las2 := " se las "
   sentpunct := '.?!'                # Punctuation which ends a
   sf := "sf"                        # Singular Feminine
   sm := "sm"                        # Singular Masculine
   sp := ' '                         # Space Cset
   spc := " "                        # Space Character
   spaces := "
   the := "the1"
   The := "The1"
   pun := ',.;:"()\'?!'              # Punctuation Cset
   punb := ',.;:"()\'?!{}[]-/'       # Punctuation Cset
   ucp := &ucase || ",.;:\"()'?!{}"  # Upper Case + Punctuation
   lcs := &lcase || "              "  # Lower Case + Spaces
   underscore := "_"
   yes := "y"

   write(&errout,"Program Tran1 is beginning on:      ",&datelin

# Open the files.
   file_name := main_args[1] || "tran1.in"
   if not (semrepin := open(file_name,"r"))                  # Un
      then stop("Cannot open ",file_name)                    #
   file_name := main_args[1] || "tran1.out"
   if not (textout := open(file_name,"w"))                   # Tr
      then stop("Cannot open ",file_name)                    #
   file_name := main_args[1] || "diags.out"
   file_name := main_args[1] || main_args[3]
   if not (diags1 := open(file_name,"w"))                    # Di
      then stop("Cannot open ",file_name)                    #

# Load the semanticon.
   semtable := semread(main_args)

# Read the untranslated text.
   while (line := semrepget(semrepin) || spc) do {    # Read unt
      linelist := []                  # This list will contain t
      if match(equalsign,line)        # Is this a paragraph brea
```

```
         then citation := line[1:-1]
         else line ? { citation := tab(find(rparen) + 1)
                       tab(many(sp))
                       while ( semtag := (tab(upto(sp))) )  do {
                          put(linelist,[semtag,semget(semtag)])
                          tab(many(sp))
                       }
         }

# Write a little diagnostic information.
         write(diags1,citation)                                    #
      if main_args[2] == yes then {
         write(textout,citation)                                   #
         every i := 1 to *linelist do {
            write(textout,1,spc,"1  :",linelist[i][1])             #
            write(textout,1,spc,"2-1:",linelist[i][2][1])          #
            write(textout,1,spc,"2-2:",linelist[i][2][2])          #
         }
      }

# Check for a line beginning with a capital letter.
      linelist[1][1] ? {
         tab(many(punb))
         if ( any(&ucase,move(1)) )
            then capitalize := yes
            else capitalize := nul
      }

# Translate the line.
      write( textout,citation," ",          # Segment the line, t
         outstring(segment1(linelist)) )    #   and write it out.
   }

# Lock up, and go home.
   close(semrepin)
   close(textout)
   close(diags1)
   write(&errout,"Program Tran1 is ending on:         ",&datelin
end

###################  Procedures  ###################

# Note: All the procedures with names like xxxx_proc() are at le
# partially language specific.  They may need to be modified dep
# on the target language.

procedure adj_move(linelist2,j,lenlinelist)
# This language specific procedure moves certain adjectives to t
# following their nouns.  It also adjusts the punctuation as nec

# The identifier pun must be a global Cset containing punctuatio
# The identifier nul must be a global containing an empty string
```

```
   local templist, pun1, pun2, tmp1, tmp2

   write(diags1,&level," Entering adj_move: j = ",j," ",outstrin
   pun1 := pun2 := nul
   templist := []
   linelist2[j][1] ? {
      pun1 := tab(many(pun))              # Get any initial punc
      linelist2[j][1] := tab(0)
   }
   while j <= lenlinelist do {
      case linelist2[j][2][1] of {
         (adj2 |
          adj2n)  : put(templist,copy(linelist2[j]))  # Add to r
         (noun |
          np   |
          n_mass) : { linelist2[j][1] ? {
                         tmp1 := (tab(many(pun)) | nul) # Leadin
                         tmp2 := (tab(upto(pun)) | tab(0)) # Sem
                         pun2 := (tab(many(pun)) | nul) # Traili
                      }
                      linelist2[j][1] := tmp1 || tmp2    # Drop
                      push(templist,copy(linelist2[j]))  # Add t
                      break
                    }
         default  : { j -:= 1 ; break }
      }
   j +:= 1
   }
   templist[1][1] := pun1 || templist[1][1]   # Append initial p
   templist[*templist][1] ||:= pun2           # Append trailing
   write(diags1,&level," Exiting adj_move: j = ",j," ",outstring
   return [j,copy(templist)]
end

procedure adj1_proc(linelist,i)
# This procedure handles adjectives which must preceed the thing

   write(diags1,&level," Entering adj1: ",outstring(linelist),"
   return copy([linelist[i][1],[linelist[i][2][1],linelist[i][2]
end

procedure adj2_proc(linelist,i)
# This procedure handles adjectives which must follow the thing

   write(diags1,&level," Entering adj2: ",outstring(linelist),"
   return copy([linelist[i][1],[linelist[i][2][1],linelist[i][2]
end

procedure agree(linelist,i)
# This language specific procedure handles agreement of adjectiv
# demonstratives, possesive pronouns, and indefinite articles wi
# governing nouns or pronouns.
```

```
# The identifier el must be a global containing the masculine si
# The identifier la must be a global containing the feminine sin
# The identifier las must be a global containing the feminine pl
# The identifier los must be a global containing the masculine p
# The identifier noun must be global and contain "n".
# The identifier np must be global and contain "np".
# The identifier n_mass must be global and contain "n-mass".
# The identifier nul must be a global containing an empty string

   local article, fs, fp, j, lenlinelist, ms, mp, png, png_save,
   local semtablelist, type
   local person, number, gender, num_gen

   write(diags1,&level," Entering agree: ",linelist[i][1],spc,
        linelist[i][2][1],spc,linelist[i][2][2],\n,"   ",
        outstring(linelist)," i: ",i)

# Find the governing noun, and look it up again in the semantico
# Also look for a possible governing pronoun.
   semtablelist := []
   lenlinelist := *linelist
   png_save := nul
   j := i + 1
   while j <= lenlinelist do {
      write(diags1,&level,spc,j," Working in agree. Looking for
         linelist[j][1],spc,
         linelist[j][2][1],spc,linelist[j][2][2])
      pos := type := png := nul
      linelist[j][2][1] ? { pos  := tab(upto(dash))            #
                                    tab(many(dash))            #
                            type := tab(upto(dash))            #
                                    tab(many(dash))            #
                            png  := (tab(upto(dash)) | tab(0)) #
                          }
      if pos == (pro) then {  # Is this a pronoun?
         write(diags1,&level,spc,j," Pronoun in agree: ",linelis
         png_save := png   # Save the person, number, and gender
      }
      if linelist[j][2][1] == (noun | n_mass | np) then { # Is t
         write(diags1,&level,spc,j," Noun in agree: ",linelist[j
         semtablelist := semget(linelist[j][1])
         semtablelist[2] ? {
            article := tab(upto(sp))  # Get the this noun's arti
            write(diags1,&level,spc,j," Article: ",article)
            break
         }
      }
      j +:= 1
   }
```

```
# Extract the possible forms.
   linelist[i][2][2] ? {
      ms := ( tab(upto(sp)) | tab(0) )  # masculine singular for
      tab(many(sp))
      fs := ( tab(upto(sp)) | tab(0) )  # feminine singular form
      tab(many(sp))
      mp := ( tab(upto(sp)) | tab(0) )  # masculine plural form
      tab(many(sp))
      fp := ( tab(upto(sp)) | tab(0) )  # feminine plural form
   }

   if j > lenlinelist
      then { # Use the pronoun's number and gender to produce ag
         num_gen := person := number := gender := nul
         png_save ? { person := move(1)
                      number := move(1)
                      gender := (move(1) | "m")
                    }
         if gender == "n" then {gender := "m"}
         num_gen := number || gender
         case num_gen of { # Produce agreement.
            sm                : linelist[i][2][2] := ms
            sf                : linelist[i][2][2] := fs
            pm                : linelist[i][2][2] := mp
            pf                : linelist[i][2][2] := fp
            default           : write(diags1,&level,
                                    ">>>Something is wrong with t
                                    num_gen)
         }
      }
      else { # Use the noun's article to produce agreement.
         case article of { # Produce agreement.
            el                : linelist[i][2][2] := ms
            la                : linelist[i][2][2] := fs
            los               : linelist[i][2][2] := mp
            las               : linelist[i][2][2] := fp
            default           : write(diags1,&level,
                                    ">>>Something is wrong with t
                                    article)
         }
      }

   write(diags1,&level," Exiting agree: ",linelist[i][1],spc,
        linelist[i][2][1],spc,linelist[i][2][2])
   return copy([linelist[i][1],[linelist[i][2][1],linelist[i][2]
end

procedure art_proc(linelist,i)
# This procedure handles definite articles.

# On input i points to an English "the".  This procedure finds t
# which "the" refers to, removes the article from that noun, and
```

```
# to the position of the English "the".  This acheives article-n
# agreement.

# The identifier noun must be global and contain "n".
# The identifier np must be global and contain "np".
# The identifier n_mass must be global and contain "n-mass".
# The identifier The must be global and contain "The1".
# The identifier the must be global and contain "the1".

   local article, j, drop, lenlinelist

   write(diags1,&level," Entering art_proc: ",linelist[i][1],spc
        linelist[i][2][1],spc,linelist[i][2][2],\n,"   ",
        outstring(linelist)," i: ",i)
   if linelist[i][1] == (the | The) then linelist[i][2][2] := ""
   lenlinelist := *linelist
   j := i + 1
   while j <= lenlinelist do {  # Look for the governing noun.
      write(diags1,&level,spc,j," Working in art_proc: ",linelis
         linelist[j][2][1],spc,linelist[j][2][2])
      if linelist[j][2][1] == (noun | n_mass | np) then {
         write(diags1,&level,spc,j," Noun in art_proc: ",linelis
         linelist[j][2][2] ? {
            article := tab(upto(sp))
            write(diags1,&level,spc,j," Article: ",article)
            tab(many(sp))
            linelist[j][2][2] := tab(0)       # Drop the article
            write(diags1,&level,spc,j," Noun: ",linelist[j][2][2
            linelist[i][2][2] := article      # Move the article
            break
         }
      }
      j +:= 1
   }
   write(diags1,&level," Exiting art_proc: ",linelist[i][1],spc,
        linelist[i][2][1],spc,linelist[i][2][2])
   return copy([linelist[i][1],[linelist[i][2][1],linelist[i][2]
end

procedure capital(outstr2)
# This procedure accepts a string and capitalizes the first lett
# that string if the variable capitalize contains "y".

# The identifier nul must be a global containing an empty string
# The identifier capatilize is a global containing a "y" if the
# should be capitalized.
# The identifier yes must be a global containing "y".

   local char1, firstpart

   char1 := firstpart := nul
   outstr2 ? {
```

```
      firstpart := tab(upto(letters))              # The citatio
      if capitalize == yes
         then char1 := map(move(1),&lcase,&ucase)  # Capitalize.
         else char1 := move(1)                     # Don't capit
      return( firstpart || char1 || tab(0) )       # Reassemble.
   }
end

procedure def_proc(linelist,i)
# This is the default procedure.  It is called when a semantic r
# needs no special treatment.

   write(diags1,&level," Entering def: ",outstring(linelist)," i
   return copy([linelist[i][1],[linelist[i][2][1],linelist[i][2]
end

procedure error1(linelist,i)
# This procedure handles undefined morphological tags (i.e. part

   write(diags1,&level," Entering error1: ",outstring(linelist),
   write(diags1,">>>",linelist[i][2][1]," is not a valid part of
   return copy([linelist[i][1],[linelist[i][2][1],linelist[i][2]
end

procedure noun_proc(linelist,i)
# This procedure handles nouns.  It removes the article from a n
# it hasn't already been removed by art_proc().

   local article

   write(diags1,&level," Entering noun_proc: ",outstring(linelis
      write(diags1,&level,spc,i," Noun_proc: ",linelist[i][2][2]
      linelist[i][2][2] ? {
         article := tab(upto(sp))
         write(diags1,&level," Article: ",article)
         tab(many(sp))
         linelist[i][2][2] := tab(0)        # Drop the article.
         write(diags1,&level,spc,i," Noun: ",linelist[i][2][2])
      }
   return copy([linelist[i][1],[linelist[i][2][1],linelist[i][2]
end

procedure order(linelist2)
# This language specific procedure corrects the word order of th
# translated text.

   local linelist3
   write(diags1,&level," Entering order: ",outstring(linelist2))

   linelist3 := []
   j := 1
   lenlinelist := *linelist2
```

```
   while j <= lenlinelist do {          # Look for words that ne
      case linelist2[j][2][1] of {
         (adj2 |
          adj2n)        : { r := adj_move(copy(linelist2),
                                  j,lenlinelist)     # Move an
                            j := r[1]   # Pointer to the last el
                            linelist3 |||:= r[2]       # The pro
                          }
         (v_fut           |
          v_fut_perf      |
          v_impf          |
          v_impf_subj     |
          v_past          |
          v_past_perf     |
          v_past_perf_subj|
          v_perf          |
          v_pres          |
          v_pres_perf     |
          v_pres_subj     |
          v_subj          |
          ve_fut          |
          ve_past         |
          ve_past_perf    |
          ve_pres         |
          ve_pres_perf    |
          ve_pres_perf_subj|
          vs_past         |
          vs_past_perf    |
          vs_pres         ) : { r := v_move(copy(linelist2),
                                     j,lenlinelist)     # Move
                                j := r[1]   # Pointer to the l
                                           # processed
                                linelist3 |||:= r[2]    # The
                                                        # elem
                              }
         default        : put(linelist3,copy(linelist2[j]))  # D
      }
   j +:= 1
   }
   write(diags1,&level," Exiting order: ",outstring(linelist3))
   return linelist3
end

procedure outstring(linelist2)
# This procedure builds up a string from the list which is its a
# Certain adjustments are made to the string.  For instance, any
# is transferred from the source language part of the list to th
# language output.  Underscores are converted to spaces, and a l
# specific procedure to perform phonological adjustments is call
# The first letter of the output string is capitalized if the va
# capitalize contains "y".
```

```
# The identifier spc must be a global containing a space.
# The identifier underscore must be a global containing an under
# The identifier pun must be a global Cset containing punctuatio
# The identifier letters must be a global Cset containing the up
# lower case alphabet.
# The identifier nul must be a global containing an empty string
# The identifier capatilize is a global containing a "y" if the
# for this line began with a capital letter.

   local linelement, outstr, pun1, pun2

   linelement := outstr := nul
   every i := 1 to *linelist2 do {
      pun1 := pun2 := nul
      linelement := copy(linelist2[i][2][2])  # Target language
      linelist2[i][1] ? {                     # English word(s)
         pun1 := tab(many(pun))               # Get any initial
         tab(upto(pun)) & ( pun2 := tab(many(pun)) )   # Get tra
         linelement := pun1 || linelement || pun2  # Append the
      }
      if linelement ~== nul
         then outstr ||:= spc || linelement  # Build up the line
   }
   outstr := map(outstr[2:0],underscore,spc)  # Change underscor
   return wrap(capital(phono_adj(outstr)),59) # Adjust phonology
end

procedure phono_adj(outstr3)
# This procedure is specific to Spanish.  It makes phonological
# like the contraction of "de el" and "a el".
# It also adjusts "le" and "les" when they precede "la".

# The identifier nul must be a global containing an empty string
# The identifier de_el2 must be a global containing " de el ".
# The identifier del2 must be a global containing " del ".
# The identifier a_el2 must be a global containing " a el ".
# The identifier al2 must be a global containing " al ".
# The identifier le2 must be a global containing " le ".
# The identifier le_la2 must be a global containing " le la ".
# The identifier le_las2 must be a global containing " le las ".
# The identifier les_la2 must be a global containing " les la ".
# The identifier les_las2 must be a global containing " les las
# The identifier les2 must be a global containing " les ".
# The identifier la2 must be a global containing " la ".
# The identifier las2 must be a global containing " las ".
# The identifier se2 must be a global containing " se ".
# The identifier se_la2 must be a global containing " se la ".
# The identifier se_las2 must be a global containing " se las ".

   local outstr2, firstpart

   outstr2 := firstpart := nul
```

```
outstr3 ? {                             # Scan for "de el" and
   while ( firstpart := tab(find(de_el2)) ) do {
      move(7)
      outstr2 ||:= firstpart || del2
   }
   outstr2 ||:= tab(0)
}
outstr3 := outstr2
outstr2 := firstpart := nul
outstr3 ? {                             # Scan for "a el" and c
   while ( firstpart := tab(find(a_el2)) ) do {
      move(6)
      outstr2 ||:= firstpart || al2
   }
   outstr2 ||:= tab(0)
}
outstr3 := outstr2
outstr2 := firstpart := nul
outstr3 ? {                             # Scan for "le la" and
   while ( firstpart := tab(find(le_la2)) ) do {
      move(7)
      outstr2 ||:= firstpart || se_la2
   }
   outstr2 ||:= tab(0)
}
outstr3 := outstr2
outstr2 := firstpart := nul
outstr3 ? {                             # Scan for "le las" and
   while ( firstpart := tab(find(le_las2)) ) do {
      move(8)
      outstr2 ||:= firstpart || se_las2
   }
   outstr2 ||:= tab(0)
}
outstr3 := outstr2
outstr2 := firstpart := nul
outstr3 ? {                             # Scan for "les la" and
   while ( firstpart := tab(find(les_la2)) ) do {
      move(8)
      outstr2 ||:= firstpart || se_la2
   }
   outstr2 ||:= tab(0)
}
outstr3 := outstr2
outstr2 := firstpart := nul
outstr3 ? {                             # Scan for "les las" an
   while ( firstpart := tab(find(les_las2)) ) do {
      move(9)
      outstr2 ||:= firstpart || se_las2
   }
   outstr2 ||:= tab(0)
}
```

```
   return outstr2
end

procedure prep_proc(linelist,i)
# This procedure handles prepositions.

   write(diags1,&level," Entering proc: ",outstring(linelist),"
   return copy([linelist[i][1],[linelist[i][2][1],linelist[i][2]
end

procedure pro_proc(linelist,i)
# This procedure handles pronouns.

   write(diags1,&level," Entering pro: ",outstring(linelist)," i
   return copy([linelist[i][1],[linelist[i][2][1],linelist[i][2]
end

procedure rel_pro_proc(linelist,i)
# This procedure handles relative pronouns.

   write(diags1,&level," Entering rel_pro: ",outstring(linelist)
   return copy([linelist[i][1],[linelist[i][2][1],linelist[i][2]
end

procedure segment1(linelist)
# This is a recursive procedure which divides a line of source t
# segments.  The entire line is considered to be a segment, and
# segments are delimited in the source line by braces.  When a s
# no longer be subdivided, it is processed by the procedure tran
# The input is a list, and the returned value is a list.

   local i, j, l, newbegin, newend, newlist, r, seg1, sublst1

   write(diags1,&level," Entering segment1: ",outstring(linelist

# Perform initializations.
   l := 0
   r := 0
   i := 1
   sublst1 := []
   newbegin := 1
   newend := 1

# Process everything preceeding the first left brace, and transl
   while i <= *linelist do {
      linelist[i][1] ? {
         while ( tab(upto(lb)) & move(1) ) do {
            l +:= 1
            if l = 1 then {
               if i > newbegin then {
                  newend := i
                  sublst1 |||:= tran(linelist[newbegin:newend])
```

```
                  newbegin := newend
                  write(diags1,&level," Top: ",outstring(sublst1
               }
            }
         }
      }

# Process (ie. further segment) everything enclosed in a balance
# of braces, but don't translate it.
      linelist[i][1] ? {
         while ( tab(upto(rb)) & move(1) ) do {
            r +:= 1
            if ( (l > 0) & (l = r) ) then {
               newend := i + 1
               newlist := linelist[newbegin:newend]
               newlist[1][1] := newlist[1][1][2:0]
               write(diags1,&level," newlist[1][1] ",newlist[1][
               j := *newlist
               newlist[j][1] := newlist[j][1][1:-1]
               write(diags1,&level," newlist[",j,"][1] ",newlist
               sublst1 |||:= segment1(newlist)
               write(diags1,&level," Bot: ",outstring(sublst1))
               l := 0
               r := 0
               newbegin := newend
               write(diags1,&level," l:",l," r:",r," newbegin/en
            }
         }
      }
   i +:= 1
   write(diags1,&level," i: ",i)
   }
   if l ~= r then write(diags1,">>>Unbalanced braces in ",citati

# Translate text to the right of the last right brace, and then
   if *sublst1 > 0
      then  {
         newend := *linelist + 1
         if newbegin < newend then {
            write(diags1,&level," There was text to the right of
            write(diags1,&level," newbegin: ",newbegin," newend:
            sublst1 |||:= tran(linelist[newbegin:newend])
         }
         write(diags1,&level," Exiting segment1 at braces exit:
            outstring(sublst1))
         return sublst1
      }

# Translate text in a line containing no braces, and then return
      else {
         write(diags1,&level," Exiting segment1 at no-braces exi
            outstring(linelist))
```

```
         return tran(copy(linelist))
      }
end

procedure semget(semtag)
# This procedure retrieves a line from a semanticon and puts
# the components of that line into a list structure.  The argume
# semtag is the key which is used to retreive the semanticon lin

# Structure of the list:
# Lines to be translated are kept in a list structure.
# Each list element is itself a list consisting of a semantic ta
# followed by another list which is the semanticon entry
# corresponding to the semantic tag.  Here is a sample list
# element:
#
#                              |---- Morphological tag
#                              |
#                              |      |----- Target language sense
# Semantic tag -----|          |      |
#                   |          |      |
#               ["feast1",["n","la fiesta"]].
#
# A list of these elements constitutes a line to be translated.

# This version of semget requires that the semanticon be in a
# table called semtable.  The table semtable must be global.

# It would be possible to write a version of semget which retrie
# semanticon lines from a file on the disk.  This would accomoda
# users of machines with limited memory, but processing speed wo
# suffer.

# The identifier ucp must be a global string containing upper ca
# plus punctuation characters.
# The identifier lcs must be a global string containing lower ca
# plus space characters.

   local semtablelist, semtag1

# Process the semantic tag (the key field of the semanticon).
   map(semtag,ucp,lcs) ? {          # Map uppercase letters to lo
      tab(many(sp))                 # Trim any leading spaces.
      semtag1 := ( tab(upto(sp)) | tab(0) )  # Get the semantic
   }

# Get the entry from the semanticon.
   semtablelist := semtable[semtag1]
   if (*semtablelist[1] > 0)
      then return copy(semtablelist)  # Return the semanticon en
      else {
         write(&errout,">>>",semtag1," is not in the semanticon.
```

```
         write(diags1,&level,">>>",semtag1," is not in the seman
         return copy([0,0])
      }
end

procedure semread(main_args)
# This procedure reads all the lines from a semanticon and check
# them for correct syntax.  The semanticon lines are converted t
# lists, and then the lists are put into the table semtable.
# The table semtable is indexed by semtag which is the first fie
# in each line of the semanticon.

# It would be possible to write a version of semget which retrie
# semanticon lines from a file on the disk.  This would accomoda
# users of machines with limited memory at the expense of proces
# speed.  In this case semread would only need to build an index
# disk based semanticon.  The index could be stored as a binary
# to facilitate fast access to the disk file via the seek() func

# This procedure requires a global identifier spc which contains
# This procedure requires a global identifier sp which contains
# The global identifier nul must contain an empty string.

   local semline, targetwds, semtag, morphtag, art1, listelement

   file_name := main_args[1] || "sem-es.dat"
   if not (semanticon := open(file_name,"r"))           # Semant
      then stop("Cannot open ",file_name)
   semtable := table(nul)
   write(&errout,"Beginning the semanticon read on:   ",&datelin
   while text := trim(read(semanticon)) do {
      art1 := targetwds := nul
      text ? {
         semtag := tab(upto(sp))
         tab(many(sp))
         morphtag := tab(upto(sp))
         tab(many(sp))
         if morphtag == noun              # Is it a noun?
            then { art1 := tab(upto(sp))
                   tab(many(sp))
                   targetwds := trim(tab(0))
                   listelement := art1 || spc || targetwds
            }
            else listelement := targetwds := trim(tab(0))
      }
      if targetwds == nul
         then write(&errout,">>>Something is wrong with: ",text)
         else {
            semline := [morphtag,listelement]
            semtable[semtag] := semline
         }
   }
```

```
   write(&errout,"The semanticon read is complete on: ",&datelin
   close(semanticon)
   return semtable
end

procedure semrepget(semrepin)
# This procedure reads and accumulates lines of the semantic rep
# of the English input text.  It reads until it encounters a bla
# It then returns the accumulated semantic representation.  This
# represent a single sentence.

# This procedure requires a global identifier spc which contains
# This procedure requires a global identifier nul which contains

     local line, lineout
     lineout := line := nul
     while line := trim(read(semrepin)) do {
        if match(equalsign,line) then return line      # Paragra
        if ( (line == nul) & (*lineout > 0) )
           then return lineout[2:0]
           else line ? { tab(many(sp))
                         lineout ||:= ( spc || tab(0) )
           }
     }
     if *lineout > 0
        then return lineout[2:0]
        else fail
end

procedure tran(linelist)
# This procedure accepts a semantic representation of a line of
# translates it into a target language.
# All the identifiers containing morphological tags are globals.

   local i, linelist2, linelist3
   write(diags1,&level," Entering tran: ",outstring(linelist))
   linelist2 := []
   every i := 1 to *linelist do {  # Process based on
      case linelist[i][2][1] of {  #   morphological tags.
         adj               : put(linelist2,def_proc(linelist,i))
         adj1              : put(linelist2,agree(linelist,i))
         adj1n             : put(linelist2,agree(linelist,i))
         adj2              : put(linelist2,agree(linelist,i))
         adj2n             : put(linelist2,agree(linelist,i))
         adv               : put(linelist2,def_proc(linelist,i))
         art_def           : put(linelist2,art_proc(linelist,i))
         art_ind           : put(linelist2,agree(linelist,i))
         com               : put(linelist2,def_proc(linelist,i))
         conj              : put(linelist2,def_proc(linelist,i))
         conj1             : put(linelist2,def_proc(linelist,i))
         dem_fs            : put(linelist2,agree(linelist,i))
         dem_fp            : put(linelist2,agree(linelist,i))
```

```
dem_ms             : put(linelist2,agree(linelist,i))
dem_mp             : put(linelist2,agree(linelist,i))
noun               : put(linelist2,noun_proc(linelist,i)
n_mass             : put(linelist2,noun_proc(linelist,i)
neg                : put(linelist2,def_proc(linelist,i))
np                 : put(linelist2,noun_proc(linelist,i)
num                : put(linelist2,def_proc(linelist,i))
prep               : put(linelist2,prep_proc(linelist,i)
prep_phra          : put(linelist2,prep_proc(linelist,i)
pro_dio_3p         : put(linelist2,pro_proc(linelist,i))
pro_dio_3p_pol     : put(linelist2,pro_proc(linelist,i))
pro_dio_3s_pol     : put(linelist2,pro_proc(linelist,i))
pro_dio_3sf        : put(linelist2,pro_proc(linelist,i))
pro_dio_3sm        : put(linelist2,pro_proc(linelist,i))
pro_do_1p          : put(linelist2,pro_proc(linelist,i))
pro_do_1s          : put(linelist2,pro_proc(linelist,i))
pro_do_2p          : put(linelist2,pro_proc(linelist,i))
pro_do_2pm_pol     : put(linelist2,pro_proc(linelist,i))
pro_do_2s          : put(linelist2,pro_proc(linelist,i))
pro_do_3pf         : put(linelist2,pro_proc(linelist,i))
pro_do_3pf_pol     : put(linelist2,pro_proc(linelist,i))
pro_do_3pm         : put(linelist2,pro_proc(linelist,i))
pro_do_3pm_pol     : put(linelist2,pro_proc(linelist,i))
pro_do_3sf         : put(linelist2,pro_proc(linelist,i))
pro_do_3sm         : put(linelist2,pro_proc(linelist,i))
pro_do_3sn         : put(linelist2,pro_proc(linelist,i))
pro_io_1p          : put(linelist2,pro_proc(linelist,i))
pro_io_1s          : put(linelist2,pro_proc(linelist,i))
pro_io_2p          : put(linelist2,pro_proc(linelist,i))
pro_io_2p_pol      : put(linelist2,pro_proc(linelist,i))
pro_io_2pm         : put(linelist2,pro_proc(linelist,i))
pro_io_2s          : put(linelist2,pro_proc(linelist,i))
pro_io_2s_pol      : put(linelist2,pro_proc(linelist,i))
pro_io_3p          : put(linelist2,pro_proc(linelist,i))
pro_io_3p_pol      : put(linelist2,pro_proc(linelist,i))
pro_io_3s_pol      : put(linelist2,pro_proc(linelist,i))
pro_io_3sf         : put(linelist2,pro_proc(linelist,i))
pro_io_3sm         : put(linelist2,pro_proc(linelist,i))
pro_op_1pf         : put(linelist2,pro_proc(linelist,i))
pro_op_1pm         : put(linelist2,pro_proc(linelist,i))
pro_op_1s          : put(linelist2,pro_proc(linelist,i))
pro_op_2p          : put(linelist2,pro_proc(linelist,i))
pro_op_2p_pol      : put(linelist2,pro_proc(linelist,i))
pro_op_2pf         : put(linelist2,pro_proc(linelist,i))
pro_op_2pm         : put(linelist2,pro_proc(linelist,i))
pro_op_2s          : put(linelist2,pro_proc(linelist,i))
pro_op_2s_pol      : put(linelist2,pro_proc(linelist,i))
pro_op_3p          : put(linelist2,pro_proc(linelist,i))
pro_op_3p_refl     : put(linelist2,pro_proc(linelist,i))
pro_op_3pf         : put(linelist2,pro_proc(linelist,i))
pro_op_3pf_pol     : put(linelist2,pro_proc(linelist,i))
pro_op_3pm         : put(linelist2,pro_proc(linelist,i))
```

```
    pro_op_3s          : put(linelist2,pro_proc(linelist,i))
    pro_op_3s_refl     : put(linelist2,pro_proc(linelist,i))
    pro_op_3sf         : put(linelist2,pro_proc(linelist,i))
    pro_op_3sf_refl    : put(linelist2,pro_proc(linelist,i))
    pro_op_3sm         : put(linelist2,pro_proc(linelist,i))
    pro_op_3sm_refl    : put(linelist2,pro_proc(linelist,i))
    pro_poss_1s        : put(linelist2,agree(linelist,i))
    pro_poss_2s        : put(linelist2,agree(linelist,i))
    pro_poss_3ms       : put(linelist2,agree(linelist,i))
    pro_poss_3p        : put(linelist2,agree(linelist,i))
    pro_refl_3p        : put(linelist2,pro_proc(linelist,i))
    pro_rel            : put(linelist2,rel_pro_proc(linelist
    pro_su_1pf         : put(linelist2,pro_proc(linelist,i))
    pro_su_1pm         : put(linelist2,pro_proc(linelist,i))
    pro_su_1s          : put(linelist2,pro_proc(linelist,i))
    pro_su_2p          : put(linelist2,pro_proc(linelist,i))
    pro_su_2p_pol      : put(linelist2,pro_proc(linelist,i))
    pro_su_2s          : put(linelist2,pro_proc(linelist,i))
    pro_su_2s_pol      : put(linelist2,pro_proc(linelist,i))
    pro_su_3pf         : put(linelist2,pro_proc(linelist,i))
    pro_su_3pf_pol     : put(linelist2,pro_proc(linelist,i))
    pro_su_3pm         : put(linelist2,pro_proc(linelist,i))
    pro_su_3s          : put(linelist2,pro_proc(linelist,i))
    sent_adv           : put(linelist2,def_proc(linelist,i))
    ve_past            : put(linelist2,def_proc(linelist,i))
    v_fut              : put(linelist2,def_proc(linelist,i))
    v_fut_perf         : put(linelist2,def_proc(linelist,i))
    v_impf             : put(linelist2,def_proc(linelist,i))
    v_impf_subj        : put(linelist2,def_proc(linelist,i))
    v_inf              : put(linelist2,def_proc(linelist,i))
    v_past             : put(linelist2,def_proc(linelist,i))
    v_past_part        : put(linelist2,def_proc(linelist,i))
    v_past_perf        : put(linelist2,def_proc(linelist,i))
    v_past_perf_subj   : put(linelist2,def_proc(linelist,i))
    v_past_prog        : put(linelist2,def_proc(linelist,i))
    v_perf             : put(linelist2,def_proc(linelist,i))
    v_pres             : put(linelist2,def_proc(linelist,i))
    v_pres_impv        : put(linelist2,def_proc(linelist,i))
    v_pres_perf        : put(linelist2,def_proc(linelist,i))
    v_pres_prog        : put(linelist2,def_proc(linelist,i))
    v_pres_subj        : put(linelist2,def_proc(linelist,i))
    v_subj             : put(linelist2,def_proc(linelist,i))
    ve_fut             : put(linelist2,def_proc(linelist,i))
    ve_past            : put(linelist2,def_proc(linelist,i))
    ve_past_perf       : put(linelist2,def_proc(linelist,i))
    ve_pres            : put(linelist2,def_proc(linelist,i))
    ve_pres_perf_subj  : put(linelist2,def_proc(linelist,i))
    vs_past            : put(linelist2,def_proc(linelist,i))
    vs_past_perf       : put(linelist2,def_proc(linelist,i))
    vs_pres            : put(linelist2,def_proc(linelist,i))
    default            : put(linelist2,error1(linelist,i))
}
```

```
   }
   linelist3 := []
   linelist3 := order(linelist2)       # Correct word order.
   write(diags1,&level," Exiting tran: ",outstring(linelist3))
   return (linelist3)
end

procedure v_move(linelist2,j,lenlinelist)
# This language specific procedure moves verbs to the position f
# direct and indirect object pronouns and negators.  It also adj
# punctuation as necessary.

# The identifier pun must be a global Cset containing punctuatio
# The identifier nul must be a global containing an empty string

   local templist, templist2, k, pun1, pun2, tmp1, tmp2

   write(diags1,&level," Entering v_move: j = ",j," ",outstring(
   pun1 := pun2 := nul
   templist  := []
   templist2 := []
   linelist2[j][1] ? {
      pun1 := tab(many(pun))              # Get any initial punc
      linelist2[j][1] := tab(0)
   }
   write(diags1,&level," v_move initial punctuation: j = ",j," p
   while j <= lenlinelist do {
      case linelist2[j][2][1] of {
         (v_fut               |
          v_fut_perf          |
          v_impf              |
          v_impf_subj         |
          v_past              |
          v_past_perf         |
          v_past_perf_subj    |
          v_perf              |
          v_pres              |
          v_pres_perf         |
          v_pres_subj         |
          v_subj              |
          ve_fut              |
          ve_past             |
          ve_past_perf        |
          ve_pres             |
          ve_pres_perf_subj   |
          vs_past             |
          vs_past_perf        |
          vs_pres)            : put(templist2,copy(linelist2[j])

         (neg                 |
         pro_dio_3p           |
```

```
            pro_dio_3p_pol  |
            pro_dio_3s_pol  |
            pro_dio_3sf     |
            pro_dio_3sm     |
            pro_do_1p       |
            pro_do_1s       |
            pro_do_2p       |
            pro_do_2pm_pol  |
            pro_do_2s       |
            pro_do_3pf      |
            pro_do_3pf_pol  |
            pro_do_3pm      |
            pro_do_3pm_pol  |
            pro_do_3sf      |
            pro_do_3sm      |
            pro_do_3sn      |
            pro_io_1p       |
            pro_io_1s       |
            pro_io_2p       |
            pro_io_2p_pol   |
            pro_io_2pm      |
            pro_io_2s       |
            pro_io_2s_pol   |
            pro_io_3p       |
            pro_io_3p_pol   |
            pro_io_3s_pol   |
            pro_io_3sf      |
            pro_io_3sm      |
            pro_refl_3p)     : put(templist,copy(linelist2[j]))
            default          : break
         }
      j +:= 1
      }
      j -:= 1
      templist[1][1] := pun1 || templist[1][1]   # Append initial p
      k := *templist
      templist[k][1] ? {
         tmp1 := (tab(many(pun)) | nul)    # Leading punctuation
         tmp2 := (tab(upto(pun)) | tab(0)) # Semantic tag
         pun2 := (tab(many(pun)) | nul)    # Trailing punctuation
      }
      templist[k][1] := tmp1 || tmp2       # Remove trailing punctu
      write(diags1,&level," v_move final punctuation: j = ",j," pun
      templist |||:= templist2             # Append the verb to the
      templist[*templist][1] ||:= pun2     # Append trailing punctu
      write(diags1,&level," Exiting v_move: j = ",j," ",outstring(t
      return [j,copy(templist)]
end
```

```
procedure wrap(line,maxlen)
# This procedure breaks a line of text up into multiple lines of

# The global nul contains an empty string
# The global sp is a Cset containing a space.
# The global spaces is a string containing spaces.
# The global citation contains the citation for the line being f

   local citation_len, indent, lineout, lineout2, sp1, word

   indent := citation_len := 0
   citation_len := *citation + 2            # Amount to indent s
   indent := spaces[1:citation_len]         # Spaces to indent s

   lineout := lineout2 := sp1 := word := nul
   line ? {
      repeat {
         word := ( tab(upto(sp)) | tab(0) )
         sp1  := ( tab(many(sp)) | nul )
         if ( (*lineout + *word) > maxlen )
            then {
               lineout2 ||:= lineout || "\n"
               lineout := indent
            }
         lineout ||:= word || sp1
         if (sp1 == nul) then return (lineout2 || lineout)
      }
   }
end
```