Selection and Use of General-Purpose Programming Languages — Program Examples
The National Bureau of Standards was established by an act of Congress on March 3, 1901. The Bureau's overall goal is to strengthen and advance the nation's science and technology and facilitate their effective application for public benefit. To this end, the Bureau conducts research and provides: (1) a basis for the nation's physical measurement system, (2) scientific and technological services for industry and government, (3) a technical basis for equity in trade, and (4) technical services to promote public safety. The Bureau's technical work is performed by the National Measurement Laboratory, the National Engineering Laboratory, the Institute for Computer Sciences and Technology, and the Center for Materials Science.

The National Measurement Laboratory

Provides the national system of physical and chemical measurement; coordinates the system with measurement systems of other nations and furnishes essential services leading to accurate and uniform physical and chemical measurement throughout the Nation's scientific community, industry, and commerce; provides advisory and research services to other Government agencies; conducts physical and chemical research; develops, produces, and distributes Standard Reference Materials; and provides calibration services. The Laboratory consists of the following centers:

- Basic Standards
- Radiation Research
- Chemical Physics
- Analytical Chemistry

The National Engineering Laboratory

Provides technology and technical services to the public and private sectors to address national needs and to solve national problems; conducts research in engineering and applied science in support of these efforts; builds and maintains competence in the necessary disciplines required to carry out this research and technical service; develops engineering data and measurement capabilities; provides engineering measurement traceability services; develops test methods and proposes engineering standards and code changes; develops and proposes new engineering practices; and develops and improves mechanisms to transfer results of its research to the ultimate user. The Laboratory consists of the following centers:

- Applied Mathematics
- Electronics and Electrical Engineering
- Manufacturing Engineering
- Building Technology
- Fire Research
- Chemical Engineering

The Institute for Computer Sciences and Technology

Conducts research and provides scientific and technical services to aid Federal agencies in the selection, acquisition, application, and use of computer technology to improve effectiveness and economy in Government operations in accordance with Public Law 89-306 (40 U.S.C. 759), relevant Executive Orders, and other directives; carries out this mission by managing the Federal Information Processing Standards Program, developing Federal ADP standards guidelines, and managing Federal participation in ADP voluntary standardization activities; provides scientific and technological advisory services and assistance to Federal agencies; and provides the technical foundation for computer-related policies of the Federal Government. The Institute consists of the following centers:

- Programming Science and Technology
- Computer Systems Engineering

The Center for Materials Science

Conducts research and provides measurements, data, standards, reference materials, quantitative understanding and other technical information fundamental to the processing, structure, properties and performance of materials; addresses the scientific basis for new advanced materials technologies; plans research around cross-country scientific themes such as nondestructive evaluation and phase diagram development; oversees Bureau-wide technical programs in nuclear reactor radiation research and nondestructive evaluation; and broadly disseminates generic technical information resulting from its programs. The Center consists of the following Divisions:

- Inorganic Materials
- Fracture and Deformation
- Polymers
- Metallurgy
- Reactor Radiation

1 Headquarters and Laboratories at Gaithersburg, MD, unless otherwise noted; mailing address Gaithersburg, MD 20899.
2 Some divisions within the center are located at Boulder, CO 80303.
3 Located at Boulder, CO, with some elements at Gaithersburg, MD.
Computer Science and Technology

NBS Special Publication 500-117, Volume 2
Selection and Use of General-Purpose Programming Languages — Program Examples

John V. Cugini

Center for Programming Science and Technology
Institute for Computer Sciences and Technology
National Bureau of Standards
Gaithersburg, MD 20899
Reports on Computer Science and Technology

The National Bureau of Standards has a special responsibility within the Federal Government for computer science and technology activities. The programs of the NBS Institute for Computer Sciences and Technology are designed to provide ADP standards, guidelines, and technical advisory services to improve the effectiveness of computer utilization in the Federal sector, and to perform appropriate research and development efforts as foundation for such activities and programs. This publication series will report these NBS efforts to the Federal computer community as well as to interested specialists in the academic and private sectors. Those wishing to receive notices of publications in this series should complete and return the form at the end of this publication.

Library of Congress Catalog Card Number: 84-601120

National Bureau of Standards Special Publication 500-117, Volume 2
CODEN: XNBSAV

U.S. GOVERNMENT PRINTING OFFICE
WASHINGTON: 1984

Selection and Use of General-Purpose Programming Languages
Volume 2 - Program Examples

John V. Cugini
Institute for Computer Sciences and Technology
National Bureau of Standards

ABSTRACT

Programming languages have been and will continue to be an important instrument for the automation of a wide variety of functions within industry and the Federal Government. Other instruments, such as program generators, application packages, query languages, and the like, are also available and their use is preferable in some circumstances.

Given that conventional programming is the appropriate technique for a particular application, the choice among the various languages becomes an important issue. There are a great number of selection criteria, not all of which depend directly on the language itself. Broadly speaking, the criteria are based on 1) the language and its implementation, 2) the application to be programmed, and 3) the user's existing facilities and software.

This study presents a survey of selection factors for the major general-purpose languages: Ada*, BASIC, C, COBOL, FORTRAN, Pascal, and PL/I. The factors covered include not only the logical operations within each language, but also the advantages and disadvantages stemming from the current computing environment, e.g., software packages, microcomputers, and standards. The criteria associated with the application and the user's facilities are explained. Finally, there is a set of program examples to illustrate the features of the various languages.

This volume includes the program examples. Volume 1 contains the discussion of language selection criteria.

Key words: Ada; alternatives to programming; BASIC; C; COBOL; FORTRAN; Pascal; PL/I; programming language features; programming languages; selection of programming language.

* Ada is a registered trademark of the U. S. Government, Ada Joint Project Office.
TABLE OF CONTENTS: Volume 2 - Program Examples

1.0 INTRODUCTION .................................................. 1
2.0 ADA .......................................................... 9
3.0 BASIC ......................................................... 33
4.0 C ............................................................. 52
5.0 COBOL ........................................................ 74
6.0 FORTRAN ...................................................... 99
7.0 PASCAL ......................................................... 135
8.0 PL/I ............................................................ 155

FIGURES:

Figure 1 - Algorithm for Program Examples. ............... 2
Figure 2 - Input Data. ........................................... 3
Figure 3 - Queries and Output. ................................. 4
1.0 INTRODUCTION

In this volume, we shall illustrate the general style of each of the languages with a program. These programs are only examples; they do not attempt to demonstrate the full capability of each language. On the other hand, the application chosen is complex enough that the programs do make significant use of several important language features, such as reading a file, interacting with a user, recursion, data abstraction, manipulation of arrays, pointers, and character strings, and some numeric calculation. Of particular note are the language features for modularizing a program of moderate size (about 1000 lines). While no application can be completely language-neutral, this variety of requirements implies a relatively unbiased example. Finally, the application deals with a well-known realm (family relationships) in order to facilitate understanding of the programs.

All of the programs solve the same problem, i.e., they accept the same input and produce output as nearly equivalent as possible. The input is a file of people, one person per record, and a series of user queries. In the file, each person's father and mother (if known), and spouse (if any) are identified. Given this information, the user may then specify any two persons in the file, and the program computes and displays the relationship (e.g., brother-in-law, second cousin) between those two. Also, based on the number and degree of common ancestors, the expected value for the proportion of common genetic material between the two is computed and displayed.

The algorithms and data structures employed are roughly equivalent, but differ in detail owing to the language differences being illustrated. Generally, user-defined names are capitalized and language-defined keywords and identifiers are written in lower-case. In all the programs a directed graph is simulated, with the vertices representing people and the edges representing different types of direct relationships. The only direct relationships are parent, child, and spouse. Starting at one vertex, a search is conducted to find the shortest path to the other vertex. The types of edges encountered along the path, together with some additional information, determine the relationship. For instance, if the shortest path between X1 and X4 is that X1 is child of X2, X2 is spouse of X3, and X3 is parent of X4, this would show that X1 and X4 are step-siblings. It is assumed that the input file has already been validated and is correct. The user's requests, however, are checked. The algorithm to determine the shortest path is adapted from [Baas78]. The overall algorithm is expressed by the pseudo-code below.

All of the programs, except the one in BASIC, have compiled and executed on at least one language processor which implements the corresponding standard or base document. The COBOL program, while conforming to both COBOL-74 and COBOL-8x, is essentially a COBOL-74 program, since it does not exploit any of the new COBOL-8x features.
Figure 1 - Algorithm for Program Examples

for each record in input PEOPLE file do
    establish entry in PERSON array
    for all previous entries do
        compare this entry to previous, looking for
        immediate relationships: parent, child, or spouse
        if relationship found
            establish link (edge) between these two persons
        end if
    end for
end for

graph is now built

while not request to stop
    prompt and read next request
    exit while-block if request to stop
    if syntax of request OK
        search for requested persons
        if exactly one of each person found
            if 1st person = 2nd person
                display "identical to self"
            else
                find shortest path between the two persons
                if no such path
                    display "unrelated"
                else
                    analyze path for named relationships:
                        path initially composed of parent, child,
                        spouse edges
                        resolve child-parent and child-spouse-parent
                        to sibling
                        resolve child-child-... and parent-parent-...
                        to descendant (child*) or ancestor (parent*)
                        resolve child*-sibling-parent* to cousin,
                        child*-sibling to nephew,
                        sibling-parent* to uncle
                    display consolidated relationships
                    compute proportion of common genetic material:
                        traverse ancestors of person1, zeroing out
                        traverse ancestors of person1, marking and
                        accumulating genetic contribution
                        traverse ancestors of person2, accumulating
                        overlap with person1
                    display results
                end if
            end if
        else
            display "duplicate name" or "not found"
        end if
    else
        display "invalid request"
    end if
end while
display "done"
Figure 2 - Input Data

This figure shows some of the input data with which the program examples were tested. The format of each record is:

<table>
<thead>
<tr>
<th>Position</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-20</td>
<td>Name of person</td>
</tr>
<tr>
<td>21-23</td>
<td>Unique 3-digit identifier of person</td>
</tr>
<tr>
<td>24</td>
<td>Gender of person</td>
</tr>
<tr>
<td>25-27</td>
<td>Identifier of father (000 if unknown)</td>
</tr>
<tr>
<td>28-30</td>
<td>Identifier of mother (000 if unknown)</td>
</tr>
<tr>
<td>31-33</td>
<td>Identifier of spouse (000 if none or unknown)</td>
</tr>
</tbody>
</table>

Example of Input Data:

John Smith       001M000000002
Mary Smith       002F003000001
Wilbur Finnegan  010M000000011
Mary Finnegan    011F000000010
James Smith      020M001002022
Wilma Smith      022F010011020
Marvin Hamlisch  031M000032000
Melvin Hamlisch  033M000032000
Martha Hamlisch  032F048043034
Murgatroyd Whatsis 034M000000032
Bentley Whatsis  035M034036000
Myrna Whozat     036F000000000
Bosworth Whatsis 037M034036000
K48              048M000000043
K43              043F041042048
K41              041M000000042
K42              042F000000041
K46              046M045000000
K45              045M048043000
K47              047M044000000
K44              044M041042000
Velorus Davis    085M000000086
Goldie Beacon    083F085086082
Ross Beacon      082M000000083
Velma Davis      086F000000085
Floyd Davis      088M085084087
Cindy Davis      084F000000000
David Beacon     121M081120000
Norma Cousins    053F082083055
Carmine Cousins  051M000000052
Maria Cousins    052F000000051
James Cousins    054M051052000
C. John Cousins  055M051052053
John Cousins     073M055053074
Janet Cousins    074F140141073
Richard Cousins  077M073074000
Paul Cousins     078M073074000
Marie Cousins    079F073074000

* * * * * *
Figure 3 - Queries and Output

This figure gives some examples of the results of running the programs.

Enter two person-identifiers (name or number), separated by semicolon. Enter "stop" to stop.
;
Incorrect request format: null field preceding semicolon.

Enter two person-identifiers (name or number), separated by semicolon. Enter "stop" to stop.
x;x;x
Incorrect request format: must be exactly one semicolon.

Enter two person-identifiers (name or number), separated by semicolon. Enter "stop" to stop.
x
First person not found.
Second person not found.

Enter two person-identifiers (name or number), separated by semicolon. Enter "stop" to stop.
111 ; 111
Christopher Delmonte is identical to himself.

Enter two person-identifiers (name or number), separated by semicolon. Enter "stop" to stop.
G6;John Smith
G6
is not related to John Smith

Enter two person-identifiers (name or number), separated by semicolon. Enter "stop" to stop.
Carmine Cousins;111
Duplicate names for first person - use numeric identifier.

Enter two person-identifiers (name or number), separated by semicolon. Enter "stop" to stop.
163;145
Shortest path between identified persons:
Linda Lackluster is child of
Millie Lackluster is child of
Anna Pittypat is parent of
Margaret Madison is spouse of
Richard Madison is child of
Victoria Pisces is parent of
Maria Gotsocks is parent of
Elzbieta Gotsocks
Condensed path:
Linda Lackluster is niece of
Richard Madison is uncle of
Elzbieta Gotsocks
Proportion of common genetic material = 0.00000E+00
Figure 3 - Queries and Output (continued)

Enter two person-identifiers (name or number), separated by semicolon. Enter "stop" to stop.

094;145

Shortest path between identified persons:
Nancy Powers       is child of
Maxine Powers      is child of
Floyd Davis        is child of
Velorus Davis      is parent of
Goldie Beacon      is parent of
Norma Cousins      is parent of
John Cousins       is spouse of
Janet Cousins      is child of
Richard Madison    is child of
Victoria Pisces    is parent of
Maria Gotsocks     is parent of
Elzbieta Gotsocks
Condensed path:
Nancy Powers       2nd half-cousin-in-law of
Janet Cousins      cousin of
Elzbieta Gotsocks

Proportion of common genetic material = 0.00000E+00

Enter two person-identifiers (name or number), separated by semicolon. Enter "stop" to stop.

036;033

Shortest path between identified persons:
Myrna Whozat       parent of
Bentley Whatsis    is child of
Murgatroyd Whatsis spouse of
Martha Hamlisch    is parent of
Melvin Hamlisch    is parent of

Condensed path:
Myrna Whozat       mother of
Bentley Whatsis    is step-brother of
Melvin Hamlisch

Proportion of common genetic material = 0.00000E+00

Enter two person-identifiers (name or number), separated by semicolon. Enter "stop" to stop.

031;033

Shortest path between identified persons:
Marvin Hamlisch    child of
Martha Hamlisch    is parent of
Melvin Hamlisch    is parent of

Condensed path:
Marvin Hamlisch    half-brother of
Melvin Hamlisch

Proportion of common genetic material = 2.50000E-01
Figure 3 - Queries and Output (continued)

Enter two person-identifiers (name or number), separated by semicolon. Enter "stop" to stop.
145;090
Shortest path between identified persons:
Elzbieta Gotsocks is child of
Maria Gotsocks is child of
U. Pisces is parent of
Richard Madison is parent of
Janet Cousins is spouse of
John Cousins is child of
Norma Cousins is child of
Goldie Beacon is child of
Velorus Davis is parent of
Floyd Davis is parent of
Maxine Powers is spouse of
Tim Powers
Condensed path:
Elzbieta Gotsocks is cousin-in-law of
John Cousins is half-cousin-in-law once removed of
Tim Powers
Proportion of common genetic material = 0.00000OE+00

Enter two person-identifiers (name or number), separated by semicolon. Enter "stop" to stop.
L6;R9
Shortest path between identified persons:
L6 is child of
L5 is child of
L4 is child of
L3 is child of
L2 is child of
L1 is child of
L0 is parent of
R1 is parent of
R2 is parent of
R3 is parent of
R4 is parent of
R5 is parent of
R6 is parent of
R7 is parent of
R8 is parent of
R9
Condensed path:
L6 is 5th half-cousin 3 times removed of
R9
Proportion of common genetic material = 3.05176E-05
Figure 3 - Queries and Output (continued)

Enter two person-identifiers (name or number), separated by semicolon. Enter "stop" to stop.

W1;R14
Shortest path between identified persons:
W1    is spouse of
L0    is parent of
R1    is parent of
R2    is parent of
R3    is parent of
R4    is parent of
R5    is parent of
R6    is parent of
R7    is parent of
R8    is parent of
R9    is parent of
R10   is parent of
R11   is parent of
R12   is parent of
R13   is parent of
R14   is parent of

Condensed path:  W1    is great*12-grand-step-father of
                 R14

Proportion of common genetic material = 0.00000E+00

Enter two person-identifiers (name or number), separated by semicolon. Enter "stop" to stop.

X8;L6
Shortest path between identified persons:
X8    is child of
X7    is child of
X6    is child of
X5    is child of
X4    is child of
X3    is spouse of
R4    is child of
R3    is child of
R2    is child of
R1    is child of
L0    is parent of
L1    is parent of
L2    is parent of
L3    is parent of
L4    is parent of
L5    is parent of
L6    is parent of

Condensed path:  X8    is great*3-grand-step-son of
                 R4    is 3rd half-cousin 2 times removed of
                 L6

Proportion of common genetic material = 0.00000E+00
Figure 3 - Queries and Output (continued)

Enter two person-identifiers (name or number), separated by semicolon. Enter "stop" to stop.
G5;G6
Shortest path between identified persons:
G5 is parent of
G6
Condensed path:
G5 is mother of
G6
Proportion of common genetic material = 5.62500E-01

Enter two person-identifiers (name or number), separated by semicolon. Enter "stop" to stop.
stop
End of relation-finder.
package RELATION_TYPES_AND_DATA is

MAX_PERSONS : constant integer := 300;
NAME_LENGTH : constant integer := 20;
-- every PERSON has a unique 3-digit IDENTIFIER
IDENTIFIER_LENGTH : constant integer := 3;
BUFFER_LENGTH : constant integer := 60;

subtype NAME_RANGE is integer range 1..NAME_LENGTH;
subtype IDENTIFIER_RANGE is integer range 1..IDENTIFIER_LENGTH;
subtype BUFFER_RANGE is integer range 1..BUFFER_LENGTH;

subtype NAME_TYPE is string (NAME_RANGE);
subtype BUFFER_TYPE is string (BUFFER_RANGE);
subtype MESSAGE_TYPE is string (1..40);

subtype INDEX_TYPE is integer range 0..MAX_PERSONS;
subtype COUNTER is integer range 0..integer'last;
subtype DIGIT_TYPE is character range '0'..'9';

type REAL is digits 6;
type IDENTIFIER_TYPE is array (IDENTIFIER_RANGE) of DIGIT_TYPE;
-- each PERSON's record in the file identifies at most three
-- others directly related: father, mother, and spouse
type GIVEN_IDENTIFIERS is (FATHER_IDENT, MOTHER_IDENT, SPOUSE_IDENT);
type RELATIVE_ARRAY is array (GIVEN_IDENTIFIERS) of IDENTIFIER_TYPE;

NULL_IDENT : constant IDENTIFIER_TYPE := "000";
REQUEST_OK : constant MESSAGE_TYPE :=
"Request OK",
REQUEST_TO_STOP : constant BUFFER_TYPE :=
"stop ";

type GENDER_TYPE is (MALE, FEMALE);
type RELATION_TYPE is (PARENT, CHILD, SPOUSE, SIBLING, UNCLE,
               NEPHEW, COUSIN, NULL_RELATION);

-- directed edges in the graph are of a given subtype
subtype EDGE_TYPE is RELATION_TYPE range PARENT..SPOUSE;
-- A node in the graph (= PERSON) has either already been reached,
-- is immediately adjacent to those reached, or farther away.
type REACHED_TYPE is (REACHED, NEARBY, NOT_SEEN);

-- each PERSON has a linked list of adjacent nodes, called neighbors

type NEIGHBOR_RECORD;
type NEIGHBOR_POINTER is access NEIGHBOR_RECORD;
type NEIGHBOR_RECORD is
record
  NEIGHBOR_INDEX : INDEX_TYPE;
  NEIGHBOR_EDGE : EDGE_TYPE;
  NEXT_NEIGHBOR : NEIGHBOR_POINTER;
end record;
All relationships are captured in the directed graph of which each record is a node.

**Type PERSON_RECORD** is

```
record
  // static information - filled from PEOPLE file:
  NAME : NAME_TYPE;
  IDENTIFIER : IDENTIFIER_TYPE;
  GENDER : GENDER_TYPE;
  // IDENTIFIERS of immediate relatives - father, mother, spouse
  RELATIVE_IDENTIFIER : RELATIVE_ARRAY;
  // head of linked list of adjacent nodes
  NEIGHBOR_LIST_HEADER : NEIGHBOR_POINTER;
  // data used when traversing graph to resolve user request:
  DISTANCE_FROM_SOURCE : REAL;
  PATH_PREDECESSOR : INDEX_TYPE;
  EDGE_TO_PREDECESSOR : EDGE_TYPE;
  REACHED_STATUS : REACHED_TYPE;
  // data used to compute common genetic material
  DESCENDANT_IDENTIFIER : IDENTIFIER_TYPE;
  DESCENDANT_GENES : REAL;
end record;
```

The PERSON array is the central repository of information about inter-relationships.

```
PERSON : array (INDEX_TYPE) of PERSON_RECORD;
```

Utility to truncate or fill with spaces

```
procedure COERCENAME_STRING (SOURCE : in string; TARGET : in out string);
```

**End RELATION_TYPES_AND_DATA;**

--- END SPECIFICATION --- BEGIN BODY --- --- --- ---

**Package body RELATION_TYPES_AND_DATA is**

```
procedure COERCENAME_STRING (SOURCE : in string; TARGET : in out string) is
  MANY_SPACES : constant string (1..100) :=
    " _" & " _";
begin
  if SOURCE\'length < TARGET\'length then
    TARGET (TARGET\'first..TARGET\'first + SOURCE\'length - 1) := SOURCE;
    TARGET (TARGET\'first + SOURCE\'length..TARGET\'last) :=
      MANY_SPACES (1..TARGET\'length - SOURCE\'length);
  else -- SOURCE longer than TARGET
    TARGET := SOURCE(SOURCE\'first..SOURCE\'first + TARGET\'length - 1);
  end if;
end COERCENAME_STRING;
```

**End RELATION_TYPES_AND_DATA;**
new compilation-unit #2: main line of execution RELATE

with RELATION_TYPES_AND_DATA, text_io, sequential_io;
use RELATION_TYPES_AND_DATA, text_io;

procedure RELATE is

-- this is the format of records in the file to be read in

type FILE_GENDER is ('M', 'F');
type FILE_PERSON_RECORD is
record
   NAME : NAME_TYPE;
   IDENTIFIER : IDENTIFIER_TYPE;
   -- 'M' for MALE and 'F' for FEMALE
   GENDER : FILE_GENDER;
   RELATIVE_IDENTIFIER : RELATIVE_ARRAY;
end record;

-- Instantiate generic package for file IO.
package PEOPLE_IO is
   new sequential_io (ELEMENT_TYPE => FILE_PERSON_RECORD);

-- These variables are used when establishing the PERSON array
-- from the PEOPLE file.
PERSON : PEOPLE_IO . FILE_TYPE;
PERSON_RECORD : FILE_PERSON_RECORD;
CURRENT, NUMBER_OF_PERS : INDEX_TYPE;
PREVIOUS_IDENT, CURRENT_IDENT : IDENTIFIER_TYPE;
RELATIONSHIP : GIVEN_IDENTIFIERS;

-- These variables are used to accept and resolve requests for
-- RELATIONSHIP information.
BUFFER_INDEX, SEMICOLON_LOCATION : BUFFER_RANGE;
REQUEST_BUFFER : BUFFER_TYPE;
PERSON1_IDENT, PERSON2_IDENT : NAME_TYPE;
PERSON1_FOUND, PERSON2_FOUND : COUNTER;
ERROR_MESSAGE : MESSAGE_TYPE;
PERSON1_INDEX, PERSON2_INDEX : INDEX_TYPE;
-- declare procedures directly invoked from RELATE:

procedure LINK_RELATIVES (FROM_INDEX : in INDEX_TYPE;
                          RELATIONSHIP : in GIVEN_IDENTIFIERS;
                          TO_INDEX : in INDEX_TYPE)
is separate;
procedure PROMPT_AND_READ is separate;
procedure CHECK_REQUEST (REQUEST_STATUS : out MESSAGE_TYPE;
                         SEMICOLON_LOCATION : out BUFFER_RANGE)
is separate;
procedure BUFFER_TO_PERSON (PERSON_ID : in out NAME_TYPE;
                           START_LOCATION,
                           STOP_LOCATION : in BUFFER_RANGE)
is separate;
procedure SEARCH_FOR_REQUESTED_PERSONS
               (PERSON1_IDENT, PERSON2_IDENT : in NAME_TYPE;
                PERSON1_INDEX, PERSON2_INDEX : out INDEX_TYPE;
                PERSON1_FOUND, PERSON2_FOUND : in out COUNTER)
is separate;
procedure FIND_RELATIONSHIP (TARGET_INDEX, SOURCE_INDEX : in INDEX_TYPE)
is separate;

-- *** execution of main sequence begins here *** --

begin

PEOPLE_IO . open (PEOPLE, PEOPLE_IO . IN_FILE, "PEOPLE.DAT");
-- CURRENT location in array being filled
CURRENT := 0;
-- This loop reads in the PEOPLE file and constructs the PERSON
-- array from it (one PERSON = one record = one array entry).
-- As records are read in, links are constructed to represent the
-- PARENT-CHILD or SPOUSE RELATIONSHIP. The array then implements
-- a directed graph which is used to satisfy subsequent user
-- requests. The file is assumed to be correct - no validation
-- is performed on it.
READ_IN_PEOPLE:
while not PEOPLE_IO . end_of_file (PEOPLE) loop
   PEOPLE_IO . read (PEOPLE, PEOPLE_RECORD);
   CURRENT := CURRENT+1;
   -- copy direct information from file to array
   PERSON (CURRENT) . NAME := PEOPLE_RECORD . NAME;
   PERSON (CURRENT) . IDENTIFIER := PEOPLE_RECORD . IDENTIFIER;
   if PEOPLE_RECORD . GENDER = 'M' then
      PERSON (CURRENT) . GENDER := MALE;
   else
      PERSON (CURRENT) . GENDER := FEMALE;
   end if;
   PERSON (CURRENT) . RELATIVE_IDENTIFIER :=
       PEOPLE_RECORD . RELATIVE_IDENTIFIER;
   -- Location of adjacent persons as yet undetermined
   PERSON (CURRENT) . NEIGHBOR_LIST_HEADER := null;
   -- Descendants as yet undetermined
   PERSON (CURRENT) . DESCENDANT_IDENTIFIER := NULL_IDENT;
   CURRENT_IDEN := PERSON (CURRENT) . IDENTIFIER;
end while;
-- Compare this PERSON against all previously entered PERSONs
-- to search for RELATIONSHIPS.
COMPARE_TO_PREVIOUS:
  for PREVIOUS in 1..CURRENT-1 loop
    PREVIOUS_IDENT := PERSON (PREVIOUS) . IDENTIFIER;
    RELATIONSHIP := FATHER_IDENT;
    -- Search for father, mother, or spouse relationship in
    -- either direction between this and PREVIOUS PERSON.
    -- Assume at most one RELATIONSHIP exists.
    TRY_ALL_RELATIONSHIPS:
      loop
        if PERSON (CURRENT) . RELATIVE_IDENTIFIER (RELATIONSHIP) =
        PREVIOUS_IDENT
          then
            LINK_RELATIVES (CURRENT, RELATIONSHIP, PREVIOUS);
            exit TRY_ALL_RELATIONSHIPS;
        else
          if CURRENT_IDENT =
            PERSON (PREVIOUS) . RELATIVE_IDENTIFIER (RELATIONSHIP)
            then
              LINK_RELATIVES (PREVIOUS, RELATIONSHIP, CURRENT);
              exit TRY_ALL_RELATIONSHIPS;
          end if;
          end if;
        if RELATIONSHIP < SPOUSE_IDENT then
          RELATIONSHIP := GIVEN_IDENTIFIERS . succ(RELATIONSHIP);
        else
          exit TRY_ALL_RELATIONSHIPS;
        end if;
      end loop TRY_ALL_RELATIONSHIPS;
  end loop COMPARE_TO_PREVIOUS;
end loop READ_IN_PEOPLE;
NUMBER_OF_PERSONS := CURRENT;
PEOPLE_IO . close (PEOPLE);

-- PERSON array is now loaded and edges between immediate relatives
-- (PARENT-CHILD or SPOUSE-SPOUSE) are established.

-- While-loop accepts requests and finds RELATIONSHIP (if any)
-- between pairs of PERSONs.
READ_AND_PROCESS_REQUEST:

loop
  PROMPT_AND_READ;
exit READ_AND_PROCESS_REQUEST when REQUEST_BUFFER = REQUEST_TO_STOP;
  CHECK_REQUEST (ERROR_MESSAGE, SEMICOLON_LOCATION);

  -- Syntax check of request completed. Now either display error
  -- message or search for the two PERSONs.

  if ERROR_MESSAGE = REQUEST_OK then
    -- Request syntactically correct -
    -- search for requested PERSONs.
    BUFFER_TO_PERSON (PERSON1_IDENT, 1, SEMICOLON_LOCATION - 1);
    BUFFER_TO_PERSON (PERSON2_IDENT, SEMICOLON_LOCATION + 1, BUFFER_LENGTH);
    SEARCH_FOR_REQUESTED_PERSONS (PERSON1_IDENT, PERSON2_IDENT,
                                   PERSON1_INDEX, PERSON2_INDEX,
                                   PERSON1_FOUND, PERSON2_FOUND);
    if (PERSON1_FOUND = 1) and (PERSON2_FOUND = 1) then
      -- Exactly one match for each PERSON - proceed to
      -- determine RELATIONSHIP, if any.
      if PERSON1_INDEX = PERSON2_INDEX then
        put ('" & PERSON (PERSON1_INDEX) . NAME & 
        " is identical to ");
        if PERSON (PERSON1_INDEX) . GENDER = MALE then
         put_line("himselt.");
        else
         put_line("herself.");
        end if;
      else
        FIND_RELATIONSHIP (PERSON1_INDEX, PERSON2_INDEX);
      end if;
    else -- either not found or more than one found
      if PERSON1_FOUND = 0 then
       put_line (" First person not found.");
      elsif PERSON1_FOUND > 1 then
       put_line (" Duplicate names for first person - use" & 
                " numeric identifier.");
      end if;
      if PERSON2_FOUND = 0 then
       put_line (" Second person not found.");
      elsif PERSON2_FOUND > 1 then
       put_line (" Duplicate names for second person - use" & 
                " numeric identifier.");
      end if;
      end if;  -- processing of syntactically legal request
    else
      put_line (" Incorrect request format: " & ERROR_MESSAGE);
    end if;
  end loop READ_AND_PROCESS_REQUEST;
  put_line (" End of relation-finder.");
end RELATE;
--- new compilation-unit #3: procedures under RELATE

separate (RELATE)
procedure LINK_RELATIVES (FROM_INDEX : in INDEX_TYPE;
   RELATIONSHIP : in GIVEN_IDENTIFIERS;
   TO_INDEX : in INDEX_TYPE) is
   -- establishes cross-indexing between immediately related PERSONs.

procedure LINK_ONE_WAY (FROM_INDEX : in INDEX_TYPE;
   THIS_EDGE : in EDGE_TYPE;
   TO_INDEX : in INDEX_TYPE) is
   -- Establishes the NEIGHBOR_RECORD from one PERSON to another

NEW_NEIGHBOR : NEIGHBOR_RECORD;

begin
   NEW_NEIGHBOR := new NEIGHBOR_RECORD
    `(NEIGHBOR_INDEX => TO_INDEX,
    NEIGHBOR_EDGE => THIS_EDGE,
    NEXT_NEIGHBOR => PERSON (FROM_INDEX). NEIGHBOR_LIST_HEADER);
    PERSON (FROM_INDEX). NEIGHBOR_LIST_HEADER := NEW_NEIGHBOR;
end;

begin -- execution of LINK_RELATIVES
   if RELATIONSHIP = SPOUSE_IDENT then
      LINK_ONE_WAY (FROM_INDEX, SPOUSE, TO_INDEX);
      LINK_ONE_WAY (TO_INDEX, SPOUSE, FROM_INDEX);
   else -- RELATIONSHIP is father or mother
      LINK_ONE_WAY (FROM_INDEX, PARENT, TO_INDEX);
      LINK_ONE_WAY (TO_INDEX, CHILD, FROM_INDEX);
   end if;
end LINK_RELATIVES;

separate (RELATE)
procedure PROMPT_AND_READ is
   -- Issues prompt for user-request, reads in request,
   -- blank-fills buffer, and skips to next line of input.

LAST_FILLED : natural;

begin
   put_line (" ");
   put_line (" -----------------------------------------------");
   put_line (" Enter two person-identifiers (name or number),");
   put_line (" separated by semicolon. Enter ""stop"" to stop.");
   get_line (REQUEST_BUFFER, LAST_FILLED);
   COERC_STRING (" ", REQUEST_BUFFER (LAST_FILLED+1..BUFFER_LENGTH));
end PROMPT_AND_READ;
separate (RELATE)
procedure CHECK_REQUEST (REQUEST_STATUS : out MESSAGE_TYPE;
                        SEMICOLON_LOCATION : out BUFFER_RANGE) is
    -- Performs syntactic check on request in buffer.
    SEMICOLON_COUNT : COUNTER;
    PERSON1_FIELD_EXISTS, PERSON2_FIELD_EXISTS : boolean;
begin
    REQUEST_STATUS := REQUEST_OK;
    SEMICOLON_LOCATION := 1;
    PERSON1_FIELD_EXISTS := false;
    PERSON2_FIELD_EXISTS := false;
    SEMICOLON_COUNT := 0;
    for BUFFER_INDEX in BUFFER_RANGE loop
        if REQUEST_BUFFER (BUFFER_INDEX) /= ';' then
            if REQUEST_BUFFER (BUFFER_INDEX) = ';' then
                SEMICOLON_LOCATION := BUFFER_INDEX;
                SEMICOLON_COUNT := SEMICOLON_COUNT + 1;
            else -- Check for non-blanks before/after semicolon.
                if SEMICOLON_COUNT < 1 then
                    PERSON1_FIELD_EXISTS := true;
                else
                    PERSON2_FIELD_EXISTS := true;
                end if;
            end if;
        end if;
    end loop;
    -- set REQUEST_STATUS, based on results of scan of REQUEST_BUFFER.
    if SEMICOLON_COUNT /= 1 then
        REQUEST_STATUS := "must be exactly one semicolon."
    elsif not PERSON1_FIELD_EXISTS then
        REQUEST_STATUS := "null field preceding semicolon."
    elsif not PERSON2_FIELD_EXISTS then
        REQUEST_STATUS := "null field following semicolon."
    end if;
end CHECK_REQUEST;

separate (RELATE)
procedure BUFFER_TO_PERSON (PERSON_ID : in out NAME_TYPE;
                           START_LOCATION, STOP_LOCATION : in BUFFER_RANGE) is
    -- fills in the PERSON_ID from the designated portion
    -- of the REQUEST_BUFFER.
    FIRST_NON_BLANK : BUFFER_RANGE;
begin
    FIRST_NON_BLANK := START_LOCATION;
    while REQUEST_BUFFER (FIRST_NON_BLANK) = ' ' loop
        FIRST_NON_BLANK := FIRST_NON_BLANK + 1;
    end loop;
    COERC_STRING (REQUEST_BUFFER (FIRST_NON_BLANK..STOP_LOCATION),
                  PERSON_ID);
end BUFFER_TO_PERSON;
separate (RELATE)
procedure SEARCH_FOR_REQUESTED_PERSONS
(PERSON1_IDENT, PERSON2_IDENT : in NAME_TYPE;
     PERSON1_INDEX, PERSON2_INDEX : out INDEX_TYPE;
     PERSON1_FOUND, PERSON2_FOUND : in out COUNTER) is
-- SEARCH_FOR_REQUESTED_PERSONS scans through the PERSON array,
-- looking for the two requested PERSONS. Match may be by NAME
-- or unique IDENTIFIER-number.
THIS_IDENT : NAME_TYPE;
begin
    PERSON1_FOUND := 0;
    PERSON2_FOUND := 0;
    PERSON1_INDEX := 0;
    PERSON2_INDEX := 0;
    SCAN_ALL_PERSONS:
for CURRENT in 1..NUMBER_OF_PERSONS loop
    -- THIS_IDENT contains CURRENT PERSON's numeric IDENTIFIER
    -- left-justified, padded with blanks.
    COERC_STRING ("", THIS_IDENT);
    for IDENTIFIER_INDEX in IDENTIFIER_RANGE loop
        THIS_IDENT (IDENTIFIER_INDEX) :=
        PERSON (CURRENT).IDENTIFIER (IDENTIFIER_INDEX);
    end loop;
    -- allow identification by name or number.
    if (PERSON1_IDENT = THIS_IDENT) or
       (PERSON1_IDENT = PERSON (CURRENT).NAME)
    then
        PERSON1_FOUND := PERSON1_FOUND + 1;
        PERSON1_INDEX := CURRENT;
    end if;
    if (PERSON2_IDENT = THIS_IDENT) or
       (PERSON2_IDENT = PERSON (CURRENT).NAME)
    then
        PERSON2_FOUND := PERSON2_FOUND + 1;
        PERSON2_INDEX := CURRENT;
    end if;
end loop SCAN_ALL_PERSONS;
end SEARCH_FOR_REQUESTED_PERSONS;
procedure FIND_RELATIONSHIP (TARGET_INDEX, SOURCE_INDEX : in INDEX_TYPE) is
  -- Finds shortest path (if any) between two PERSONs and
  -- determines their RELATIONSHIP based on immediate relations
  -- traversed in path. PERSON array simulates a directed graph,
  -- and algorithm finds shortest path, based on following
  -- weights: PARENT-CHILD edge = 1.0
  -- SPOUSE-SPOUSE edge = 1.8

  type SEARCH_TYPE is (SEARCHING, SUCCEEDED, FAILED);

  SEARCH_STATUS : SEARCH_TYPE;
  THIS_NODE, ADJACENT_NODE, BEST_NEARBY_INDEX, LAST_NEARBY_INDEX : INDEX_TYPE;
  NEARBY_NODE : array (INDEX_TYPE) of INDEX_TYPE;
  THIS_EDGE : EDGE_TYPE;
  THIS_NEIGHBOR : NEIGHBOR_POINTER;
  RELATIONSHIP : GIVEN_IDENTIFIERS;
  MINIMAL_DISTANCE : REAL;

begin  -- execution of FIND_RELATIONSHIP
  -- initialize PERSON-array for processing -
  -- mark all nodes as not seen
  for PERSON_INDEX in 1..NUMBER_OF_PERSONS loop
    PERSON (PERSON_INDEX).REACHED_STATUS := NOT_SEEN;
  end loop;
  THIS_NODE := SOURCE_INDEX;
  -- mark source node as REACHED
  PERSON (THIS_NODE).REACHED_STATUS := REACHED;
  PERSON (THIS_NODE).DISTANCE_FROM_SOURCE := 0.0;
  -- no NEARBY nodes exist yet
  LAST_NEARBY_INDEX := 0;
  if THIS_NODE = TARGET_INDEX then
    SEARCH_STATUS := SUCCEEDED;
  else
    SEARCH_STATUS := SEARCHING;
  end if;
-- Loop keeps processing closest-to-source, unREACHED node
-- until target REACHED, or no more connected nodes.
SEARCH_FOR_TARGET:
while SEARCH_STATUS = SEARCHING loop
  -- Process all nodes adjacent to THIS_NODE
  THIS_NEIGHBOR := PERSON (THIS_NODE) . NEIGHBOR_LIST_HEADER;
  while THIS_NEIGHBOR /= null loop
    PROCESS_ADJACENT_NODE (THIS_NODE,
      THIS_NEIGHBOR . NEIGHBOR_INDEX,
      THIS_NEIGHBOR . NEIGHBOR_EDGE);
    THIS_NEIGHBOR := THIS_NEIGHBOR . NEXT_NEIGHBOR;
  end loop;

  -- All nodes adjacent to THIS_NODE are set. Now search for
  -- shortest-distance unREACHED (but NEARBY) node to process next.
  if LAST_NEARBY_INDEX = 0 then
    SEARCH_STATUS := FAILED;
  else
    -- determine next node to process
    MINIMAL_DISTANCE := 1.0e+18;
    for PERSON_INDEX in 1..LAST_NEARBY_INDEX loop
      if PERSON (NEARBY_NODE (PERSON_INDEX)) . DISTANCE_FROM_SOURCE
      < MINIMAL_DISTANCE
        then
          BEST_NEARBY_INDEX := PERSON_INDEX;
          MINIMAL_DISTANCE :=
            PERSON (NEARBY_NODE (PERSON_INDEX)) . DISTANCE_FROM_SOURCE;
        end if;
    end loop;
    -- establish new THIS_NODE
    THIS_NODE := NEARBY_NODE (BEST_NEARBY_INDEX);
    -- change THIS_NODE from being NEARBY to REACHED
    PERSON (THIS_NODE) . REACHED_STATUS := REACHED;
    -- remove THIS_NODE from NEARBY list
    NEARBY_NODE (BEST_NEARBY_INDEX) := NEARBY_NODE (LAST_NEARBY_INDEX);
    LAST_NEARBY_INDEX := LAST_NEARBY_INDEX - 1;
    if THIS_NODE = TARGET_INDEX then
      SEARCH_STATUS := SUCCEEDED;
    end if;
  end if;
end loop SEARCH_FOR_TARGET;

-- Shortest path between PERSONs now established. Next task is
-- to translate path to English description of RELATIONSHIP.
if SEARCH_STATUS = FAILED then
  put_line ('' & PERSON (TARGET_INDEX) . NAME & '' is not related to '' &
  PERSON (SOURCE_INDEX) . NAME);
else
  -- success - parse path to find and display RELATIONSHIP
  RESOLVE_PATH_TO_ENGLISH;
  COMPUTE_COMMON_GENES (SOURCE_INDEX, TARGET_INDEX);
end if;
end FIND_RELATIONSHIP;
--- new compilation-unit #4: procedures under FIND_RELATIONSHIP

separate (RELATE . FIND_RELATIONSHIP)

procedure PROCESS_ADJACENT_NODE (BASE_NODE, NEXT_NODE : in INDEX_TYPE;
   NEXT_BASE_EDGE : in EDGE_TYPE) is
   -- NEXT_NODE is adjacent to last-REACHED_node (= BASE_NODE).
   -- if NEXT_NODE already REACHED, do nothing.
   -- If previously seen, check whether path thru BASE_NODE is
   -- shorter than current path to NEXT_NODE, and if so re-link
   -- next to base.
   -- If not previously seen, link next to base node.

   WEIGHT_THIS_EDGE, DISTANCE_THRU_BASE_NODE : REAL;

   procedure LINK_NEXT_NODE_TO_BASE_NODE is
      -- link next to base by re-setting its predecessor index to
      -- point to base, note type of edge, and re-set distance
      -- as it is through base node.
   begin -- execution of LINK_NEXT_NODE_TO_BASE_NODE
      PERSON (NEXT_NODE) . DISTANCE_FROM_SOURCE := DISTANCE_THRU_BASE_NODE;
      PERSON (NEXT_NODE) . PATH_PREDECESSOR := BASE_NODE;
      PERSON (NEXT_NODE) . EDGE_TO_PREDECESSOR := NEXT_BASE_EDGE;
   end LINK_NEXT_NODE_TO_BASE_NODE;

   begin -- execution of PROCESS_ADJACENT_NODE
      if PERSON (NEXT_NODE) . REACHED_STATUS /= REACHED then
         if NEXT_BASE_EDGE = SPOUSE then
            WEIGHT_THIS_EDGE := 1.8;
         else
            WEIGHT_THIS_EDGE := 1.0;
         end if;
      end if;
      DISTANCE_THRU_BASE_NODE := WEIGHT_THIS_EDGE +
      PERSON (BASE_NODE) . DISTANCE_FROM_SOURCE;
      if PERSON (NEXT_NODE) . REACHED_STATUS = NOT_SEEN then
         PERSON (NEXT_NODE) . REACHED_STATUS := NEARBY;
         LAST_NEARBY_INDEX := LAST_NEARBY_INDEX + 1;
         NEARBY_NODE (LAST_NEARBY_INDEX) := NEXT_NODE;
         LINK_NEXT_NODE_TO_BASE_NODE;
      else -- REACHED_STATUS = NEARBY
         if DISTANCE_THRU_BASE_NODE < PERSON (NEXT_NODE) . DISTANCE_FROM_SOURCE
            then
               LINK_NEXT_NODE_TO_BASE_NODE;
            end if;
      end if;
   end if;
end PROCESS_ADJACENT_NODE;
separate (RELATE . FIND_RELATIONSHIP)

procedure RESOLVE_PATH_TO_ENGLISH is
   -- RESOLVE_PATH_TO_ENGLISH condenses the shortest path to a
   -- series of RELATIONSHIPS for which there are English
   -- descriptions.
   -- Key persons are the ones in the RELATIONSHIP path which remain
   -- after the path is condensed.

   type SIBLING_TYPE is (STEP, HALF, FULL);

   type KEY_PERSON_RECORD (RELATION_TO_NEXT : RELATION_TYPE := PARENT) is
      record
         PERSON_INDEX : INDEX_TYPE;
         GENERATION_GAP : COUNTER;
         PROXIMITY : SIBLING_TYPE;
         case RELATION_TO_NEXT is
            when COUSIN => COUSIN_RANK : COUNTER;
            when others => null;
         end case;
      end record;

   -- these variables are used to generate KEY_PERSONs
   GENERATION_COUNT : COUNTER;
   THIS_COUSIN_RANK : COUNTER;
   THIS_PROXIMITY : SIBLING_TYPE;

   -- these variables are used to condense the path
   KEY_PERSON : array (INDEX_TYPE) of KEY_PERSON_RECORD;
   KEY_RELATION, LATER_KEY_RELATION, PRIMARY_RELATION,
      NEXT_PRIMARY_RELATION : RELATION_TYPE;
   KEY_INDEX, LATER_KEY_INDEX, PRIMARY_INDEX : INDEX_TYPE;
   ANOTHER_ELEMENT_POSSIBLE : boolean;

   function FULL_SIBLING (INDEX1, INDEX2 : in INDEX_TYPE)
      return boolean is
      -- Determines whether two PERSONs are full siblings, i.e.,
      -- have the same two parents.
      begin
      return
         PERSON (INDEX1) . RELATIVE_IDENTIFIER (FATHER_IDENT) /= NULL_IDENT and
         PERSON (INDEX1) . RELATIVE_IDENTIFIER (MOTHER_IDENT) /= NULL_IDENT and
         PERSON (INDEX1) . RELATIVE_IDENTIFIER (FATHER_IDENT) =
            PERSON (INDEX2) . RELATIVE_IDENTIFIER (FATHER_IDENT) and
         PERSON (INDEX1) . RELATIVE_IDENTIFIER (MOTHER_IDENT) =
            PERSON (INDEX2) . RELATIVE_IDENTIFIER (MOTHER_IDENT);
   end FULL_SIBLING;
procedure CONDENSE_KEY_PERSONS (AT_INDEX : in INDEX_TYPE;
GAPE SIZE : in COUNTER)
is
-- CONDENSE_KEY_PERSONS condenses superfluous entries from the
-- KEY_PERSON array, starting at AT_INDEX.

RECEIVE_INDEX, SEND_INDEX : INDEX_TYPE;

begin
RECEIVE_INDEX := AT_INDEX;
loop
RECEIVE_INDEX := RECEIVE_INDEX + 1;
SEND_INDEX := RECEIVE_INDEX + GAP_SIZE;
KEY_PERSON (RECEIVE_INDEX) := KEY_PERSON (SEND_INDEX);
exit when KEY_PERSON (SEND_INDEX) . RELATION_TO_NEXT = NULL_RELATION;
end loop;
end CONDENSE_KEY_PERSONS;

procedure DISPLAY_RELATION (FIRST_INDEX, LAST_INDEX, PRIMARY_INDEX
: in INDEX_TYPE)
is separate;

begin -- execution of RESOLVE_PATH_TO_ENGLISH
put_line (" Shortest path between identified persons: ");
THIS_NODE := TARGET_INDEX;
KEY_INDEX := 1;
-- Display path and initialize KEY_PERSON array from path elements.
TRAVERSE_SHORTEST_PATH:
while THIS_NODE /= SOURCE_INDEX loop
put (" " & PERSON (THIS_NODE) . NAME & " is ");
case PERSON (THIS_NODE) . EDGE_TO_PREDECESSOR is
when PARENT =>
put_line ("parent of ");
KEY_PERSON (KEY_INDEX) :=
(PERSON_INDEX => THIS_NODE,
GENERATION_GAP => 1,
PROXIMITY => FULL,
RELATION_TO_NEXT => PARENT);
when CHILD =>
put_line ("child of ");
KEY_PERSON (KEY_INDEX) :=
(PERSON_INDEX => THIS_NODE,
GENERATION_GAP => 1,
PROXIMITY => FULL,
RELATION_TO_NEXT => CHILD);
when SPOUSE =>
put_line ("spouse of ");
KEY_PERSON (KEY_INDEX) :=
(PERSON_INDEX => THIS_NODE,
GENERATION_GAP => 0,
PROXIMITY => FULL,
RELATION_TO_NEXT => SPOUSE);
end case;
KEY_INDEX := KEY_INDEX + 1;
THIS_NODE := PERSON (THIS_NODE) . PATH_PREDECESSOR;
end loop TRAVERSE_SHORTEST_PATH;
put_line( "& PERSON (THIS NODE) . NAME);
KEY_PERSON (KEY_INDEX) :=
    (PERSON_INDEX => THIS_NODE,
     GENERATION_GAP => 0,
     PROXIMITY => FULL,
     RELATION_TO_NEXT => NULL_RELATION);
KEY_PERSON (KEY_INDEX + 1) :=
    (PERSON_INDEX => 0,
     GENERATION_GAP => 0,
     PROXIMITY => FULL,
     RELATION_TO_NEXT => NULL_RELATION);
-- Resolve CHILD-PARENT and CHILD-SPOUSE-PARENT relations
-- to SIBLING relations.
KEY_INDEX := 1;
FIND_SIBLINGS:
    while KEY_PERSON (KEY_INDEX) . RELATION_TO_NEXT /= NULL_RELATION loop
        if KEY_PERSON (KEY_INDEX) . RELATION_TO_NEXT = CHILD then
            LATER_KEY_RELATION := KEY_PERSON (KEY_INDEX + 1) . RELATION_TO_NEXT;
            if LATER_KEY_RELATION = PARENT then
                -- found either full or half SIBLINGS
                if FULL_SIBLING (KEY_PERSON (KEY_INDEX) . PERSON_INDEX,
                                 KEY_PERSON (KEY_INDEX + 2) . PERSON_INDEX) then
                    THIS_PROXIMITY := FULL;
                else
                    THIS_PROXIMITY := HALF;
                end if;
                KEY_PERSON (KEY_INDEX) :=
                    (PERSON_INDEX => KEY_PERSON (KEY_INDEX) . PERSON_INDEX,
                     GENERATION_GAP => 0,
                     PROXIMITY => THIS_PROXIMITY,
                     RELATION_TO_NEXT => SIBLING);
                CONDENSE_KEY_PERSONS (KEY_INDEX, 1);
            elsif (LATER_KEY_RELATION = SPOUSE) and
                    (KEY_PERSON (KEY_INDEX + 2) . RELATION_TO_NEXT = PARENT) then
                -- found step-SIBLINGS
                KEY_PERSON (KEY_INDEX) :=
                    (PERSON_INDEX => KEY_PERSON (KEY_INDEX) . PERSON_INDEX,
                     GENERATION_GAP => 0,
                     PROXIMITY => STEP,
                     RELATION_TO_NEXT => SIBLING);
                CONDENSE_KEY_PERSONS (KEY_INDEX, 2);
            end if; -- LATER_KEY_RELATION = PARENT
        end if;
        RELATION_TO_NEXT = CHILD
        KEY_INDEX := KEY_INDEX + 1;
    end loop FIND_SIBLINGS;
-- Resolve CHILD-CHILD--... and PARENT-PARENT--... relations to
-- direct descendant or ancestor relations.
KEY_INDEX := 1;
FIND_ANCESTORS_OR_DESCENDANTS:
    while KEY_PERSON (KEY_INDEX) . RELATION_TO_NEXT /= NULL_RELATION loop
        if (KEY_PERSON (KEY_INDEX) . RELATION_TO_NEXT = CHILD) or
           (KEY_PERSON (KEY_INDEX) . RELATION_TO_NEXT = PARENT)
            then
                LATER_KEY_INDEX := KEY_INDEX + 1;
                while KEY_PERSON (LATER_KEY_INDEX) . RELATION_TO_NEXT =
                KEY_PERSON (KEY_INDEX) . RELATION_TO_NEXT loop
                    LATER_KEY_INDEX := LATER_KEY_INDEX + 1;
                end loop;
                GENERATION_COUNT := LATER_KEY_INDEX - KEY_INDEX;
                if GENERATION_COUNT > 1 then -- compress generations
                    KEY_PERSON (KEY_INDEX) . GENERATION_GAP := GENERATION_COUNT;
                    CONDENSE_KEY_PERSONS (KEY_INDEX, GENERATION_COUNT - 1);
                end if;
            end if;
        end loop
    end loop FIND_ANCESTORS_OR_DESCENDANTS;
-- Resolve CHILD-SIBLING-PARENT to COUSIN,
-- CHILD-SIBLING to NEPHEW,
-- SIBLING-PARENT to UNCLE.

KEY_INDEX := 1;

FIND_COUSINS_NEPHWS_UNCLES:
while KEY_PERSON (KEY_INDEX) . RELATION_TO_NEXT /= NULL_RELATION loop
  LATER_KEY_RELATION := KEY_PERSON (KEY_INDEX + 1) . RELATION_TO_NEXT;
  if (KEY_PERSON (KEY_INDEX) . RELATION_TO_NEXT = CHILD) and
    (LATER_KEY_RELATION = SIBLING)
    then -- COUSIN or NEPHEW
      if KEY_PERSON (KEY_INDEX + 2) . RELATION_TO_NEXT = PARENT then
        if KEY_PERSON (KEY_INDEX) . GENERATION_GAP <
          KEY_PERSON (KEY_INDEX + 2) . GENERATION_GAP
          then
            THIS_COUSIN_RANK :=
              KEY_PERSON (KEY_INDEX) . GENERATION_GAP;
            else
              THIS_COUSIN_RANK :=
              KEY_PERSON (KEY_INDEX + 2) . GENERATION_GAP;
            end if;
        KEY_PERSON (KEY_INDEX) :=
          (PERSON_INDEX => KEY_PERSON (KEY_INDEX) . PERSON_INDEX,
           GENERATION_GAP => abs (KEY_PERSON (KEY_INDEX) . GENERATION_GAP -
                                KEY_PERSON (KEY_INDEX + 2) . GENERATION_GAP),
           PROXIMITY => KEY_PERSON (KEY_INDEX + 1) . PROXIMITY,
           RELATION_TO_NEXT => COUSIN,
           COUSIN_RANK => THIS_COUSIN_RANK); condense_key_persons (KEY_INDEX, 2);
      else -- found NEPHEW
        KEY_PERSON (KEY_INDEX) :=
          (PERSON_INDEX => KEY_PERSON (KEY_INDEX) . PERSON_INDEX,
           GENERATION_GAP => KEY_PERSON (KEY_INDEX + 1) . GENERATION_GAP,
           PROXIMITY => KEY_PERSON (KEY_INDEX) . PROXIMITY,
           RELATION_TO_NEXT => NEPHEW);
        condense_key_persons (KEY_INDEX, 1);
      end if;
    elseif KEY_PERSON (KEY_INDEX) . RELATION_TO_NEXT = SIBLING and
      LATER_KEY_RELATION = PARENT
    then -- found UNCLE
      KEY_PERSON (KEY_INDEX) :=
        (PERSON_INDEX => KEY_PERSON (KEY_INDEX) . PERSON_INDEX,
         GENERATION_GAP => KEY_PERSON (KEY_INDEX + 1) . GENERATION_GAP,
         PROXIMITY => KEY_PERSON (KEY_INDEX) . PROXIMITY,
         RELATION_TO_NEXT => UNCLE);
      condense_key_persons (KEY_INDEX, 1);
    end if;
  end if;
  KEY_INDEX := KEY_INDEX + 1;
end loop FIND_COUSINS_NEPHWS_UNCLES;
-- Loop below will pick out valid adjacent strings of elements
-- to be displayed.  KEY_INDEX points to first element,
-- LATER_KEY_INDEX to last element, and PRIMARY_INDEX to the
-- element which determines the primary English word to be used.
-- Associativity of adjacent elements in condensed table
-- is based on English usage.

KEY_INDEX := 1;
put_line (" Condensed path:";
CONSOLIDATE_ADJACENT_PERSONS:
while KEY_PERSON (KEY_INDEX) . RELATION_TO_NEXT /= NULL_RELATION loop
KEY_RELATION := KEY_PERSON (KEY_INDEX) . RELATION_TO_NEXT;
LATER_KEY_INDEX := KEY_INDEX;
PRIMARY_INDEX := KEY_INDEX;
if KEY_PERSON (KEY_INDEX + 1) . RELATION_TO_NEXT /= NULL_RELATION then
   -- seek multi-element combination
   ANOTHER_ELEMENT_POSSIBLE := true;
   if KEY_RELATION = SPOUSE then
      LATER_KEY_INDEX := LATER_KEY_INDEX + 1;
      PRIMARY_INDEX := LATER_KEY_INDEX;
      if (KEY_PERSON (LATER_KEY_INDEX) . RELATION_TO_NEXT = SIBLING) or
         (KEY_PERSON (LATER_KEY_INDEX) . RELATION_TO_NEXT = COUSIN)
      then
         -- Nothing can follow SPOUSE-SIBLING or SPOUSE-COUSIN
         ANOTHER_ELEMENT_POSSIBLE := false;
      end if;
   end if;
end if;

-- PRIMARY_INDEX is now correctly set. Next if-statement
-- determines if a following SPOUSE relation should be
-- appended to this combination or left for the next
-- combination.
if ANOTHER_ELEMENT_POSSIBLE and
   (KEY_PERSON (PRIMARY_INDEX + 1) . RELATION_TO_NEXT = SPOUSE)
then
   -- Only a SPOUSE can follow a Primary
   -- check primary preceding and following SPOUSE.
   PRIMARY_RELATION :=
      KEY_PERSON (PRIMARY_INDEX) . RELATION_TO_NEXT;
   NEXT_PRIMARY_RELATION :=
      KEY_PERSON (PRIMARY_INDEX + 2) . RELATION_TO_NEXT;
   if (NEXT_PRIMARY_RELATION = NEPHEW or
      NEXT_PRIMARY_RELATION = COUSIN or
      NEXT_PRIMARY_RELATION = NULL_RELATION)
   or (PRIMARY_RELATION = NEPHEW)
   or ( (PRIMARY_RELATION = SIBLING or
        PRIMARY_RELATION = PARENT)
      and NEXT_PRIMARY_RELATION /= UNCLE )
   then
      -- append following SPOUSE with this combination.
      LATER_KEY_INDEX := LATER_KEY_INDEX + 1;
   end if;
end if;   -- multi-element combination
DISPLAY_RELATION (KEY_INDEX, LATER_KEY_INDEX, PRIMARY_INDEX);
KEY_INDEX := LATER_KEY_INDEX + 1;
end loop CONSOLIDATE_ADJACENT_PERSONS;
put_line (" & PERSON (KEY_PERSON (KEY_INDEX) . PERSON_INDEX) . NAME);
end;   -- RESOLVE_PATH_TO_ENGLISH
--- new compilation-unit #5: procedures under RESOLVE_PATH_TO_ENGLISH

separate (RELATE . FIND_RELATIONSHIP . RESOLVE_PATH_TO_ENGLISH)
procedure DISPLAY_RELATION (FIRST_INDEX, LAST_INDEX, PRIMARY_INDEX
   : in INDEX_TYPE) is
   -- DISPLAY_RELATION takes 1, 2, or 3 adjacent elements in the
   -- condensed table and generates the English description of
   -- the relation between the first and last + 1 elements.

   INLAW : boolean;
   THIS_PROXIMITY : SIBLING_TYPE;
   THIS_GENDER : GENDER_TYPE;
   FIRST_RELATION, LAST_RELATION, PRIMARY_RELATION
      : RELATION_TYPE;
   THIS_GENERATION_GAP, THIS_COUSIN_RANK
      : COUNTER;

   -- need to instantiate package to display integer values
   package COUNTER_IO is
      new integer_io (COUNTER);
begin -- execution of DISPLAY_RELATION
  FIRST_RELATION := KEY_PERSON (FIRST_INDEX) . RELATION_TO_NEXT;
  LAST_RELATION := KEY_PERSON (LAST_INDEX) . RELATION_TO_NEXT;
  PRIMARY_RELATION := KEY_PERSON (PRIMARY_INDEX) . RELATION_TO_NEXT;
  -- set THIS_PROXIMITY
  if ((PRIMARY_RELATION = PARENT) and (FIRST_RELATION = SPOUSE)) or
     ((PRIMARY_RELATION = CHILD) and (LAST_RELATION = SPOUSE))
  then
    THIS_PROXIMITY := STEP;
  elsif PRIMARY_RELATION = SIBLING or
       PRIMARY_RELATION = UNCLE or
       PRIMARY_RELATION = NEPHEW or
       PRIMARY_RELATION = COUSIN
  then
    THIS_PROXIMITY := KEY_PERSON (PRIMARY_INDEX) . PROXIMITY;
  else
    THIS_PROXIMITY := FULL;
  end if;
  -- set THIS_GENERATION_GAP
  if PRIMARY_RELATION = PARENT or
     PRIMARY_RELATION = CHILD or
     PRIMARY_RELATION = UNCLE or
     PRIMARY_RELATION = NEPHEW or
     PRIMARY_RELATION = COUSIN
  then
    THIS_GENERATION_GAP := KEY_PERSON (PRIMARY_INDEX) . GENERATION_GAP;
  else
    THIS_GENERATION_GAP := 0;
  end if;
  -- set INLAW
  INLAW := false;
  if (FIRST_RELATION = SPOUSE) and
     (PRIMARY_RELATION = SIBLING or
      PRIMARY_RELATION = CHILD or
      PRIMARY_RELATION = NEPHEW or
      PRIMARY_RELATION = COUSIN)
  then
    INLAW := true;
  elsif (LAST_RELATION = SPOUSE) and
     (PRIMARY_RELATION = SIBLING or
      PRIMARY_RELATION = PARENT or
      PRIMARY_RELATION = UNCLE or
      PRIMARY_RELATION = COUSIN)
  then
    INLAW := true;
  end if;
  -- set THIS_COUSIN_RANK
  if PRIMARY_RELATION = COUSIN then
    THIS_COUSIN_RANK := KEY_PERSON (PRIMARY_INDEX) . COUSIN_RANK;
  end if;
— parameters are set - now generate display.

put (" " & PERSON (KEY_PERSON (FIRST_INDEX) . PERSON_INDEX) . NAME & " is ");
if PRIMARY_RELATION = PARENT or PRIMARY_RELATION = CHILD or PRIMARY_RELATION = UNCLE or PRIMARY_RELATION = NEPHEW then
 -- display generation-qualifier
   if THIS_GENERATION_GAP >= 3 then
     put ("great");
     if THIS_GENERATION_GAP > 3 then
       put ("*");
       COUNTER_IO put (THIS_GENERATION_GAP - 2, width => 1);
     end if;
     put ("-");
   end if;
   if THIS_GENERATION_GAP >= 2 then
     put ("grand-");
   end if;
elsif (PRIMARY_RELATION = COUSIN) and then (THIS_COUSIN_RANK > 1) then
   COUNTER_IO put (THIS_COUSIN_RANK, width => 1);
   case THIS_COUSIN_RANK mod 10 is
     when 1 => put ("st ");
     when 2 => put ("nd ");
     when 3 => put ("rd ");
     when others => put ("th ");
   end case;
end if;

if THIS_PROXIMITY = STEP then
  put ("step-");
elsif THIS_PROXIMITY = HALF then
  put ("half-");
end if;
THIS_GENDER := PERSON (KEY_PERSON (FIRST_INDEX) . PERSON_INDEX) . GENDER;
case PRIMARY_RELATION is
    when PARENT => if THIS_GENDER = MALE then put ("father");
                  else put ("mother");
                  end if;
    when CHILD   => if THIS_GENDER = MALE then put ("son");
                  else put ("daughter");
                  end if;
    when SPOUSE  => if THIS_GENDER = MALE then put ("husband");
                  else put ("wife");
                  end if;
    when SIBLING => if THIS_GENDER = MALE then put ("brother");
                  else put ("sister");
                  end if;
    when UNCLE   => if THIS_GENDER = MALE then put ("uncle");
                  else put ("aunt");
                  end if;
    when NEPHEW  => if THIS_GENDER = MALE then put ("nephew");
                  else put ("niece");
                  end if;
    when COUSIN  => put ("cousin");
    when others  => put ("null");
end case;

if INLAW then
    put ("-in-law");
end if;

if (PRIMARY_RELATION = COUSIN) and (THIS_GENERATION_GAP > 0) then
    if THIS_GENERATION_GAP > 1 then
        put (" ");
        COUNTER_IO . put (THIS_GENERATION_GAP, width => 1);
        put (" times removed");
    else
        put (" once removed");
    end if;
end if;

put_line (" of");
end DISPLAY_RELATION;
new compilation-unit #6: procedures under FIND_RELATIONSHIP

separate (RELATE . FIND_RELATIONSHIP)

procedure COMPUTE_COMMON_GENES (INDEX1, INDEX2 : in INDEX_TYPE) is
  -- COMPUTE_COMMON_GENES assumes that each ancestor contributes
  -- half of the genetic material to a PERSON. It finds common
  -- ancestors between two PERSONs and computes the expected
  -- value of the PROPORTION of common material.

  COMMON_PROPORTION : REAL;

package REAL_IO is
  new FLOAT_IO (REAL);

procedure ZERO_PROPORTION (ZERO_INDEX : in INDEX_TYPE) is
  -- ZERO_PROPORTION recursively seeks out all ancestors and
  -- zeros them out.

  THIS_NEIGHBOR : NEIGHBOR_POINTER;

begin
  PERSON (ZERO_INDEX). DESCENDANT_GENES := 0.0;
  THIS_NEIGHBOR := PERSON (ZERO_INDEX). NEIGHBOR_LIST_HEADER;
  while THIS_NEIGHBOR /= null loop
    if THIS_NEIGHBOR. NEIGHBOR_EDGE = PARENT then
      ZERO_PROPORTION (THIS_NEIGHBOR . NEIGHBOR_INDEX);
    end if;
    THIS_NEIGHBOR := THIS_NEIGHBOR . NEXT_NEIGHBOR;
  end loop;
end ZERO_PROPORTION;

procedure MARK_PROPORTION (MARKER : in IDENTIFIER_TYPE;
  PROPORTION : in REAL;
  MARKED_INDEX : in INDEX_TYPE) is
  -- MARK_PROPORTION recursively seeks out all ancestors and
  -- marks them with the sender's PROPORTION of shared
  -- genetic material. This PROPORTION is diluted by one-half
  -- for each generation.

  THIS_NEIGHBOR : NEIGHBOR_POINTER;

begin
  PERSON (MARKED_INDEX). DESCENDANT_IDENTIFIER := MARKER;
  PERSON (MARKED_INDEX). DESCENDANT_GENES :=
    PERSON (MARKED_INDEX). DESCENDANT_GENES + PROPORTION;
  THIS_NEIGHBOR := PERSON (MARKED_INDEX). NEIGHBOR_LIST_HEADER;
  while THIS_NEIGHBOR /= null loop
    if THIS_NEIGHBOR. NEIGHBOR_EDGE = PARENT then
      MARK_PROPORTION (MARKER, PROPORTION / 2.0,
                       THIS_NEIGHBOR . NEIGHBOR_INDEX);
    end if;
    THIS_NEIGHBOR := THIS_NEIGHBOR . NEXT_NEIGHBOR;
  end loop;
end MARK_PROPORTION;
procedure CHECK_COMMON_PROPORTION
(COMMON_PROPORTION : in out REAL;
MATCH_IDENTIFIER : in IDENTIFIER_TYPE;
PROPORTION : in REAL;
ALREADY_COUNTED : in REAL;
CHECK_INDEX : in INDEX_TYPE) is
  -- CHECK_COMMON_PROPORTION searches all the ancestors of
  -- CHECK_INDEX to see if any have been marked, and if so
  -- adds the appropriate amount to COMMON_PROPORTION.

THIS_NEIGHBOR : NEIGHBOR_POINTER;
THIS_CONTRIBUTION : REAL;

begin
  if PERSON (CHECK_INDEX).DESCENDANT_IDENTIFIER = MATCH_IDENTIFIER then
    -- Increment COMMON_PROPORTION by the contribution of
    -- this common ancestor, but discount for the contribution
    -- of less remote ancestors already counted.
    THIS_CONTRIBUTION := PERSON (CHECK_INDEX).DESCENDANT_GENES
                          * PROPORTION;
    COMMON_PROPORTION := COMMON_PROPORTION
                         + THIS_CONTRIBUTION - ALREADY_COUNTED;
  else
    THIS_CONTRIBUTION := 0.0;
  end if;
  THIS_NEIGHBOR := PERSON (CHECK_INDEX).NEIGHBOR_LIST_HEADER;
  while THIS_NEIGHBOR /= null loop
    if THIS_NEIGHBOR.NEIGHBOR_EDGE = PARENT then
      CHECK_COMMON_PROPORTION (COMMON_PROPORTION, MATCH_IDENTIFIER, PROPORTION / 2.0,
                               THIS_CONTRIBUTION / 4.0,
                               THIS_NEIGHBOR.NEIGHBOR_INDEX);
    end if;
    THIS_NEIGHBOR := THIS_NEIGHBOR.NEXT_NEIGHBOR;
  end loop;
end CHECK_COMMON_PROPORTION;

begin
  -- COMPUTE_COMMON_GENES
  -- First zero out all ancestors to allow adding. This is necessary
  -- because there might be two paths to an ancestor.
  ZERO_PROPORTION (INDEX1);
  -- now mark with shared PROPORTION
  MARK_PROPORTION (PERSON (INDEX1).IDENTIFIER, 1.0, INDEX1);
  COMMON_PROPORTION := 0.0;
  CHECK_COMMON_PROPORTION (COMMON_PROPORTION,
                           PERSON (INDEX1).IDENTIFIER, 1.0, 0.0, INDEX2);
  put ("Proportion of common genetic material = ");
  REAL_10. put (COMMON_PROPORTION, fore => 1, aft => 5, exp => 3);
  put_line (" ");
end COMPUTE_COMMON_GENES;
3.0 BASIC

Because of the unavailability of a standard implementation, the BASIC program could not be tested directly. However, a syntactically non-standard version, which is believed to be logically equivalent, was tested.

10000 ! --- program-unit number 1 ----
10010 !
10020 program RELATE
10030 !
10040 ! declare subs to be used by this program-unit
10050 !
10060 declare external sub FIND_RELATIONSHIP
10070 declare sub LINK_RELATIVES, LINK_ONE_WAY, PROMPT_AND_READ
10080 declare sub CHECK_REQUEST, SEARCH_FOR_REQUESTED_PERSONS
10090 !
10100 option base 1
10110 !
10120 ! Define global objects
10130 !
10140 data 300
10150 read MAX_PERSONS
10160 !
10170 data 1, 2 ! for truth values
10180 read TRUE, FALSE
10190 !
10200 ! each PERSON's record in the file identifies at most three
10210 ! others directly related: father, mother, and spouse
10220 data 1, 2, 3
10230 read FATHER_IDENT, MOTHER_IDENT, SPouse_IDENT
10240 !
10250 data M, F
10260 read MALE$, FEMALE$
10270 !
10280 data 000
10290 read NULL_IDENT$
10300 !
10310 data 1, 2, 3, 4, 5, 6, 7, 8
10320 read PARENT, CHILD, SPOUSE, SIBLING, UNCLE, NEPHEW
10325 read COUSIN, NULL_RELATION
10330 !
10340 ! A node in the graph (= PERSON) has either already been reached,
10350 ! is immediately adjacent to those reached, or farther away.
10360 data 1, 2, 3
10370 read REACHED, NEARBY, NOT_SEEN
10380 !
The following data arrays are the central repository of information about inter-relationships. All relationships are captured in the directed graph of which each record is a node.

Static information - filled from PEOPLE file:

- `NAME$ (300)`, `IDENTIFIER$ (300)`, `GENDER$ (300)`
- `IDENTIFIER$s of immediate relatives - father, mother, spouse`

- `RELATIVE_IDENTIFIER$ (300,3)`

Pointers to immediate neighbors in graph:

- `NEIGHBOR_COUNT (300)`
- `NEIGHBOR_INDEX (300,20)`, `NEIGHBOR_EDGE (300,20)`

Data used when traversing graph to resolve user request:

- `DISTANCE_FROM_SOURCE (300)`, `PATH_PREDECESSOR (300)`
- `EDGE_TO_PREDECESSOR (300)`, `REACHED_STATUS (300)`

Data used to compute common genetic material:

- `DESCENDANT_IDENTIFIER$ (300)`, `DESCENDANT_GENES (300)`

Stop, Request OK

Read `REQUEST_TO_STOP$, REQUEST_OK$`

End initialization
10640 ! begin main line of execution
10650 !
10660 open #1: name "PEOPLE.DAT", access input, retype native, &
& organization sequential
10670 !
10680 ! This loop reads in the PEOPLE file and constructs the person
10690 ! array from it (one person = one set of array entries).
10700 ! As records are read in, links are constructed to represent the
10710 ! PARENT-CHILD or SPOUSE RELATIONSHIP. The array then implements
10720 ! a directed graph which is used to satisfy subsequent user
10730 ! requests. The file is assumed to be correct - no validation
10740 ! is performed on it.
10750 !
10760 for CURRENT = 1 to MAX_PERS0NS
10770 read #1, if missing then exit for,
&
& with "string*20, string*3, string*1, 3 of string*3": &
& NAME$ (CURRENT), IDENTIFIER$ (CURRENT), GENDER$ (CURRENT), &
& RELATIVE_IDENTIFIER$ (CURRENT, FATHER_IDENT), &
& RELATIVE_IDENTIFIER$ (CURRENT, MOTHER_IDENT), &
& RELATIVE_IDENTIFIER$ (CURRENT, SPOUSE_IDENT)
10780 let NAME$ (CURRENT) = rtrim$ (NAME$ (CURRENT))
10790 ! Location of adjacent persons as yet undetermined
10800 let NEIGHBOR_COUNT (CURRENT) = 0
10810 ! Descendants as yet undetermined
10820 let DESCENDANT_IDENTIFIER$ (CURRENT) = NULL_IDENT$
10830 let CURRENT_IDENT$ = IDENTIFIER$ (CURRENT)
10840 ! Compare this PERSON against all previously entered PERSONS
10850 ! to search for RELATIONSHIPS.
10860 !
10870 for PREVIOUS = 1 to CURRENT - 1
10880 ! Search for father, mother, or spouse relationship in
10890 ! either direction between this and PREVIOUS person.
10900 ! Assume at most one RELATIONSHIP exists.
10910 for RELATIONSHIP = FATHER_IDENT to SPOUSE_IDENT
10920 if RELATIVE_IDENTIFIER$ (CURRENT, RELATIONSHIP) &
& = PREVIOUS_IDENT$ then
10930 call LINK_RELATIVES (CURRENT, RELATIONSHIP, PREVIOUS)
10940 exit for
10950 elseif RELATIVE_IDENTIFIER$ (PREVIOUS, RELATIONSHIP) &
& = CURRENT_IDENT$ then
10960 call LINK_RELATIVES (PREVIOUS, RELATIONSHIP, CURRENT)
10970 exit for
10980 end if
10990 next RELATIONSHIP
11000 next PREVIOUS
11010 next CURRENT
11020 let NUMBER_OF_PERS0NS = CURRENT - 1
11030 close #1
11040 !
11050 ! Person arrays are now loaded and edges between immediate relatives
11060 ! (PARENT-CHILD or SPOUSE-SPouse) are established.
11070 !
Do-loop accepts requests and finds relationship (if any) between pairs of PERSONs.

```
11080 !
11090 !
11100 do
11110 do-loop
11120 call PROMPT_AND_READ
11130 if REQUEST_BUFFER$ = REQUEST_TO_STOP$ then exit do
11140 call CHECK_REQUEST (REQUEST_MESSAGE$, PERSON1_IDENT$, PERSON2_IDENT$)
11150 !
11160 ! Syntax check of request completed. Now either display error
11170 ! message or search for the two PERSONs.
11180 !
11190 if ERROR_MESSAGE$ = REQUEST_OK$ then
11200 ! request syntactically correct
11210 call SEARCH_FOR_REQUESTED_PERSONS(PERSON1_IDENT$, PERSON2_IDENT$, &
11220 & PERSON1_INDEX, PERSON2_INDEX, &
11230 & PERSON1_FOUND, PERSON2_FOUND)
11240 if PERSON1_INDEX = PERSON2_INDEX then
11250 ! Exactly one match for each PERSON - proceed to
11260 ! determine RELATIONSHIP, if any.
11270 print ""; NAME$ (PERSON1_INDEX); " is identical to ";
11280 if GENDER$ (PERSON1_INDEX) = MALE$ then
11290 print "himself."
11300 else
11310 print "herself."
11320 end if
11330 else
11340 call FIND_RELATIONSHIP &
11350 & (PERSON1_INDEX, PERSON2_INDEX, NUMBER_OF_PERSONS, &
11360 & NAME$, IDENTIFIER$, GENDER$, RELATIVE_IDENTIFIER$, &
11370 & NEIGHBOR_COUNT, NEIGHBOR_INDEX, NEIGHBOR_EDGE, &
11380 & DISTANCE_FROM_SOURCE, PATH_PREDECESSOR, &
11390 & EDGE_TO_PREDECESSOR, REACHED_STATUS, &
11400 & DESCENDANT_IDENTIFIER$, DESCENDANT_GENES)
11410 end if
11420 else ! either not found or more than one found
11430 if PERSON1_FOUND = 0 then
11440 print "First person not found."
11450 elseif PERSON1_FOUND > 1 then
11460 print " Duplicate names for first person -";
11470 print " use numeric identifier."
11480 end if
11490 if PERSON2_FOUND = 0 then
11500 print "Second person not found."
11510 elseif PERSON2_FOUND > 1 then
11520 print " Duplicate names for second person -";
11530 print " use numeric identifier."
11540 end if
11550 else
11560 print " Incorrect request format: "; ERROR_MESSAGE$
11570 end if
11580 loop
11590 print " End of relation-finder."
11600 stop
11610 ! end of main line of execution; internal subs follow
```
11570 !
11580 sub LINK_RELATIVES (FROM_INDEX, RELATIONSHIP, TO_INDEX)
11590 ! establishes cross-indexing between immediately related PERSONs.
11600 !
11610 if RELATIONSHIP = SPOUSE IDENT then
11620 call LINK_ONE_WAY (FROM_INDEX, SPOUSE, TO_INDEX)
11630 call LINK_ONE_WAY (TO_INDEX, SPOUSE, FROM_INDEX)
11640 else ! RELATIONSHIP is father or mother
11650 call LINK_ONE_WAY (FROM_INDEX, PARENT, TO_INDEX)
11660 call LINK_ONE_WAY (TO_INDEX, CHILD, FROM_INDEX)
11670 end if
11680 end sub
11690 !
11700 sub LINK_ONE_WAY (FROM_INDEX, THIS_EDGE, TO_INDEX)
11710 ! Establishes the neighbor entries from one person to another
11720 !
11730 let NEXT_NEIGHBOR = NEIGHBOR COUNT (FROM_INDEX) + 1
11740 let NEIGHBOR_COUNT (FROM_INDEX) = NEXT_NEIGHBOR
11750 let NEIGHBOR_INDEX (FROM_INDEX, NEXT_NEIGHBOR) = TO_INDEX
11760 let NEIGHBOR_EDGE (FROM_INDEX, NEXT_NEIGHBOR) = THIS_EDGE
11770 end sub
11780 !
11790 sub PROMPT_AND_READ
11800 ! Issues prompt for user-request, reads in request,
11810 ! blank-fills buffer, and skips to next line of input.
11820 !
11830 print
11840 print "-----------------------------------------------"
11850 print " Enter two person-identifiers (name or number),"
11860 print " separated by semicolon. Enter "stop" to stop."
11870 line input REQUEST_BUFFER$
11880 end sub
11890 !
11900 sub CHECK_REQUEST (REQUEST_STATUS$, PERSON1_IDENT$, PERSON2_IDENT$)
11910 ! Performs syntactic check on request in buffer
11920 ! and fills in identifiers of the two requested persons.
11930 !
11940 let SEMICOLON_LOCATION = pos (REQUEST BUFFER$, ";")
11950 let PERSON1_IDENT$ = ltrim$ (rtrim$ & (REQUEST BUFFER$ (1 : SEMICOLON_LOCATION - 1)))
11960 let PERSON2_IDENT$ = ltrim$ (rtrim$ & (REQUEST BUFFER$ (SEMICOLON_LOCATION + 1 : len (REQUEST BUFFER$))))
11970 if SEMICOLON_LOCATION = 0 or pos (PERSON2_IDENT$, ";") <> 0 then
11980 let REQUEST_STATUS$ = "must be exactly one semicolon."
11990 elseif PERSON1_IDENT$ = "" then
12000 let REQUEST_STATUS$ = "null field preceding semicolon."
12010 elseif PERSON2_IDENT$ = "" then
12020 let REQUEST_STATUS$ = "null field following semicolon."
12030 else
12040 let REQUEST_STATUS$ = REQUEST_OK$
12050 end if
12060 end sub
12070 !
12080 sub SEARCH_FOR_REQUESTED_PERSONS (PERSON1_IDENT$, PERSON2_IDENT$, & & PERSON1_INDEX, PERSON2_INDEX, & & PERSON1_FOUND, PERSON2_FOUND)
12090 ! SEARCH_FOR_REQUESTED_PERSONS scans through the PERSON array, 12100 ! looking for the two requested PERSONs. Match may be by NAME 12110 ! or unique IDENTIFIER-number 12120 !
12130 let PERSON1_FOUND = 0
12140 let PERSON2_FOUND = 0
12150 let PERSON1_INDEX = 0
12160 let PERSON2_INDEX = 0
12170 for CURRENT = 1 to NUMBER_OF_PERSONS
12180 ! allow identification by name or identifier 12190 if IDENTIFIER$ (CURRENT) = PERSON1_IDENT$ & & or NAME$ (CURRENT) = PERSON1_IDENT$ then
12200 let PERSON1_INDEX = CURRENT
12210 let PERSON1_FOUND = PERSON1_FOUND + 1
12220 end if
12230 if IDENTIFIER$ (CURRENT) = PERSON2_IDENT$ & & or NAME$ (CURRENT) = PERSON2_IDENT$ then
12240 let PERSON2_INDEX = CURRENT
12250 let PERSON2_FOUND = PERSON2_FOUND + 1
12260 end if
12270 next CURRENT
12280 end sub
12290 end ! of main program unit - external procedures follow
12300 !
Finds shortest path (if any) between two PERSONs and traversed in path. PERSON array simulates a directed graph, and algorithm finds shortest path, based on following weights: PARENT-CHILD edge = 1.0
SPouse-SPOUSE edge = 1.8

A node in the graph (= PERSON) has either already been reached, is immediately adjacent to those reached, or farther away.
12760 data 1, 2, 3 ! values for search status
12770 read SEARCHING, SUCCEEDED, FAILED
12780 !
12790 data 1, 2, 3 ! values for sibling proximity
12800 read STEP, HALF, FULL
12810 !
12820 ! The following arrays contain information on key persons.
12830 ! Key persons are the ones in the RELATIONSHIP path which remain
12840 ! after the path is condensed.
12850 !
12860 dim RELATION TO NEXT (300), PERSON_INDEX (300), GENERATION_GAP (300)
12870 dim PROXIMITY (300), COUSIN_RANK (300)
12880 !
12890 ! keeps track of current NEARBY nodes in graph search
12900 dim NEARBY_NODE (300)
12910 !
12920 ! begin main line of execution of FIND_RELATIONSHIP
12930 !
12940 ! initialize PERSON-array for processing -
12950 ! mark all nodes as not seen
12960 for THIS_NODE = 1 to NUMBER_OF_PERSONS
12970   let REACHED_STATUS (THIS_NODE) = NOT_SEEN
12980 next THIS_NODE
12990 !
13000 let THIS_NODE = SOURCE_INDEX
13010 ! mark source node as REACHED
13020 let REACHED_STATUS (THIS_NODE) = REACHED
13030 let DISTANCE_FROM_SOURCE (THIS_NODE) = 0
13040 ! no nearby nodes exist yet
13050 let LAST_NEARBY_INDEX = 0
13060 if THIS_NODE = TARGET_INDEX then
13070   let SEARCH_STATUS = SUCCEEDED
13080 else
13090   let SEARCH_STATUS = SEARCHING
13100 end if
13110 !
13120 ! Loop keeps processing closest-to-source, unREACHED node
13130 ! until target REACHED, or no more connected nodes.
13140 do while SEARCH_STATUS = SEARCHING
13150 ! Process all nodes adjacent to THIS_NODE
13160 ! for THIS_NEIGHBOR = 1 to NEIGHBOR_COUNT (THIS_NODE)
13170 ! call PROCESS_ADJACENT_NODE (THIS_NODE, 
13180 & NEIGHBOR_INDEX (THIS_NODE, THIS_NEIGHBOR), 
13190 & NEIGHBOR_EDGE (THIS_NODE, THIS_NEIGHBOR))
13180 next THIS_NEIGHBOR
13190 ! All nodes adjacent to THIS_NODE are set. Now search for
13200 ! shortest-distance unREACHED (but NEARBY) node to process next.
13210 if LAST_NEARBY_INDEX = 0 then
13220 ! let SEARCH_STATUS = FAILED
13230 else ! determine next node to process
13240 let MINIMAL_DISTANCE = 1.0E+18
13250 ! now find closest unreached node
13260 ! for THIS_NEARBY_INDEX = 1 to LAST_NEARBY_INDEX
13270 let NEXT_NODE = NEARBY_NODE (THIS_NEARBY_INDEX)
13280 ! if DISTANCE_FROM_SOURCE (NEXT_NODE) < MINIMAL_DISTANCE then
13290 ! let BEST_NEARBY_INDEX = THIS_NEARBY_INDEX
13300 ! let MINIMAL_DISTANCE = DISTANCE_FROM_SOURCE (NEXT_NODE)
13310 end if
13320 next THIS_NEARBY_INDEX
13330 ! establish new THIS_NODE
13340 let THIS_NODE = NEARBY_NODE (BEST_NEARBY_INDEX)
13350 ! change THIS_NODE from being NEARBY to REACHED
13360 let REACHED_STATUS (THIS_NODE) = REACHED
13370 ! remove THIS_NODE from NEARBY list
13380 let NEARBY_NODE (BEST_NEARBY_INDEX) = 
13390 & NEARBY_NODE (LAST_NEARBY_INDEX)
13400 if THIS_NODE = TARGET_INDEX then let SEARCH_STATUS = SUCCEEDED
13410 end if
13420 loop
13430 !
13440 ! Shortest path between PERSONs now established. Next task is
13450 ! to translate path to English description of RELATIONSHIP.
13460 if SEARCH_STATUS = FAILED then
13470 print ""; NAME$ (TARGET_INDEX); " is not related to "; & 
13480 & NAME$ (SOURCE_INDEX)
13490 else
13500 ! success - parse path to find and display RELATIONSHIP
13510 call RESOLVE_PATH_TO_ENGLISH
13520 call COMPUTE_COMMON_GENES (SOURCE_INDEX, TARGET_INDEX, 
13530 & IDENTIFIERS$, NEIGHBOR_COUNT, NEIGHBOR_INDEX, NEIGHBOR_EDGE, 
13540 & DESCENDANT_IDENTIFIERS$, DESCENDANT_GENES)
13550 end if
13560 exit sub
13570 !
13580 ! end of main line of execution of FIND_RELATIONSHIP
SUB PROCESS_ADJACENT_NODE (BASE_NODE, NEXT_NODE, NEXT_BASE_EDGE)

NEXT_NODE is adjacent to last-REACHED node (= BASE_NODE).

if NEXT_NODE already REACHED, do nothing.

If previously seen, check whether path thru BASE_NODE is
shorter than current path to NEXT_NODE, and if so re-link
next to base.

If not previously seen, link next to base node.

if NEXT_BASE_EDGE = SPOUSE then
    let WEIGHT_THIS_EDGE = 1.8
else
    let WEIGHT_THIS_EDGE = 1.0
end if

if REACHED_STATUS (NEXT_NODE) <> REACHED then
    let DISTANCE_THRU_BASE_NODE = WEIGHT_THIS_EDGE + DISTANCE_FROM_SOURCE (BASE_NODE)
    if REACHED_STATUS (NEXT_NODE) = NOT_SEEN then
        let REACHED_STATUS (NEXT_NODE) = NEARBY
        let LAST_NEARBY_INDEX = LAST_NEARBY_INDEX + 1
        let NEARBY_NODE (LAST_NEARBY_INDEX) = NEXT_NODE
        ! link next to base by re-setting its predecessor index to
        ! as it is through base node.
        let DISTANCE_FROM_SOURCE (NEXT_NODE) = DISTANCE_THRU_BASE_NODE
        let PATH_PREDECESSOR (NEXT_NODE) = BASE_NODE
        let EDGE_TO_PREDECESSOR (NEXT_NODE) = NEXT_BASE_EDGE
    else ! REACHED_STATUS = NEARBY
        if DISTANCE_THRU_BASE_NODE < DISTANCE_FROM_SOURCE (NEXT_NODE) then
            ! link next to base by re-setting its predecessor index to
            ! point to base, note type of edge, and re-set distance
            ! as it is through base node.
            let DISTANCE_FROM_SOURCE (NEXT_NODE) = DISTANCE_THRU_BASE_NODE
            let PATH_PREDECESSOR (NEXT_NODE) = BASE_NODE
            let EDGE_TO_PREDECESSOR (NEXT_NODE) = NEXT_BASE_EDGE
        end if
    end if
end if
end sub
sub RESOLVE_PATH_TO_ENGLISH
RESOLVE_PATH_TO_ENGLISH condenses the shortest path to a series of RELATIONSHIPS for which there are English descriptions.
Key persons are the ones in the RELATIONSHIP path which remain after the path is condensed.
print " Shortest path between identified persons: ",
let THIS_NODE = TARGET_INDEX
print path and initialize KEY_PERSON array from path elements,
as shortest path is traversed.
let KEY_INDEX = 1
do until THIS_NODE = SOURCE_INDEX
let PERSON_INDEX (KEY_INDEX) = THIS_NODE
let PROXIMITY (KEY_INDEX) = FULL
let RELATION_TO_NEXT (KEY_INDEX) = EDGE_TO_PREDECESSOR (THIS_NODE)
print "; NAME$ (THIS_NODE); tab(23); "is ";
if EDGE_TO_PREDECESSOR (THIS_NODE) = SPOUSE then
let GENERATION_GAP (KEY_INDEX) = 0
else
let GENERATION_GAP (KEY_INDEX) = 1
if EDGE_TO_PREDECESSOR (THIS_NODE) = PARENT then
print "parent of"
else ! edge is child-type
print "child of"
end if
else
let KEY_INDEX = KEY_INDEX + 1
let THIS_NODE = PATH_PREDECESSOR (THIS_NODE)
loop
print "; NAME$ (THIS_NODE)
let PERSON_INDEX (KEY_INDEX) = THIS_NODE
let RELATION_TO_NEXT (KEY_INDEX) = NULL_RELATION
let RELATION_TO_NEXT (KEY_INDEX + 1) = NULL_RELATION
14340 ! Resolve CHILD-PARENT and CHILD-SPOUSE-PARENT relations
14350 ! to SIBLING relations.
14360 let KEY_INDEX = 1
14370 do until RELATION_TO_NEXT (KEY_INDEX) = NULL_RELATION
14380   if RELATION_TO_NEXT (KEY_INDEX) = CHILD then
14390     let LATER_KEY_RELATION = RELATION_TO_NEXT (KEY_INDEX + 1)
14400     if LATER_KEY_RELATION = PARENT then
14410       ! found either full or half SIBLINGS
14420       let GENERATION_GAP (KEY_INDEX) = 0
14430       let RELATION_TO_NEXT (KEY_INDEX) = SIBLING
14440       let PROXIMITY (KEY_INDEX) = &
14450       & SIBLING_PROXIMITY (PERSON_INDEX (KEY_INDEX), &
14460       & PERSON_INDEX (KEY_INDEX + 2))
14470       call CONDENSE_KEY_PERSONS (KEY_INDEX, 1)
14480     else
14490       if LATER_KEY_RELATION = SPOUSE and
14500       & RELATION_TO_NEXT (KEY_INDEX + 2) = PARENT then
14510       ! found step-siblings
14520       let GENERATION_GAP (KEY_INDEX) = 0
14530       let RELATION_TO_NEXT (KEY_INDEX) = SIBLING
14540       let PROXIMITY (KEY_INDEX) = STEP
14550       call CONDENSE_KEY_PERSONS (KEY_INDEX, 2)
14560     end if
14570   end if
14580 end if
14590 let KEY_INDEX = KEY_INDEX + 1
14600 loop
14610 ! Resolve CHILD-CHILD-... and PARENT-PARENT-... relations to
14620 ! direct descendant or ancestor relations.
14630 let KEY_INDEX = 1
14640 do until RELATION_TO_NEXT (KEY_INDEX) = NULL_RELATION
14650   if RELATION_TO_NEXT (KEY_INDEX) = CHILD or
14660   & RELATION_TO_NEXT (KEY_INDEX) = PARENT then
14670     let LATER_KEY_INDEX = KEY_INDEX + 1
14680     do while RELATION_TO_NEXT (LATER_KEY_INDEX) &
14690     & = RELATION_TO_NEXT (KEY_INDEX)
14700     let LATER_KEY_INDEX = LATER_KEY_INDEX + 1
14710 loop
14720 let GENERATION_COUNT = LATER_KEY_INDEX - KEY_INDEX
14730 if GENERATION_COUNT > 1 then ! compress generations
14740   let GENERATION_GAP (KEY_INDEX) = GENERATION_COUNT
14750   call CONDENSE_KEY_PERSONS (KEY_INDEX, GENERATION_COUNT - 1)
14760 end if
14770 end if
14780 let KEY_INDEX = KEY_INDEX + 1
14790 loop
14800 !
14770 ! Resolve CHILD-SIBLING-PARENT to COUSIN,
14780 ! CHILD-SIBLING to NEPHEW,
14790 ! SIBLING-PARENT to UNCLE.
14800 let KEY_INDEX = 1
14810 do until RELATION_TO_NEXT (KEY_INDEX) = NULL_RELATION
14820 let LATER_KEY_RELATION = RELATION_TO_NEXT (KEY_INDEX + 1)
14830 if RELATION_TO_NEXT (KEY_INDEX) = CHILD &
               and LATER_KEY_RELATION = SIBLING then
14840  ! found COUSIN or NEPHEW
14850  if RELATION_TO_NEXT (KEY_INDEX + 2) = PARENT then
14860  ! found cousin
14870  let GAP1 = GENERATION_GAP (KEY_INDEX)
14880  let GAP2 = GENERATION_GAP (KEY_INDEX + 2)
14890  let COUSIN_RANK (KEY_INDEX) = min (GAP1, GAP2)
14900  let GENERATION_GAP (KEY_INDEX) = abs (GAP1 - GAP2)
14910  let PROXIMITY (KEY_INDEX) = PROXIMITY (KEY_INDEX + 1)
14920  let RELATION_TO_NEXT (KEY_INDEX) = COUSIN
14930  call CONDENSE_KEY_PERSONS (KEY_INDEX, 2)
14940 else  ! found NEPHEW
14950  let PROXIMITY (KEY_INDEX) = PROXIMITY (KEY_INDEX + 1)
14960  let RELATION_TO_NEXT (KEY_INDEX) = NEPHEW
14970  call CONDENSE_KEY_PERSONS (KEY_INDEX, 1)
14980 end if
14990 else
15000  if RELATION_TO_NEXT (KEY_INDEX) = SIBLING &
               and LATER_KEY_RELATION = PARENT then UNCLE
15010  ! found UNCLE
15020  let GENERATION_GAP (KEY_INDEX) =
               &
               GENERATION_GAP (KEY_INDEX + 1)
15030  let RELATION_TO_NEXT (KEY_INDEX) = UNCLE
15040  call CONDENSE_KEY_PERSONS (KEY_INDEX, 1)
15050 end if
15060 end if
15070 let KEY_INDEX = KEY_INDEX + 1
15080 loop
15090 !
Loop below will pick out valid adjacent strings of elements to be printed. KEY_INDEX points to first element, LATER_KEY_INDEX to last element, and PRIMARY_INDEX to the element which determines the primary English word to be used. Associativity of adjacent elements in condensed table is based on English usage.

print " Condensed path:"
let KEY_INDEX = 1
do until RELATION_TO_NEXT (KEY_INDEX) = NULL_RELATION
  let KEY_RELATION = RELATION_TO_NEXT (KEY_INDEX)
  let LATER_KEY_INDEX, PRIMARY_INDEX = KEY_INDEX
  if RELATION_TO_NEXT (KEY_INDEX + 1) <> NULL_RELATION then
    ! seek multi-element combination
    let ANOTHER_ELEMENT_POSSIBLE = TRUE
    if KEY_RELATION = SPOUSE then
      let LATER_KEY_INDEX = LATER_KEY_INDEX + 1
      let PRIMARY_INDEX = LATER_KEY_INDEX
      if RELATION_TO_NEXT (LATER_KEY_INDEX) = SIBLING or &
      & RELATION_TO_NEXT (LATER_KEY_INDEX) = COUSIN then
        ! nothing can follow spouse-sibling or spouse-cousin
        let ANOTHER_ELEMENT_POSSIBLE = FALSE
      end if
    end if
    ! PRIMARY_INDEX is now correctly set. Next if-statement determines if a following SPOUSE relation should be appended to this combination or left for the next combination.
    if RELATION_TO_NEXT (PRIMARY_INDEX + 1) = SPOUSE and &
    & ANOTHER_ELEMENT_POSSIBLE = TRUE then
      ! Only a SPOUSE can follow a Primary
      ! check primary preceding and following SPOUSE.
      let PRIMARY_RELATION = RELATION_TO_NEXT (PRIMARY_INDEX)
      let NEXT_PRIMARY_RELATION = RELATION_TO_NEXT (PRIMARY_INDEX + 2)
      if (NEXT_PRIMARY_RELATION = NEPHEW or &
      & NEXT_PRIMARY_RELATION = COUSIN or &
      & NEXT_PRIMARY_RELATION = NULL_RELATION) &
      & or (PRIMARY_RELATION = NEPHEW) &
      & or ( PRIMARY_RELATION = SIBLING or &
      & PRIMARY_RELATION = PARENT) &
      & and NEXT_PRIMARY_RELATION <> UNCLE ) then
        ! append following SPOUSE with this combination
        let LATER_KEY_INDEX = LATER_KEY_INDEX + 1
      end if
    end if
  end if
  end do
end if
multi-element combination
call DISPLAY_RELATION (KEY_INDEX, LATER_KEY_INDEX, PRIMARY_INDEX)
let KEY_INDEX = LATER_KEY_INDEX + 1
loop
print " "; NAME$ (PERSON_INDEX (KEY_INDEX))
end sub
end of RESOLVE_PATH_TO_ENGLISH
function SIBLING_PROXIMITY (INDEX1, INDEX2)
! Determines whether two PERSONs are full siblings, i.e.,
! have the same two parents.
if RELATIVE_IDENTIFIER$ (INDEX1, FATHER_IDENT) <> NULL_IDENT$ and
& RELATIVE_IDENTIFIER$ (INDEX1, MOTHER_IDENT) <> NULL_IDENT$ and
& RELATIVE_IDENTIFIER$ (INDEX1, FATHER_IDENT) =
& RELATIVE_IDENTIFIER$ (INDEX2, FATHER_IDENT) and
& RELATIVE_IDENTIFIER$ (INDEX1, MOTHER_IDENT) =
& RELATIVE_IDENTIFIER$ (INDEX2, MOTHER_IDENT) then
let SIBLING_PROXIMITY = FULL
else
let SIBLING_PROXIMITY = HALF
end if
end function
!
sub CONDENSE_KEY_PERSONS (AT_INDEX, GAP_SIZE)
!
sub DISPLAY_RELATION (FIRST_INDEX, LAST_INDEX, PRIMARY_INDEX)
!
set THIS_PROXIMITY
if (PRIMARY_RELATION = PARENT and FIRST_RELATION = SPOUSE) or
& (PRIMARY_RELATION = CHILD and LAST_RELATION = SPOUSE) then
let THIS_PROXIMITY = STEP
else
if PRIMARY_RELATION = SIBLING or
& PRIMARY_RELATION = UNCLE or
& PRIMARY_RELATION = NEPHEW or
& PRIMARY_RELATION = COUSIN then
let THIS_PROXIMITY = PROXIMITY (PRIMARY_INDEX)
else
let THIS_PROXIMITY = FULL
end if
16000 **!** set THIS\_GENERATION\_GAP
16010 if PRIMARY\_RELATION = PARENT or &
16015 & PRIMARY\_RELATION = CHILD or &
16020 & PRIMARY\_RELATION = UNCLE or &
16025 & PRIMARY\_RELATION = NEPHEW or &
16030 & PRIMARY\_RELATION = COUSIN then
16035 let THIS\_GENERATION\_GAP = GENERATION\_GAP (PRIMARY\_INDEX)
16040 else
16045 let THIS\_GENERATION\_GAP = 0
16050 end if
16060 **!**
16070 **!** set INLAW
16080 if (FIRST\_RELATION = SPOUSE) and &
16085 & (PRIMARY\_RELATION = SIBLING or &
16090 & PRIMARY\_RELATION = CHILD or &
16095 & PRIMARY\_RELATION = NEPHEW or &
16100 & PRIMARY\_RELATION = COUSIN) then
16105 let INLAW = TRUE
16110 else
16115 if (LAST\_RELATION = SPOUSE) and &
16120 & (PRIMARY\_RELATION = SIBLING or &
16125 & PRIMARY\_RELATION = PARENT or &
16130 & PRIMARY\_RELATION = UNCLE or &
16135 & PRIMARY\_RELATION = COUSIN) then
16140 let INLAW = TRUE
16150 else
16155 let INLAW = FALSE
16160 end if
16170 **!**
16180 **!** set THIS\_COUSIN\_RANK
16190 if PRIMARY\_RELATION = COUSIN then
16200 let THIS\_COUSIN\_RANK = COUSIN\_RANK (PRIMARY\_INDEX)
16210 else
16215 let THIS\_COUSIN\_RANK = 0
16220 end if
16230 **!**
16240 **!**
16250 **!** parameters are set – now generate display.
16260 **!**
16270 print " "; NAME$ (PERSON\_INDEX (FIRST\_INDEX)); tab(23); " is ";
16280 if PRIMARY\_RELATION = PARENT or &
16285 & PRIMARY\_RELATION = CHILD or &
16290 & PRIMARY\_RELATION = UNCLE or &
16295 & PRIMARY\_RELATION = NEPHEW then
16300 ! print generation-qualifier
16310 if THIS\_GENERATION\_GAP >= 3 then
16315 print "great";
16320 if THIS\_GENERATION\_GAP > 3 then
16325 print "*"; str$ (THIS\_GENERATION\_GAP - 2);
16330 end if
16340 end if
16350 print "-";
16360 end if
16370 if THIS\_GENERATION\_GAP >= 2 then print "grand-";
elseif PRIMARY_RELATION = COUSIN and THIS_COUSIN_RANK > 1 then
    print str$ (THIS_COUSIN_RANK);
    select case mod (THIS_COUSIN_RANK, 10)
    case 1
        print "st ";
    case 2
        print "nd ";
    case 3
        print "rd ";
    case else
        print "th ";
    end select
end if

if THIS_PROXIMITY = STEP then
    print "step-";
elseif THIS_PROXIMITY = HALF then
    print "half-";
else
    let THIS_GENDER$ = GENDER$ (PERSON_INDEX (FIRST_INDEX))
    select case PRIMARY_RELATION
    case 1 ! PARENT
        if THIS_GENDER$ = MALE$ then print "father"; else print "mother";
    case 2 ! CHILD
        if THIS_GENDER$ = MALE$ then print "son"; else print "daughter";
    case 3 ! SPOUSE
        if THIS_GENDER$ = MALE$ then print "husband"; else print "wife";
    case 4 ! SIBLING
        if THIS_GENDER$ = MALE$ then print "brother"; else print "sister";
    case 5 ! UNCLE
        if THIS_GENDER$ = MALE$ then print "uncle"; else print "aunt";
    case 6 ! NEPHEW
        if THIS_GENDER$ = MALE$ then print "nephew"; else print "niece";
    case 7 ! COUSIN
        print "cousin";
    case else
        print "null";
    end select
    print "of"
end sub ! end of internal sub DISPLAY_RELATION
end sub ! end of external sub FIND_RELATIONSHIP
16930 ! ---- program-unit number 3 ----
16940 !
16950 external sub COMPUTE_COMMON_GENES (INDEX1, INDEX2, IDENTIFIER$ ()),
&
&   NEIGHBOR_COUNT (), NEIGHBOR_INDEX (,), NEIGHBOR_EDGE (),
&
&   DESCENDANT_IDENTIFIER$ (,), DESCENDANT_GENES ()
16960 !
16970 ! COMPUTE_COMMON_GENES assumes that each ancestor contributes
16980 ! half of the genetic material to a person. It finds common
16990 ! ancestors between two persons and computes the expected
17000 ! value of the PROPORTION of common material.
17010 !
17020 declare sub ZERO_PROPORTION, MARK_PROPORTION, CHECK_COMMON_PROPORTION
17030 !
17035 option base 1
17040 !
17045 data 1, 2, 3, 4, 5, 6, 7, 8
17050 read PARENT, CHILD, SPOUSE, SIBLING, UNCLE, NEPHEW
17055 read COUSIN, NULL_RELATION
17057 !
17060 ! Begin main line of execution of COMPUTE_COMMON_GENES
17065 !
17070 ! First zero out all ancestors to allow adding. This is necessary
17075 ! because there might be two paths to an ancestor.
17080 call ZERO_PROPORTION (INDEX1, 0)
17090 ! now mark with shared PROPORTION
17100 call MARK_PROPORTION (IDENTIFIER$ (INDEX1), 1.0, INDEX1, 0)
17110 let COMMON_PROPORTION = 0.0
17120 call CHECK_COMMON_PROPORTION (COMMON_PROPORTION, &
&   IDENTIFIER$ (INDEX1), 1.0, 0.0, INDEX2, 0)
17130 print using " Proportion of common genetic material = #.#####^":
&
&   COMMON_PROPORTION
17140 !
17150 ! End main line of execution of COMPUTE_COMMON_GENES
17160 !
17170 sub ZERO_PROPORTION (ZERO_INDEX, THIS_NEIGHBOR)
17180 ! ZERO_PROPORTION recursively seeks out all ancestors and
17190 ! zeros them out
17200 let DESCENDANT_GENES (ZERO_INDEX) = 0.0
17210 for THIS_NEIGHBOR = 1 to NEIGHBOR_COUNT (ZERO_INDEX)
17220   if NEIGHBOR_EDGE (ZERO_INDEX, THIS_NEIGHBOR) = PARENT then
17230     call ZERO_PROPORTION &
&   (NEIGHBOR_INDEX (ZERO_INDEX, THIS_NEIGHBOR), 0)
17240   end if
17250 next THIS_NEIGHBOR
17260 end sub ! ZERO_PROPORTION
17270 !
17280 sub MARK_PROPORTION (MARKER$, PROPORTION, MARKED_INDEX, THIS_NEIGHBOR)
17290 ! MARK_PROPORTION recursively seeks out all ancestors and
17300 ! marks them with the sender's PROPORTION of shared
17310 ! genetic material. This PROPORTION is diluted by one-half
17320 ! for each generation
17330 let DESCENDANT_IDENTIFIER$ (MARKED_INDEX) = MARKER$
17340 let DESCENDANT_GENES (MARKED_INDEX) = &
17350 for THIS_NEIGHBOR = 1 to NEIGHBOR_COUNT (MARKED_INDEX)
17360 if NEIGHBOR_EDGE (MARKED_INDEX, THIS_NEIGHBOR) = PARENT then
17370 call MARK_PROPORTION (MARKER$, PROPORTION / 2.0, &
17380 ! & NEIGHBOR_INDEX (MARKED_INDEX, THIS_NEIGHBOR), 0)
17390 end if
17400 end sub ! MARK_PROPORTION
17410 !
17420 sub CHECK_COMMON_PROPORTION (COMMON_PROPORTION, MATCH_IDENTIFIER$, &
17430 ! PROPORTION, ALREADY_COUNTED, CHECK_INDEX, THIS_NEIGHBOR)
17440 ! CHECK_COMMON_PROPORTION searches all the ancestors of
17450 ! CHECK_INDEX to see if any have been marked, and if so
17460 ! adds the appropriate amount to COMMON_PROPORTION
17470 ! Increment COMMON_PROPORTION by the contribution of
17480 ! this common ancestor, but discount for the contribution
17490 ! of less remote ancestors already counted
17500 let THIS_CONTRIBUTION = DESCENDANT_GENES (CHECK_INDEX) * PROPORTION
17510 let COMMON_PROPORTION = COMMON_PROPORTION &
17520 ! & THIS_CONTRIBUTION - ALREADY_COUNTED
17530 else
17540 ! let THIS_CONTRIBUTION = 0.0
17550 end if
17560 for THIS_NEIGHBOR = 1 to NEIGHBOR_COUNT (CHECK_INDEX)
17570 if NEIGHBOR_EDGE (CHECK_INDEX, THIS_NEIGHBOR) = PARENT then
17580 call CHECK_COMMON_PROPORTION (COMMON_PROPORTION, &
17590 ! & MATCH_IDENTIFIER$, PROPORTION / 2.0, &
17600 ! & THIS_CONTRIBUTION / 4.0, &
17610 ! & NEIGHBOR_INDEX (CHECK_INDEX, THIS_NEIGHBOR), 0)
17620 end if
17630 next THIS_NEIGHBOR
17640 end sub ! end of internal sub CHECK_COMMON_PROPORTION
17650 end sub ! end of external sub COMPUTE_COMMON_GENES
4.0 C

The identifiers NULL and FILE are capitalized, even though they are supplied by the standard run-time library, because identifiers in C are case-sensitive, e.g., "null" is not equivalent to "NULL".

/* Bring in standard routines for run-time support */
#include <stdio.h>

/* Global types and objects */

typedef short int BOOLEAN;
#define TRUE 1
#define FALSE 0
#define EQUALS 0

#define NULL_ID "000"
#define NULL_chr '\0'

#define MAX_PERS 300
#define NAME_LEN 20
  /* every PERSON has a unique 3-digit IDENT */
#define ID_LEN 3
#define BUF_LEN 60

/* Use "+ 1" when treating type as variable-length - extra character used to hold NULL_chr termination character. */
typedef char NAME_TYP [NAME_LEN + 1];
typedef char BUF_TYPE [BUF_LEN + 1];
typedef char MSG_TYPE [40 + 1];
typedef char ID_TYPE [ID_LEN + 1];

typedef int INDX_TYP, COUNTER;

/* each PERSON's record in the file identifies at most three others directly related: father, mother, and spouse */
typedef short int GIVEN_ID;
#define FATHR_ID 0
#define MOTHUR_ID 1
#define SPOUS_ID 2
#define MAX_GVEN 3

typedef ID_TYPE REL.ARRAY [MAX_GVEN];
#define REQ_OK "Request OK"
#define REQ_STOP "stop"

typedef char GNDR_TYP;
#define MALE 'M'
#define FEMALE 'F'
typedef unsigned int REL_TYPE;
/* Values defined as octal powers of two to facilitate comparisons of one relation with several possibilities. */
#define PARENT 0001
#define CHILD 0002
#define SPOUSE 0004
#define SIBLING 0010
#define UNCLE 0020
#define NEPHEW 0040
#define COUSIN 0100
#define NULL_REL 0200

/* directed edges in the graph are of a given type */
typedef REL_TYPE EDG_TYPE;
/* A node in the graph (= PERSON) has either already been reached, is immediately adjacent to those reached, or farther away. */
typedef short int REACH_TYPE;
#define REACHED 1
#define NEARBY 2
#define NOT_SEEN 3

/* each PERSON has a linked list of adjacent nodes, called neighbors */
typedef struct NBR_NODE
{  IND_TYPE NBR_DEX;
   EDG_TYPE NBR_EDGE;
   struct NBR_NODE *NEXT_NBR;
} NBR_REC, *NBR_PTR;

/* All relationships are captured in the directed graph of which each record is a node. */
typedef struct
{
  /* static information - filled from PEOPLE file: */
  NAME_TYPE NAME;
  ID_TYPE IDENT;
  GNDR_TYPE GENDER;
  /* IDENTs of immediate relatives - father, mother, spouse */
  REL_ARRY REL_ID;
  /* head of linked list of adjacent nodes */
  NBR_PTR NBR_HDR;
  /* data used when traversing graph to resolve user request: */
  float DIST_SRC;
  IND_TYPE PATHPRED;
  EDG_TYPE EDG_PRED;
  REACH_TYPE REACH_ST;
  /* data used to compute common genetic material */
  ID_TYPE DSC_ID;
  float DSC_GENE;
} PERS_REC;
/* the PERSON array is the central repository of information about inter-relationships. */
PERS_REC PERSON [MAX_PERS];
INDX_TYP NUM_PERS;

/* Key persons are the ones in the REL_SHIP path which remain after the path is condensed. */
typedef short int SIB_TYPE;
#define STEP 1
#define HALF 2
#define FULL 3
typedef struct
   { REL_TYPE REL_NEXT;
   INDX_TYP PERS_DEX;
   COUNTER GEN_GAP;
   SIB_TYPE PROXIMTY;
   COUNTER CUZ_RANK;
   }
   KEY_REC;

/****** Main line of execution RELATE ******/
main()
{
      /* These variables are used when establishing the PERSON array from the PEOPLE file. */
      FILE *fopen(), *PEOPLE;
      register INDX_TYP CURRENT, PREVIOUS;
      ID_TYPE PREV_ID, CUR_ID;
      GIVEN_ID REL_SHIP;
      char INF_BUF[100];

      /* These variables are used to accept and resolve requests for REL_SHIP information. */
      COUNTER SEMI_LOC;
      BUF_TYPE REQ_BUF;
      BUF_TYPE P1_IDENT, P2_IDENT;
      COUNTER P1_FOUND, P2_FOUND;
      MSG_TYPE ERR_MSG;
      INDX_TYP P1_INDEX, P2_INDEX;
/* *** execution of main sequence begins here *** */

PEOPLE = fopen("PEOPLE.DAT", "r");
/* This loop reads in the PEOPLE file and constructs the PERSON
array from it (one PERSON == one record == one array entry).
As records are read in, links are constructed to represent the
PARENT-CHILD or SPOUSE REL_SHIP. The array then implements
a directed graph which is used to satisfy subsequent user
requests. The file is assumed to be correct - no validation
is performed on it. */

READ_PEO:
for (CURRENT = 0; ; CURRENT++)
{
    /* copy direct information from file to array */
    if (FXD_GETC (PERSON [CURRENT] . NAME, PEOPLE, NAME_LEN)
        == EOF)
        break;
    FXD_GETC (PERSON [CURRENT] . IDENT, PEOPLE, ID_LEN);
    FXD_GETC (&(PERSON [CURRENT] . GENDER), PEOPLE, 1);
    for (REL_SHIP = FATHR_ID; REL_SHIP < MAX_GVEN; REL_SHIP++)
        FXD_GETC (PERSON [CURRENT] . REL_ID [REL_SHIP], PEOPLE, ID_LEN);
    /* flush remainder of record */
    fgets (1NP_BUF, 100, PEOPLE);
    /* Location of adjacent persons as yet undetermined */
    PERSON [CURRENT] . NBR_HDR = NULL;
    /* Descendants as yet undetermined */
    strcpy (PERSON [CURRENT] . DSC_ID, NULL_ID);
    /* Compare this PERSON against all previously entered PERSONs
    to search for REL_SHIPS. */
    strcpy (CUR_ID, PERSON [CURRENT] . IDENT);
    CMP_PREV:
    for (PREVIOUS = 0; PREVIOUS < CURRENT; PREVIOUS++)
    {
        strcpy (PREV_ID, PERSON [PREVIOUS] . IDENT);
        /* Search for father, mother, or spouse relationship in
        either direction between this and PREVIOUS PERSON.
        Assume at most one REL_SHIP exists. */
        TRY_RELS:
        for (REL_SHIP = FATHR_ID; REL_SHIP < MAX_GVEN; REL_SHIP++)
        {
            if (STREQ (PREV_ID, PERSON [CURRENT] . REL_ID [REL_SHIP]))
                { LINK_REL (CURRENT, REL_SHIP, PREVIOUS);
                break;
            } else
                if (STREQ (CUR_ID, PERSON [PREVIOUS] . REL_ID [REL_SHIP]))
                    { LINK_REL (PREVIOUS, REL_SHIP, CURRENT);
                        break;
                    } /* end TRY_RELS */
                } /* end CMP_PREV */
        } /* end READ_PEO */
    NUM_PERS = CURRENT;
    fclose (PEOPLE);
/* PERSON array is now loaded and edges between immediate relatives (PARENT-CHILD or SPOUSE-SPOUSE) are established.

While-loop accepts requests and finds REL_SHIP (if any) between pairs of PERSONS. */

PROC_REQ:
while (TRUE)
{
PROMPT (REQ_BUF);
if (STREQ (REQ_BUF, REQ_STOP))
break;
SEMI_LOC = CHK_RQST (REQ_BUF, ERR_MSG);

/* Syntax check of request completed. Now either display error * 
message or search for the two PERSONS. */
if (STREQ (ERR_MSG, REQ_OK))
{
  /* Request syntactically correct - search for requested PERSONS. */
  REQ_BUF [SEMI_LOC] = NULL CHR;
  BUF_PERS (REQ_BUF, 0, P1_IDENT);
  BUF_PERS (REQ_BUF, SEMI_LOC + 1, P2_IDENT);
  SEEK_PER (P1_IDENT, P2_IDENT, &P1_INDEX, &P2_INDEX, &P1_FOUND, &P2_FOUND);

  if (P1_FOUND == 1 && P2_FOUND == 1)
  /* Exactly one match for each PERSON - proceed to 
determine REL_SHIP, if any. */
  if (P1_INDEX == P2_INDEX)
    printf ("%ls is identical to %8s \n", 
            PERSON [P1_INDEX]. NAME,
            (PERSON [P1_INDEX]. GENDER == MALE) ? 
                "himself." : "herself.");
  else
    FIND_REL (P1_INDEX, P2_INDEX);
  else /* either not found or more than one found */
    if (P1_FOUND == 0)
      printf ("First person not found.\n");
    else if (P1_FOUND > 1)
      
      printf ("Duplicate names for first person -");
      printf (" use numeric identifier.\n");
    
    if (P2_FOUND == 0)
      printf ("Second person not found.\n");
    else if (P2_FOUND > 1)
      
      printf ("Duplicate names for second person -");
      printf (" use numeric identifier.\n");
  
}
/* end processing of syntactically legal request */
else
  printf ("Incorrect request format: %ls \n", ERR_MSG);
} /* end PROC_REQ loop */
printf ("End of relation-finder. \n");

/* End of main line of RELATE */
/* procedures under RELATE */

FXD_GETC (RECEIVER, SENDING, GET_LEN)

char *RECEIVER;
FILE *SENDING;
int GET_LEN;

{ register int CHAR_CNT;

for (CHAR_CNT = 0;
CHAR_CNT < GET_LEN && (*RECEIVER++ = getc (SENDING)) != EOF ; )
if (CHAR_CNT >= GET_LEN)
{
    *RECEIVER = NULL_CHR;
    return !EOF;
}
else
    return EOF;
}

STREQ (STRING1, STRING2)
/* compare for equality, ignore trailing spaces */

register char *STRING1, *STRING2;

{ register char *LONGER;

for (; *STRING1 == *STRING2; STRING1++, STRING2++)
    if (*STRING1 == NULL_CHR)
        return TRUE;
if (*STRING1 == NULL_CHR)
    LONGER = STRING2;
else
    if (*STRING2 == NULL_CHR)
        LONGER = STRING1;
else
    return FALSE;
for (; *LONGER++ == ' '; )
    return (*--LONGER == NULL_CHR);
LINK_REL (FROM_DEX, REL_SHIP, TO_INDEX)
    /* establishes cross-indexing between immediately related PERSONs. */
    register INDX_TYP FROM_DEX, TO_INDEX;
    register GIVEN_ID REL_SHIP;

    { /* execution of LINK_REL */
      if (REL_SHIP == SPOUS_ID)
        {
          LINK_ONE (FROM_DEX, SPOUSE, TO_INDEX);
          LINK_ONE (TO_INDEX, SPOUSE, FROM_DEX);
        }
      else /* REL_SHIP is father or mother */
        {
          LINK_ONE (FROM_DEX, PARENT, TO_INDEX);
          LINK_ONE (TO_INDEX, CHILD, FROM_DEX);
        }
    }

LINK_ONE (FROM_DEX, THIS_EDG, TO_INDEX)
    /* Establishes the NBR_REC from one PERSON to another */
    INDX_TYP FROM_DEX, TO_INDEX;
    EDG_TYPE THIS_EDG;

    { register NBR_PTR NEW_NBR;

      NEW_NBR = (NBR_REC *) calloc(1, sizeof(NBR_REC));
      NEW_NBR -> NBR_DEX = TO_INDEX;
      NEW_NBR -> NBR_EDGE = THIS_EDG;
      NEW_NBR -> NEXT_NBR = PERSON [FROM_DEX].NBR_HDR;
      PERSON [FROM_DEX].NBR_HDR = NEW_NBR;
    }

PROMPT (REQ_BUF)
    /* Issues prompt for user-request, reads in request,
       blank-fills buffer, and skips to next line of input. */
    BUF_TYPE REQ_BUF;

    {
      printf (" \n");
      printf (" ----------------------------------------------------\n");
      printf (" Enter two person-identifiers (name or number),\n");
      printf (" separated by semicolon. Enter \"stop\" to stop.\n");
      fgets (REQ_BUF, BUF_LEN, stdin);
      for ( ; *REQ_BUF++ != '\n' ; ) ;
      *---REQ_BUF = '\0';
    }
CHK_RQST (REQ_BUF, REQ_STAT)
   /* Performs syntactic check on request in buffer. */

   BUF_TYPE        REQ_BUF;
   MSG_TYPE        REQ_STAT;

   COUNTER        SEMI_LOC  = 1,
                   SEMI_CNT  = 0;
   register COUNTER BUF_DEX;
   BOOLEAN        P1_EXIST = FALSE,
                   P2_EXIST = FALSE;

   strcpy (REQ_STAT, REQ_OK);
   for (BUF_DEX = 0; BUF_DEX < BUF_LEN && REQ_BUF [BUF_DEX]; BUF_DEX++)
   {  
      if (REQ_BUF [BUF_DEX] != '\n')
         if (REQ_BUF [BUF_DEX] == ';')
            {                    
               SEMI_LOC = BUF_DEX;
               SEMI_CNT  = SEMI_CNT + 1;
            }
      else /* Check for non-blanks before/after semicolon. */
         if (SEMI_CNT < 1)
            P1_EXIST = TRUE;
         else
            P2_EXIST = TRUE;
   }

   /* set REQ_STAT, based on results of scan of REQ_BUF. */
   if (SEMI_CNT != 1)
      strcpy (REQ_STAT, "must be exactly one semicolon.");
   else if ( ! P1_EXIST)
      strcpy (REQ_STAT, "null field preceding semicolon.");
   else if ( ! P2_EXIST)
      strcpy (REQ_STAT, "null field following semicolon.");
   return SEMI_LOC;

BUF_PERS (REQ_BUF, BUF_DEX, PERS_ID)
   /* fills in the PERS_ID from the designated portion of the REQ_BUF, deleting leading blanks. */

   BUF_TYPE        REQ_BUF;
   register COUNTER BUF_DEX;
   NAME_TYP        PERS_ID;

   for ( ; REQ_BUF [BUF_DEX++] == '\n'; )
      strcpy (PERS_ID, &REQ_BUF [--BUF_DEX]);
SEEK_PER (P1_IDENT, P2_IDENT, P1_INDEX, P2_INDEX,
     P1_FOUND, P2_FOUND)
/* SEEK_PER scans through the PERSON array,
   looking for the two requested PERSONs. Match may be by NAME
   or unique IDENT-number. */

BUF_TYPE       P1_IDENT, P2_IDENT;
INDX_TYP       #P1_INDEX, #P2_INDEX;
COUNTER        #P1_FOUND, #P2_FOUND;

{ register INDX_TYP CURRENT;

  #P1_INDEX = 0;
  #P2_INDEX = 0;
  #P1_FOUND = 0;
  #P2_FOUND = 0;

SCAN_PER:
  for (CURRENT = 0; CURRENT < NUM_PERS; CURRENT++)
    { /* allow identification by name or number. */
      if (STREQ (P1_IDENT, PERSON [CURRENT]. IDENT) ||
          STREQ (P1_IDENT, PERSON [CURRENT]. NAME))
      { (*P1_FOUND)++;
        #P1_INDEX = CURRENT;
      }
      if (STREQ (P2_IDENT, PERSON [CURRENT]. IDENT) ||
          STREQ (P2_IDENT, PERSON [CURRENT]. NAME))
      { (*P2_FOUND)++;
        #P2_INDEX = CURRENT;
      }
    } /* end SCAN_PER loop */
} /* end of SEEK_PER */
FIND_REL (TARG_DEX, SRCE_DEX)
/* Finds shortest path (if any) between two PERSONs and
determines their REL_SHIP based on immediate relations
traversed in path. PERSON array simulates a directed graph,
and algorithm finds shortest path, based on following
weights: PARENT-CHILD edge = 1.0
SPOUSE-SPOUSE edge = 1.8 */

INDX_TYP TARG_DEX, SRCE_DEX;

{ register INDX_TYP PERS_DEX;
  INDX_TYP THIS_NOD, BEST_DEX, LST_NRBY,
  NBR_PTR [MAX_PERS];
  register NBR_PTR THIS_NBR;
  float MIN_DIST;
  typedef short int SRCH_TYP;
  #define SEARCHING 1
  #define SUCCESS 2
  #define FAILED 3

SRCH_TYP SRCH_ST;

/* begin execution of FIND_REL */

/* initialize PERSON-array for processing -
mark all nodes as not seen */
for (PERS_DEX = 0; PERS_DEX < NUM_PERS; PERS_DEX++)
  PERSON [PERS_DEX] . REACH_ST = NOT_SEEN;
THIS_NOD = SRCE_DEX;
/* mark source node as REACHED */
PERSON [THIS_NOD] . REACH_ST = REACHED;
PERSON [THIS_NOD] . DIST_SRC = 0.0;
/* no NEARBY nodes exist yet */
LST_NRBY = -1;
SRCH_ST = (THIS_NOD == TARG_DEX) ? SUCCESS : SEARCHING;
/* Loop keeps processing closest-to-source, unREACHED node until target REACHED, or no more connected nodes. */

SEEKTARG:
while (SRCH_ST == SEARCHING)
{
    /* Process all nodes adjacent to THIS_NOD */
    for (THIS_NBR = PERSON [THIS_NOD] . NBR_HDR;
        THIS_NBR != NULL;
        THIS_NBR = THIS_NBR -> NEXT_NBR)
        PROC_ADJ (THIS_NOD, THIS_NBR -> NBR_DEX, THIS_NBR -> NBR_EDGE,
                     NRBY ND, &LST NRBY);
/* All nodes adjacent to THIS_NOD are set. Now search for shortest-distance unREACHED (but NEARBY) node to process next. */
    if (LST NRBY == -1)
        SRCH_ST = FAILED;
    else /* determine next node to process */
    {
        MIN_DIST = 1.0E+18;
        for (PERS_DEX = 0; PERS_DEX <= LST_NRBY; PERS_DEX++)
            if (PERSON [NRBY ND [PERS_DEX]] . DIST_SRC < MIN_DIST)
                {  
                    BEST_DEX = PERS DEX;
                    MIN DIST = PERSON [NRBY ND [PERS DEX]] . DIST SRC;
                }
/* establish new THIS_NOD */
    THIS_NOD = NRBY ND [BEST_DEX];
/* change THIS_NOD from being NEARBY to REACHED */
    PERSON [THIS_NOD] . REACH_ST = REACHED;
/* remove THIS_NOD from NEARBY list */
    NRBY ND [BEST_DEX] = NRBY ND [LST NRBY--];
    if (THIS_NOD == TARG_DEX)
        SRCH_ST = SUCCESS;
    }
} /* end SEEKTARG loop */

/* Shortest path between PERSONs now established. Next task is to translate path to English description of REL_SHIP. */
if (SRCH_ST == FAILED)
    printf ("%ls is not related to %ls\n", 
                PERSON [TARG_DEX] . NAME, PERSON [SRCE DEX] . NAME);
else  /* success - parse path to find and display REL_SHIP */
    {  
        RESOLVE (SRCE DEX, TARG DEX);
        CMPT GNS (SRCE DEX, TARG DEX);
    }
} /* end FIND_REL */
/* procedures under FIND_REL */

PROC_ADJ (BASENODE, NXT_NODE, N_B_EDGE, NRBY_ND, LST_NRBY)

/* NXT_NODE is adjacent to last-REACHED node (== BASENODE).
   If NXT_NODE already REACHED, do nothing.
   If previously seen, check whether path thru BASENODE is
   shorter than current path to NXT_NODE, and if so re-link
   next to base.
   If not previously seen, link next to base node. */

register INDX_TYP NXT_NODE;
INDX_TYP BASENODE, NRBY_ND[], *LST_NRBY;
EDG_TYPE N_B_EDGE;
{
  float WGHTE DG, DIST_BAS;

  /* begin execution of PROC_ADJ */
  if (PERSON [NXT_NODE] . REACH_ST != REACHED)
  {
    WGHTE DG = (N_B_EDGE == SPOUSE) ? 1.8 : 1.0;
    DIST_BAS = WGHTE DG + PERSON [BASENODE] . DIST_SRC;
    if (PERSON [NXT_NODE] . REACH_ST == NOT_SEEN)
    {
      PERSON [NXT_NODE] . REACH_ST = NEARBY;
      NRBY_ND [++ *LST_NRBY] = NXT_NODE;
      /* link next to base by re-setting its predecessor index to
         point to base, note type of edge, and re-set distance
         as it is through base node. */
      PERSON [NXT_NODE] . DIST_SRC = DIST_BAS;
      PERSON [NXT_NODE] . PATHPRED = BASENODE;
      PERSON [NXT_NODE] . EDG_PRED = N_B_EDGE;
    }
  } else /* REACH_ST = NEARBY */
  if (DIST_BAS < PERSON [NXT_NODE] . DIST_SRC)
  {
    /* link next to base by re-setting its predecessor index to
       point to base, note type of edge, and re-set distance
       as it is through base node. */
    PERSON [NXT_NODE] . DIST_SRC = DIST_BAS;
    PERSON [NXT_NODE] . PATHPRED = BASENODE;
    PERSON [NXT_NODE] . EDG_PRED = N_B_EDGE;
  }
}
/* end PROC_ADJ */
RESOLVE (SRCE_DEX, TARG_DEX)
/* RESOLVE condenses the shortest path to a
series of REL_SHIPS for which there are English
descriptions. */

INDEX_TYP SRCE_DEX, TARG_DEX;
{ /* these variables are used to generate KEY_PERSs */
  COUNTER GEN_CNT;
  /* these variables are used to condense the path */
  KEY_REC KEY_PERS [MAX_PERS];
  REL_Type KEY_REL, LKEY_REL, PRIM_REL, NXT_PRIM;
  register INDEX_TYP KEY_DEX;
  INDEX_TYP LKEY_DEX, PRIM_DEX, THIS_NOD;
  BOOLEAN SEEKMORE;

  /* begin execution of RESOLVE */
  printf ("Shortest path between identified persons: \n");
  /* Display path and initialize KEY_PERS array from path elements. */
  TRAVERSE:
    for (THIS_NOD = TARG_DEX, KEY_DEX = 0; THIS_NOD != SRCE_DEX;
      THIS_NOD = PERSON [THIS_NOD] . PATHPRED, KEY_DEX++)
      {
        printf ("%ls is ", PERSON [THIS_NOD] . NAME);
        KEY_PERS [KEY_DEX] . PERS_DEX = THIS_NOD;
        KEY_PERS [KEY_DEX] . PROXIMTY = FULL;
        KEY_PERS [KEY_DEX] . REL_NEXT = PERSON [THIS_NOD] . EDG_PRED;
        switch (PERSON [THIS_NOD] . EDG_PRED)
          {
            case PARENT: printf ("parent of\n");
              KEY_PERS [KEY_DEX] . GEN_GAP = 1;
              break;
            case CHILD: printf ("child of\n");
              KEY_PERS [KEY_DEX] . GEN_GAP = 1;
              break;
            case SPOUSE: printf ("spouse of\n");
              KEY_PERS [KEY_DEX] . GEN_GAP = 0;
              break;
          } /* end switch */
      } /* end TRAVERSE loop */
  printf ("%ls\n", PERSON [THIS_NOD] . NAME);
  KEY_PERS [KEY_DEX] . PERS_DEX = THIS_NOD;
  KEY_PERS [KEY_DEX] . REL_NEXT = NULL_REL;
  KEY_PERS [KEY_DEX + 1] . REL_NEXT = NULL_REL;
/* Resolve CHILD-PARENT and CHILD-SPOUSE-PARENT relations to SIBLING relations. */

FIND_SIB:
for (KEY_DEX = 0; KEY_PERS [KEY_DEX] . REL_NEXT != NULL_REL; KEY_DEX++)
{
if (KEY_PERS [KEY_DEX] . REL_NEXT == CHILD)
{
  LKEY_REL = KEY_PERS [KEY_DEX + 1] . REL_NEXT;
  if (LKEY_REL == PARENT)
    /* found either full or half SIBLINGS */
    BOOLEAN FULL_SIB();
  KEY_PERS [KEY_DEX] . PROXIMITY =
      FULL_SIB (KEY_PERS [KEY_DEX] . PERS_DEX,
                KEY_PERS [KEY_DEX + 2] . PERS_DEX)
    ? FULL : HALF;
  KEY_PERS [KEY_DEX] . GEN_GAP = 0;
  KEY_PERS [KEY_DEX] . REL_NEXT = SIBLING;
  CONDENSE (KEY_DEX, 1, KEY_PERS);
}
else
  if (LKEY_REL == SPOUSE
      && KEY_PERS [KEY_DEX + 2] . REL_NEXT == PARENT)
    /* found step-SIBLINGS */
    { /* found step-SIBLINGS */
      KEY_PERS [KEY_DEX] . GEN_GAP = 0;
      KEY_PERS [KEY_DEX] . PROXIMITY = STEP;
      KEY_PERS [KEY_DEX] . REL_NEXT = SIBLING;
      CONDENSE (KEY_DEX, 2, KEY_PERS);
    }
  } /* end if REL_NEXT == CHILD */
} /* end FIND_SIB loop */

/* Resolve CHILD-CHILD-... and PARENT-PARENT-... relations to direct descendant or ancestor relations. */

FIND_ANC:
for (KEY_DEX = 0; KEY_PERS [KEY_DEX] . REL_NEXT != NULL_REL; KEY_DEX++)
{
if (KEY_PERS [KEY_DEX] . REL_NEXT == CHILD ||
    KEY_PERS [KEY_DEX] . REL_NEXT == PARENT)
{
  if (LKEY_DEX = KEY_DEX + 1;
      KEY_PERS [LKEY_DEX] . REL_NEXT == KEY_PERS [KEY_DEX] . REL_NEXT;
      LKEY_DEX++);
  GEN_CNT = LKEY_DEX - KEY_DEX;
  if (GEN_CNT > 1) /* compress generations */
    { /* compress generations */
      KEY_PERS [KEY_DEX] . GEN_GAP = GEN_CNT;
      CONDENSE (KEY_DEX, GEN_CNT - 1, KEY_PERS);
    }
  } /* end if REL_NEXT == CHILD or PARENT */
} /* end FIND_ANC loop */
/* Resolve CHILD-SIBLING-PARENT to COUSIN,
CHILD-SIBLING to NEPHEW,
SIBLING-PARENT to UNCLE. */

FIND_CUZ:
for (KEY_DEX = 0; KEY_PERS[KEY_DEX] . REL_NEXT != NULL_REL; KEY_DEX++)
{
  LKEY_REL = KEY_PERS[KEY_DEX + 1] . REL_NEXT;
  if (KEY_PERS[KEY_DEX] . REL_NEXT == CHILD && LKEY_REL == SIBLING)
    { /* COUSIN or NEPHEW */
      if (KEY_PERS[KEY_DEX + 2] . REL_NEXT == PARENT)
        { /* found COUSIN */
          COUNTER GAP1, GAP2;
          GAP1 = KEY_PERS[KEY_DEX] . GEN_GAP;
          GAP2 = KEY_PERS[KEY_DEX + 2] . GEN_GAP;
          KEY_PERS[KEY_DEX] . PROXIMTY = KEY_PERS[KEY_DEX + 1] . PROXIMTY;
          KEY_PERS[KEY_DEX] . GEN_GAP = (GAP1 < GAP2) ? (GAP2 - GAP1) : (GAP1 - GAP2);
          KEY_PERS[KEY_DEX] . CUZ_RANK = (GAP1 < GAP2) ? GAP1 : GAP2;
          KEY_PERS[KEY_DEX] . REL_NEXT = COUSIN;
          CONDENSE (KEY_DEX, 2, KEY_PERS);
        }
      else /* found NEPHEW */
        { /* found NEPHEW */
          KEY_PERS[KEY_DEX] . PROXIMTY = KEY_PERS[KEY_DEX + 1] . PROXIMTY;
          KEY_PERS[KEY_DEX] . REL_NEXT = NEPHEW;
          CONDENSE (KEY_DEX, 1, KEY_PERS);
        }
    }
  else /* end COUSIN or NEPHEW */
else
  if (KEY_PERS[KEY_DEX] . REL_NEXT == SIBLING && LKEY_REL == PARENT)
    { /* found UNCLE */
      KEY_PERS[KEY_DEX] . GEN_GAP = KEY_PERS[KEY_DEX + 1] . GEN_GAP;
      KEY_PERS[KEY_DEX] . REL_NEXT = UNCLE;
      CONDENSE (KEY_DEX, 1, KEY_PERS);
    }
} /* end FIND_CUZ loop */
/* Loop below will pick out valid adjacent strings of elements to be displayed. KEY_DEX points to first element, LKEY_DEX to last element, and PRIM_DEX to the element which determines the primary English word to be used. Associativity of adjacent elements in condensed table is based on English usage. */

printf("Condensed path:\n");
COND LadIT:
for (KEY_DEX = 0; KEY_PERS [KEY_DEX] . REL_NEXT != NULL_REL;
    KEY_DEX = LKEY_DEX + 1)
{
    KEY_REL = KEY_PERS [KEY_DEX] . REL_NEXT;
    LKEY_DEX = KEY_DEX;
    PRIM_DEX = KEY_DEX;
    if (KEY_PERS [KEY_DEX + 1] . REL_NEXT != NULL_REL)
    { /* seek multi-element combination */
        SEEKMORE = TRUE;
        if (KEY_REL == SPOUSE)
        {
            PRIM_DEX = ++LKEY_DEX;
            /* Nothing can follow SPOUSE-SIBLING or SPOUSE-COUSIN */
            SEEKMORE = !(KEY_PERS [LKEY_DEX] . REL_NEXT & (SIBLING | COUSIN));
        }
        /* PRIM_DEX is now correctly set. Next if-statement determines if a following SPOUSE relation should be appended to this combination or left for the next combination. */
        if (SEEKMORE && KEY_PERS [PRIM_DEX + 1] . REL_NEXT == SPOUSE)
        { /* Only a SPOUSE can follow a Primary; check primary preceding and following SPOUSE. */
            PRIM_REL = KEY_PERS [PRIM_DEX] . REL_NEXT;
            NXT_PRIM = KEY_PERS [PRIM_DEX + 2] . REL_NEXT;
            if ((NXT_PRIM & (NEPHEW | COUSIN | NULL_REL))
                || (PRIM_REL == NEPHEW)
                || ((PRIM_REL & (SIBLING | PARENT)) && NXT_PRIM != UNCLE ))
            /* append following SPOUSE with this combination. */
                LKEY_DEX++;
        }
    }
    /* end multi-element combination */
    SHOW_REL (KEY_DEX, LKEY_DEX, PRIM_DEX, KEY_PERS);
} /* end CONSLIDT loop */
printf("%ls\n", PERSON [KEY_PERS [KEY_DEX] . PERS_DEX] . NAME);
} /* end of RESOLVE */
BOOLEAN FULL_SIB (INDEX1, INDEX2)
/* Determines whether two PERSONs are full siblings, i.e.,
have the same two parents. */
register INDEX_TYP INDEX1, INDEX2;
{
    return
    ! STREQ (PERSON [INDEX1] . REL_ID [FATHR_ID], NULL_ID) &&
    ! STREQ (PERSON [INDEX1] . REL_ID [MOTHR_ID], NULL_ID) &&
    STREQ (PERSON [INDEX1] . REL_ID [FATHR_ID],
            PERSON [INDEX2] . REL_ID [FATHR_ID]) &&
    STREQ (PERSON [INDEX1] . REL_ID [MOTHR_ID],
            PERSON [INDEX2] . REL_ID [MOTHR_ID]);
}

CONDENSE (AT_INDEX, GAP_SIZE, KEY_PERS)
/* CONDENSE condenses superfluous entries from the
KEY_PERS array, starting at AT_INDEX. */
register INDEX_TYP AT_INDEX;
COUNTER GAP_SIZE;
KEY_REC KEY_PERS [];
{
    register INDEX_TYP SENDDEX;
    do
    {
        AT_INDEX++;
        SENDDEX = AT_INDEX + GAP_SIZE;
        KEY_PERS [AT_INDEX] = KEY_PERS [SENDDEX];
    }
    while (KEY_PERS [SENDDEX] . REL_NEXT != NULL_REL);
}
/* procedures under RESOLVE */

SHOW_REL (FRST_DEX, LAST_DEX, PRIM_DEX, KEY_PERS)
/* SHOW_REL takes 1, 2, or 3 adjacent elements in the condensed table and generates the English description of the relation between the first and last + 1 elements. */

INDX_TYP FRST_DEX, LAST_DEX, PRIM_DEX;
KEY_REC KEY_PERS[];
{ BOOLEAN INLAW;
SIB_TYP THIS_PRX;
GNDR_TYP THIS_GND;
short int SUFFIX;
register REL_TYPE FRST_REL, LAST_REL, PRIM_REL;
COUNTER THIS_GAP, THIS_CUZ;

FRST_REL = KEY_PERS[FRST_DEX].REL_NEXT;
LAST_REL = KEY_PERS[LAST_DEX].REL_NEXT;
PRIM_REL = KEY_PERS[PRIM_DEX].REL_NEXT;

/* set THIS_PRX */
if (((PRIM_REL == PARENT && FRST_REL == SPOUSE) ||
(PRIM_REL == CHILD && LAST_REL == SPOUSE))
   THIS_PRX = STEP;
else
   if (PRIM_REL & (SIBLING | UNCLE | NEPHEW | COUSIN))
      THIS_PRX = KEY_PERS[PRIM_DEX].PROXIMTY;
   else
      THIS_PRX = FULL;

/* set THIS_GAP */
if (PRIM_REL & (PARENT | CHILD | UNCLE | NEPHEW | COUSIN))
   THIS_GAP = KEY_PERS[PRIM_DEX].GEN_GAP;
else
   THIS_GAP = 0;

/* set INLAW */
INLAW = FALSE;
if (FRST_REL == SPOUSE && (PRIM_REL & (SIBLING | CHILD | NEPHEW | COUSIN)))
   INLAW = TRUE;
else
   if (LAST_REL == SPOUSE &&
       (PRIM_REL & (SIBLING | PARENT | UNCLE | COUSIN)))
      INLAW = TRUE;

/* set THIS_CUZ */
if (PRIM_REL == COUSIN)
   THIS_CUZ = KEY_PERS[PRIM_DEX].CUZ_RANK;
else
   THIS_CUZ = 0;
/* parameters are set - now generate display. */

printf ("%ls is ", PERSON [KEY_PERS [FRST_DEX] . PERS_DEX] . NAME);
if (PRIM_REL & (PARENT | CHILD | UNCLE | NEPHEW))
{ /* display generation-qualifier */

  if (THIS_GAP >= 3)
  {
    printf ("great");
    if (THIS_GAP > 3)
      printf ("*%ld", THIS_GAP - 2);
    printf ("-");
  }

  if (THIS_GAP >= 2)
    printf ("grand-");

  else
    if (PRIM_REL == COUSIN & THIS_CUZ > 1)
    {
      printf ("%ld", THIS_CUZ);
      SUFFIX = THIS_CUZ % 10;
      switch (SUFFIX)
      {
        case 1: printf ("st "); break;
        case 2: printf ("nd "); break;
        case 3: printf ("rd "); break;
        default: printf ("th "); break;
      }
    }

  if (THIS_PRX == STEP)
    printf ("step-");
  else
    if (THIS_PRX == HALF)
      printf ("half-");
THIS_GND = PERSON [KEY_PERS [FRST_DEX] . PERS_DEX] . GENDER;
switch (PRIM_REL) {
    case PARENT : if (THIS_GND == MALE) printf ("father");
                  else printf ("mother");
                  break;
    case CHILD : if (THIS_GND == MALE) printf ("son");
                else printf ("daughter");
                break;
    case SPOUSE : if (THIS_GND == MALE) printf ("husband");
                  else printf ("wife");
                  break;
    case SIBLING: if (THIS_GND == MALE) printf ("brother");
                  else printf ("sister");
                  break;
    case UNCLE  : if (THIS_GND == MALE) printf ("uncle");
                  else printf ("aunt");
                  break;
    case NEPHEW : if (THIS_GND == MALE) printf ("nephew");
                  else printf ("niece");
                  break;
    case COUSIN : printf ("cousin");
                  break;
    default     : printf ("null");
                  break;
}

if (INLAW)  printf ("-in-law");

if (PRIM_REL == COUSIN && THIS_GAP > 0)  
if (THIS_GAP > 1)   printf (" %ld times removed", THIS_GAP);  
else   printf (" once removed");

printf (" of
n");
} /* end of SHOW_REL */
/* procedures under FIND_REL */

CMPT_GNS (INDEX1, INDEX2)
/* CMPT_GNS assumes that each ancestor contributes half of the genetic material to a PERSON. It finds common ancestors between two PERSONs and computes the expected value of the PROPORTN of common material. */

register INDX_TYP INDEX1, INDEX2;

{ float COM_PROP;

/* First zero out all ancestors to allow adding. This is necessary because there might be two paths to an ancestor. */
ZERO_PRO (INDEX1);

/* now mark with shared PROPORTN */
MARK_PRO (PERSON [INDEX1] . IDENT, 1.0, INDEX1);
COM_PROP = 0.0;
CHK_COM ( & COM_PROP, PERSON [INDEX1] . IDENT, 1.0, 0.0, INDEX2);
printf ("Proportion of common genetic material = %1.5e \n", COM_PROP);
}

/* end of CMPT_GNS */

ZERO_PRO (ZERO_DEX)
/* ZERO_PRO recursively seeks out all ancestors and zeros them out. */

register INDX_TYP ZERO_DEX;

{ register NBR_PTR THIS_NBR;

PERSON [ZERO_DEX] . DSC_GENE = 0.0;
for (THIS_NBR = PERSON [ZERO_DEX] . NBR_HDR; THIS_NBR != NULL;
    THIS_NBR = THIS_NBR -> NEXT_NBR)
    {
        if (THIS_NBR -> NBR_EDGE == PARENT)
            ZERO_PRO (THIS_NBR -> NBR_DEX);
    }
}

/* end of ZERO_PRO */
MARK_PRO (MARKER, PROPRTN, MARK_DEX)
/* MARK_PRO recursively seeks out all ancestors and marks them with the sender's PROPRTN of shared genetic material. This PROPRTN is diluted by one-half for each generation. */

ID_TYPE MARKER;
float PROPRTN;
INDX_TYP MARK_DEX;

{ register NBR_PTR THIS_NBR;

strcpy (PERSON [MARK_DEX]. DSC_ID, MARKER);
PERSON [MARK_DEX]. DSC_GENE += PROPRTN;
for (THIS_NBR = PERSON [MARK_DEX]. NBR_HDR;
THIS_NBR != NULL;
THIS_NBR = THIS_NBR -> NEXT_NBR)
{
if (THIS_NBR -> NBR_EDGE == PARENT)
MARK_PRO (MARKER, PROPRTN / 2.0, THIS_NBR -> NBR_DEX);
}
} /* end of MARK_PRO */

CHK_COM (COM_PTR, MATCH_ID, PROPRTN, COUNTED, CHK_DEX)
/* CHK_COM searches all the ancestors of CHK_DEX to see if any have been marked, and if so adds the appropriate amount to *COM_PTR. */

float *COM_PTR, PROPRTN, COUNTED;
ID_TYPE MATCH_ID;
INDX_TYP CHK_DEX;

{ register NBR_PTR THIS_NBR;
register float CONTRIB;

if (STREQ (PERSON [CHK_DEX]. DSC_ID, MATCH_ID))
{ /* Increment *COM_PTR by the contribution of this common ancestor, but discount for the contribution of less remote ancestors already counted. */
CONTRIB = PERSON [CHK_DEX]. DSC_GENE * PROPRTN;
*COM_PTR += CONTRIB - COUNTED;
}
else
CONTRIB = 0.0;
for (THIS_NBR = PERSON [CHK_DEX]. NBR_HDR;
THIS_NBR != NULL;
THIS_NBR = THIS_NBR -> NEXT_NBR)
{
if (THIS_NBR -> NBR_EDGE == PARENT)
CHK_COM (COM_PTR, MATCH_ID, PROPRTN / 2.0,
CONTRIB / 4.0, THIS_NBR -> NBR_DEX);
}
} /* end of CHK_COM */
5.0 COBOL

In keeping with the general convention of the examples, language-supplied keywords and identifiers are written in lower case in the program. To conform strictly to the COBOL-74 standard, however, programs must use only upper-case letters.

* ---- Compilation unit number 1 ---- *

identification division.
program-id. RELATE.

environment division.

configuration section.
source-computer. VAX-11.
object-computer. VAX-11.

input-output section.
file-control.
select PEOPLE assign to "PEOPLE.DAT",
    file status is PEOPLE-STATUS.

data division.

file section.
fd PEOPLE
    label records are standard.
01 PEOPLE-RECORD.
    05 NAME pic X(20).
    05 IDENTIFIER pic 999.
*** "M" for MALE and "F" for FEMALE
    05 GENDER pic X.
    05 IMMEDIATE-RELATIONS.
        10 RELATIVE-IDENTIFIER occurs 3 times pic 999.

working-storage section.

77 ARG-PERSON1-INDEX pic 999.
77 ARG-PERSON2-INDEX pic 999.

01 PEOPLE-STATUS.
    05 STATUS-1 pic X.
        88 END-OF-PEOPLE-FILE value "1".
    05 STATUS-2 pic X.

* Define global objects

01 TRUTH-VALUES.
    05 IS-TRUE pic X value "T".
    05 IS-FALSE pic X value "F".

01 SPECIAL-IDENT-VALUE.
    05 NULL-IDENT pic 999 value 000.
* each PERSON's record in the file identifies at most three
* others directly related: father, mother, and spouse

01 GIVEN-IDENTIFIERS.
  05 FATHER-IDENT  pic 9  value 1.
  05 MOTHER-IDENT  pic 9  value 2.
  05 SPOUSE-IDENT  pic 9  value 3.

01 GENDER-TYPE.
  05 MALE         pic X  value "M".
  05 FEMALE       pic X  value "F".

01 RELATION-TYPE.
  05 PARENT       pic 9  value 1.
  05 CHILD        pic 9  value 2.
  05 SPOUSE       pic 9  value 3.
  05 SIBLING      pic 9  value 4.
  05 UNCLE        pic 9  value 5.
  05 NEPHEW       pic 9  value 6.
  05 COUSIN       pic 9  value 7.
  05 NULL-RELATION pic 9  value 8.

* A node in the graph (= PERSON) has either already been reached,
* is immediately adjacent to those reached, or farther away.

01 REACHED-TYPE.
  05 REACHED      pic 9  value 1.
  05 NEARBY       pic 9  value 2.
  05 NOT-SEEN     pic 9  value 3.
* the PERSON array is the central repository of information
* about inter-relationships.
* All relationships are captured in the directed graph of which
* each record is a node.

01 PERSON-TABLE.
  05 NUMBER-OF-PERSONS usage index.
  05 PERSON occurs 300 times
     indexed by CURRENT, PREVIOUS,
     FROM-INDEX, TO-INDEX,
     PERSON1-INDEX, PERSON2-INDEX.

*** static information - filled from PEOPLE file:
  10 NAME pic X(20).
  10 IDENTIFIER pic 999.
  10 GENDER pic X.

*** IDENTIFIERS of immediate relatives - father, mother, spouse
  10 IMMEDIATE-RELATIONS.
     15 RELATIVE-IDENTIFIER occurs 3 times indexed by RELATIONSHIP
         pic 999.

*** pointers to immediate neighbors in graph
  10 NEIGHBOR-COUNT pic 99.
  10 NEIGHBOR-RECORD occurs 20 times indexed by NEXT-NEIGHBOR.
     15 NEIGHBOR-INDEX usage index.
     15 NEIGHBOR-EDGE pic 9.

*** data used when traversing graph to resolve user request:
  10 DISTANCE-FROM-SOURCE pic 9999999V9.
  10 PATH-PREDECESSOR usage index.
  10 EDGE-TO-PREDECESSOR pic 9.
  10 REACHED-STATUS pic 9.

*** data used to compute common genetic material
  10 DESCENDANT-IDENTIFIER pic 999.
  10 DESCENDANT-GENES pic 9V99999999.

* These variables are used to accept and resolve requests for
* RELATIONSHIP information.
01 RELATIONSHIP-WORK-ITEMS.
  05 REQUEST-BUFFER pic X(60).
     88 REQUEST-TO-STOP value "stop".
  05 PERSON1-IDENT pic X(20).
  05 PERSON2-IDENT pic X(20).
  05 PERSON1-FOUND pic 999.
  05 PERSON2-FOUND pic 999.
  05 ERROR-MESSAGE pic X(40).
  05 REQUEST-OK pic X(40) value "Request OK".

01 AUXILIARY-VARIABLES.
  05 RELATION-LOOP-DONE pic X.
     88 RELATION-LOOP-IS-DONE value "T".
  05 TEMP-INDEX usage index.
  05 THIS-EDGE pic 9.
  05 LEADING-SPACES pic 99.
  05 SEMICOLON-COUNT pic 99.
  05 CURRENT-IDENT pic 999.
  05 PREVIOUS-IDENT pic 999.
  05 TEMP-IDENT pic X(20).
procedure division.
MAIN-LINE.
    open input PEOPLE.
    read PEOPLE at end perform NULL.

* This loop reads in the PEOPLE file and constructs the PERSON
* array from it (one PERSON = one record = one array entry).
* As records are read in, links are constructed to represent the
* PARENT-CHILD or SPOUSE RELATIONSHIP. The array then implements
* a directed graph which is used to satisfy subsequent user
* requests. The file is assumed to be correct - no validation
* is performed on it.

perform READ-IN-PEOPLE thru READ-IN-PEOPLE-EXIT
    varying CURRENT from 1 by 1 until END-OF-PEOPLE-FILE.
    set CURRENT down by 1.
    set NUMBER-OF-PERSONS to CURRENT.
    close PEOPLE.

* PERSON array is now loaded and edges between immediate relatives
* (PARENT-CHILD or SPOUSE-SPOUSE) are established.

    perform PROMPT-AND-READ.

* While-loop accepts requests and finds RELATIONSHIP (if any)
* between pairs of PERSONS.

    perform READ-AND-PROCESS-REQUEST thru READ-AND-PROCESS-REQUEST-EXIT
        until REQUEST-TO-STOP.
        display " End of relation-finder."
    stop run.

READ-IN-PEOPLE.
*** copy direct information from file to array
    move corresponding PEOPLE-RECORD to PERSON (CURRENT).
    move IMMEDIATE-RELATIONS of PEOPLE-RECORD
        to IMMEDIATE-RELATIONS of PERSON (CURRENT).
*** Location of adjacent persons as yet undetermined
    move zero to NEIGHBOR-COUNT of PERSON (CURRENT).
*** Descendants as yet undetermined
    move NULL-IDENT to DESCENDANT-IDENTIFIER of PERSON (CURRENT).
    move IDENTIFIER of PERSON (CURRENT) to CURRENT-IDENT.
*** Compare this PERSON against all previously entered PERSONS
*** to search for RELATIONSHIPS.
    perform COMPARE-TO-PREVIOUS varying PREVIOUS from 1 by 1
        until PREVIOUS not < CURRENT.
    read PEOPLE at end perform NULL.
READ-IN-PEOPLE-EXIT.
    exit.

NULL.
    exit.
COMPARE-TO-PREVIOUS.
move IDENTIFIER of PERSON (PREVIOUS) to PREVIOUS-IDENT.
*** Search for father, mother, or spouse relationship in
*** either direction between this and PREVIOUS PERSON.
*** Assume at most one RELATIONSHIP exists.
move IS-FALSE to RELATION-LOOP-DONE.
perform TRY-ALL-RELATIONSHIPS
  varying RELATIONSHIP from FATHER-IDENT by 1
  until RELATIONSHIP > SPOUSE-IDENT or RELATION-LOOP-IS-DONE.
TRY-ALL-RELATIONSHIPS.
  if RELATIVE-IDENTIFIER of PERSON (CURRENT, RELATIONSHIP) =
    PREVIOUS-IDENT
  set FROM-INDEX to CURRENT
  set TO-INDEX to PREVIOUS
  perform LINK-RELATIVES
  move IS-TRUE to RELATION-LOOP-DONE
else
  if CURRENT-IDENT =
    RELATIVE-IDENTIFIER of PERSON (PREVIOUS, RELATIONSHIP)
  set FROM-INDEX to PREVIOUS
  set TO-INDEX to CURRENT
  perform LINK-RELATIVES
  move IS-TRUE to RELATION-LOOP-DONE.
LINK-RELATIVES.
* establishes cross-indexing between immediately related PERSONs.

  if RELATIONSHIP = SPOUSE-IDENT
  move SPOUSE to THIS-EDGE
  perform LINK-ONE-WAY
  set TEMP-INDEX to FROM-INDEX
  set FROM-INDEX to TO-INDEX
  set TO-INDEX to TEMP-INDEX
  perform LINK-ONE-WAY
else
  * RELATIONSHIP is father or mother
  move PARENT to THIS-EDGE
  perform LINK-ONE-WAY
  move CHILD to THIS-EDGE
  set TEMP-INDEX to FROM-INDEX
  set FROM-INDEX to TO-INDEX
  set TO-INDEX to TEMP-INDEX
  perform LINK-ONE-WAY.
LINK-ONE-WAY.
*** Establishes the NEIGHBOR-RECORD from one PERSON to another
add 1 to NEIGHBOR-COUNT of PERSON (FROM-INDEX).
set NEXT-NEIGHBOR to NEIGHBOR-COUNT of PERSON (FROM-INDEX).
set NEIGHBOR-INDEX of PERSON (FROM-INDEX, NEXT-NEIGHBOR)
to TO-INDEX.
move THIS-EDGE
to NEIGHBOR-EDGE of PERSON (FROM-INDEX, NEXT-NEIGHBOR).
PROMPT-AND-READ.
* Issues prompt for user-request, reads in request,
  * blank-fills buffer, and skips to next line of input.

  display " ".
  display " ----------------------------------".
  display " Enter two person-identifiers (name or number), ".
  display " separated by semicolon. Enter "stop" to stop. ".
  move spaces to REQUEST-BUFFER.
  accept REQUEST-BUFFER.

READ-AND-PROCESS-REQUEST.
perform CHECK-REQUEST.

*** Syntax check of request completed. Now either display error
*** message or search for the two PERSONs.

if ERROR-MESSAGE = REQUEST-OK
  perform PROCESS-LEGAL-REQUEST
else
  display " Incorrect request format: ", ERROR-MESSAGE.
  perform PROMPT-AND-READ.
  READ-AND-PROCESS-REQUEST-EXIT.
  exit.

CHECK-REQUEST.
* Performs syntactic check on request in buffer
  * and fills in identifiers of the two requested persons.

  move zero to SEMICOLON-COUNT.
  inspect REQUEST-BUFFER tallying SEMICOLON-COUNT
    for all ";".
  if SEMICOLON-COUNT not = 1
    move "must be exactly one semicolon." to ERROR-MESSAGE
  else
    move zero to LEADING-SPACES
    inspect REQUEST-BUFFER tallying LEADING-SPACES
      for leading spaces
    add 1 to LEADING-SPACES
    unstring REQUEST-BUFFER delimited by " ";
      into PERSON1-IDENT, TEMP-IDENT
      with pointer LEADING-SPACES
    if PERSON1-IDENT = spaces
      move "null field preceding semicolon." to ERROR-MESSAGE
    else
      if TEMP-IDENT = spaces
        move "null field following semicolon." to ERROR-MESSAGE
      else
        move zero to LEADING-SPACES
        inspect TEMP-IDENT tallying LEADING-SPACES
          for leading spaces
        add 1 to LEADING-SPACES
        unstring TEMP-IDENT into PERSON2-IDENT
          with pointer LEADING-SPACES
        move REQUEST-OK to ERROR-MESSAGE.
PROCESS-LEGAL-REQUEST.
*** search for requested PERSONs.
  move zero to PERSON1-FOUND, PERSON2-FOUND.
  perform SCAN-ALL-PERSONS varying CURRENT from 1 by 1
till CURRENT > NUMBER-OF-PERSONS.
  if PERSON1-FOUND = 1 and PERSON2-FOUND = 1
*** Exactly one match for each PERSON - proceed to
*** determine RELATIONSHIP, if any.
  if PERSON1-INDEX = PERSON2-INDEX
    if GENDER of PERSON (PERSON1-INDEX) = MALE
      display "", NAME of PERSON (PERSON1-INDEX),
      " is identical to himself."
    else
      display "", NAME of PERSON (PERSON1-INDEX),
      " is identical to herself."
  else
    set ARG-PERSON1-INDEX to PERSON1-INDEX
    set ARG-PERSON2-INDEX to PERSON2-INDEX
    call "FINDREL" using
    ARG-PERSON1-INDEX, ARG-PERSON2-INDEX, PERSON-TABLE
  else
 *** either not found or more than one found
  perform MISSING-OR-DUPLICATE-PERSONS.

SCAN-ALL-PERSONS.
  if PERSON1-IDENT = NAME  of PERSON (CURRENT) or
    IDENTIFIER of PERSON (CURRENT)
    set PERSON1-INDEX to CURRENT
    add 1 to PERSON1-FOUND.
  if PERSON2-IDENT = NAME  of PERSON (CURRENT) or
    IDENTIFIER of PERSON (CURRENT)
    set PERSON2-INDEX to CURRENT
    add 1 to PERSON2-FOUND.

MISSING-OR-DUPLICATE-PERSONS.
  if PERSON1-FOUND = zero
    display " First person not found."
  else
    if PERSON1-FOUND > 1
      display " Duplicate names for first person - use",
      " numeric identifier."
    if PERSON2-FOUND = zero
      display " Second person not found."
    else
      if PERSON2-FOUND > 1
        display " Duplicate names for second person - use",
        " numeric identifier.".
* ---- Compilation unit number 2 ----

identification division.
program-id. FINDREL.

Finds shortest path (if any) between two PERSONs and
determines their RELATIONSHIP based on immediate relations
traversed in path. PERSON array simulates a directed graph,
and algorithm finds shortest path, based on following
weights: PARENT-CHILD edge = 1.0
SPouse-SpoUSE edge = 1.8

environment division.
configuration section.
source-computer. VAX-11.
object-computer. VAX-11.

data division.
working-storage section.

* Define global objects

01 TRUTH-VALUES.
   05 IS-TRUE pic X value "T".
   05 IS-FALSE pic X value "F".

* each PERSON's record in the file identifies at most three
* others directly related: father, mother, and spouse
01 GIVEN-IDENTIFIERS.
   05 FATHER-IDENT pic 9 value 1.
   05 MOTHER-IDENT pic 9 value 2.
   05 SPOUSE-IDENT pic 9 value 3.

01 GENDER-TYPE.
   05 MALE pic X value "M".
   05 FEMALE pic X value "F".

01 RELATION-TYPE.
   05 PARENT pic 9 value 1.
   05 CHILD pic 9 value 2.
   05 SPOUSE pic 9 value 3.
   05 SIBLING pic 9 value 4.
   05 UNCLE pic 9 value 5.
   05 NEPHEW pic 9 value 6.
   05 COUSIN pic 9 value 7.
   05 NULL-RELATION pic 9 value 8.
* A node in the graph (= PERSON) has either already been reached,
* is immediately adjacent to those reached, or farther away.

01 REACHED-TYPE.
  05 REACHED pic 9 value 1.
  05 NEARBY pic 9 value 2.
  05 NOT-SEEN pic 9 value 3.

01 SEARCH-TYPE.
  05 SEARCHING pic 9 value 1.
  05 SUCCEEDED pic 9 value 2.
  05 FAILED pic 9 value 3.

01 SIBLING-TYPE.
  05 STEP pic 9 value 1.
  05 HALF pic 9 value 2.
  05 FULL pic 9 value 3.

01 KEY-PERSON-TABLE.
  05 KEY-PERSON occurs 300 times
     indexed by KEY-INDEX, LATER-KEY-INDEX, PRIMARY-INDEX,
     FIRST-INDEX, LAST-INDEX, RECEIVE-INDEX, SEND-INDEX.
    10 RELATION-TO-NEXT pic 9.
    10 PERSON-INDEX usage index.
    10 GENERATION-GAP pic 999.
    10 PROXIMITY pic 9.
    10 COUSIN-RANK pic 999.

01 AUXILIARY-VARIABLES.
*** these variables are used to find the shortest path
  05 WEIGHT-THIS-EDGE pic 99V9.
  05 DISTANCE-THRU-BASE-NODE pic 999999V9.
  05 SEARCH-STATUS pic 9.
  05 NEARBY-NODE usage index, occurs 300 times,
     indexed by THIS-NEARBY-INDEX, BEST-NEARBY-INDEX, LAST-NEARBY-INDEX.
  05 THIS-EDGE pic 9.
  05 NEXT-BASE-EDGE pic 9.
  05 MINIMAL-DISTANCE pic 9999999V9.
  05 DISPLAY-BUFFER pic X(70).
  05 DISPLAY-POINTER pic 99.
  05 NULL-IDENT pic 999 value 000.

*** these variables are used to condense the path
  05 KEY-RELATION pic 9.
  05 LATER-KEY-RELATION pic 9.
  05 PRIMARY-RELATION pic 9.
  05 FIRST-RELATION pic 9.
  05 LAST-RELATION pic 9.
  05 NEXT-PRIMARY-RELATION pic 9.
  05 GAP-SIZE pic 999.
  05 ANOTHER-ELEMENT-POSSIBLE pic X.
  88 ANOTHER-ELEMENT-IS-POSSIBLE value "T".
*** these variables are used to generate KEY-PERSONs and for DISPLAY

05 GENERATION-COUNT pic 999.
05 TEMP-NUMBER pic 999.
05 THIS-COUSIN-RANK pic 999.
05 THIS-PROXIMITY pic 9.
05 THIS-GENDER pic X.
05 THIS-GENERATION-GAP pic 999.
05 SUFFIX-INDICATOR pic 9.
05 TWO-DIGIT-FIELD pic 29.
05 INLAW pic X.

88 RELATION-IS-INLAW value "T".

05 MALE-NAME-VALUES.
  10 filler pic X(8) value "father ".
  10 filler pic X(8) value "son ".
  10 filler pic X(8) value "husband ".
  10 filler pic X(8) value "brother ".
  10 filler pic X(8) value "uncle ".
  10 filler pic X(8) value "nephew ".
  10 filler pic X(8) value "cousin ".
  10 filler pic X(8) value "null ".

05 MALE-NAME-TABLE redefines MALE-NAME-VALUES.
  10 PRIMARY-MALE-NAME pic X(8) occurs 8 times indexed by MALE-INDEX.

05 FEMALE-NAME-VALUES.
  10 filler pic X(8) value "mother ".
  10 filler pic X(8) value "daughter".
  10 filler pic X(8) value "wife ".
  10 filler pic X(8) value "sister ".
  10 filler pic X(8) value "aunt ".
  10 filler pic X(8) value "niece ".
  10 filler pic X(8) value "cousin ".
  10 filler pic X(8) value "null ".

05 FEMALE-NAME-TABLE redefines FEMALE-NAME-VALUES.
  10 PRIMARY-FEMALE-NAME pic X(8) occurs 8 times indexed by FEMALE-INDEX.
linkage section.

77 PARM-TARGET-INDEX pic 999.
77 PARM-SOURCE-INDEX pic 999.

01 PERSON-TABLE.
   05 NUMBER-OF-PERSONS usage index.
   05 PERSON occurs 300 times
      indexed by INDEX1, INDEX2, TARGET-INDEX, SOURCE-INDEX,
      BASE-NODE, THIS-NODE, NEXT-NODE.
*** static information - filled from PEOPLE file:
   10 NAME pic X(20).
   10 IDENTIFIER pic 999.
   10 GENDER pic X.
*** IDENTIFIERS of immediate relatives - father, mother, spouse
   10 IMMEDIATE-RELATIONS.
      15 RELATIVE-IDENTIFIER occurs 3 times indexed by RELATIONSHIP
         pic 999.
*** pointers to immediate neighbors in graph
   10 NEIGHBOR-COUNT pic 99.
   10 NEIGHBOR-RECORD occurs 20 times indexed by THIS-NEIGHBOR.
      15 NEIGHBOR-INDEX usage index.
      15 NEIGHBOR-EDGE pic 9.
*** data used when traversing graph to resolve user request:
   10 DISTANCE-FROM-SOURCE pic 9999999.
   10 PATH-PREDECESSOR usage index.
   10 EDGE-TO-PREDECESSOR pic 9.
   10 REACHED-STATUS pic 9.
*** data used to compute common genetic material
   10 DESCENDANT-IDENTIFIER pic 999.
   10 DESCENDANT-GENES pic 999999999.

procedure division using
   PARM-TARGET-INDEX, PARM-SOURCE-INDEX, PERSON-TABLE.

MAIN-LINE.
   set TARGET-INDEX to PARM-TARGET-INDEX.
   set SOURCE-INDEX to PARM-SOURCE-INDEX.
*** initialize PERSON-array for processing -
*** mark all nodes as not seen
   perform MARK-AS-NOT-SEEN varying THIS-NODE from 1 by 1
      until THIS-NODE > NUMBER-OF-PERSONS.
   set THIS-NODE to SOURCE-INDEX.
*** mark source node as REACHED
   move REACHED to REACHED-STATUS of PERSON (THIS-NODE).
   move zero to DISTANCE-FROM-SOURCE of PERSON (THIS-NODE).
*** no nearby nodes exist yet
   set LAST-NEARBY-INDEX to 1.
   set LAST-NEARBY-INDEX down by 1.
   if THIS-NODE = TARGET-INDEX
      move SUCCEEDED to SEARCH-STATUS
   else
      move SEARCHING to SEARCH-STATUS.
*** Loop keeps processing closest-to-source, unREACHED node until target REACHED, or no more connected nodes.
perform SEARCH-FOR-TARGET until SEARCH-STATUS not = SEARCHING.

*** Shortest path between PERSONs now established. Next task is to translate path to English description of RELATIONSHIP.
if SEARCH-STATUS = FAILED
   display "", NAME of PERSON (TARGET-INDEX), " is not related to ", NAME of PERSON (SOURCE-INDEX)
else
   success - parse path to find and display RELATIONSHIP
   perform RESOLVE-PATH-TO-ENGLISH
   call "COMGENES" using
      PARM-SOURCE-INDEX, PARM-TARGET-INDEX, PERSON-TABLE.
   END-OF-FINDREL.
   exit program.

MARK-AS-NOT-SEEN.
   move NOT-SEEN to REACHED-STATUS of PERSON (THIS-NODE).

SEARCH-FOR-TARGET.
*** Process all nodes adjacent to THIS-NODE
perform PROCESS-ADJACENT-NODE varying THIS-NEIGHBOR from 1 by 1
   until THIS-NEIGHBOR > NEIGHBOR-COUNT of PERSON (THIS-NODE).
*** All nodes adjacent to THIS-NODE are set. Now search for shortest-distance unREACHED (but NEARBY) node to process next.
   if LAST-NEARBY-INDEX = zero
      move FAILED to SEARCH-STATUS
   else
      determine next node to process
      move 9999999 to MINIMAL-DISTANCE
      perform FIND-CLOSEST-UNREACHED-NODE varying THIS-NEARBY-INDEX from 1 by 1
      until THIS-NEARBY-INDEX > LAST-NEARBY-INDEX
***
establish new THIS-NODE
set THIS-NODE to NEARBY-NODE (BEST-NEARBY-INDEX)
*** change THIS-NODE from being NEARBY to REACHED
move REACHED to REACHED-STATUS of PERSON (THIS-NODE)
*** remove THIS-NODE from NEARBY list
set NEARBY-NODE (BEST-NEARBY-INDEX) to NEARBY-NODE (LAST-NEARBY-INDEX)
set LAST-NEARBY-INDEX down by 1
if THIS-NODE = TARGET-INDEX
   move SUCCEEDED to SEARCH-STATUS.
PROCESS-ADJACENT-NODE.
set BASE-NODE to THIS-NODE.
set NEXT-NODE to NEIGHBOR-INDEX of PERSON (BASE-NODE, THIS-NEIGHBOR).
move NEIGHBOR-EDGE of PERSON (BASE-NODE, THIS-NEIGHBOR) to NEXT-BASE-EDGE.
*** NEXT-NODE is adjacent to last-REACHED node (= BASE-NODE).
*** if NEXT-NODE already REACHED, do nothing.
*** If previously seen, check whether path thru BASE-NODE is next to base.
*** If not previously seen, link next to base node.
if NEXT-BASE-EDGE = SPOUSE
  move 1.8 to WEIGHT-THIS-EDGE
else
  move 1.0 to WEIGHT-THIS-EDGE.
if REACHED-STATUS of PERSON (NEXT-NODE) not = REACHED
  add WEIGHT-THIS-EDGE, DISTANCE-FROM-SOURCE of PERSON (BASE-NODE) giving DISTANCE-THRU-BASE-NODE
if REACHED-STATUS of PERSON (NEXT-NODE) = NOT-SEEN
  move NEARBY to REACHED-STATUS of PERSON (NEXT-NODE)
  set LAST-NEARBY-INDEX up by 1
  set NEARBY-NODE (LAST-NEARBY-INDEX) to NEXT-NODE
  perform LINK-NEXT-NODE-TO-BASE-NODE
else
  *** REACHED-STATUS = NEARBY
  if DISTANCE-THRU-BASE-NODE < DISTANCE-FROM-SOURCE of PERSON (NEXT-NODE)
    perform LINK-NEXT-NODE-TO-BASE-NODE.

LINK-NEXT-NODE-TO-BASE-NODE.
*** link next to base by re-setting its predecessor index to point to base, note type of edge, and re-set distance
*** as it is through base node.
move DISTANCE-THRU-BASE-NODE to DISTANCE-FROM-SOURCE of PERSON (NEXT-NODE).
set PATH-PREDECESSOR of PERSON (NEXT-NODE) to BASE-NODE.
move NEXT-BASE-EDGE to EDGE-TO-PREDECESSOR of PERSON (NEXT-NODE).

FIND-CLOSEST-UNREACHED-NODE.
set NEXT-NODE to NEARBY-NODE (THIS-NEARBY-INDEX).
if DISTANCE-FROM-SOURCE of PERSON (NEXT-NODE) < MINIMAL-DISTANCE
  set BEST-NEARBY-INDEX to THIS-NEARBY-INDEX
  move DISTANCE-FROM-SOURCE of PERSON (NEXT-NODE) to MINIMAL-DISTANCE.
RESOLVE-PATH-TO-ENGLISH.
*** RESOLVE-PATH-TO-ENGLISH condenses the shortest path to a
*** series of RELATIONSHIPS for which there are English
*** descriptions.
*** Key persons are the ones in the RELATIONSHIP path which remain
*** after the path is condensed.

display " Shortest path between identified persons: ".
set THIS-NODE to TARGET-INDEX.
*** Display path and initialize KEY-PERSON array from path elements.
perform TRAVERSE-SHORTEST-PATH varying KEY-INDEX from 1 by 1
   until THIS-NODE = SOURCE-INDEX.
display " ", NAME of PERSON (THIS-NODE).
set PERSON-INDEX of KEY-PERSON (KEY-INDEX) to THIS-NODE.
move NULL-RELATION to RELATION-TO-NEXT of KEY-PERSON (KEY-INDEX).
move NULL-RELATION to RELATION-TO-NEXT of KEY-PERSON (KEY-INDEX + 1).
*** Resolve CHILD-_PARENT and CHILD-SPouse-PARENT relations
*** to SIBLING relations.
perform FIND-SIBLINGS varying KEY-INDEX from 1 by 1
   until RELATION-TO-NEXT of KEY-PERSON (KEY-INDEX) = NULL-RELATION.
*** Resolve CHILD-CHILD-... and PARENT-PARENT-... relations to
*** direct descendant or ancestor relations.
perform FIND-ANCESTORS-OR-DESCENDANTS varying KEY-INDEX from 1 by 1
   until RELATION-TO-NEXT of KEY-PERSON (KEY-INDEX) = NULL-RELATION.
*** Resolve CHILD-SIBLING-PARENT to COUSIN,
*** CHILD-SIBLING to NEPHEW,
*** SIBLING-PARENT to UNCLE.
perform FIND-COUSINS-NEPHEWS-UNCLES varying KEY-INDEX from 1 by 1
   until RELATION-TO-NEXT of KEY-PERSON (KEY-INDEX) = NULL-RELATION.
*** Loop below will pick out valid adjacent strings of elements
*** to be displayed.  KEY-INDEX points to first element,
*** LATER-KEY-INDEX to last element, and PRIMARY-INDEX to the
*** element which determines the primary English word to be used.
*** Associativity of adjacent elements in condensed table
*** is based on English usage.
set KEY-INDEX to 1.
display " Condensed path: ".
perform CONSOLIDATE-ADJACENT-PERSONS
   until RELATION-TO-NEXT of KEY-PERSON (KEY-INDEX) = NULL-RELATION
set THIS-NODE to PERSON-INDEX of KEY-PERSON (KEY-INDEX).
display " ", NAME of PERSON (THIS-NODE).
*** end of RESOLVE-PATH-TO-ENGLISH
TRAVERSE-SHORTEST-PATH.
set PERSON-INDEX of KEY-PERSON (KEY-INDEX) to THIS-NODE.
move FULL to PROXIMITY of KEY-PERSON (KEY-INDEX).
move EDGE-TO-PREDECESSOR of PERSON (THIS-NODE) to RELATION-TO-NEXT of KEY-PERSON (KEY-INDEX).
if EDGE-TO-PREDECESSOR of PERSON (THIS-NODE) = SPOUSE
move zero to GENERATION-GAP of KEY-PERSON (KEY-INDEX)
display "", NAME of PERSON (THIS-NODE), "is spouse of"
else
move 1 to GENERATION-GAP of KEY-PERSON (KEY-INDEX)
if EDGE-TO-PREDECESSOR of PERSON (THIS-NODE) = PARENT
display "", NAME of PERSON (THIS-NODE), "is parent of"
else
*** edge is child-type
display "", NAME of PERSON (THIS-NODE), "is child of".
set THIS-NODE to PATH-PREDECESSOR of PERSON (THIS-NODE).

FIND-SIBLINGS.
if RELATION-TO-NEXT of KEY-PERSON (KEY-INDEX) = CHILD
move RELATION-TO-NEXT of KEY-PERSON (KEY-INDEX + 1) to LATER-KEY-RELATION
if LATER-KEY-RELATION = PARENT
*** then found either full or half SIBLINGS
perform SET-UP-FULL-HALF-SIBLING
else
if LATER-KEY-RELATION = SPOUSE and
RELATION-TO-NEXT of KEY-PERSON (KEY-INDEX + 2) = PARENT
*** then found step-siblings
move zero to GENERATION-GAP of KEY-PERSON (KEY-INDEX)
move STEP to PROXIMITY of KEY-PERSON (KEY-INDEX)
move SIBLING to RELATION-TO-NEXT of KEY-PERSON (KEY-INDEX)
move 2 to GAP-SIZE
perform CONDENSE-KEY-PERSONS.

SET-UP-FULL-HALF-SIBLING.
*** Determines whether two PERSONs are full siblings, i.e.,
*** have the same two parents.
set INDEX1 to PERSON-INDEX of KEY-PERSON (KEY-INDEX).
set INDEX2 to PERSON-INDEX of KEY-PERSON (KEY-INDEX + 2).
if (NULL-IDENT not =
RELATIVE-IDENTIFIER of PERSON (INDEX1, FATHER-IDENT)
and RELATIVE-IDENTIFIER of PERSON (INDEX1, MOTHER-IDENT))
and (RELATIVE-IDENTIFIER of PERSON (INDEX1, FATHER-IDENT) =
RELATIVE-IDENTIFIER of PERSON (INDEX2, FATHER-IDENT))
and (RELATIVE-IDENTIFIER of PERSON (INDEX1, MOTHER-IDENT) =
RELATIVE-IDENTIFIER of PERSON (INDEX2, MOTHER-IDENT))
move FULL to PROXIMITY of KEY-PERSON (KEY-INDEX)
else
move HALF to PROXIMITY of KEY-PERSON (KEY-INDEX).
move zero to GENERATION-GAP of KEY-PERSON (KEY-INDEX).
move SIBLING to RELATION-TO-NEXT of KEY-PERSON (KEY-INDEX).
move 1 to GAP-SIZE.
perform CONDENSE-KEY-PERSONS.
FIND-ANCESTORS-OR-DESCENDANTS.
    if RELATION-TO-NEXT of KEY-Person (KEY-INDEX) = CHILD or PARENT
      perform NULL varying LATER-KEY-INDEX from KEY-INDEX by 1
      until RELATION-TO-NEXT of KEY-Person (LATER-KEY-INDEX) not =
        RELATION-TO-NEXT of KEY-Person (KEY-INDEX)
      set GENERATION-COUNT to LATER-KEY-INDEX
      set TEMP-NUMBER to KEY-INDEX
      subtract TEMP-NUMBER from GENERATION-COUNT
    if GENERATION-COUNT > 1
      ***
      compress generations
      move GENERATION-COUNT to GENERATION-GAP of KEY-Person (KEY-INDEX)
      subtract 1 from GENERATION-COUNT giving GAP-SIZE
      perform CONDENSE-KEY-PERSONS.

FIND-COUSINS-NEPHEWS-UNCLES.
    move RELATION-TO-NEXT of KEY-Person (KEY-INDEX + 1) to LATER-KEY-RELATION
    if RELATION-TO-NEXT of KEY-Person (KEY-INDEX) = CHILD and
      LATER-KEY-RELATION = SIBLING
      *** then COUSIN or NEPHEW
        if RELATION-TO-NEXT of KEY-Person (KEY-INDEX + 2) = PARENT
          perform FOUND-COUSIN
        else
          ***
          found NEPHEW
          move PROXIMITY of KEY-Person (KEY-INDEX + 1) to
          PROXIMITY of KEY-Person (KEY-INDEX)
          move NEPHEW to RELATION-TO-NEXT of KEY-Person (KEY-INDEX)
          move 1 to GAP-SIZE
          perform CONDENSE-KEY-PERSONS
        else
          if RELATION-TO-NEXT of KEY-Person (KEY-INDEX) = SIBLING and
            LATER-KEY-RELATION = PARENT
            *** then found UNCLE
              move GENERATION-GAP of KEY-Person (KEY-INDEX + 1) to
              GENERATION-GAP of KEY-Person (KEY-INDEX)
              move UNCLE to RELATION-TO-NEXT of KEY-Person (KEY-INDEX)
              move 1 to GAP-SIZE
              perform CONDENSE-KEY-PERSONS.
          FOUND-COUSIN.
            if GENERATION-GAP of KEY-Person (KEY-INDEX)
              < GENERATION-GAP of KEY-Person (KEY-INDEX + 2)
              move GENERATION-GAP of KEY-Person (KEY-INDEX)
              to COUSIN-RANK of KEY-Person (KEY-INDEX)
            else
              move GENERATION-GAP of KEY-Person (KEY-INDEX + 2)
              to COUSIN-RANK of KEY-Person (KEY-INDEX).
              ***
              subtract moves in absolute value since GENERATION-GAP is unsigned
              subtract GENERATION-GAP of KEY-Person (KEY-INDEX + 2)
              from GENERATION-GAP of KEY-Person (KEY-INDEX).
              move PROXIMITY of KEY-Person (KEY-INDEX + 1)
              to PROXIMITY of KEY-Person (KEY-INDEX).
              move COUSIN to RELATION-TO-NEXT of KEY-Person (KEY-INDEX).
              move 2 to GAP-SIZE.
              perform CONDENSE-KEY-PERSONS.
            NULL.
            exit.
CONDENSE-KEY-PERSONS.

*** CONDENSE-KEY-PERSONS condenses superfluous entries from the
*** KEY-PERSON array, starting at KEY-INDEX.

set RECEIVE-INDEX to KEY-INDEX.
set RECEIVE-INDEX up by 1.
set SEND-INDEX to RECEIVE-INDEX.
set SEND-INDEX up by GAP-SIZE.
perform SLIDE-IT-DOWN varying RECEIVE-INDEX from RECEIVE-INDEX by 1
until RELATION-TO-NEXT of KEY-PERSON (RECEIVE-INDEX - 1)
= NULL-RELATION.

SLIDE-IT-DOWN.
move KEY-PERSON (SEND-INDEX) to KEY-PERSON (RECEIVE-INDEX).
set SEND-INDEX up by 1.

CONSOLIDATE-ADJACENT-PERSONS.
move RELATION-TO-NEXT of KEY-PERSON (KEY-INDEX) to KEY-RELATION.
set LATER-KEY-INDEX, PRIMARY-INDEX to KEY-INDEX.
if RELATION-TO-NEXT of KEY-PERSON (KEY-INDEX + 1) not = NULL-RELATION
perform SEEK-MULTI-ELEMENT-COMBINATION.
set FIRST-INDEX to KEY-INDEX.
set LAST-INDEX to LATER-KEY-INDEX.
perform DISPLAY-RELATION.
set KEY-INDEX to LATER-KEY-INDEX.
set KEY-INDEX up by 1.

SEEK-MULTI-ELEMENT-COMBINATION.
move IS-TRUE to ANOTHER-ELEMENT-POSSIBLE.
if KEY-RELATION = SPOUSE
  set LATER-KEY-INDEX up by 1
  set PRIMARY-INDEX up by 1
  if RELATION-TO-NEXT of KEY-PERSON (LATER-KEY-INDEX)
    = SIBLING or COUSIN
    *** then nothing can follow spouse-sibling or spouse-cousin
    move IS-FALSE to ANOTHER-ELEMENT-POSSIBLE.
*** PRIMARY-INDEX is now correctly set. Next if-statement
*** determines if a following SPOUSE relation should be
*** appended to this combination or left for the next
*** combination.
if RELATION-TO-NEXT of KEY-PERSON (PRIMARY-INDEX + 1) = SPOUSE
  and ANOTHER-ELEMENT-IS-POSSIBLE
  *** Only a SPOUSE can follow a Primary
  check primary preceding and following SPOUSE.
  move RELATION-TO-NEXT of KEY-PERSON (PRIMARY-INDEX)
  to PRIMARY-RELATION
  move RELATION-TO-NEXT of KEY-PERSON (PRIMARY-INDEX + 2)
  to NEXT-PRIMARY-RELATION
  if (NEXT-PRIMARY-RELATION = NEPHEW or COUSIN or NULL-RELATION)
    or (PRIMARY-RELATION = NEPHEW)
    or ( (PRIMARY-RELATION = SIBLING or PARENT)
      and NEXT-PRIMARY-RELATION not = UNCLE )
    *** then append following SPOUSE with this combination.
    set LATER-KEY-INDEX up by 1.
DISPLAY-RELATION.

*** DISPLAY-RELATION takes 1, 2, or 3 adjacent elements in the condensed table and generates the English description of the relation between the first and last + 1 elements.

move RELATION-TO-NEXT of KEY-PERSON (FIRST-INDEX) to FIRST-RELATION.
move RELATION-TO-NEXT of KEY-PERSON (LAST-INDEX) to LAST-RELATION.
move RELATION-TO-NEXT of KEY-PERSON (PRIMARY-INDEX) to PRIMARY-RELATION.

*** set THIS-PROXIMITY
if (PRIMARY-RELATION = PARENT and FIRST-RELATION = SPOUSE) or (PRIMARY-RELATION = CHILD and LAST-RELATION = SPOUSE)
move STEP to THIS-PROXIMITY
else
if PRIMARY-RELATION = SIBLING or UNCLE or NEPHEW or COUSIN
move PROXIMITY of KEY-PERSON (PRIMARY-INDEX) to THIS-PROXIMITY
else
move FULL to THIS-PROXIMITY.

*** set THIS GENERATION GAP
if PRIMARY-RELATION = PARENT or CHILD or UNCLE or NEPHEW or COUSIN
move GENERATION GAP of KEY-PERSON (PRIMARY-INDEX) to THIS GENERATION GAP
else
move zero to THIS GENERATION GAP.

*** set INLAW
if (FIRST-RELATION = SPOUSE) and (PRIMARY-RELATION = SIBLING or CHILD or NEPHEW or COUSIN)
move IS TRUE to INLAW
else
if (LAST-RELATION = SPOUSE) and (PRIMARY-RELATION = SIBLING or PARENT or UNCLE or COUSIN)
move IS TRUE to INLAW
else
move IS FALSE to INLAW.

*** set THIS COUSIN RANK
if PRIMARY-RELATION = COUSIN
move COUSIN RANK of KEY PERSON (PRIMARY INDEX) to THIS COUSIN RANK
else
move zero to THIS COUSIN RANK.
*** parameters are set - now generate display.

set THIS-NODE to PERSON-INDEX of KEY-PERSON (FIRST-INDEX).
move spaces to DISPLAY-BUFFER.
move 1 to DISPLAY-POINTER.
string " ", NAME of PERSON (THIS-NODE), " is "
delimited by size
into DISPLAY-BUFFER with pointer DISPLAY-POINTER.
if PRIMARY-RELATION = PARENT or CHILD or UNCLE or NEPHEW
perform GENERATE-GENERATION-QUALIFIER
else
if (PRIMARY-RELATION = COUSIN) and (THIS-COUSIN-RANK > 1)
move THIS-COUSIN-RANK to TWO-DIGIT-FIELD
string TWO-DIGIT-FIELD delimited by size into DISPLAY-BUFFER
with pointer DISPLAY-POINTER
divide THIS-COUSIN-RANK by 10 giving TEMP-NUMBER
remainder SUFFIX-INDICATOR
if SUFFIX-INDICATOR = 1
string "st " delimited by size
into DISPLAY-BUFFER with pointer DISPLAY-POINTER
else if SUFFIX-INDICATOR = 2
string "nd " delimited by size
into DISPLAY-BUFFER with pointer DISPLAY-POINTER
else if SUFFIX-INDICATOR = 3
string "rd " delimited by size
into DISPLAY-BUFFER with pointer DISPLAY-POINTER
else
string "th " delimited by size
into DISPLAY-BUFFER with pointer DISPLAY-POINTER.
if THIS-PROXIMITY = STEP
string "step-" delimited by size
into DISPLAY-BUFFER with pointer DISPLAY-POINTER
else
if THIS-PROXIMITY = HALF
string "half-" delimited by size
into DISPLAY-BUFFER with pointer DISPLAY-POINTER.
set THIS-NODE to PERSON-INDEX of KEY-PERSON (FIRST-INDEX).
move GENDER of PERSON (THIS-NODE) to THIS-GENDER.
set MALE-INDEX, FEMALE-INDEX to PRIMARY-RELATION.
if THIS-GENDER = MALE
string PRIMARY-MALE-NAME (MALE-INDEX) delimited by space
into DISPLAY-BUFFER with pointer DISPLAY-POINTER
else
string PRIMARY-FEMALE-NAME (FEMALE-INDEX) delimited by space
into DISPLAY-BUFFER with pointer DISPLAY-POINTER.
if RELATION-IS-INLAW
string "-in-law" delimited by size
into DISPLAY-BUFFER with pointer DISPLAY-POINTER.
if (PRIMARY-RELATION = COUSIN) and (THIS-GENERATION-GAP > 0)
  if THIS-GENERATION-GAP > 1
    move THIS-GENERATION-GAP to TWO-DIGIT-FIELD
    string " ", TWO-DIGIT-FIELD, " times removed"
    delimited by size
    into DISPLAY-BUFFER with pointer DISPLAY-POINTER
  else
    string " once removed" delimited by size
    into DISPLAY-BUFFER with pointer DISPLAY-POINTER.
  string " of" delimited by size
  into DISPLAY-BUFFER with pointer DISPLAY-POINTER.
display DISPLAY-BUFFER.

GENERATE-GENERATION-QUALIFIER.
  if THIS-GENERATION-GAP not < 3
    string "great" delimited by size
    into DISPLAY-BUFFER with pointer DISPLAY-POINTER
  if THIS-GENERATION-GAP > 3
    subtract 2 from THIS-GENERATION-GAP giving TWO-DIGIT-FIELD
    string "*, TWO-DIGIT-FIELD, "-" delimited by size
    into DISPLAY-BUFFER with pointer DISPLAY-POINTER
  else
    string "-" delimited by size
    into DISPLAY-BUFFER with pointer DISPLAY-POINTER.
  if THIS-GENERATION-GAP not < 2
    string "grand-" delimited by size
    into DISPLAY-BUFFER with pointer DISPLAY-POINTER.
* ---- Compilation unit number 3 ----

identification division.
program-id. COMGENES.

* COMGENES assumes that each ancestor contributes
* half of the genetic material to a PERSON. It finds common
* ancestors between two PERSONs and computes the expected
* value of the PROPORTION of common material.

environment division.

configuration section.
source-computer. VAX-11.
object-computer. VAX-11.

data division.
working-storage section.

01 RELATION-TYPE.
   05 PARENT    pic 9    value 1.
   05 CHILD     pic 9    value 2.
   05 SPOUSE    pic 9    value 3.
   05 SIBLING   pic 9    value 4.
   05 UNCLE     pic 9    value 5.
   05 NEPHEW    pic 9    value 6.
   05 COUSIN    pic 9    value 7.
   05 NULL-RELATION pic 9    value 8.

01 AUXILIARY-VARIABLES.
   05 COMMON-PROPORTION   pic 9V9999999999.
   05 MATCH-IDENTIFIER    pic 999.
   05 TEN-DIGIT-FIELD     pic 9.9999999999.

01 STACKED-VARIABLES.
*** used to simulate recursion
   05 STACK-ENTRY occurs 50 times indexed by STACK-INDEX.
      10 PROPORTION    pic 9V9999999999.
      10 THIS-CONTRIBUTION pic 9V9999999999.
      10 ALREADY-COUNTED pic 9V9999999999.
      10 PERSON-INDEX   usage index.
      10 NEXT-NEIGHBOR  pic 999.
linkage section.

77 PARM-INDEX1 pic 999.
77 PARM-INDEX2 pic 999.

01 PERSON-TABLE.
  05 NUMBER-OF-PERSONS usage index.
  05 PERSON occurs 300 times indexed by INDEX1, INDEX2, THIS-NODE.

*** static information - filled from PEOPLE file:
  10 NAME pic X(20).
  10 IDENTIFIER pic 999.
  10 GENDER pic X.

*** IDENTIFIERS of immediate relatives - father, mother, spouse
  10 IMMEDIATE-RELATIONS.
     15 RELATIVE-IDENTIFIER occurs 3 times indexed by RELATIONSHIP pic 999.

*** pointers to immediate neighbors in graph
  10 NEIGHBOR-COUNT pic 99.
  10 NEIGHBOR-RECORD occurs 20 times indexed by THIS-NEIGHBOR.
     15 NEIGHBOR-INDEX usage index.
     15 NEIGHBOR-EDGE pic 9.

*** data used when traversing graph to resolve user request:
  10 DISTANCE-FROM-SOURCE pic 99999999.
  10 PATH-PREDECESSOR usage index.
  10 EDGE-TO-PREDECESSOR pic 9.
  10 REACHED-STATUS pic 9.

*** data used to compute common genetic material
  10 DESCENDANT-IDENTIFIER pic 999.
  10 DESCENDANT-GENES pic 9V99999999.
procedure division using
   PARM-INDEX1, PARM-INDEX2, PERSON-TABLE.

MAIN-LINE.
   set INDEX1 to PARM-INDEX1.
   set INDEX2 to PARM-INDEX2.
*** First zero out all ancestors to allow adding. This is necessary
*** because there might be two paths to an ancestor.
   set STACK-INDEX to 1.
   set PERSON-INDEX (STACK-INDEX) to INDEX1.
   move zero to NEXT-NEIGHBOR (STACK-INDEX).
   perform ZERO-PROPORTION until STACK-INDEX < 1.

*** now mark with shared PROPORTION
   move IDENTIFIER of PERSON (INDEX1) to MATCH-IDENTIFIER.
   set STACK-INDEX to 1.
   set PERSON-INDEX (STACK-INDEX) to INDEX1.
   move zero to NEXT-NEIGHBOR (STACK-INDEX).
   move 1.0 to PROPORTION (STACK-INDEX).
   perform MARK-PROPORTION until STACK-INDEX < 1.

*** traverse ancestor tree for INDEX2, summing overlap
*** with marked tree of INDEX1
   move zero to COMMON-PROPORTION
   set STACK-INDEX to 1.
   set PERSON-INDEX (STACK-INDEX) to INDEX2.
   move IDENTIFIER of PERSON (INDEX1) to MATCH-IDENTIFIER.
   move zero to NEXT-NEIGHBOR (STACK-INDEX).
   move 1.0 to PROPORTION (STACK-INDEX).
   move zero to ALREADY-COUNTED (STACK-INDEX).
   perform CHECK-COMMON-PROPORTION until STACK-INDEX < 1.
   move COMMON-PROPORTION to TEN-DIGIT-FIELD.
   display " Proportion of common genetic material = ", TEN-DIGIT-FIELD.

END-OF-COMGENES.
   exit program.

ZERO-PROPORTION.
*** ZERO-PROPORTION recursively seeks out all ancestors and
*** zeros them out.
   set THIS-NODE to PERSON-INDEX (STACK-INDEX).
   if NEXT-NEIGHBOR (STACK-INDEX) = zero
      move zero to DESCENDANT-GENES of PERSON (THIS-NODE)
      move 1 to NEXT-NEIGHBOR (STACK-INDEX).
   perform NULL
      varying THIS-NEIGHBOR from NEXT-NEIGHBOR (STACK-INDEX) by 1
      until THIS-NEIGHBOR > NEIGHBOR-COUNT (THIS-NODE)
      or NEIGHBOR-EDGE (THIS-NODE, THIS-NEIGHBOR) = PARENT.
   if THIS-NEIGHBOR > NEIGHBOR-COUNT (THIS-NODE)
*** then no more ancestors
   set STACK-INDEX down by 1
else
*** set up for next ancestor
   set NEXT-NEIGHBOR (STACK-INDEX) to THIS-NEIGHBOR
   add 1 to NEXT-NEIGHBOR (STACK-INDEX)
   set STACK-INDEX up by 1
   set PERSON-INDEX (STACK-INDEX)
      to NEIGHBOR-INDEX (THIS-NODE, THIS-NEIGHBOR)
   move zero to NEXT-NEIGHBOR (STACK-INDEX).
MARK-PROPORTION.
*** MARK-PROPORTION recursively seeks out all ancestors and
*** marks them with the sender's PROPORTION of shared
*** genetic material. This PROPORTION is diluted by one-half
*** for each generation.

set THIS-NODE to PERSON-INDEX (STACK-INDEX).
if NEXT-NEIGHBOR (STACK-INDEX) = zero
  move MATCH-IDENTIFIER
    to DESCENDANT-IDENTIFIER of PERSON (THIS-NODE)
  add PROPORTION (STACK-INDEX)
    to DESCENDANT-GENES of PERSON (THIS-NODE)
  move 1 to NEXT-NEIGHBOR (STACK-INDEX).
perform NULL
  varying THIS-NEIGHBOR from NEXT-NEIGHBOR (STACK-INDEX) by 1
  until THIS-NEIGHBOR > NEIGHBOR-COUNT (THIS-NODE)
    or NEIGHBOR-EDGE (THIS-NODE, THIS-NEIGHBOR) = PARENT.
if THIS-NEIGHBOR > NEIGHBOR-COUNT (THIS-NODE)
  *** then no more ancestors
  set STACK-INDEX down by 1
else
  *** set up for next ancestor
  set NEXT-NEIGHBOR (STACK-INDEX) to THIS-NEIGHBOR
  add 1 to NEXT-NEIGHBOR (STACK-INDEX)
  set STACK-INDEX up by 1
  set PERSON-INDEX (STACK-INDEX)
    to NEIGHBOR-INDEX (THIS-NODE, THIS-NEIGHBOR)
  move zero to NEXT-NEIGHBOR (STACK-INDEX)
  divide PROPORTION (STACK-INDEX - 1) by 2 giving
    PROPORTION (STACK-INDEX).
CHECK-COMMON-PROPORTION.
*** CHECK-COMMON-PROPORTION searches all the ancestors of
*** CHECK-INDEX to see if any have been marked, and if so
*** adds the appropriate amount to COMMON-PROPORTION.

set THIS-NODE to PERSON-INDEX (STACK-INDEX).
if NEXT-NEIGHBOR (STACK-INDEX) = zero
  move 1 to NEXT-NEIGHBOR (STACK-INDEX)
  if DESCENDANT-IDENTIFIER of PERSON (THIS-NODE) = MATCH-IDENTIFIER
    *** Increment COMMON-PROPORTION by the contribution of
    *** this common ancestor, but discount for the contribution
    *** of less remote ancestors already counted.
    multiply DESCENDANT-GENES of PERSON (THIS-NODE)
      by PROPORTION (STACK-INDEX)
      giving THIS-CONTRIBUTION (STACK-INDEX)
    compute COMMON-PROPORTION = COMMON-PROPORTION
    + THIS-CONTRIBUTION (STACK-INDEX)
    - ALREADY-COUNTED (STACK-INDEX)
  else
    move zero to THIS-CONTRIBUTION (STACK-INDEX).
perform NULL
  varying THIS-NEIGHBOR from NEXT-NEIGHBOR (STACK-INDEX) by 1
  until THIS-NEIGHBOR > NEIGHBOR-COUNT (THIS-NODE)
    or NEIGHBOR-EDGE (THIS-NODE, THIS-NEIGHBOR) = PARENT.
  if THIS-NEIGHBOR > NEIGHBOR-COUNT (THIS-NODE)
    *** then no more ancestors
      set STACK-INDEX down by 1
  else
    *** set up for next ancestor
      set NEXT-NEIGHBOR (STACK-INDEX) to THIS-NEIGHBOR
      add 1 to NEXT-NEIGHBOR (STACK-INDEX)
      set STACK-INDEX up by 1
      set PERSON-INDEX (STACK-INDEX)
        to NEIGHBOR-INDEX (THIS-NODE, THIS-NEIGHBOR)
      move zero to NEXT-NEIGHBOR (STACK-INDEX)
      divide PROPORTION (STACK-INDEX - 1) by 2 giving
        PROPORTION (STACK-INDEX)
      divide THIS-CONTRIBUTION (STACK-INDEX - 1) by 4 giving
        ALREADY-COUNTED (STACK-INDEX).

NULL.
  exit.
6.0 FORTRAN

In keeping with the general convention of the examples, language-supplied keywords and identifiers are written in lower case in the program. To conform strictly to the FORTRAN standard, however, programs must use only upper-case letters.

program RELATE

c Establish global constants

integer MAXPRS, NAMLEN, IDLEN, BUFLEN,
1 MSGLEN, MAXNBR, MAXGVN
parameter (MAXPRS = 300, NAMLEN = 20, IDLEN = 3, BUFLEN = 60,
1 MSGLEN = 40, MAXNBR = 20, MAXGVN = 3)

ccharacter NULLID*(IDLEN)
parameter (NULLID = "000")

Each PERSON's record in the file identifies at most three others directly related: father, mother, and spouse

integer FATHID, MOTHID, SPOUID
parameter (FATHID = 1, MOTHID = 2, SPOUID = 3)

ccharacter REQOK*10, REQSTP*A
parameter (REQOK = "Request OK", REQSTP = "stop")

ccharacter MALE*1, FEMALE*1
parameter (MALE = "M", FEMALE = "F")

integer PARENT, CHILD, SPOUSE, SIBLNG,
1 UNCLE, NEPHEW, COUSIN, NULLRL
parameter (PARENT = 1, CHILD = 2, SPOUSE = 3, SIBLNG = 4,
1 UNCLE = 5, NEPHEW = 6, COUSIN = 7, NULLRL = 8)

These common blocks hold the PERSON array, which is global to the entire program.

common /PERNUM/ NBRCNT, NBRD exacerb, NBRRED, DSTSRC, PATHPR,
1 EDGRD, RCHST, DSCGEN, NUMPER

common /PERCHR/ NAME, IDENT, GENDER, RELID, DSCID
The following data items constitute the PERSON array, which is the central repository of information about inter-relationships.

- **Static information** - filled from PEOPLE file:
  - CHARACTER*(NAMLEN) NAME (MAXPRS)
  - CHARACTER*(IDLEN) IDENT (MAXPRS)
  - CHARACTER*1 GENDER (MAXPRS)

- **IDENTs of immediate relatives** - father, mother, spouse:
  - CHARACTER*(IDLEN) RELID (MAXPRS, MAXGVN)

- **Pointers to immediate neighbors in graph**:
  - INTEGER NBRCNT (MAXPRS)
  - INTEGER NBRDEX (MAXPRS, MAXNBR)
  - INTEGER NBREDG (MAXPRS, MAXNBR)

- **Data used when traversing graph to resolve user request**:
  - REAL DSTSRC (MAXPRS)
  - INTEGER PATHPR (MAXPRS)
  - INTEGER EDGPRD (MAXPRS)
  - INTEGER RCHST (MAXPRS)

- **Data used to compute common genetic material**:
  - CHARACTER*(IDLEN) DSCID (MAXPRS)
  - REAL DSCGEN (MAXPRS)

- **NUMPER** keeps track of the actual number of persons:
  - INTEGER NUMPER

*** End of declarations for common data ***

These variables are used when establishing the PERSON array from the PEOPLE file:

- INTEGER CURRNT, PRVDEX
- CHARACTER*(IDLEN) PREVID, CURRID
- INTEGER RELSHP

These variables are used to accept and resolve requests for RELSHP information:

- INTEGER BUFDEX, SEMLOC
- CHARACTER*(BUFLEN) REQBUF
- CHARACTER*(NAMLEN) P1IDNT, P2IDNT
- INTEGER P1FND, P2FND
- CHARACTER*(MSGLEN) ERRMSG
- INTEGER P1DEX, P2DEX
- CHARACTER*7 PRNOUN
c *** execution of main sequence begins here ***

open (unit=10, file=`PEOPLE.DAT`, status=`old`, form=`formatted`)  

c This loop reads in the PEOPLE file and constructs the PERSON  
c array from it (one PERSON = one record = one array entry).  
c As records are read in, links are constructed to represent the  
PARENT-CHILD or SPOUSE relationship. The array then implements  
a directed graph which is used to satisfy subsequent user  
c requests. The file is assumed to be correct − no validation  
c is performed on it.

do 110 CURRNT=1, MAXPRS
   c copy direct information from file to array
   read (unit=10, fmt=`(a20, a3, a1, 3a3)`, end=111)  
      NAME(CURRNT), IDENT(CURRNT), GENDER(CURRNT),  
      ((RELID(CURRNT,ITEMP), ITEMP=FATHID, SPOUID))  
   c Location of adjacent persons as yet undetermined
   NBRCNT (CURRNT) = 0
   c Descendants as yet undetermined
   DSCID (CURRNT) = NULLID
   c Compare this PERSON against all previously entered PERSONs
   c to search for relationships.
   CURRID = IDENT (CURRNT)
   do 120 PRVDEX = 1, CURRNT-1  
      PREVID = IDENT (PRVDEX)  
      c Search for father, mother, or spouse relationship in  
      c either direction between this and previous PERSON.  
      c Assume at most one relationship exists.
      do 130 RELSHP = FATHID, SPOUID  
         if (PREVID .eq. RELID (CURRNT, RELSHP)) then  
            call LNKREL (CURRNT, RELSHP, PRVDEX)  
            goto 131  
         else if (CURRID .eq. RELID (PRVDEX, RELSHP)) then  
            call LNKREL (PRVDEX, RELSHP, CURRNT)  
            goto 131  
         end if  
      130 continue
      131 continue
      120 continue
   110 continue
   NUMPER = CURRNT - 1
   close (unit=10, status=`keep`)  

c PERSON array is now loaded and edges between immediate relatives  
c (PARENT-CHILD or SPOUSE-SPOUSE) are established.
c Loop accepts requests and finds relationship (if any) between pairs of PERSONs.

200 continue
    call PROMPT (REQBUF)
    if (REQBUF .eq. REQSTP) goto 201
    call CHKRQS (REQBUF, ERRMSG, P1IDNT, P2IDNT)

    c Syntax check of request completed. Now either display error message or search for the two PERSONs.
    if (ERRMSG .eq. REQOK) then
        Request syntactically correct - search for requested PERSONs.
        call SEEKPR (P1IDNT, P2IDNT, P1DEX, P2DEX,
                     P1FND, P2FND)
        if (P1FND .eq. 1 .and. P2FND .eq. 1) then
            Exactly one match for each PERSON - proceed to determine relationship, if any.
            if (P1DEX .eq. P2DEX) then
                if (GENDER (P1DEX) .eq. MALE) then
                    PRNOUN = 'himself'
                else
                    PRNOUN = 'herself'
                end if
                write (unit=*, fmt=9002) NAME (P1DEX), PRNOUN
            else
                call FINDRL (P1DEX, P2DEX)
            end if
        else
            either not found or more than one found
            if (P1FND .eq. 0) then
                write (unit=*, fmt='(" First person not found.")')
            else if (P1FND .gt. 1) then
                write (unit=*,
                       fmt='(" Duplicate names for first person",
                        " - use numeric identifier.")')
            end if
            if (P2FND .eq. 0) then
                write (unit=*, fmt='(" Second person not found.")')
            else if (P2FND .gt. 1) then
                write (unit=*,
                       fmt='(" Duplicate names for second person",
                        " - use numeric identifier.")')
            end if
        end if
    end if
    end processing of syntactically legal request
    else
        write (unit=*, fmt=9004) ERRMSG
        format ('Incorrect request format: ', a40)
    end if
    goto 200
201 continue
    write (unit=*, fmt='(" End of relation-finder.")')
    c End of main line of RELATE
c procedures under RELATE

    subroutine LNKREL (FRMDEX, RELSHIP, TODEX)
    c establishes cross-indexing between immediately related PERSONs.
    integer FRMDEX, TODEX, RELSHIP
    c Each PERSON's record in the file identifies at most three
    c others directly related: father, mother, and spouse
    integer FATHID, MOTHID, SPOUID
    parameter (FATHID = 1, MOTHID = 2, SPOUID = 3)
    integer PARENT, CHILD, SPOUSE, SIBLNG,
        UNCLE, NEPHEW, COUSIN, NULLRL
    parameter (PARENT = 1, CHILD = 2, SPOUSE = 3, SIBLNG = 4,
        UNCLE = 5, NEPHEW = 6, COUSIN = 7, NULLRL = 8)
    if (RELSHP .eq. SPOUID) then
        call LNKONE (FRMDEX, SPOUSE, TODEX)
        call LNKONE (TODEX, SPOUSE, FRMDEX)
    else
        c RELSHIP is father or mother
        call LNKONE (FRMDEX, PARENT, TODEX)
        call LNKONE (TODEX, CHILD, FRMDEX)
    end if
end
subroutine LNKONE (FRMDEX, THSEDG, TODEX)
c Establishes the NBR pointers from one PERSON to another
integer FRMDEX, TODEX, THSEDG

integer MAXPRS, NAMLEN, IDLEN, BUFLEN,
1 MSGLEN, MAXNBR, MAXGVN
parameter (MAXPRS = 300, NAMLEN = 20, IDLEN = 3, BUFLEN = 60,
1 MSGLEN = 40, MAXNBR = 20, MAXGVN = 3)

character NULLID*(IDLEN)
parameter (NULLID = '000')

c These common blocks hold the PERSON array, which is global to
c the entire program.

common /PERNUM/ NBRCNT, NBRDEX, NBREDG, DSTSRC, PATHPR,
1 EDGPRD, RCHST, DSCGEN, NUMPER

common /PERCHR/ NAME, IDENT, GENDER, RELID, DSCID

c The following data items constitute the PERSON array, which
c is the central repository of information about inter-relationships.

c static information - filled from PEOPLE file
character*(NAMLEN) NAME (MAXPRS)
character*(IDLEN) IDENT (MAXPRS)
character*1 GENDER (MAXPRS)

c IDENTs of immediate relatives - father, mother, spouse
character*(IDLEN) RELID (MAXPRS, MAXGVN)


c pointers to immediate neighbors in graph
integer NBRCNT (MAXPRS)
integer NBRDEX (MAXPRS, MAXNBR)
integer NBREDG (MAXPRS, MAXNBR)

c data used when traversing graph to resolve user request:
real DSTSRC (MAXPRS)
integer PATHPR (MAXPRS)
integer EDGPRD (MAXPRS)
integer RCHST (MAXPRS)

c data used to compute common genetic material
character*(IDLEN) DSCID (MAXPRS)
real DSCGEN (MAXPRS)

c NUMPER keeps track of the actual number of persons
integer NUMPER

c *** end of declarations for common data ***

ITEMP = NBRCNT (FRMDEX) + 1
NBRCNT (FRMDEX) = ITEMP
NBRDEX (FRMDEX, ITEMP) = TODEX
NBREDG (FRMDEX, ITEMP) = THSEDG
end
subroutine PROMPT (REQBUF)
c  Issues prompt for user-request, reads in request,
c  blank-fills buffer, and skips to next line of input.

character*(*)  REQBUF

write (unit=*, fmt=9001)
9001 format (/,, 'Enter two person-identifiers (name or number),'
  2 /,, ' separated by semicolon. Enter "stop" to stop.' )

*** NOTE THAT THIS IS NOT A STANDARD WAY TO READ A LINE FROM
*** THE TERMINAL (see section 12.9.5.2.1). THE STANDARD
*** PROVIDES NO SUCH CAPABILITY.

read (unit=*, fmt=¨(a60)¨) REQBUF
end

subroutine CHKQRS (REQBUF, REQST, P1IDNT, P2IDNT)
c  Performs syntactic check on request in buffer.

integer  MAXPRS, NAMLEN, IDLEN, BUFLEN,
1  MSGLEN, MAXNBR, MAXGVN
parameter (MAXPRS = 300, NAMLEN = 20, IDLEN = 3, BUFLEN = 60,
1  MSGLEN = 40, MAXNBR = 20, MAXGVN = 3)

character  NULLID*(IDLEN)
parameter (NULLID = ¨000¨)

character  REQOK*10, REQSTP*4
parameter (REQOK = ¨Request OK¨, REQSTP = ¨stop¨)

character* (NAMLEN)  REQBUF*(BUFLEN), REQST*(MSGLEN)
character*(NAMLEN)  P1IDNT, P2IDNT, LTRIM
integer       SEMLOC

SEMLOC = INDEX (REQBUF,¨;¨)
P2IDNT = REQBUF (SEMLOC+1 : BUFLEN)

c  set REQST, based on results of scan of REQBUF, and

c  fill in P1IDNT and P2IDNT.
if (SEMLOC .eq. 0 .or. INDEX (P2IDNT, ';') .ne. 0) then
   REQST = 'must be exactly one semicolon.'
else
   if (SEMLOC .eq. 1) then
      P1IDNT = '
   else
      P1IDNT = REQBUF (1 : SEMLOC-1)
   end if
   if (P1IDNT .eq. ' ') then
      REQST = 'null field preceding semicolon.'
   else if (P2IDNT .eq. ' ') then
      REQST = 'null field following semicolon.'
   else
      REQST = REQOK
      P1IDNT = LTRIM (P1IDNT)
      P2IDNT = LTRIM (P2IDNT)
   end if
end if
end if
end character

character(*) function LTRIM (STRING)

   LTRIM deletes leading spaces and returns the resulting value.

   character(*) STRING

   do 100 ITEMP = 1, len(STRING)
      if (STRING (ITEMP : ITEMP) .ne. ' ') goto 101
   100 continue
  101 continue
   LTRIM = STRING (ITEMP : len(STRING))
end

subroutine SEEKPR (P1IDNT, P2IDNT, P1DEX, P2DEX, P1FND, P2FND)

   SEEKPR scans through the PERSON array, looking for the two
   requested PERSONs. Match may be by NAME or unique IDENT-number.

   integer MAXPRS, NAMLEN, IDLEN, BUFLEN,
   MSGLEN, MAXNBR, MAXGVN
   parameter (MAXPRS = 300, NAMLEN = 20, IDLEN = 3, BUFLEN = 60,
   MSGLEN = 40, MAXNBR = 20, MAXGVN = 3)

   character NULLID*(IDLEN)
   parameter (NULLID = '000')

   character*(NAMLEN) P1IDNT, P2IDNT
   integer P1DEX, P2DEX, P1FND, P2FND

   integer CURRNT
These common blocks hold the PERSON array, which is global to the entire program.

```
common /PERNUM/ NBRCNT, NBRDEX, NBREDG, DSTSRC, PATHPR,
                 1
             EDGPRD, RCHST, DSCGEN, NUMPER
```

```
common /PERCHR/ NAME, IDENT, GENDER, RELID, DSCID
```

The following data items constitute the PERSON array, which is the central repository of information about inter-relationships.

The static information - filled from PEOPLE file

```
character*(NAMLEN) NAME (MAXPRS)
character*(IDLEN) IDENT (MAXPRS)
character*1 GENDER (MAXPRS)
```

IDENTs of immediate relatives - father, mother, spouse

```
character*(IDLEN) RELID (MAXPRS, MAXGVN)
```

Pointers to immediate neighbors in graph

```
integer NBRCNT (MAXPRS)
integer NBRDEX (MAXPRS, MAXNBR)
integer NBREDG (MAXPRS, MAXNBR)
```

data used when traversing graph to resolve user request:

```
real DSTSRC (MAXPRS)
integer PATHPR (MAXPRS)
integer EDGPRD (MAXPRS)
integer RCHST (MAXPRS)
```

data used to compute common genetic material

```
character*(IDLEN) DSCID (MAXPRS)
real DSCGEN (MAXPRS)
```

NUMPER keeps track of the actual number of persons

```
integer NUMPER
```

*** end of declarations for common data ***

```
P1DEX = 0
P2DEX = 0
P1FND = 0
P2FND = 0
do 100 CURRNT = 1, NUMPER
```

Allow identification by name or number.

```
if (P1IDNT .eq. IDENT (CURRNT)) then
    P1FND = P1FND + 1
    P1DEX = CURRNT
end if
```

```
if (P2IDNT .eq. IDENT (CURRNT)) then
    P2FND = P2FND + 1
    P2DEX = CURRNT
end if
```

```
100 continue
end
subroutine FINDRL (TRGDEX, SRCDEX)
c Finds shortest path (if any) between two PERSONs and
c determines their relationship based on immediate relations

c traversed in path. PERSON array simulates a directed graph,
c and algorithm finds shortest path, based on following

c weights: PARENT-CHILD edge = 1.0

c SPouse-SPOUSE edge = 1.8

integer TRGDEX, SRCDEX

integer MAXPRS, NAMLEN, IDLEN, BUFLEN,
  MSGLEN, MAXNBR, MAXGVN
parameter (MAXPRS = 300, NAMLEN = 20, IDLEN = 3, BUFLEN = 60,
  MSGLEN = 40, MAXNBR = 20, MAXGVN = 3)

c A node in the graph (= PERSON) has either already been reached,
c is immediately adjacent to those reached, or farther away.

integer REACHD, NEARBY, NOSEEN
parameter (REACHD = 1, NEARBY = 2, NOSEEN = 3)

These common blocks hold the PERSON array, which is global to
c the entire program.

common /PERNUM/ NBRCNT, NBRDEX, NBREDG, DSTSRC, PATHPR,
  EDGPRD, RCHST, DSCGEN, NUMPER

common /PERCHR/ NAME, IDENT, GENDER, RELID, DSCID

The following data items constitute the PERSON array, which

c is the central repository of information about inter-relationships.

c static information - filled from PEOPLE file
  character*(NAMLEN) NAME (MAXPRS)
  character*(IDLEN) IDENT (MAXPRS)
  character*1 GENDER (MAXPRS)

identS of immediate relatives - father, mother, spouse
  character*(IDLEN) RELID (MAXPRS, MAXGVN)

pointers to immediate neighbors in graph
  integer NBRCNT (MAXPRS)
  integer NBRDEX (MAXPRS, MAXNBR)
  integer NBREDG (MAXPRS, MAXNBR)

data used when traversing graph to resolve user request:
  real DSTSRC (MAXPRS)
  integer PATHPR (MAXPRS)
  integer EDGPRD (MAXPRS)
  integer RCHST (MAXPRS)

data used to compute common genetic material
  character*(IDLEN) DSCID (MAXPRS)
  real DSCGEN (MAXPRS)

NUMPER keeps track of the actual number of persons

integer NUMPER

*** end of declarations for common data ***
integer PERDEX, THSNOD, ADJNOD, BSTDEX, LASTNR, NEARND (MAXPRS)
integer THSEDG, THSNBR
integer RELSHP
real MINDIS
integer SRCHNG, SUCCES, FAILED
parameter (SRCHNG = 1, SUCCES = 2, FAILED = 3)
integer SRCHST

c begin execution of FINDRL

c initialize PERSON-array for processing -
c mark all nodes as not seen
do 100 PERDEX = 1, NUMPER
   RCHST (PERDEX) = NOSEEN
100 continue
THSNOD = SRCDEX
c mark source node as reached
RCHST (THSNOD) = REACHD
DSTSRC (THSNOD) = 0.0
c no NEARBY nodes exist yet
LASTNR = 0
if (THSNOD .eq. TRGDEX) then
   SRCHST = SUCCES
else
   SRCHST = SRCHNG
end if
Loop keeps processing closest-to-source, unreached node until target reached, or no more connected nodes.

```
200 continue
   if (SRCHST .ne. SRCHNG) goto 201
c Process all nodes adjacent to THSNOD
do 210 THSNBR = 1, NBRCNT (THSNOD)
   call PROCAD (THSNOD, NBRDEX (THSNOD, THSNBR),
               NBREDG (THSNOD, THSNBR), NEARND, LASTNR)
210 continue
``` 

All nodes adjacent to THSNOD are set. Now search for shortest-distance unreached (but NEARBY) node to process next.

```
   if (LASTNR .eq. 0) then
      SRCHST = FAILED
   else
      determine next node to process
      MINDIS = 1.0E+18
      do 220 PERDEX = 1, LASTNR
         if (DSTSRC (NEARND (PERDEX)) .lt. MINDIS) then
            BSTDEX = PERDEX
            MINDIS = DSTSRC (NEARND (PERDEX))
         end if
      220 continue
      establish new THSNOD
      THSNOD = NEARND (BSTDEX)
      change THSNOD from being NEARBY to reached
      RCHST (THSNOD) = REACHD
      remove THSNOD from NEARBY list
      NEARND (BSTDEX) = NEARND (LASTNR)
      LASTNR = LASTNR - 1
      if (THSNOD .eq. TRGDEX) SRCHST = SUCCES
   end if
201 continue
``` 

Shortest path between PERSONS now established. Next task is to translate path to English description of relationship.

```
   if (SRCHST .eq. FAILED) then
      write (unit=*, fmt=9001) NAME (TRGDEX), NAME (SRCDEX)
   9001 format (a22, ' is not related to ', a20)
   else
      success - parse path to find and display relationship
      call RESOLV (SRCDEX, TRGDEX)
      compute proportion of common genetic material
      call CMPTGN (SRCDEX, TRGDEX)
   end if
end
c procedures under FINDRL

subroutine PROCAD (BASNOD, NXTNOD, NBEDGE, NEARND, LASTNR)
c NXTNOD is adjacent to last-reached node (= BASNOD).
c If NXTNOD already reached, do nothing.
c If previously seen, check whether path thru BASNOD is
c shorter than current path to NXTNOD, and if so re-link
c next to base.
c If not previously seen, link next to base node.

integer NXTNOD, BASNOD, NEARND(*), LASTNR
integer NBEDGE

integer MAXPRS, NAMLEN, IDLEN, BUFLEN,
  MSGLEN, MAXNBR, MAXGVN
parameter (MAXPRS = 300, NAMLEN = 20, IDLEN = 3, BUFLEN = 60,
  MSGLEN = 40, MAXNBR = 20, MAXGVN = 3)

character NULLID*(IDLEN)
parameter (NULLID = '000')

c A node in the graph (= PERSON) has either already been reached,
c is immediately adjacent to those reached, or farther away.

integer REACHD, NEARBY, NOSEEN
parameter (REACHD = 1, NEARBY = 2, NOSEEN = 3)

c These common blocks hold the PERSON array, which is global to
c the entire program.
common /PERNUM/ NBRCNT, NBRDEX, NBREDG, DSTSRC, PATHPR,
  EDGPRD, RCHST, DSCGEN, NUMPER

common /PERCHR/ NAME, IDENT, GENDER, RELID, DSCID

c The following data items constitute the PERSON array, which
c is the central repository of information about inter-relationships.

c static information - filled from PEOPLE file
  character*(NAMLEN) NAME (MAXPRS)
  character*(IDLEN) IDENT (MAXPRS)
  character*1 GENDER (MAXPRS)

c IDENTs of immediate relatives - father, mother, spouse
  character*(IDLEN) RELID (MAXPRS, MAXGVN)

c pointers to immediate neighbors in graph
  integer NBRCNT (MAXPRS)
  integer NBRDEX (MAXPRS, MAXNBR)
  integer NBREDG (MAXPRS, MAXNBR)

  c data used when traversing graph to resolve user request:
    real DSTSRC (MAXPRS)
    integer PATHPR (MAXPRS)
    integer EDGPRD (MAXPRS)
    integer RCHST (MAXPRS)

  c data used to compute common genetic material
    character*(IDLEN) DSCID (MAXPRS)
    real DSCGEN (MAXPRS)
c NUMPER keeps track of the actual number of persons
integer NUMPER

c *** end of declarations for common data ***

real WGHTEG, DSTBAS

begin execution of PROCAD
if (RCHST (NXTNOD) .ne. REACHD) then
  if (NBEDGE .eq. SPOUSE) then
    WGHTEG = 1.8
  else
    WGHTEG = 1.0
  end if
else
  DSTBAS = WGHTEG + DSTSRC (BASNOD)
end if

if (RCHST (NXTNOD) .eq. NOSEEN) then
  change status of THSNOD from not-seen to NEARBY
  RCHST (NXTNOD) = NEARBY
  LASTNR = LASTNR + 1
  NEARND (LASTNR) = NXTNOD
end if

link next to base by re-setting its predecessor index to
point to base, note type of edge, and re-set distance
as it is through base node.
DSTSRC (NXTNOD) = DSTBAS
PATHPR (NXTNOD) = BASNOD
EDGPRD (NXTNOD) = NBEDGE
else
  RCHST is NEARBY
  if (DSTBAS .lt. DSTSRC (NXTNOD)) then
    link next to base by re-setting its predecessor index to
point to base, note type of edge, and re-set distance
as it is through base node.
DSTSRC (NXTNOD) = DSTBAS
PATHPR (NXTNOD) = BASNOD
EDGPRD (NXTNOD) = NBEDGE
  end if
end if
end if
end
subroutine RESOLV (SRCDEX, TRGDEX)

RESOLV condenses the shortest path to a series of relationships for which there are English descriptions.

integer SRCDEX, TRGDEX

c Establish global constants

integer MAXPRS, NAMLEN, IDLEN, BUFLEN,
1 MSGLEN, MAXNBR, MAXGVN
parameter (MAXPRS = 300, NAMLEN = 20, IDLEN = 3, BUFLEN = 60,
1 MSGLEN = 40, MAXNBR = 20, MAXGVN = 3)

c character NULLID*(IDLEN)
parameter (NULLID = '000')

c character MALE*1, FEMALE*1
parameter (MALE = 'M', FEMALE = 'F')

c integer PARENT, CHILD, SPOUSE, SIBLNG,
1 UNCLE, NEPHEW, COUSIN, NULLRL
parameter (PARENT = 1, CHILD = 2, SPOUSE = 3, SIBLNG = 4,
1 UNCLE = 5, NEPHEW = 6, COUSIN = 7, NULLRL = 8)

c sibling proximity can have three values

integer STEP, HALF, FULL
parameter (STEP = 1, HALF = 2, FULL = 3)

c These common blocks hold the PERSON array, which is global to the entire program.

common /PERNUM/ NBRCNT, NBRDEX, NBREDG, DSTSRC, PATHPR,
1 EDGPRD, RCHST, DSCGEN, NUMPER

common /PERCHR/ NAME, IDENT, GENDER, RELID, DSCID
The following data items constitute the PERSON array, which is the central repository of information about inter-relationships.

- **Static information** - filled from PEOPLE file:
  - Character*(NAMLEN) NAME (MAXPRS)
  - Character*(IDLEN) IDENT (MAXPRS)
  - Character*1 GENDER (MAXPRS)

- **Identifiers** of immediate relatives - father, mother, spouse:
  - Character*(IDLEN) RELID (MAXPRS, MAXGVN)

- **Pointers** to immediate neighbors in graph:
  - Integer NBRCNT (MAXPRS)
  - Integer NBRDEX (MAXPRS, MAXNBR)
  - Integer NBREDG (MAXPRS, MAXNBR)

- **Data used when traversing graph to resolve user request**:
  - Real DSTSRC (MAXPRS)
  - Integer PATHPR (MAXPRS)
  - Integer EDGPRD (MAXPRS)
  - Integer RCHST (MAXPRS)

- **Data used to compute common genetic material**:
  - Character*(IDLEN) DSCID (MAXPRS)
  - Real DSCGEN (MAXPRS)

- **NUMPER** keeps track of the actual number of persons:
  - Integer NUMPER

*** end of declarations for common data ***

- These variables are used to generate key-person data:
  - Integer GENCNT, THSCUZ
  - Integer THSPRX

- These variables are used to condense the path:

common /KEYPER/ RELNXT, PERDEX, GENGAP, PRXMTY, CUZRNK

- Key persons are the ones in the relationship path which remain after the path is condensed.

- Integer RELNXT (MAXPRS)
  - Integer PERDEX (MAXPRS)
  - Integer GENGAP (MAXPRS)
  - Integer PRXMTY (MAXPRS)
  - Integer CUZRNK (MAXPRS)
  - Integer KEYREL, LATREL, PRIREL, NXPRI
  - Integer KEYDEX, LATDEX, PRIDEX, THSNOD
  - Integer GAP1, GAP2

- Logical SEEKMR, FULSIB
begin execution of RESOLV

write (unit=*,
   fmt=¨(¨Shortest path between identified persons: ´¨)¨)

Display path and initialize key person arrays from path elements.

THSNOD = TRGDEX

do 100 KEYDEX = 1, MAXPRS
   if (THSNOD .eq. SRCDEX) goto 101
   PERDEX (KEYDEX) = THSNOD
   PRXMTY (KEYDEX) = FULL
   RELNXT (KEYDEX) = EDGPRD (THSNOD)
   if (EDGPRD (THSNOD) .eq. SPOUSE) then
      write (unit=*, fmt=¨(a22, ´¨ is spouse of¨¨)¨) NAME (THSNOD)
      GENGAP (KEYDEX) = 0
   else
      GENGAP (KEYDEX) = 1
      if (EDGPRD (THSNOD) .eq. PARENT) then
         write (unit=*, fmt=¨(a22, ´¨ is parent of¨¨)¨) NAME (THSNOD)
      else
         write (unit=*, fmt=¨(a22, ´¨ is child of¨¨)¨) NAME (THSNOD)
      end if
   end if
   THSNOD = PATHPR (THSNOD)
100 continue

101 continue

write (unit=*, fmt=¨(a22)¨) NAME (THSNOD)
PERDEX (KEYDEX) = THSNOD
RELNXT (KEYDEX) = NULLRL
RELNXT (KEYDEX + 1) = NULLRL
(resolve CHILD-PARENT and CHILD-SPOUSE-PARENT relations to SIBLING relations.

do 200 KEYDEX = 1, MAXPRS
  if (RELNXT (KEYDEX) .eq. NULLRL) goto 201
  if (RELNXT (KEYDEX) .eq. CHILD) then
    LATREL = RELNXT (KEYDEX + 1)
    if (LATREL .eq. PARENT) then
      LATREL = RELNXT (KEYDEX + 1)
    else
      PRXMTY (KEYDEX) = FULL
    end if
  end if
  LENX = RELNXT (KEYDEX)
  if (LENX .ne. RELNXT (KEYDEX)) then
    if (FULSIB (PERDEX (KEYDEX), PERDEX (KEYDEX + 2)) then
      PRXMTY (KEYDEX) = FULL
    else
      PRXMTY (KEYDEX) = HALF
    end if
  end if
  GENLEN = LENX - KEYDEX
  if (GENLEN .gt. 1) then
    GENLEN = GENLEN - 1
  else
    GENLEN = 0
  end if
  RELNXT (KEYDEX) = SIBLING
  call CONDNS (KEYDEX, 1)
  else if (LATREL .eq. SPOUSE .and. RELNXT (KEYDEX + 1) .eq. PARENT) then
    found step-SIBLING
    GEN = GEN + 1
    GENLEN = GENLEN - 1
    RELNXT (KEYDEX) = SIBLING
    call CONDNS (KEYDEX, 2)
  end if
end if

200 continue
201 continue

(resolve CHILD-CHILD-... and PARENT-PARENT-... relations to direct descendant or ancestor relations.

do 300 KEYDEX = 1, MAXPRS
  if (RELNXT (KEYDEX) .eq. NULLRL) goto 301
  if (RELNXT (KEYDEX) .eq. CHILD .or.
  1 RELNXT (KEYDEX) .eq. PARENT) then
    do 310 LATDEX = KEYDEX + 1, MAXPRS
      if (RELNXT (LATDEX) .ne. RELNXT (KEYDEX)) goto 311
    310 continue
    311 continue
    GENCN = LATDEX - KEYDEX
    if (GENCN .gt. 1) then
      compress generations
      GEN = GENCN
      call CONDNS (KEYDEX, GENCN - 1)
    end if
  end if
end if
300 continue
301 continue
c resolve CHILD-SIBLING-PARENT to COUSIN,
c CHILD-SIBLING to NEPHEW,
c SIBLING-PARENT to UNCLE.
do 400 KEYDEX = 1, MAXPRS
   if (RELNXT (KEYDEX) .eq. NULLRL) goto 401
   LATREL = RELNXT (KEYDEX + 1)
   if (RELNXT (KEYDEX) .eq. CHILD .and. LATREL .eq. SIBLING) then
      found COUSIN or NEPHEW
      PRXMTY (KEYDEX) = PRXMTY (KEYDEX + 1)
   if (RELNXT (KEYDEX + 2) .eq. PARENT) then
      found COUSIN
      GAP1 = GENGAP (KEYDEX)
      GAP2 = GENOP (KEYDEX + 2)
      GENOP (KEYDEX) = abs (GAP1 - GAP2)
      CUZRNK (KEYDEX) = min (GAP1, GAP2)
      RELNXT (KEYDEX) = COUSIN
      call CONDNS (KEYDEX, 2)
   else
      found NEPHEW
      RELNXT (KEYDEX) = NEPHEW
      call CONDNS (KEYDEX, 1)
   end if
else
   if (RELNXT (KEYDEX) .eq. SIBLING .and.
      LATREL .eq. PARENT) then
      found UNCLE
      GENOP (KEYDEX) = GENOP (KEYDEX + 1)
      RELNXT (KEYDEX) = UNCLE
      call CONDNS (KEYDEX, 1)
   end if
end if
400 continue
401 continue
Loop below will pick out valid adjacent strings of elements to be displayed. KEYDEX points to first element, LATDEX to last element, and PRIDEX to the element which determines the primary English word to be used. Associativity of adjacent elements in condensed table is based on English usage.

```fortran
KEYDEX = 1
write (unit=*, fmt='(a, c1)'' Condensed path:'')
500 continue
 if (RELNXT (KEYDEX) .eq. NULLRL) goto 501
 KEYREL = RELNXT (KEYDEX)
 LATDEX = KEYDEX
 PRIDEX = KEYDEX
 if (RELNXT (KEYDEX + 1) .ne. NULLRL) then
  seek multi-element combination
 SEEKMR = .true.
  if (KEYREL .eq. SPOUSE) then
   LATDEX = LATDEX + 1
   PRIDEX = LATDEX
  end if
  Nothing can follow SPOUSE-SIBLING or SPOUSE-COUSIN
 SEEKMR = .not. (RELNXT (LATDEX) .eq. SIBLING .or.
   RELNXT (LATDEX) .eq. COUSIN)
1 end if
 end if
 PRIDEX is now correctly set. Next if-statement determines if a following SPOUSE relation should be appended to this combination or left for the next combination.
 if (SEEKMR .and. RELNXT (PRIDEX + 1) .eq. SPOUSE) then
  Only a SPOUSE can follow a Primary.
  Check primary preceding and following SPOUSE.
 PRIREL = RELNXT (PRIDEX)
 NXTPRI = RELNXT (PRIDEX + 2)
 if ((NXTPRI .eq. NEPHEW .or.
  NXTPRI .eq. COUSIN .or.
  NXTPRI .eq. NULLRL)
  .or. (PRIREL .eq. NEPHEW)
  .or. ((PRIREL .eq. SIBLING .or. PRIREL .eq. PARENT)
  .and. NXTPRI .ne. UNCLE )) then
 append following SPOUSE with this combination.
 LATDEX = LATDEX + 1
 end if
 end if
 end multi-element combination
 call SHOWRE (KEYDEX, LATDEX, PRIDEX)
 KEYDEX = LATDEX + 1
 goto 500
501 continue
 write (unit=*, fmt='(a22)') NAME (PERDEX (KEYDEX))
end
 end of RESOLV
logical function FULSIB (INDEX1, INDEX2)
c  Determines whether two PERSONs are full siblings, i.e.,
c  have the same two parents.

integer INDEX1, INDEX2

integer    MAXPRS, NAMLEN, IDLEN, BUFLEN,
            MSGLEN, MAXNBR, MAXGVN
parameter (MAXPRS = 300, NAMLEN = 20, IDLEN = 3, BUFLEN = 60,
            MSGLEN = 40, MAXNBR = 20, MAXGVN = 3)

character   NULLID*(IDLEN)
parameter (NULLID = '000')

integer    FATHID, MOTHID, SPOUID
parameter (FATHID = 1, MOTHID = 2, SPOUID = 3)

c  These common blocks hold the PERSON array, which is global to
   the entire program.
common /PERNUM/ NBRCT, NBRDEX, NBREDG, DSTSRC, PATHPR,
                EDGPRD, RCHST, DSCGEN, NUMPER

common /PERCHR/ NAME, IDENT, GENDER, RELID, DSCID

c  The following data items constitute the PERSON array, which
   is the central repository of information about inter-relationships.

c static information - filled from PEOPLE file
character*(NAMLEN) NAME (MAXPRS)
character*(IDLEN) IDENT (MAXPRS)
character*1 GENDER (MAXPRS)
c RELID of immediate relatives - father, mother, spouse
character*(IDLEN) RELID (MAXPRS, MAXGVN)
c pointers to immediate neighbors in graph
integer NBRCT (MAXPRS)
integer NBRDEX (MAXPRS, MAXNBR)
integer NBREDG (MAXPRS, MAXNBR)
c data used when traversing graph to resolve user request:
real DSTSRC (MAXPRS)
integer PATHPR (MAXPRS)
integer EDGPRD (MAXPRS)
integer RCHST (MAXPRS)
c data used to compute common genetic material
character*(IDLEN) DSCID (MAXPRS)
real DSCGEN (MAXPRS)
c NUMPER keeps track of the actual number of persons
integer NUMPER

c *** end of declarations for common data ***

FULSIB =
  1 RELID (INDEX1, FATHID) .ne. NULLID .and. 
  2 RELID (INDEX1, MOTHID) .ne. NULLID .and. 
  3 RELID (INDEX1, FATHID) .eq. RELID (INDEX2, FATHID) .and. 
  4 RELID (INDEX1, MOTHID) .eq. RELID (INDEX2, MOTHID) 
end
subroutine CONDNS (ATDEX, GAPSIZ)
c CONDNS condenses superfluous entries from the
c key person arrays, starting at ATDEX.

text 1

integer MAXPRS, NAMLEN, IDLEN, BUFLEN,
1 MSGLEN, MAXNBR, MAXGVN
parameter (MAXPRS = 300, NAMLEN = 20, IDLEN = 3, BUFLEN = 60,
1 MSGLEN = 40, MAXNBR = 20, MAXGVN = 3)

text 1

character NULLID*(IDLEN)
parameter (NULLID = '000')

text 1

integer PARENT, CHILD, SPOUSE, SIBLNG,
1 UNCLE, NEPHEW, COUSIN, NULLRL
parameter (PARENT = 1, CHILD = 2, SPOUSE = 3, SIBLNG = 4,
1 UNCLE = 5, NEPHEW = 6, COUSIN = 7, NULLRL = 8)

text 1

common /KEYPER/ RELNXT, PERDEX, GENGAP, PRXMTY, CUZRNK
c Key persons are the ones in the relationship path which remain
c after the path is condensed.

text 1

integer RELNXT (MAXPRS)
integer PERDEX (MAXPRS)
integer GENGAP (MAXPRS)
integer PRXMTY (MAXPRS)
integer CUZRNK (MAXPRS)

integer ATDEX, GAPSIZ, SENDEX, RCVDEX

RCVDEX = ATDEX
100 continue

RCVDEX = RCVDEX + 1
SENDEX = RCVDEX + GAPSIZ
RELNXT (RCVDEX) = RELNXT (SENDEX)
PERDEX (RCVDEX) = PERDEX (SENDEX)
GENGAP (RCVDEX) = GENGAP (SENDEX)
PRXMTY (RCVDEX) = PRXMTY (SENDEX)
CUZRNK (RCVDEX) = CUZRNK (SENDEX)
if (RELNXT (SENDEX) .ne. NULLRL) goto 100

end
c procedures under RESOLV

subroutine SHOWRE (FSTDEX, LSTDEX, PRIDEX)
c SHOWRE takes 1, 2, or 3 adjacent elements in the
c condensed table and generates the English description of
c the relation between the first and last + 1 elements.

c Establish global constants

integer MAXPRS, NAMLEN, IDLEN, BUFLEN,
   MSGLEN, MAXNBR, MAXGVN
parameter (MAXPRS = 300, NAMLEN = 20, IDLEN = 3, BUFLEN = 60,
   MSGLEN = 40, MAXNBR = 20, MAXGVN = 3)
character NULLID*(IDLEN)
parameter (NULLID = '000')
character MALE*1, FEMALE*1
parameter (MALE = 'M', FEMALE = 'F')
integer PARENT, CHILD, SPOUSE, SIBLNG,
   UNCLE, NEPHEW, COUSIN, NULLRL
parameter (PARENT = 1, CHILD = 2, SPOUSE = 3, SIBLNG = 4,
   UNCLE = 5, NEPHEW = 6, COUSIN = 7, NULLRL = 8)

c sibling proximity can have three values

integer STEP, HALF, FULL
parameter (STEP = 1, HALF = 2, FULL = 3)

c These common blocks hold the PERSON array, which is global to
the entire program.
common /PERNUM/ NBRCNT, NBRDEx, NBRDEx, DSTSRC, PATHPR,
   EDGPRD, RCHST, DSCGEN, NUMPER
common /PERCHR/ NAME, IDENT, GENDER, RELID, DSCID
c The following data items constitute the PERSON array, which
c is the central repository of information about inter-relationships.

c static information – filled from PEOPLE file
  character*(NAMLEN)  NAME  (MAXPRS)
  character*(IDLEN)    IDENT  (MAXPRS)
  character*1        GENDER  (MAXPRS)

c IDENTs of immediate relatives – father, mother, spouse
  character*(IDLEN)    RELID  (MAXPRS, MAXGVN)

c pointers to immediate neighbors in graph
  integer               NBRCNT  (MAXPRS)
  integer               NBRDEX  (MAXPRS, MAXNBR)
  integer               NBREDG  (MAXPRS, MAXNBR)

c data used when traversing graph to resolve user request:
  real                  DSTSRC  (MAXPRS)
  integer               PATHPR  (MAXPRS)
  integer               EDGPRD  (MAXPRS)
  integer               RCHST   (MAXPRS)

c data used to compute common genetic material
  character*(IDLEN)    DSCID   (MAXPRS)
  real                  DSCGEN  (MAXPRS)

c NUMPER keeps track of the actual number of persons

  integer               NUMPER

  common /KEYPER/ RELNXT, PERDEX, GENGAP, PRXMTY, CUZRNK

  c Key persons are the ones in the relationship path which remain
  c after the path is condensed.

  integer               RELNXT  (MAXPRS)
  integer               PERDEX  (MAXPRS)
  integer               GENGAP  (MAXPRS)
  integer               PRXMTY  (MAXPRS)
  integer               CUZRNK  (MAXPRS)

  c *** end of declarations for common data ***

  logical               INLAW
  integer               THSPRX, THSGAP, THSCUZ
  character              TWODIG*2
  integer               SUFPTR
  character              SUFCHR*12
  integer               FSTDSEX, LSTDSEX, PRIDEX
  integer               FSTREL, LSTREL, PRIREL
  character*75           OUTBUF
  integer               OUTPTR
c begin execution of SHOWRE

    FSTREL = RELNXT (FSTDEX)
    LSTREL = RELNXT (LSTDEX)
    PRIREL = RELNXT (PRIDEX)

c    set THSPRX
    if ((PRIREL .eq. PARENT .and. FSTREL .eq. SPOUSE) .or. 
     1 (PRIREL .eq. CHILD .and. LSTREL .eq. SPOUSE)) then
        THSPRX = STEP
    else
        if (PRIREL .eq. SIBLNG .or. PRIREL .eq. UNCLE .or. 
         1 PRIREL .eq. NEPHEW .or. PRIREL .eq. COUSIN) then
            THSPRX = PRXMTY (PRIDEX)
        else
            THSPRX = FULL
        end if
    end if

c    set THSGAP
    if (PRIREL .eq. PARENT .or. PRIREL .eq. CHILD .or. 
     1 PRIREL .eq. UNCLE .or. PRIREL .eq. NEPHEW .or. 
     2 PRIREL .eq. COUSIN) then
        THSGAP = GENGAP (PRIDEX)
    else
        THSGAP = 0
    end if

c    set INLAW
    if (FSTREL .eq. SPOUSE .and. 
     1 (PRIREL .eq. SIBLNG .or. PRIREL .eq. CHILD .or. 
     2 PRIREL .eq. NEPHEW .or. PRIREL .eq. COUSIN)) then
        INLAW = .true.
    else
        if (LSTREL .eq. SPOUSE .and. 
         1 (PRIREL .eq. SIBLNG .or. PRIREL .eq. PARENT .or. 
         2 PRIREL .eq. UNCLE .or. PRIREL .eq. COUSIN)) then
            INLAW = .true.
        else
            INLAW = .false.
        end if
    end if

c    set THSCUZ
    if (PRIREL .eq. COUSIN) then
        THSCUZ = CUZRNK (PRIDEX)
    else
        THSCUZ = 0
    end if
c parameters are set - now generate display.

OUTBUF = NAME (PERDEX (FSTDEX)) // ' is '
OUTPTR = NAMLEN + 5
if (PRIREL .eq. PARENT .or. PRIREL .eq. CHILD .or.
  PRIREL .eq. UNCLE .or. PRIREL .eq. NEPHEW) then
c display generation-qualifier
  if (THSGAP .ge. 3) then
    call APPEND (OUTBUF, OUTPTR, 'great')
    if (THSGAP .gt. 3) then
      write (unit=TWODIG, fmt='(i2)') THSGAP - 2
      call APPEND (OUTBUF, OUTPTR, '*' // TWODIG)
    end if
    call APPEND (OUTBUF, OUTPTR, '-')
  end if
  if (THSGAP .ge. 2) then
    call APPEND (OUTBUF, OUTPTR, 'grand-')
  end if
else
  if (PRIREL .eq. CUSIN .and. THSCUZ .gt. 1) then
c display cousin-degree
    write (unit=TWODIG, fmt='(i2)') THSCUZ
    call APPEND (OUTBUF, OUTPTR, TWODIG)
    SUFPTR = mod (THSCUZ, 10)
    if (SUFPTR .gt. 3) SUFPTR = 0
    SUF PTR = 3 * SUF PTR + 1
    SUFCHR = 'th st nd rd '
    call APPEND (OUTBUF, OUTPTR, SUFCHR (SUFPTR : SUF PTR + 2))
  end if
end if

if (THSPRX .eq. STEP) then
  call APPEND (OUTBUF, OUTPTR, 'step-')
else
  if (THSPRX .eq. HALF) then
    call APPEND (OUTBUF, OUTPTR, 'half-')
  end if
end if
if (GENDER (PERDEX (FSTDEX)) .eq. MALE) then
  goto (201,202,203,204,205,206,297,298), PRIREL
  continue
    call APPEND (OUTBUF, OUTPTR, 'father')
    goto 300
  continue
    call APPEND (OUTBUF, OUTPTR, 'son')
    goto 300
  continue
    call APPEND (OUTBUF, OUTPTR, 'husband')
    goto 300
  continue
    call APPEND (OUTBUF, OUTPTR, 'brother')
    goto 300
  continue
    call APPEND (OUTBUF, OUTPTR, 'uncle')
    goto 300
  continue
    call APPEND (OUTBUF, OUTPTR, 'nephew')
    goto 300
else
  gender is FEMALE
  goto (251,252,253,254,255,256,297,298), PRIREL
  continue
    call APPEND (OUTBUF, OUTPTR, 'mother')
    goto 300
  continue
    call APPEND (OUTBUF, OUTPTR, 'daughter')
    goto 300
  continue
    call APPEND (OUTBUF, OUTPTR, 'wife')
    goto 300
  continue
    call APPEND (OUTBUF, OUTPTR, 'sister')
    goto 300
  continue
    call APPEND (OUTBUF, OUTPTR, 'aunt')
    goto 300
  continue
    call APPEND (OUTBUF, OUTPTR, 'niece')
    goto 300
end if
  goto 297
  continue
    call APPEND (OUTBUF, OUTPTR, 'cousin')
    goto 300
  continue
    call APPEND (OUTBUF, OUTPTR, 'null')
    goto 300
  continue
if (INLAW) call APPEND (OUTBUF, OUTPTR, 'in-law')

if (PRIREL .eq. COUSIN .and. THSGAP .gt. 0) then
  if (THSGAP .gt. 1) then
    write (unit=TWODIG, fmt='(i2)') THSGAP
    call APPEND (OUTBUF, OUTPTR, '/TWODIG/ times removed')
  else
    call APPEND (OUTBUF, OUTPTR, ' once removed')
  end if
end if

call APPEND (OUTBUF, OUTPTR, ' of')
write (unit=*, fmt='(a77)') OUTBUF
end

subroutine APPEND (STRING, PTR, ADDEND)
c APPEND appends the contents of ADDEND to STRING in the position
  indicated by PTR, and increments PTR

character STRING*(*), ADDEND*(*)
integer PTR, ADDLEN

ADDLEN = len (ADDEND)
STRING (PTR : PTR + ADDLEN - 1) = ADDEND
PTR = PTR + ADDLEN
end

subroutine CMPTGN (INDEX1, INDEX2)
c CMPTGN assumes that each ancestor contributes
  half of the genetic material to a PERSON. It finds common
  ancestors between two PERSONs and computes the expected
  value of the proportion of common material.

integer INDEX1, INDEX2

integer MAXPRS, NAMLEN, IDLEN, BUFLEN,
  MSGLEN, MAXNBR, MAXGVN
parameter (MAXPRS = 300, NAMLEN = 20, IDLEN = 3, BUFLEN = 60,
  MSGLEN = 40, MAXNBR = 20, MAXGVN = 3)

character NULLID*(IDLEN)
parameter (NULLID = '000')

c These common blocks hold the PERSON array, which is global to
c the entire program.

common /PERNUM/ NBRCNT, NBRDAX, NBREDG, DSTSRC, PATHPR,
  EDGPRD, RCHST, DSCGEN, NUMPER

common /PERCHR/ NAME, IDENT, GENDER, RELID, DSCID
The following data items constitute the PERSON array, which is the central repository of information about inter-relationships.

c static information - filled from PEOPLE file
  character*(NAMLEN)  NAME  (MAXPRS)
  character*(IDLEN)   IDENT  (MAXPRS)
  character*         GENDER (MAXPRS)

c IDENTs of immediate relatives - father, mother, spouse
  character*(IDLEN)  RELID  (MAXPRS, MAXCVN)

c pointers to immediate neighbors in graph
  integer           NBRCNT  (MAXPRS)
  integer           NBRDEX  (MAXPRS, MAXNBR)
  integer           NBREDG  (MAXPRS, MAXNBR)

c data used when traversing graph to resolve user request:
  real              DSTSRC  (MAXPRS)
  integer           PATHPR  (MAXPRS)
  integer           EDGPRD  (MAXPRS)
  integer           RCHST   (MAXPRS)

c data used to compute common genetic material
  character*(IDLEN)  DSCID   (MAXPRS)
  real              DSCGEN  (MAXPRS)

c NUMPER keeps track of the actual number of persons
  integer           NUMPER

c STACK is common to the routines which calculate genetic overlap.

c It is used to implement recursive traversal of the ancestor trees.

  integer   STKSIZ
  parameter (STKSIZ = 50)

  common /STACK/ PROPTN, CONTRB, COUNTD, PERDEX, NXTNBR,
               STKPTR

  real     PROPTN   (STKSIZ)
  real     CONTRB   (STKSIZ)
  real     COUNTD   (STKSIZ)
  integer  PERDEX   (STKSIZ)
  integer  NXTNBR   (STKSIZ)
  integer  STKPTR

c *** end of declarations for common data ***

  real     COMPRP
First zero out all ancestors to allow adding. This is necessary because there might be two paths to an ancestor.

```
STKPTR = 1
PERDEX (STKPTR) = INDEX1
NXTNBR (STKPTR) = 0
```

```
100 continue
    call ZERPRO
    if (STKPTR .ge. 1) goto 100
```

```
101 continue
```

```
c now mark with shared PROPTN
STKPTR = 1
PERDEX (STKPTR) = INDEX1
NXTNBR (STKPTR) = 0
PROPTN (STKPTR) = 1.0
```

```
200 continue
    call MRKPRO (IDENT (INDEX1))
    if (STKPTR .ge. 1) goto 200
```

```
201 continue
```

```
c traverse ancestor tree for INDEX2. summing overlap with marked tree of INDEX1
COMPRP = 0.0
STKPTR = 1
PERDEX (STKPTR) = INDEX2
NXTNBR (STKPTR) = 0
PROPTN (STKPTR) = 1.0
COUNTD (STKPTR) = 0.0
```

```
300 continue
    call CHKCOM (COMPRP, IDENT (INDEX1))
    if (STKPTR .ge. 1) goto 300
```

```
301 continue
```

```
write (unit=*, fmt=9001) COMPRP
```

```
9001 format(´Proportion of common genetic material = `.p, e12.5e2)´ end
```

subroutine ZERPRO

```
c ZERPRO recursively seeks out all ancestors and zeros them out.
```

```
integer MAXPRS, NAMLEN, IDLEN, BUFLEN,
1 MSGLEN, MAXNBR, MAXGVN
parameter (MAXPRS = 300, NAMLEN = 20, IDLEN = 3, BUFLEN = 60,
1 MSGLEN = 40, MAXNBR = 20, MAXGVN = 3)
```

```
character NULLID*(IDLEN)
parameter (NULLID = ´000´)
```

```
integer PARENT, CHILD, SPOUSE, SIBLNG,
1 UNCLE, NEPHEW, COUSIN, NULLRL
parameter (PARENT = 1, CHILD = 2, SPOUSE = 3, SIBLNG = 4,
1 UNCLE = 5, NEPHEW = 6, COUSIN = 7, NULLRL = 8)
```
These common blocks hold the PERSON array, which is global to the entire program.

common /PERNUM/ NBRCNT, NBRDEX, NBREDG, DSTSRC, PATHPR, EDGPRD, RCHST, DSCGEN, NUMPER

common /PERCHR/ NAME, IDENT, GENDER, RELID, DSCID

The following data items constitute the PERSON array, which is the central repository of information about inter-relationships.

- static information - filled from PEOPLE file
  - character*(NAMLEN) NAME (MAXPRS)
  - character*(IDLEN) IDENT (MAXPRS)
  - character*1 GENDER (MAXPRS)

- IDENTs of immediate relatives - father, mother, spouse
  - character*(IDLEN) RELID (MAXPRS, MAXGVN)

- pointers to immediate neighbors in graph
  - integer NBRCNT (MAXPRS)
  - integer NBRDEX (MAXPRS, MAXNBR)
  - integer NBREDG (MAXPRS, MAXNBR)

- data used when traversing graph to resolve user request:
  - real DSTSRC (MAXPRS)
  - integer PATHPR (MAXPRS)
  - integer EDGPRD (MAXPRS)
  - integer RCHST (MAXPRS)

- data used to compute common genetic material
  - character*(IDLEN) DSCID (MAXPRS)
  - real DSCGEN (MAXPRS)

- NUMPER keeps track of the actual number of persons
  - integer NUMPER

- STACK is common to the routines which calculate genetic overlap.
- It is used to implement recursive traversal of the ancestor trees.
  - integer STKSIZ
  - parameter (STKSIZ = 50)

common /STACK/ PROPTN, CONTRB, COUNTD, PERDEX, NXTNBR, STKPTR

- real PROPTN (STKSIZ)
- real CONTRB (STKSIZ)
- real COUNTD (STKSIZ)
- integer PERDEX (STKSIZ)
- integer NXTNBR (STKSIZ)
- integer STKPTR

*** end of declarations for common data ***
integer ZERDEX, THSNBR
ZERDEX = PERDEX (STKPTR)
if (NXTNBR (STKPTR) .eq. 0) then
   DSCGEN (ZERDEX) = 0.0
   NXTNBR (STKPTR) = 1
end if
do 100 THSNBR = NXTNBR (STKPTR), NBRCNT (ZERDEX)
   if (NBREDG (ZERDEX, THSNBR) .eq. PARENT) goto 101
100 continue
101 continue
if (THSNBR .gt. NBRCNT (ZERDEX)) then
   c no more ancestors from this person
   STKPTR = STKPTR - 1
else
   c set up for next ancestor
   NXTNBR (STKPTR) = THSNBR + 1
   STKPTR = STKPTR + 1
   PERDEX (STKPTR) = NBRDEX (ZERDEX, THSNBR)
   NXTNBR (STKPTR) = 0
end if
end

subroutine MRKPRO (MARKER)
c MRKPRO recursively seeks out all ancestors and
marks them with the sender's proportion of shared
c genetic material. This proportion is diluted by one-half
c for each generation.

integer MAXPRS, NAMLEN, IDLEN, BUFLEN,
1 MSGLEN, MAXNBR, MAXGVN
parameter (MAXPRS = 300, NAMLEN = 20, IDLEN = 3, BUFLEN = 60,
1 MSGLEN = 40, MAXNBR = 20, MAXGVN = 3)

c These common blocks hold the PERSON array, which is global to
c the entire program.
common /PERNUM/ NBRCNT, NBRDEX, NBREDG, DSTSRC, PATHPR,
1 EDGPRD, RCHST, DSCGEN, NUMPER
common /PERCHR/ NAME, IDENT, GENDER, RELID, DSCID
The following data items constitute the PERSON array, which is the central repository of information about inter-relationships.

c static information - filled from PEOPLE file
    character*(NAMLEN) NAME (MAXPRS)
    character*(IDLEN) IDENT (MAXPRS)
    character*1 GENDER (MAXPRS)

c IDENTs of immediate relatives - father, mother, spouse
    character*(IDLEN) RELID (MAXPRS, MAXGVN)

c pointers to immediate neighbors in graph
    integer NBRCNT (MAXPRS)
    integer NBRDEX (MAXPRS, MAXNBR)
    integer NBREDG (MAXPRS, MAXNBR)

c data used when traversing graph to resolve user request:
    real DSTSRC (MAXPRS)
    integer PATHPR (MAXPRS)
    integer EDGPRD (MAXPRS)
    integer RCHST (MAXPRS)

c data used to compute common genetic material
    character*(IDLEN) DSCID (MAXPRS)
    real DSCGEN (MAXPRS)

c NUMPER keeps track of the actual number of persons
    integer NUMPER

c STACK is common to the routines which calculate genetic overlap.
    c It is used to implement recursive traversal of the ancestor trees.

    integer STKSIZ
    parameter (STKSIZ = 50)

    common /STACK/ PROPTN, CONTRB, COUNTD, PERDEX, NXTNBR, 1  
        STKPTR

    real PROPTN (STKSIZ)
    real CONTRB (STKSIZ)
    real COUNTD (STKSIZ)
    integer PERDEX (STKSIZ)
    integer NXTNBR (STKSIZ)
    integer STKPTR

c *** end of declarations for common data ***

    character*(IDLEN) MARKER
    integer MRKDEX, THSNBR
MRKDEX = PERDEX (STKPTR)
if (NXTNBR (STKPTR).eq. 0) then
  DSCID (MRKDEX) = MARKER
  DSCGEN (MRKDEX) = DSCGEN (MRKDEX) + PROPTN (STKPTR)
  NXTNBR (STKPTR) = 1
end if
do 100 THTSNBR = NXTNBR (STKPTR), NBRCNT (MRKDEX)
  if (NBREDG (MRKDEX, THTSNBR).eq. PARENT) goto 101
100 continue
101 continue
if (THTSNBR.gt. NBRCNT (MRKDEX)) then
  c  no more ancestors from this person
  STKPTR = STKPTR - 1
else
  c  set up for next ancestor
  NXTNBR (STKPTR) = THTSNBR + 1
  STKPTR = STKPTR + 1
  PERDEX (STKPTR) = NBRDEX (MRKDEX, THTSNBR)
  NXTNBR (STKPTR) = 0
  PROPTN (STKPTR) = PROPTN (STKPTR - 1) / 2.0
end if
end

subroutine CHKCOM (COMPRP, MTCHID)
c  CHKCOM searches all the ancestors of CHKDEX to see if any have
  been marked, and if so adds the appropriate amount to COMPRP.
integer MAXPRS, NAMLEN, IDLEN, BUFLEN,
  MSGLEN, MAXNBR, MAXGVN
parameter (MAXPRS = 300, NAMLEN = 20, IDLEN = 3, BUFLEN = 60,
  MSGLEN = 40, MAXNBR = 20, MAXGVN = 3)
character NULLID*(IDLEN)
parameter (NULLID = '000')
integer PARENT, CHILD, SPOUSE, SIBLNG,
  UNCLE, NEPHEW, COUSIN, NULLRL
parameter (PARENT = 1, CHILD = 2, SPOUSE = 3, SIBLNG = 4,
  UNCLE = 5, NEPHEW = 6, COUSIN = 7, NULLRL = 8)
c These common blocks hold the PERSON array, which is global to
  the entire program.
common /PERNUM/ NBRCNT, NBRDEX, NBREDG, DSTSRC, PATHPR,
  EDGPRD, RCHST, DSCGEN, NUMPER
common /PERCHR/ NAME, IDENT, GENDER, RELID, DSCID
The following data items constitute the PERSON array, which is the central repository of information about inter-relationships.

**Static information** - filled from PEOPLE file
- character*(NAMLEN): NAME (MAXPRS)
- character*(IDLEN): IDENT (MAXPRS)
- character*: GENDER (MAXPRS)

**Identifiers of immediate relatives** - father, mother, spouse
- character*(IDLEN): RELID (MAXPRS, MAXGVN)

**Pointers to immediate neighbors in graph**
- integer: NBRCT (MAXPRS)
- integer: NBRDXX (MAXPRS, MAXNBR)
- integer: NBREDG (MAXPRS, MAXNBR)

**Data used when traversing graph to resolve user request:**
- real: DSTSRC (MAXPRS)
- integer: PATHPR (MAXPRS)
- integer: EDGPRD (MAXPRS)
- integer: RCHST (MAXPRS)

**Data used to compute common genetic material**
- character*(IDLEN): DSCID (MAXPRS)
- real: DSCGEN (MAXPRS)

**NUMPER** keeps track of the actual number of persons
- integer: NUMPER

**STACK** is common to the routines which calculate genetic overlap.
- It is used to implement recursive traversal of the ancestor trees.

**Parameters**
- integer: STKSIZ
- parameter (STKSIZ = 50)

**Common Blocks**
- real: PROPTN (STKSIZ)
- real: CONTRB (STKSIZ)
- real: COUNTD (STKSIZ)
- integer: PERDEX (STKSIZ)
- integer: NXTNBR (STKSIZ)
- integer: STKPTR

*** End of declarations for common data ***

- real: COMPRP
- character*(IDLEN): MTCHID
- integer: CHKDEX
CHKDEX = PERDEX (STKPTR)

if (NXTNBR (STKPTR).eq. 0) then
    NXTNBR (STKPTR) = 1
    if (DSCID (CHKDEX).eq. MTCHID) then
        c
        Increment COMPRP by the contribution of this
        common ancestor, but discount for the contribution
        of less remote ancestors already counted.
        CONTRB (STKPTR) = DSCGEN (CHKDEX) * PROPTN (STKPTR)
        COMPRP = COMPRP + CONTRB (STKPTR) - COUNTD (STKPTR)
    else
        CONTRB (STKPTR) = 0.0
    end if
end if

do 100 THSNBR = NXTNBR (STKPTR), NBRCNT (CHKDEX)
    if (NBREDG (CHKDEX, THSNBR).eq. PARENT) goto 101
100 continue

101 continue

if (THSNBR.gt. NBRCNT (CHKDEX)) then
    c
    no more ancestors from this person
    STKPTR = STKPTR - 1
else
    c
    set up for next ancestor
    NXTNBR (STKPTR) = THSNBR + 1
    STKPTR = STKPTR + 1
    PERDEX (STKPTR) = NBRDEX (CHKDEX, THSNBR)
    NXTNBR (STKPTR) = 0
    PROPTN (STKPTR) = PROPTN (STKPTR - 1) / 2.0
    COUNTD (STKPTR) = CONTRB (STKPTR - 1) / 4.0
end if

end
7.0 PASCAL

User-defined identifiers are written in mixed upper and lower case, rather than all upper-case, because Pascal provides no separator character, such as "-" or "_" for identifiers. Therefore, upper-case letters are used for readability, e.g., `EdgeToPredecessor` is used in Pascal where `EDGE_TO_PREDECESSOR` is used in most of the other languages.

```pascal
program Relate (input, output, People);

const
  MaxPersons = 300;
  NameLength = 20;
  { every Person has a unique 3-digit Identifier }
  IdentifierLength = 3;
  BufferLength = 60;
  RequestOk = 'Request OK'
  RequestToStop = 'stop'

type
  IdentifierRange = 1..IdentifierLength;
  BufferRange = 1..BufferLength;
  NameRange = 1..NameLength;
  DigitType = '0'..'9';
  NameType = packed array [NameRange] of char;
  BufferType = packed array [BufferRange] of char;
  MessageType = packed array [1..40] of char;
  IdentifierType = array [IdentifierRange] of DigitType;
  { each Person's record in the file identifies at most three others directly related: father, mother, and spouse }
  GivenIdentifiers = (FatherIdent, MotherIdent, SpouseIdent);
  RelativeArray = array [GivenIdentifiers] of IdentifierType;
  Counter = 0..maxint;

{ this is the format of records in the file to be read in }
FilePersonRecord = record
  Name : NameType;
  Identifier : IdentifierType;
  { 'M' for Male and 'F' for Female }
  Gender : char;
  RelativeIdentifier : RelativeArray
end;
```
IndexType = 0..MaxPersons;
GenderType = (Male, Female);
RelationType = (Parent, Child, Spouse, Sibling, Uncle, Nephew, Cousin, NullRelation);

{ directed edges in the graph are of a given type }
EdgeType = Parent..Spouse;

{ A node in the graph (= Person) has either already been reached, is immediately adjacent to those reached, or farther away. }
ReachedType = (Reached, Nearby, NotSeen);

{ each Person has a linked list of adjacent nodes, called neighbors }
NeighborPointer = 'NeighborRecord;

NeighborRecord = record
  NeighborIndex : IndexType;
  NeighborEdge : EdgeType;
  NextNeighbor : NeighborPointer
end;

{ All Relationships are captured in the directed graph of which each record is a node. }
PersonRecord = record
  { static information - filled from People file: }
  Name : NameType;
  Identifier : IdentifierType;
  Gender : GenderType;
  { Identifiers of immediate relatives - father, mother, spouse }
  RelativeIdentifier : RelativeArray;
  { head of linked list of adjacent nodes }
  NeighborListHeader : NeighborPointer;
  { data used when traversing graph to resolve user request: }
  DistanceFromSource : real;
  PathPredecessor : IndexType;
  EdgeToPredecessor : EdgeType;
  ReachedStatus : ReachedType;
  { data used to compute common genetic material }
  DescendantIdentifier : IdentifierType;
  DescendantGenes : real
end;

var
  { The Person array is the central repository of information about inter-relationships. }
  Person : array [IndexType] of PersonRecord;

  { These variables are used when establishing the Person array from the People file. }
  People : file of FilePersonRecord;
  Current, Previous, NumberOfPersons : IndexType;
  IdentifierIndex : IdentifierRange;
  PreviousIdent, CurrentIdent, NullIdent : IdentifierType;
  Relationship : GivenIdentifiers;
  RelationLoopDone : boolean;
{ These variables are used to accept and resolve requests for Relationship information. }
BufferIndex, SemicolonLocation
  : BufferRange;
RequestBuffer   : BufferType;
Person1Ident, Person2Ident
  : NameType;
Person1Found, Person2Found
  : Counter;
ErrorMessage   : MessageType;
Person1Index, Person2Index
  : IndexType;

function IdentsEqual (Identa, Identb: IdentifierType) : boolean;
{ Determines whether two numeric Person-Identifiers are equal. A function is necessary because the "=" operator does not work for arrays of anything but char. }
var
  Index   : 1..IdentifierLength;
begin
  IdentsEqual := true;
  for Index := 1 to IdentifierLength do
    if Identa [Index] <> Identb [Index] then
      IdentsEqual := false
  end;  { IdentsEqual }

procedure LinkRelatives (FromIndex : IndexType;
    Relationship : GivenIdentifiers;
    ToIndex     : IndexType);
{ establishes cross-indexing between immediately related Persons. }

procedure LinkOneWay (FromIndex : IndexType;
    ThisEdge   : EdgeType;
    ToIndex   : IndexType);
{ Establishes the NeighborRecord from one Person to another }

var
    NewNeighbor : NeighborPointer;

begin
    new (NewNeighbor);
    with NewNeighbor do
    begin
        NeighborIndex := ToIndex;
        NeighborEdge := ThisEdge;
        NextNeighbor := Person [FromIndex] . NeighborListHeader
    end;
    Person [FromIndex] . NeighborListHeader := NewNeighbor
end;

begin  { execution of LinkRelatives }
if Relationship = SpouseIdent then
begin
    LinkOneWay (FromIndex, Spouse, ToIndex);
    LinkOneWay (ToIndex, Spouse, FromIndex)
end
else  { Relationship is Mother or Father }
begin
    LinkOneWay (FromIndex, Parent, ToIndex);
    LinkOneWay (ToIndex, Child, FromIndex)
end
end;  { LinkRelatives }

procedure PromptAndRead;
{ Issues prompt for user-request, reads in request,
    blank-fills buffer, and skips to next line of input. }

var
    BufferIndex : BufferRange;

begin
    writeln ('');
    writeln ('-----------------------------');
    writeln ('Enter two person-identifiers (name or number),
    separated by semicolon. Enter "stop" to stop.');
    for BufferIndex := 1 to BufferLength do
        if eoln(input) then
            RequestBuffer [BufferIndex] := '
        else
            read (input, RequestBuffer [BufferIndex] );
    readln(input)
end;  { PromptAndRead }
procedure CheckRequest (var RequestStatus : MessageType;
var SemicolonLocation : BufferRange);
{ Performs syntactic check on request in buffer. }
begin
RequestStatus := RequestOk;
Person1FieldExists := false;
Person2FieldExists := false;
SemicolonCount := 0;
for BufferIndex := 1 to BufferLength do
  if RequestBuffer [BufferIndex] <> '"' then
    if RequestBuffer [BufferIndex] = '"' then
      begin
        SemicolonLocation := BufferIndex;
        SemicolonCount := SemicolonCount + 1
      end
    else  { Check for non-blanks before/after semicolon. }
      if SemicolonCount < 1 then
        Person1FieldExists := true
      else
        Person2FieldExists := true;
{ set RequestStatus, based on results of scan of RequestBuffer. }
if SemicolonCount <> 1 then
  RequestStatus := '"must be exactly one semicolon.
else
  if not Person1FieldExists then
    RequestStatus := '"null field preceding semicolon.
  else
    if not Person2FieldExists then
      RequestStatus := '"null field following semicolon.
end;  { CheckRequest }

procedure BufferToPerson (var PersonId : NameType;
StartLocation, StopLocation : BufferRange);
{ fills in the PersonId from the designated portion
of the RequestBuffer. }
begin
BufferIndex := StartLocation;
while RequestBuffer [BufferIndex] = '"' do
  BufferIndex := BufferIndex + 1;
for PersonIndex := 1 to NameLength do
  if BufferIndex > StopLocation then
    PersonId [PersonIndex] := '"'
  else
    begin
      PersonId [PersonIndex] := RequestBuffer [BufferIndex];
      BufferIndex := BufferIndex + 1
    end
end;  { BufferToPerson }
procedure SearchForRequestedPersons (Person1Ident, Person2Ident : NameType;
  var Person1Index, Person2Index : IndexType;
  var Person1Found, Person2Found : Counter);
{ SearchForRequestedPersons scans through the Person array,
  looking for the two requested persons. Match may be by name
  or unique identifier-number. }
var
  Current : IndexType;
  ThisIdent : NameType;
  IdentifierIndex : IdentifierRange;
begin
  Person1Found := 0;
  Person2Found := 0;
  ThisIdent := ' ';
  for Current := 1 to NumberOfPersons do
    with Person [Current] do
      begin
        { ThisIdent contains Current Person's numeric Identifier
          left-justified, padded with blanks. }  
        for IdentifierIndex := 1 to IdentifierLength do
          ThisIdent [IdentifierIndex] := Identifier [IdentifierIndex];
        { allow identification by name or number. }
        if (Person1Ident = ThisIdent) or (Person1Ident = Name) then
          begin
            Person1Found := Person1Found + 1;
            Person1Index := Current
          end;
        if (Person2Ident = ThisIdent) or (Person2Ident = Name) then
          begin
            Person2Found := Person2Found + 1;
            Person2Index := Current
          end
      end  { with Person [Current] }
end;  { SearchForRequestedPersons }

procedure FindRelationship (TargetIndex, SourceIndex : IndexType);
{ Finds shortest path (if any) between two Persons and
  determines their Relationship based on immediate relations
  traversed in path. Person array simulates a directed graph,
  and algorithm finds shortest path, based on following
  weights: Parent-Child edge = 1.0
            Spouse-Spouse edge = 1.8 }
var
  SearchStatus : (Searching, Succeeded, Failed);
  PersonIndex, ThisNode, AdjacentNode, BestNearbyIndex, LastNearbyIndex
    : IndexType;
  NearbyNode    : array [IndexType] of IndexType;
  ThisEdge      : EdgeType;
  ThisNeighbor  : NeighborPointer;
  Relationship  : GivenIdentifiers;
  MinimalDistance : real;
procedure ProcessAdjacentNode (BaseNode, NextNode : IndexType;
   NextBaseEdge : EdgeType);
{ NextNode is adjacent to last-reached node (= BaseNode).
  if NextNode already Reached, do nothing.
  If previously seen, check whether path thru base node is
  shorter than current path to NextNode, and if so re-link
  next to base.
  If not previously seen, link next to base node. }
var
  WeightThisEdge, DistanceThruBaseNode :
  real;

procedure LinkNextNodeToBaseNode;
{ link next to base by re-setting its predecessor Index to
  point to base, note type of edge, and re-set distance
  as it is through base node. }
begin { execution of LinkNextNodeToBaseNode }
  with Person [NextNode] do begin
    begin { execution of ProcessAdjacentNode }
      with Person [NextNode] do begin
        if ReachedStatus <> Reached then begin
          if NextBaseEdge = Spouse then
            WeightThisEdge := 1.8
          else
            WeightThisEdge := 1.0;
          DistanceThruBaseNode := WeightThisEdge +
          Person [BaseNode] . DistanceFromSource;
          if ReachedStatus = NotSeen then begin
            ReachedStatus := Nearby;
            LastNearbyIndex := LastNearbyIndex + 1;
            NearbyNode [LastNearbyIndex] := NextNode;
            LinkNextNodeToBaseNode
          end
        else { ReachedStatus = Nearby }
          if DistanceThruBaseNode < DistanceFromSource then
            LinkNextNodeToBaseNode;
        end { if ReachedStatus <> Reached }
      end; { ProcessAdjacentNode }
    end; { execution of ProcessAdjacentNode }
  end; { execution of LinkNextNodeToBaseNode }
end; { LinkNextNodeToBaseNode }
procedure ResolvePathToEnglish;
{
ResolvePathToEnglish condenses the shortest path to a series of Relationships for which there are English descriptions. }
type
{
Key Persons are the ones in the Relationship path which remain after the path is condensed. }
SiblingType = (Step, Half, Full);
KeyPersonRecord = record
PersonIndex : IndexType;
GenerationGap : Counter;
Proximity : SiblingType;
case RelationToNext : RelationType of
Parent, Child, Spouse, Sibling, Uncle, Nephew, NullRelation : ();
Cousin : (CousinRank : Counter)
end;
var
{ these variables are used to condense the path }
KeyPerson : array [IndexType] of KeyPersonRecord;
KeyRelation, LaterKeyRelation, PrimaryRelation, NextPrimaryRelation :
RelationType;
GenerationCount : Counter;
KeyIndex, LaterKeyIndex, PrimaryIndex : IndexType;
AnotherElementPossible : boolean;

function FullSibling (Index1, Index2 : IndexType) : boolean;
{ Determines whether two Persons are full siblings, i.e., have the same two Parents. }
var
IdentIndex : 1..IdentifierLength;
begin
with Person [Index1] do
FullSibling :=
(not IdentsEqual (RelativeIdentifier [FatherIdent], NullIdent)) and
(not IdentsEqual (RelativeIdentifier [MotherIdent], NullIdent)) and
(IdentsEqual (RelativeIdentifier [FatherIdent],
Person [Index2] . RelativeIdentifier [FatherIdent])) and
(IdentsEqual (RelativeIdentifier [MotherIdent],
Person [Index2] . RelativeIdentifier [MotherIdent]))
end; { FullSibling }

procedure CondenseKeyPersons (AtIndex : IndexType; GapSize : Counter);
{ CondenseKeyPersons condenses superfluous entries from the KeyPerson array, starting at AtIndex. }
var
ReceiveIndex, SendIndex : IndexType;
begin
ReceiveIndex := AtIndex;
repeat
ReceiveIndex := ReceiveIndex + 1;
SendIndex := ReceiveIndex + GapSize;
KeyPerson [ReceiveIndex] := KeyPerson [SendIndex];
until KeyPerson [SendIndex] . RelationToNext = NullRelation
end; { CondenseKeyPersons }
procedure DisplayRelation (FirstIndex, LastIndex, PrimaryIndex : IndexType);
    { DisplayRelation takes 1, 2, or 3 adjacent elements in the condensed table and generates the English description of the relation between the first and last + 1 elements. }
var
    Inlaw : boolean;
    ThisProximity : SiblingType;
    ThisGender : GenderType;
    SuffixIndicator : 0..9;
    FirstRelation, LastRelation, PrimaryRelation : RelationType;
    ThisGenerationGap, ThisCousinRank : Counter;
beginn   { execution of DisplayRelation }
    FirstRelation := KeyPerson [FirstIndex] . RelationToNext;
    LastRelation := KeyPerson [LastIndex] . RelationToNext;
    PrimaryRelation := KeyPerson [PrimaryIndex] . RelationToNext;
    { set ThisProximity }
    if ((PrimaryRelation = Parent) and (FirstRelation = Spouse)) or
       ((PrimaryRelation = Child) and (LastRelation = Spouse))
    then
        ThisProximity := Step
    else
        if PrimaryRelation in [Sibling, Uncle, Nephew, Cousin]
        then
            ThisProximity := KeyPerson [PrimaryIndex] . Proximity
        else
            ThisProximity := Full;
    { set ThisGenerationGap }
    if PrimaryRelation in [Parent, Child, Uncle, Nephew, Cousin]
    then
        ThisGenerationGap := KeyPerson [PrimaryIndex] . GenerationGap
    else
        ThisGenerationGap := 0;
    { set Inlaw }
    Inlaw := false;
    if (FirstRelation = Spouse) and
       (PrimaryRelation in [Sibling, Child, Nephew, Cousin] )
    then
        Inlaw := true;
    if (LastRelation = Spouse) and
       (PrimaryRelation in [Sibling, Parent, Uncle, Cousin] )
    then
        Inlaw := true;
    { set ThisCousinRank }
    if PrimaryRelation = Cousin then
        ThisCousinRank := KeyPerson [PrimaryIndex] . CousinRank
    else
        ThisCousinRank := 0;
{ parameters are set – now generate display. }

write (" ", Person [KeyPerson [FirstIndex] . PersonIndex] . Name, 
   " is ");
if PrimaryRelation in [Parent, Child, Uncle, Nephew] then
   begin { write generation-qualifier }
      if ThisGenerationGap >= 3 then
         begin
            write ("great");
            if ThisGenerationGap > 3 then
               write ("*", ThisGenerationGap - 2 : 1);
            write ("-")
         end;
      if ThisGenerationGap >= 2 then
         write ("grand-")
   end
else
   if (PrimaryRelation = Cousin) and (ThisCousinRank > 1) then
      begin
         write (ThisCousinRank : 1);
         SuffixIndicator := ThisCousinRank mod 10;
         case SuffixIndicator of
            1 : write ("st ");
            2 : write ("nd ");
            3 : write ("rd ");
            0, 4, 5, 6, 7, 8, 9
               : write ("th ")
         end
   end;

if ThisProximity = Step then
   write ("step-")
else
   if ThisProximity = Half then
      write ("half-")
end;

ThisGender := Person [KeyPerson [FirstIndex] . PersonIndex] . Gender; 
case PrimaryRelation of
   Parent : if ThisGender = Male then write ("father")
           else write ("mother");
   Child : if ThisGender = Male then write ("son")
           else write ("daughter");
   Spouse : if ThisGender = Male then write ("husband")
            else write ("wife");
   Sibling : if ThisGender = Male then write ("brother")
            else write ("sister");
   Uncle : if ThisGender = Male then write ("uncle")
             else write ("aunt");
   Nephew : if ThisGender = Male then write ("nephew")
             else write ("niece");
   Cousin : write ("cousin");
   NullRelation : write ("null");
end; { case }
if Inlaw then
    write ('-in-law');

if (PrimaryRelation = Cousin) and (ThisGenerationGap > 0) then
    if ThisGenerationGap > 1 then
        write (' ', ThisGenerationGap : 1, ' times removed')
    else
        write (' once removed');

writeln (' of')
end; { DisplayRelation }

begin { execution of ResolvePathToEnglish }
writeln (' Shortest path between identified persons: ');
ThisNode := TargetIndex;
KeyIndex := 1;
{ Display path and initialize KeyPerson array from path elements. }
while ThisNode <> SourceIndex do
    with Person [ThisNode] do
        begin
            write (' ', Name, ' is ');
            case EdgeToPredecessor of
                Parent : writeln ('parent of');
                Child : writeln ('child of');
                Spouse : writeln ('spouse of')
            end;
            KeyPerson [KeyIndex] . PersonIndex := ThisNode;
            KeyPerson [KeyIndex] . RelationToNext := EdgeToPredecessor;
            if EdgeToPredecessor = Spouse then
                KeyPerson [KeyIndex] . GenerationGap := 0
            else { Parent or Child }
                KeyPerson [KeyIndex] . GenerationGap := 1;
                KeyIndex := KeyIndex + 1;
                ThisNode := PathPredecessor
            end;
            writeln(' ', Person [ThisNode] . Name);
            KeyPerson [KeyIndex] . PersonIndex := ThisNode;
            KeyPerson [KeyIndex] . RelationToNext := NullRelation;
            KeyPerson [KeyIndex + 1] . RelationToNext := NullRelation;
{ Resolve Child-Parent and Child-Spouse-Parent relations to Sibling relations. }

KeyIndex := 1;
while KeyPerson [KeyIndex] . RelationToNext <> NullRelation do
with KeyPerson [KeyIndex] do
begin
    if RelationToNext = Child then
        begin
            LaterKeyRelation := KeyPerson [KeyIndex + 1] . RelationToNext;
            if LaterKeyRelation = Parent then
                begin
                    RelationToNext := Sibling;
                    if FullSibling (PersonIndex, KeyPerson [KeyIndex + 2] . PersonIndex)
                        then
                            Proximity := Full
                        else
                            Proximity := Half;
                        CondenseKeyPersons (KeyIndex, 1)
                end { processing of full/half siblings }
        else
            if (LaterKeyRelation = Spouse) and
            (KeyPerson [KeyIndex + 2] . RelationToNext = Parent)
                then { found step-siblings }
                begin
                    RelationToNext := Sibling;
                    Proximity := Step;
                    CondenseKeyPersons (KeyIndex, 2)
                end { processing of step-siblings }
        end; { if RelationToNext = Child }
    KeyIndex := KeyIndex + 1
end; { with KeyPerson [KeyIndex] }

{ Resolve Child-Child-... and Parent-Parent-... relations to direct descendant or ancestor relations. }
KeyIndex := 1;
while KeyPerson [KeyIndex] . RelationToNext <> NullRelation do
with KeyPerson [KeyIndex] do
begin
    if (RelationToNext = Child) or (RelationToNext = Parent) then
        begin
            LaterKeyIndex := KeyIndex + 1;
            while KeyPerson [LaterKeyIndex] . RelationToNext = RelationToNext do
                LaterKeyIndex := LaterKeyIndex + 1;
            GenerationCount := LaterKeyIndex - KeyIndex;
            if GenerationCount > 1 then
                begin
                    { compress generations }
                    GenerationGap := GenerationCount;
                    CondenseKeyPersons (KeyIndex, GenerationCount - 1)
                end
        end; { if RelationToNext = Child or Parent }
    KeyIndex := KeyIndex + 1
end; { with KeyPerson [KeyIndex] }
{ Resolve Child-Sibling-Parent to Cousin,
   Child-Sibling to Nephew,
   Sibling-Parent to Uncle. }

KeyIndex := 1;
while KeyPerson [KeyIndex] . RelationToNext <> NullRelation do
with KeyPerson [KeyIndex] do
begin
   LaterKeyRelation := KeyPerson [KeyIndex + 1] . RelationToNext;
   if (RelationToNext = Child) and
   (LaterKeyRelation = Sibling)
then
   { Cousin or Nephew }
if KeyPerson [KeyIndex + 2] . RelationToNext = Parent then
   { found Cousin }
begin
   RelationToNext := Cousin;
   Proximity := KeyPerson [KeyIndex + 1] . Proximity;
   if GenerationGap < KeyPerson [KeyIndex + 2] . GenerationGap
then
   CousinRank := GenerationGap
else
   CousinRank := KeyPerson [KeyIndex + 2] . GenerationGap;
   GenerationGap := abs (GenerationGap -
   KeyPerson [KeyIndex + 2] . GenerationGap);
   CondenseKeyPersons (KeyIndex, 2)
end
else
{ found Nephew }
begin
   RelationToNext := Nephew;
   Proximity := KeyPerson [KeyIndex + 1] . Proximity;
   CondenseKeyPersons (KeyIndex, 1)
end
else
{ not Cousin or Nephew }
if (RelationToNext = Sibling) and (LaterKeyRelation = Parent)
then
{ found Uncle }
begin
   RelationToNext := Uncle;
   GenerationGap := KeyPerson [KeyIndex + 1] . GenerationGap;
   CondenseKeyPersons (KeyIndex, 1)
end;
KeyIndex := KeyIndex + 1
end;  { with KeyPerson [KeyIndex] }
{ Loop below will pick out valid adjacent strings of elements to be displayed. KeyIndex points to first element, LaterKeyIndex to last element, and PrimaryIndex to the element which determines the primary English word to be used. Associativity of adjacent elements in condensed table is based on English usage. }

KeyIndex := 1;
writeln (' Condensed path:');
while KeyPerson [KeyIndex] . RelationToNext <> NullRelation do
begin
  KeyRelation := KeyPerson [KeyIndex] . RelationToNext;
  LaterKeyIndex := KeyIndex;
  PrimaryIndex := KeyIndex;
  if KeyPerson [KeyIndex + 1] . RelationToNext <> NullRelation then begin
    { seek multi-element combination }
    AnotherElementPossible := true;
    if KeyRelation = Spouse then begin
      LaterKeyIndex := LaterKeyIndex + 1;
      PrimaryIndex := LaterKeyIndex;
      if (KeyPerson [LaterKeyIndex] . RelationToNext = Sibling) or
        (KeyPerson [LaterKeyIndex] . RelationToNext = Cousin) then
        { Nothing can follow Spouse-Sibling or Spouse-Cousin }
        AnotherElementPossible := false
    end;
    { PrimaryIndex is now correctly set. Next if-statement determines if a following Spouse relation should be appended to this combination or left for the next combination. }
    if AnotherElementPossible and
      (KeyPerson [PrimaryIndex + 1] . RelationToNext = Spouse) then
      { Only a Spouse can follow a Primary }
    then begin
      { check primary preceding and following Spouse. }
      PrimaryRelation :=
        KeyPerson [PrimaryIndex] . RelationToNext;
      NextPrimaryRelation :=
        KeyPerson [PrimaryIndex + 2] . RelationToNext;
      if (NextPrimaryRelation in [Nephew, Cousin, NullRelation] )
        or (PrimaryRelation = Nephew)
        or ( ( PrimaryRelation in [Sibling, Parent] )
             and (NextPrimaryRelation <> Uncle ) ) then
        { append following Spouse with this combination. }
        LaterKeyIndex := LaterKeyIndex + 1
    end
    { multi-element combination }
  end;
  DisplayRelation (KeyIndex, LaterKeyIndex, PrimaryIndex);
  KeyIndex := LaterKeyIndex + 1
end; { while }
writeln ('", Person [KeyPerson [KeyIndex] . PersonIndex] . Name)
procedure ComputeCommonGenes (Index1, Index2 : IndexType);
{ ComputeCommonGenes assumes that each ancestor contributes half of the genetic material to a Person. It finds common ancestors between two Persons and computes the expected value of the Proportion of common material. }

var
CommonProportion : real;

procedure ZeroProportion (ZeroIndex : IndexType);
{ ZeroProportion recursively seeks out all ancestors and zeros them out. }

var
ThisNeighbor : NeighborPointer;

begin
with Person [ZeroIndex] do
begin
DescendantGenes := 0.0;
ThisNeighbor := NeighborListHeader
end;
while ThisNeighbor <> nil do
with ThisNeighbor^ do
begin
if NeighborEdge = Parent then
ZeroProportion (NeighborIndex);
ThisNeighbor := NextNeighbor
end  { with }
end;  { ZeroProportion }

procedure MarkProportion (Marker : IdentifierType;
Proportion : real; MarkedIndex : IndexType);
{ MarkProportion recursively seeks out all ancestors and marks them with the sender's Proportion of shared genetic material. This Proportion is diluted by one-half for each generation. }

var
ThisNeighbor : NeighborPointer;

begin
with Person [MarkedIndex] do
begin
DescendantIdentifier := Marker;
DescendantGenes := DescendantGenes + Proportion;
ThisNeighbor := NeighborListHeader
end;
while ThisNeighbor <> nil do
with ThisNeighbor^ do
begin
if NeighborEdge = Parent then
MarkProportion (Marker, Proportion / 2.0,
NeighborIndex );
ThisNeighbor := NextNeighbor
end
end;  { MarkProportion }
procedure CheckCommonProportion
(var CommonProportion : real;
    MatchIdentifier : IdentifierType;
    Proportion : real;
    AlreadyCounted : real;
    CheckIndex : IndexType);
{ CheckCommonProportion searches all the ancestors of
  CheckIndex to see if any have been marked, and if so
  adds the appropriate amount to CommonProportion. }
var
  ThisNeighbor : NeighborPointer;
  ThisContribution : real;
begin
  with Person [CheckIndex] do begin
    begin
      if IdentsEqual (DescendantIdentifier, MatchIdentifier) then begin
        { Increment CommonProportion by the contribution of
          this common ancestor, but discount for the contribution
          of less remote ancestors already counted. }
        ThisContribution := DescendantGenes * Proportion;
        CommonProportion := CommonProportion +
          ThisContribution - AlreadyCounted
      end
    else
      ThisContribution := 0.0;
    end;
    begin
      ThisNeighbor := NeighborListHeader
    end; { with Person [CheckIndex] }
  end;
  while ThisNeighbor <> nil do
  with ThisNeighbor do begin
    if NeighborEdge = Parent then
    begin
      CheckCommonProportion (CommonProportion,
                           MatchIdentifier, Proportion / 2.0,
                           ThisContribution / 4.0,
                           NeighborIndex);
      ThisNeighbor := NextNeighbor
    end
  end; { CheckCommonProportion }
begin { ComputeCommonGenes }
{ First zero out all ancestors to allow adding. This is necessary
  because there might be two paths to an ancestor. }
  ZeroProportion (Index1);
{ now mark with shared Proportion }
  MarkProportion ( Person [Index1] . Identifier, 1.0, Index1);
  CommonProportion := 0.0;
  CheckCommonProportion (CommonProportion,
                        Person [Index1] . Identifier, 1.0, 0.0, Index2);
  writeln ('Proportion of common genetic material = ',
            CommonProportion : 12)
end; { ComputeCommonGenes }
begin  { execution of FindRelationship } 
{ initialize Person-array for processing - 
  mark all nodes as not seen }
for PersonIndex := 1 to NumberOfPersons do 
  Person [PersonIndex] . ReachedStatus := NotSeen;
{ mark source node as Reached }
ThisNode := SourceIndex;
with Person [ThisNode] do 
  begin
    ReachedStatus := Reached;
    DistanceFromSource := 0.0
  end;
{ no Nearby nodes exist yet }
LastNearbyIndex := 0;
if ThisNode = TargetIndex then
  SearchStatus := Succeeded
else
  SearchStatus := Searching;
{ Loop keeps processing closest-to-source, unreached node 
  until target Reached, or no more connected nodes. } 
while SearchStatus = Searching do
  begin
  { Process all nodes adjacent to ThisNode }
  ThisNeighbor := Person [ThisNode] . NeighborListHeader;
  while ThisNeighbor <> nil do 
    with ThisNeighbor do
      begin
        ProcessAdjacentNode (ThisNode, NeighborIndex, NeighborEdge);
        ThisNeighbor := NextNeighbor
      end;
{ All nodes adjacent to ThisNode are set. Now search for 
  shortest-distance unreached (but Nearby) node to process next. }
if LastNearbyIndex = 0 then
  SearchStatus := Failed
else 
  begin
    MinimalDistance := 1.0e+18;
    for PersonIndex := 1 to LastNearbyIndex do
      with Person [NearbyNode [PersonIndex]] do
        if DistanceFromSource < MinimalDistance then
          begin
            BestNearbyIndex := PersonIndex;
            MinimalDistance := DistanceFromSource
          end;
{ Establish new ThisNode }
ThisNode := NearbyNode [BestNearbyIndex];
{ change ThisNode from being Nearby to Reached }
Person [ThisNode] . ReachedStatus := Reached;
{ remove ThisNode from Nearby list }
NearbyNode [BestNearbyIndex] := NearbyNode [LastNearbyIndex];
LastNearbyIndex := LastNearbyIndex - 1;
if ThisNode = TargetIndex then
  SearchStatus := Succeeded
end { determination of next node to process }
ed;} { while SearchStatus = Searching }
{ Shortest path between Persons now established. Next task is to translate path to English description of Relationship. }

if SearchStatus = Failed then
    writeln ('  Person [TargetIndex]. Name, ' is not related to ' ,
         Person [SourceIndex]. Name)
else { success - parse path to find and display Relationship }
    begin
        ResolvePathToEnglish;
        ComputeCommonGenes (SourceIndex, TargetIndex)
    end
end;  { FindRelationship }

{ *** execution of main sequence begins here *** }

begin
    for IdentifierIndex := 1 to IdentifierLength do
        Nullldent [IdentifierIndex] := '0';
    reset (People);
    { Current location in array being filled }
    Current := 0;
    { This loop reads in the People file and constructs the Person array from it (one Person = one record = one array entry).
    As records are read in, links are constructed to represent the Parent-Child or Spouse relationship. The array then implements a directed graph which is used to satisfy subsequent user requests. The file is assumed to be correct - no validation is performed on it. }
    while not eof(People) do
        begin
            Current := Current+1;
            with Person [Current] do
                begin
                    { copy direct information from file to array }
                    Name := People`. Name;
                    Identifier := People`. Identifier;
                    if People`. Gender = 'M' then
                        Gender := Male
                    else
                        Gender := Female;
                    RelativeIdentifier := People `. RelativeIdentifier;
                    { Location of adjacent persons as yet undetermined }
                    NeighborListHeader := nil;
                    { Descendants as yet undetermined. }
                    DescendantIdentifier := NullIden;
                    CurrentIdentifier := Identifier;
{ Compare this Person against all previously entered Persons to search for Relationships. }

for Previous := 1 to (Current-1) do
begin
PreviousIdent := Person [Previous]. Identifier;
RelationLoopDone := false;
Relationship := FatherIdent;
{ Search for father, mother, or spouse Relationship in either direction between this and previous Person. Assume at most one Relationship exists. }
repeat
  if IdentsEqual (RelativeIdentifier [Relationship], PreviousIdent) then
    begin
      LinkRelatives (Current, Relationship, Previous);
      RelationLoopDone := true
    end
  else
    if IdentsEqual (CurrentIdent,
      Person [Previous]. RelativeIdentifier [Relationship])
    then
      begin
        LinkRelatives (Previous, Relationship, Current);
        RelationLoopDone := true
      end;
      if Relationship < SpouseIdent then
        Relationship := succ(Relationship)
      else
        RelationLoopDone := true;
    until RelationLoopDone
  end; { for Previous }
get(People)
end { with Person [Current] }
end; { while not eof(People) }
NumberOfPersons := eof;

{ Person array is now loaded and edges between immediate relatives (Parent-Child or Spouse-Spouse) are established. }

While-loop accepts requests and finds Relationship (if any) between pairs of Persons. }
reset(input);
PromptAndRead;
while RequestBuffer <> RequestToStop do
  { The following code retrieves and validates a user request for the Relationship between two identified Persons. }
begin
  CheckRequest (ErrorMessage, SemicolonLocation);
  { Syntax check of request completed. Now either display error message or search for the two Persons. }
  if ErrorMessage = RequestOk then
    begin
      { Request syntactically correct - search for requested Persons. }
      BufferToPerson (Person1Ident, 1, SemicolonLocation - 1);
      BufferToPerson (Person2Ident, SemicolonLocation + 1, BufferLength);
      SearchForRequestedPersons (Person1Ident, Person2Ident, Person1Index, Person2Index, Person1Found, Person2Found);
      if (Person1Found = 1) and (Person2Found = 1) then
        { Exactly one match for each Person - proceed to determine Relationship, if any. }
        if Person1Index = Person2Index then
          begin
            write ("\". Person [Person1Index] . Name, " is identical to \"");
            if Person [Person1Index] . Gender = Male then writeln("himself.");
            else writeln("herself.");
          end
        else
          FindRelationship (Person1Index, Person2Index)
      else
        { either not found or more than one found }
        begin
          if Person1Found = 0 then
            writeln ("First person not found.");
          else
            if Person1Found > 1 then
              writeln("Duplicate names for first person - use", " numeric identifier.");
            if Person2Found = 0 then
              writeln("Second person not found.");
            else
              if Person2Found > 1 then
                writeln("Duplicate names for second person - use", " numeric identifier.");
          end
        end
    end
  else
    writeln ("Incorrect request format: \", ErrorMessage);
PromptAndRead
end; { while RequestBuffer }
writeln ("End of relation-finder.");
end.
8.0 PL/I

In keeping with the general convention of the examples, language-supplied keywords and identifiers are written in lower case in the program. To conform strictly to the PL/I standard, however, programs must use only upper-case letters. In the following program, the logical "Not" operator is represented by the graphic character "~".

RELATE: procedure options (main);

/* Begin declaration of global data */

declare
   /* Used to index relative array, pointing to immediate relatives */
   ( FATHER_IDENT   initial (1),
     MOTHER_IDENT   initial (2),
     SPOUSE_IDENT   initial (3),
     PARENT         initial (1),
     CHILD          initial (2),
     SPOUSE         initial (3),
     SIBLING        initial (4),
     UNCLE          initial (5),
     NEPHEW         initial (6),
     COUSIN          initial (7),
     NULL_RELATION  initial (8),
     REACHED        initial (1),
     NEARBY         initial (2),
     NOT_SEEN       initial (3))
fixed binary (4,0),

   /* Used as mnemonics to represent truth-values */
   ( TRUE           initial ('1'b),
     FALSE          initial ('0'b))
bit (1),

   /* Used to control user requests. */
   ( REQUEST_OK    character (10) initial ('Request OK'),
     REQUEST_TO_STOP character (4) initial ('stop'),
     MALE            initial ('M'),
     FEMALE          initial ('F'))
character (1);
declare  
/* the PERSON array is the central repository of information  
about inter-relationships. */
/* All relationships are captured in the directed graph of which  
each record is a node. */
01 PERSON dimension (1:300),
  /* static information - filled from PEOPLE file: */
  05 NAME character (20),
  05 IDENTIFIER picture '999',
  05 GENDER character (1),
  /* IDENTIFIERS of immediate relatives - father, mother, spouse */
  05 RELATIVE_IDENTIFIER (1:3) picture '999',
  /* head of linked list of adjacent nodes */
  05 NEIGHBOR_LIST_HEADER pointer,
  /* data used when traversing graph to resolve user request: */
  05 DISTANCE_FROM_SOURCE float decimal (6),
  05 PATH_PREDECESSOR fixed binary (10,0),
  05 EDGE_TO_PREDECESSOR fixed binary (4,0),
  05 REACHED_STATUS fixed binary (4,0),
  /* data used to compute common genetic material */
  05 DESCENDANT_IDENTIFIER picture '999',
  05 DESCENDANT_GENES float decimal (6);

declare  
/* each PERSON has a linked list of adjacent nodes, called neighbors */
01 NEIGHBOR_RECORD based (NEW_NEIGHBOR),
  05 NEIGHBOR_INDEX fixed binary (10,0),
  05 NEIGHBOR_EDGE fixed binary (4,0),
  05 NEXT_NEIGHBOR pointer;

/* End declaration of global data. */

declare  
/* This is the format of records in the file to be read in. */
01 PEOPLE_RECORD,
  05 NAME character (20),
  05 IDENTIFIER picture '999',
  /* 'M' for MALE and 'F' for FEMALE */
  05 GENDER character (1),
  05 RELATIVE_IDENTIFIER (1:3) picture '999';

declare  
/* These variables are used when establishing the PERSON array  
from the PEOPLE file. */
PEOPLE file record sequential input,
(CURRENT, PREVIOUS, NUMBER_OF_PERSONS) fixed binary (10,0),
(PREVIOUS_IDENT, CURRENT_IDENT) picture '999',
NULL_IDENT picture '999' static initial (000),
RELATIONSHIP fixed binary (4,0),
RELATION LOOP DONE bit (1),
END_OF_PEOPLE bit (1);
declare

/* These variables are used to accept and resolve requests for RELATIONSHIP information. */
sysin file record input environment (AREAD),
(BUFFER_INDEX, SEMICOLON_LOCATION)
   fixed binary (10,0),
REQUEST_BUFFER    character (60) varying,
(PERSON1_IDENT, PERSON2_IDENT)
   character (20),
(PERSON1_FOUND, PERSON2_FOUND)
   fixed binary (10,0),
ERROR_MESSAGE     character (40),
(PERSON1_INDEX, PERSON2_INDEX)
   fixed binary (10,0);

/* This on-block captures exceptions from the following code */
on endfile (PEOPLE)
   begin;
      END_OF_PEOPLE = TRUE;
   end;
open file (PEOPLE) title ('PEOPLE.DAT');
END_OF_PEOPLE = FALSE;

/* This loop reads in the PEOPLE file and constructs the PERSON array from it (one PERSON = one record = one array entry). As records are read in, links are constructed to represent the PARENT-CHILD or SPOUSE RELATIONSHIP. The array then implements a directed graph which is used to satisfy subsequent user requests. The file is assumed to be correct - no validation is performed on it. */
read file (PEOPLE) into (PEOPLE_RECORD);
READ_IN_PEOPLE:
do CURRENT = 1 to 300 while (~ END_OF_PEOPLE);
    /* copy direct information from file to array */
    PERSON (CURRENT) = PEOPLE_RECORD, by name;
    /* Location of adjacent persons as yet undetermined. */
    PERSON (CURRENT) . NEIGHBOR_LIST_HEADER = null();
    /* Descendants as yet undetermined */
    PERSON (CURRENT) . DESCENDANT_IDENTIFIER = NULL_IDENT;
    CURRENT_IDENT = PERSON (CURRENT) . IDENTIFIER;
    /* Compare this PERSON against all previously entered PERSONs to search for RELATIONSHIPS. */
    COMPARE_TO_PREVIOUS:
do PREVIOUS = 1 to (CURRENT-1);
    PREVIOUS_IDENT = PERSON (PREVIOUS) . IDENTIFIER;
    RELATION_LOOP_DONE = FALSE;
    /* Search for father, mother, or spouse relationship in either direction between this and PREVIOUS PERSON. Assume at most one RELATIONSHIP exists. */
    TRY_ALL_RELATIONSHIPS:
do RELATIONSHIP = FATHER_IDENT to SPOUSE_IDENT while (~ RELATION_LOOP_DONE);
        if PERSON (CURRENT) . RELATIVE_IDENTIFIER (RELATIONSHIP) = PREVIOUS_IDENT then
            do;
                call LINK_RELATIVES (CURRENT, RELATIONSHIP, PREVIOUS);
                RELATION_LOOP_DONE = TRUE;
            end;
        else
            if CURRENT_IDENT = PERSON (PREVIOUS) . RELATIVE_IDENTIFIER (RELATIONSHIP) then
                do;
                    call LINK_RELATIVES (PREVIOUS, RELATIONSHIP, CURRENT);
                    RELATION_LOOP_DONE = TRUE;
                end;
            end TRY_ALL_RELATIONSHIPS;
        end
    end COMPARE_TO_PREVIOUS;
read file (PEOPLE) into (PEOPLE_RECORD);
end READ_IN_PEOPLE;
NUMBER_OF_PERSONS = CURRENT - 1;
close file (PEOPLE);

/* PERSON array is now loaded and edges between immediate relatives (PARENT-CHILD or SPOUSE-SPOUSE) are established. */
While-loop accepts requests and finds RELATIONSHIP (if any) between pairs of PERSONs. */

call PROMPT_AND_READ();
READ_AND_PROCESS_REQUEST:
do while (REQUEST_BUFFER ^= REQUEST_TO_STOP);
  /* The following code retrieves and validates a user request
   for the RELATIONSHIP between two identified PERSONs. */
call CHECK_REQUEST (ERROR_MESSAGE, SEMICOLON_LOCATION);
  /* Syntax check of request completed. Now either display error
   message or search for the two PERSONs. */
if ERROR_MESSAGE = REQUEST_OK then
  do; /* Request syntactically correct -
    search for requested PERSONs. */
call BUFFER_TO_PERSON (PERSON1_IDENT, 1, SEMICOLON_LOCATION - 1);
call BUFFER_TO_PERSON (PERSON2_IDENT, SEMICOLON_LOCATION + 1,
  length (REQUEST_BUFFER));
call SEARCH_FOR_REQUESTED_PERSONS (PERSON1_IDENT, PERSON2_IDENT,
  PERSON1_INDEX, PERSON2_INDEX, PERSON1_FOUND, PERSON2_FOUND);
  if (PERSON1_FOUND = 1) & (PERSON2_FOUND = 1) then
    /* Exactly one match for each PERSON - proceed to
    determine RELATIONSHIP, if any. */
    if PERSON1_INDEX = PERSON2_INDEX then
      if PERSON (PERSON1_INDEX) . GENDER = MALE then
        put skip list (" " || PERSON (PERSON1_INDEX) . NAME ||
          " is identical to himself.");
      else
        put skip list (" " || PERSON (PERSON1_INDEX) . NAME ||
          " is identical to herself.");
      else
        call FIND_RELATIONSHIP (PERSON1_INDEX, PERSON2_INDEX);
    else /* either not found or more than one found */
      do;
        if PERSON1_FOUND = 0 then
          put skip list (" First person not found.");
        else
          if PERSON1_FOUND > 1 then
            put skip list (" Duplicate names for first person - use" ||
              " numeric identifier.");
          if PERSON2_FOUND = 0 then
            put skip list (" Second person not found.");
          else
            if PERSON2_FOUND > 1 then
              put skip list (" Duplicate names for second person - use" ||
                " numeric identifier.");
            end;
        end; /* processing of syntactically legal request */
      else
        put skip list (" Incorrect request format: " || ERROR_MESSAGE);
call PROMPT_AND_READ();
end READ_AND_PROCESS_REQUEST;
put skip list (" End of relation-finder.");
/* End execution of main sequence RELATE
procedures under RELATE begin here */

LINK_RELATIVES: procedure (FROM_INDEX, RELATIONSHIP, TO_INDEX);

declare
  FROM_INDEX    fixed binary (10,0),
  RELATIONSHIP  fixed binary (4,0),
  TO_INDEX      fixed binary (10,0);

/* begin execution of LINK_RELATIVES */

if RELATIONSHIP = SPOUSE_IDENT then
  do;
    call LINK_ONE_WAY (FROM_INDEX, SPOUSE, TO_INDEX);
    call LINK_ONE_WAY (TO_INDEX, SPOUSE, FROM_INDEX);
  end;
else /* RELATIONSHIP is mother or father */
  do;
    call LINK_ONE_WAY (FROM_INDEX, PARENT, TO_INDEX);
    call LINK_ONE_WAY (TO_INDEX, CHILD, FROM_INDEX);
  end;

LINK_ONE_WAY: procedure (FROM_INDEX, THIS_EDGE, TO_INDEX);

declare
  FROM_INDEX    fixed binary (10,0),
  THIS_EDGE     fixed binary (4,0),
  TO_INDEX      fixed binary (10,0);

declare
  NEW_NEIGHBOR  pointer;

/* begin execution of LINK_ONE_WAY */
allocate NEIGHBOR_RECORD set (NEW_NEIGHBOR);
NEW_NEIGHBOR -> NEIGHBOR_INDEX = TO_INDEX;
NEW_NEIGHBOR -> NEIGHBOR_EDGE = THIS_EDGE;
NEW_NEIGHBOR -> NEXT_NEIGHBOR =
    PERSON (FROM_INDEX) . NEIGHBOR_LIST_HEADER;
PERSON (FROM_INDEX) . NEIGHBOR_LIST_HEADER = NEW_NEIGHBOR;
end LINK_ONE_WAY;
end LINK_RELATIVES;

PROMPT_AND_READ: procedure;
/* Issues prompt for user-request, reads in request,
   blank-fills buffer, and skips to next line of input. */

declare
  BUFFER_INDEX    fixed binary (10,0),
  SEMICOLON_COUNT fixed binary (4,0);
/* begin execution of PROMPT_AND_READ */
put skip (2) list ('-----------------------------------------------');
put skip list ('Enter two person-identifiers (name or number),');
put skip list (' separated by semicolon. Enter "stop" to stop.');
put skip list ('');

/* The use of sysin for record-oriented, rather than stream-oriented,
input may not be considered to be standard usage. It is done here
because stream input cannot recognize line boundaries, so as to
read an entire line from the terminal. */
read file (sysin) into (REQUEST BUFFER);
end PROMPT_AND_READ;

CHECK_REQUEST: procedure (REQUESTSTATUS, SEMICOLON_LOCATION);
/* Performs syntactic check on request in buffer. */
declare
  REQUESTSTATUS   character (40),
  SEMICOLON_LOCATION fixed binary (10,0);

/* begin execution of CHECK_REQUEST */
SEMICOLON_LOCATION = index (REQUEST BUFFER, ';');
if SEMICOLON_LOCATION = 0 |
  index (substr (REQUEST BUFFER, SEMICOLON_LOCATION + 1), ';') > 0
then
  REQUESTSTATUS = 'must be exactly one semicolon.';
else
  if before (REQUEST BUFFER, ';') = ' ' then
    REQUESTSTATUS = 'null field preceding semicolon.';
  else
    if after (REQUEST BUFFER, ';') = ' ' then
      REQUESTSTATUS = 'null field following semicolon.';
    else
      REQUESTSTATUS = REQUEST_OK;
end CHECK_REQUEST;

BUFFER_TO_PERSON: procedure (PERSON_ID, START_LOCATION, STOP_LOCATION);
/* fills in the PERSON_ID from the designated portion
of the REQUEST_BUFFER. */
declare
  PERSON_ID   character (20),
  (START_LOCATION, STOP_LOCATION)
    fixed binary (10,0);

 declare
  FIRST_NON_BLANK fixed binary (10,0);

/* begin execution of BUFFER_TO_PERSON */
do FIRST_NON_BLANK = START_LOCATION to STOP_LOCATION
  while (substr (REQUEST BUFFER, FIRST_NON_BLANK, 1) = ' ');
end;
PERSON_ID = substr (REQUEST_BUFFER, FIRST_NON_BLANK,
  STOP_LOCATION - FIRST_NON_BLANK + 1);
end BUFFER_TO_PERSON;
SEARCH_FOR_REQUESTED_PERSONS: procedure (PERSON1_IDENT, PERSON2_IDENT, PERSON1_INDEX, PERSON2_INDEX, PERSON1_FOUND, PERSON2_FOUND);

/* SEARCH_FOR_REQUESTED_PERSONS scans through the PERSON array, looking for the two requested PERSONs. Match may be by NAME or unique IDENTIFIER-number. */
declare
  (PERSON1_IDENT, PERSON2_IDENT) character (20),
  (PERSON1_INDEX, PERSON2_INDEX) fixed binary (10,0),
  (PERSON1_FOUND, PERSON2_FOUND) fixed binary (10,0);
declare
  THIS_IDENT character (20),
  CURRENT fixed binary (10,0);
/* begin execution of SEARCH_FOR_REQUESTED_PERSONS */
PERSON1_FOUND = 0;
PERSON2_FOUND = 0;
SCAN_ALL_PERSONS:
docURRENT = 1 to NUMBER_OF_PERSONS;
  /* THIS_IDENT contains CURRENT PERSON's numeric IDENTIFIER left-justified, padded with blanks. */
  THIS_IDENT = PERSON (CURRENT). IDENTIFIER;
  /* allow identification by name or number. */
  if (PERSON1_IDENT = THIS_IDENT) |
    (PERSON1_IDENT = PERSON (CURRENT). NAME)
    then
      do;
      PERSON1_FOUND = PERSON1_FOUND + 1;
      PERSON1_INDEX = CURRENT;
      end;
    if (PERSON2_IDENT = THIS_IDENT) |
      (PERSON2_IDENT = PERSON (CURRENT). NAME)
      then
        do;
        PERSON2_FOUND = PERSON2_FOUND + 1;
        PERSON2_INDEX = CURRENT;
        end;
      end SCAN_ALL_PERSONS;
end SEARCH_FOR_REQUESTED_PERSONS;

/* End of utility procedures under RELATE. */
FIND_RELATIONSHIP does major work of program: determines relationship between any two people in PERSON array. */

FIND_RELATIONSHIP: procedure (TARGET_INDEX, SOURCE_INDEX);
/* Finds shortest path (if any) between two PERSONs and determines their RELATIONSHIP based on immediate relations traversed in path. PERSON array simulates a directed graph, and algorithm finds shortest path, based on following weights: PARENT-CHILD edge = 1.0
SPouse-SPouse edge = 1.8 */
declare (TARGET_INDEX, SOURCE_INDEX) fixed binary (10,0);
declare SEARCH_STATUS character (1),
/* values for SEARCH_STATUS */
(SEARCHING initial ('?'),
SUCCEEDED initial ('!'),
FAILED initial ('X')) character (1),
(PERSON_INDEX, THIS_NODE, ADJACENT_NODE, BEST_NEARBY_INDEX,
LAST_NEARBY_INDEX) fixed binary (10,0),
NEARBY_NODE dimension (1:300) fixed binary (10,0),
THIS_EDGE fixed binary (4,0),
THIS_NEIGHBOR pointer,
RELATIONSHIP fixed binary (4,0),
MINIMAL_DISTANCE float decimal (6);

/* begin execution of FIND_RELATIONSHIP */
/* initialize PERSON-array for processing - mark all nodes as not seen */
PERSON . REACHED_STATUS = NOT_SEEN;
/* mark source node as REACHED */
THIS_NODE = SOURCE_INDEX;
PERSON (THIS_NODE) . REACHED_STATUS = REACHED;
PERSON (THIS_NODE) . DISTANCE_FROM_SOURCE = 0.0;
/* no NEARBY nodes exist yet */
LAST_NEARBY_INDEX = 0;
if THIS_NODE = TARGET_INDEX then
  SEARCH_STATUS = SUCCEEDED;
else
  SEARCH_STATUS = SEARCHING;
/* Loop keeps processing closest-to-source, unREACHED node  
    until target REACHED, or no more connected nodes. */
SEARCH_FOR_TARGET:
do while (SEARCH_STATUS = SEARCHING);
    /* Process all nodes adjacent to THIS_NODE */
    THIS_NEIGHBOR = PERSON (THIS_NODE) . NEIGHBOR_LIST_HEADER;
do while (THIS_NEIGHBOR ^= null());
call PROCESS_ADJACENT_NODE (THIS_NODE,  
    THIS_NEIGHBOR -> NEIGHBOR_INDEX,  
    THIS_NEIGHBOR -> NEIGHBOR_EDGE);
    THIS_NEIGHBOR = THIS_NEIGHBOR -> NEXT_NEIGHBOR; 
end;

/* All nodes adjacent to THIS_NODE are set. Now search for  
    shortest-distance unREACHED (but NEARBY) node to process next. */
if LAST_NEARBY_INDEX = 0 then
    SEARCH_STATUS = FAILED;
else
do;
    MINIMAL_DISTANCE = 1.0e+18;
do PERSON_INDEX = 1 to LAST_NEARBY_INDEX;
    if PERSON (NEARBY_NODE (PERSON_INDEX)) . DISTANCE_FROM_SOURCE  
        < MINIMAL_DISTANCE then
do;
        BEST_NEARBY_INDEX = PERSON_INDEX;
        MINIMAL_DISTANCE =  
            PERSON (NEARBY_NODE (PERSON_INDEX)) . DISTANCE_FROM_SOURCE;
    end;
end;  /* PERSON_INDEX loop */
/* establish new THIS_NODE */
THIS_NODE = NEARBY_NODE (BEST_NEARBY_INDEX);  
/* change THIS_NODE from being NEARBY to REACHED */
PERSON (THIS_NODE) . REACHED_STATUS = REACHED;
/* remove THIS_NODE from nearby list */
NEARBY_NODE (BEST_NEARBY_INDEX) = NEARBY_NODE (LAST_NEARBY_INDEX);
LAST_NEARBY_INDEX = LAST_NEARBY_INDEX - 1;
if THIS_NODE = TARGET_INDEX then
    SEARCH_STATUS = SUCCEEDED;
end;  /* determination of next node to process */
end SEARCH_FOR_TARGET;

/* Shortest path between PERSONs now established. Next task is  
    to translate path to English description of RELATIONSHIP. */
if SEARCH_STATUS = FAILED then
    put skip list (' ', PERSON (TARGET_INDEX) . NAME, ' is not related to  
        PERSON (SOURCE_INDEX) . NAME);
else  /* success - parse path to find and display RELATIONSHIP */
do;
call RESOLVE_PATH_TO_ENGLISH;
call COMPUTE_COMMON_GENES (SOURCE_INDEX, TARGET_INDEX);
end;

/* End execution of FIND_RELATIONSHIP. */
Utility procedures begin here. */

PROCESS_ADJACENT_NODE: procedure (BASE_NODE, NEXT_NODE, NEXT_BASE_EDGE);
/* NEXT_NODE is adjacent to last-REACHED node (= BASE_NODE).
    if NEXT_NODE already REACHED, do nothing.
    If previously seen, check whether path thru BASE_NODE is
    shorter than current path to NEXT_NODE, and if so re-link
    next to base.
    If not previously seen, link next to base node. */
declare
    (BASE_NODE, NEXT_NODE) fixed binary (10,0),
    NEXT_BASE_EDGE fixed binary (4,0);
declare
    (WEIGHT_THIS_EDGE, DISTANCE_THRU_BASE_NODE)
        float decimal (6);

/* begin execution of PROCESS_ADJACENT_NODE */
if PERSON (NEXT_NODE). REACHED_STATUS ^= REACHED then
    do;
        if NEXT_BASE_EDGE = SPOUSE then
            WEIGHT_THIS_EDGE = 1.8;
        else
            WEIGHT_THIS_EDGE = 1.0;
        DISTANCE_THRU_BASE_NODE = WEIGHT_THIS_EDGE +
            PERSON (BASE_NODE). DISTANCE_FROM_SOURCE;
        if PERSON (NEXT_NODE). REACHED_STATUS = NOT_SEEN then
            do;
                PERSON (NEXT_NODE). REACHED_STATUS = NEARBY;
                LAST_NEARBY_INDEX = LAST_NEARBY_INDEX + 1;
                NEARBY_NODE^ (LAST_NEARBY_INDEX) = NEXT_NODE;
                call LINK_NEXT_NODE_TO_BASE_NODE;
            end;
        else /* REACHED_STATUS = NEARBY */
            if DISTANCE_THRU_BASE_NODE <
                PERSON (NEXT_NODE). DISTANCE_FROM_SOURCE then
                call LINK_NEXT_NODE_TO_BASE_NODE;
            end; /* if REACHED_STATUS not = REACHED */
end /* if PERSON (NEXT_NODE). REACHED_STATUS ^= REACHED */

LINK_NEXT_NODE_TO_BASE_NODE: procedure;
/* link next to base by re-setting its predecessor index to
    point to base, note type of edge, and re-set distance
    as it is through base node. */
/* begin execution of LINK_NEXT_NODE_TO_BASE_NODE */
PERSON (NEXT_NODE). DISTANCE_FROM_SOURCE = DISTANCE_THRU_BASE_NODE;
PERSON (NEXT_NODE). PATH_PREDECESSOR = BASE_NODE;
PERSON (NEXT_NODE). EDGE_TO_PREDECESSOR = NEXT_BASE_EDGE;
end LINK_NEXT_NODE_TO_BASE_NODE;

end PROCESS_ADJACENT_NODE;

/* End utility procedures under FIND_RELATIONSHIP.*/
Begin two major procedures: RESOLVE_PATH_TO_ENGLISH and COMPUTE_COMMONGENES. */

RESOLVE_PATH_TO_ENGLISH: procedure;
/* RESOLVE_PATH_TO_ENGLISH condenses the shortest path to a series of RELATIONSHIPS for which there are English descriptions. */
/* Key persons are the ones in the RELATIONSHIP path which remain after the path is condensed. */
declare
/* values for sibling proximity */
(STEP initial ('S'),
HALF initial ('H'),
FULL initial ('F')) character (1);
declare
01 KEY_PERSON dimension (1:300),
 05 PERSON_INDEX fixed binary (10,0),
 05 GENERATION_GAP fixed binary (10,0),
 05 PROXIMITY character (1),
 05 RELATION_TO_NEXT fixed binary (4,0),
 05 COUSIN_RANK fixed binary (10,0);
declare
/* these variables are used to condense the path */
(KEY_RELATION, LATER_KEY_RELATION, PRIMARY_RELATION,
NEXT_PRIMARY_RELATION) fixed binary (4,0),
GENERATION_COUNT fixed binary (10,0),
(KEY_INDEX, LATER_KEY_INDEX, PRIMARY_INDEX)
fixed binary (10,0),
ANOTHER_ELEMENT_POSSIBLE bit (1);

/* begin execution of RESOLVE_PATH_TO_ENGLISH */
put skip list (" Shortest path between identified persons: ");
THIS_NODE = TARGET_INDEX;
/* Display path and initialize KEY_PERSON array from path elements. */
TRAVVERSE_SHORTEST_PATH:
do KEY_INDEX = 1 to 300 while (THIS_NODE ^= SOURCE_INDEX);
begin;
declare
  EDGE_TYPE dimension (1:3) character (9) static
    initial ('parent of', 'child of', 'spouse of');
put skip list (" II PERSON (THIS_NODE). NAME || is ||
  EDGE_TYPE (PERSON (THIS_NODE). EDGE_TO_PREDECESSOR));
end;
KEY_PERSON (KEY_INDEX) . PERSON_INDEX = THIS_NODE;
KEY_PERSON (KEY_INDEX) . RELATION_TO_NEXT =
  PERSON (THIS_NODE) . EDGE_TO_PREDECESSOR;
if PERSON (THIS_NODE) . EDGE_TO_PREDECESSOR = SPOUSE then
  KEY_PERSON (KEY_INDEX) . GENERATION_GAP = 0;
else
  KEY_PERSON (KEY_INDEX) . GENERATION_GAP = 1;
THIS_NODE = PERSON (THIS_NODE) . PATH_PREDECESSOR;
end TRAVVERSE_SHORTEST_PATH;
put skip list(" II PERSON (THIS_NODE). NAME");
KEY_PERSON (KEY_INDEX) . PERSON_INDEX = THIS_NODE;
KEY_PERSON (KEY_INDEX) . RELATION_TO_NEXT = NULL_RELATION;
KEY_PERSON (KEY_INDEX + 1) . RELATION_TO_NEXT = NULL_RELATION;
/* Resolve CHILD-PARENT and CHILD-SPOUSE-PARENT relations
to SIBLING relations. */

FIND_SIBLINGS:
do KEY_INDEX = 1 to 300
while (KEY_PERSON (KEY_INDEX) . RELATION_TO_NEXT ~= NULL_RELATION);
if KEY_PERSON (KEY_INDEX) . RELATION_TO_NEXT = CHILD then
do;
LATER_KEY_RELATION = KEY_PERSON (KEY_INDEX + 1) . RELATION_TO_NEXT;
if LATER_KEY_RELATION = PARENT then
/* found either full or half SIBLINGS */
do;
KEY_PERSON (KEY_INDEX) . RELATION_TO_NEXT = SIBLING;
if FULL_SIBLING (KEY_PERSON (KEY_INDEX) . PERSON_INDEX,
KEY_PERSON (KEY_INDEX + 2) . PERSON_INDEX)
then
KEY_PERSON (KEY_INDEX) . PROXIMITY = FULL;
else
KEY_PERSON (KEY_INDEX) . PROXIMITY = HALF;
call CONDENSE_KEY_PERSONS (KEY_INDEX, 1);
end; /* processing of full/half SIBLINGS */
else
if (LATER_KEY_RELATION = SPOUSE) &
(KEY_PERSON (KEY_INDEX + 2) . RELATION_TO_NEXT = PARENT)
then /* found step-SIBLINGS */
do;
KEY_PERSON (KEY_INDEX) . RELATION_TO_NEXT = SIBLING;
KEY_PERSON (KEY_INDEX) . PROXIMITY = STEP;
call CONDENSE_KEY_PERSONS (KEY_INDEX, 2);
end; /* processing of step-SIBLINGS */
end; /* if RELATION_TO_NEXT = CHILD */
end FIND_SIBLINGS;
/* Resolve CHILD-CHILD-... and PARENT-PARENT-... relations to
direct descendant or ancestor relations. */

FIND_ANCESTORS_OR_DESCENDANTS:
do KEY_INDEX = 1 to 300
while (KEY_PERSON (KEY_INDEX) . RELATION_TO_NEXT ~= NULL_RELATION);
if (KEY_PERSON (KEY_INDEX) . RELATION_TO_NEXT = CHILD ) |
(KEY_PERSON (KEY_INDEX) . RELATION_TO_NEXT = PARENT)
then
do;
LATER_KEY_INDEX = KEY_INDEX + 1 to 300
while (KEY_PERSON (LATER_KEY_INDEX) . RELATION_TO_NEXT =
KEY_PERSON (KEY_INDEX) . RELATION_TO_NEXT);
end;
GENERATION_COUNT = LATER_KEY_INDEX - KEY_INDEX;
if GENERATION_COUNT > 1 then
do; /* compress generations */
KEY_PERSON (KEY_INDEX) . GENERATION_GAP = GENERATION_COUNT;
call CONDENSE_KEY_PERSONS (KEY_INDEX, GENERATION_COUNT - 1);
end;
end; /* if RELATION_TO_NEXT = CHILD or PARENT */
end FIND_ANCESTORS_OR_DESCENDANTS;
/* Resolve CHILD-SIBLING-PARENT to COUSIN, CHILD-SIBLING to NEPHEW, SIBLING-PARENT to UNCLE. */

FIND_COUSINS_NEPHES_NECLES:
do  KEY_INDEX = 1 to 300
  while (KEY_PERSON (KEY_INDEX) . RELATION_TO_NEXT  = NULL_RELATION);
    LATER_KEY_RELATION = KEY_PERSON (KEY_INDEX + 1) . RELATION_TO_NEXT;
    if (KEY_PERSON (KEY_INDEX) . RELATION_TO_NEXT = CHILD) &
      (LATER_KEY_RELATION = SIBLING)
      then /* COUSIN or NEPHEW */
        if KEY_PERSON (KEY_INDEX + 2) . RELATION_TO_NEXT = PARENT then
          /* Found COUSIN */
          do;
            KEY_PERSON (KEY_INDEX) . RELATION_TO_NEXT = COUSIN;
            KEY_PERSON (KEY_INDEX) . PROXIMITY =
              KEY_PERSON (KEY_INDEX + 1) . PROXIMITY;
            KEY_PERSON (KEY_INDEX) . COUSIN_RANK =
              min (KEY_PERSON (KEY_INDEX) . GENERATION_GAP,
              KEY_PERSON (KEY_INDEX + 2) . GENERATION_GAP);
            KEY_PERSON (KEY_INDEX) . GENERATION_GAP =
              abs (KEY_PERSON (KEY_INDEX) . GENERATION_GAP -
              KEY_PERSON (KEY_INDEX + 2) . GENERATION_GAP);
            call CONDENSE_KEY_PERSONS (KEY_INDEX, 2);
          end;
        else /* Found NEPHEW */
          do;
            KEY_PERSON (KEY_INDEX) . RELATION_TO_NEXT = NEPHEW;
            KEY_PERSON (KEY_INDEX) . PROXIMITY =
              KEY_PERSON (KEY_INDEX + 1) . PROXIMITY;
            call CONDENSE_KEY_PERSONS (KEY_INDEX, 1);
          end;
        else /* Not COUSIN or NEPHEW */
          if (KEY_PERSON (KEY_INDEX) . RELATION_TO_NEXT = SIBLING) &
            (LATER_KEY_RELATION = PARENT)
            then /* Found UNCLE */
              do;
                KEY_PERSON (KEY_INDEX) . RELATION_TO_NEXT = UNCLE;
                KEY_PERSON (KEY_INDEX) . GENERATION_GAP =
                  KEY_PERSON (KEY_INDEX + 1) . GENERATION_GAP;
                call CONDENSE_KEY_PERSONS (KEY_INDEX, 1);
              end;
          end
  end
end FIND_COUSINS_NEPHES_NECLES;
/* Loop below will pick out valid adjacent strings of elements to be displayed. KEY_INDEX points to first element, LATER_KEY_INDEX to last element, and PRIMARY_INDEX to the element which determines the primary English word to be used. Associativity of adjacent elements in condensed table is based on English usage. */

KEY_INDEX = 1;

put skip list (' Condensed path:');
CONSOLIDATE_ADJACENT_PERSONS:
do while (KEY_PERSON (KEY_INDEX) . RELATION_TO_NEXT ~= NULL_RELATION);
   KEY_RELATION = KEY_PERSON (KEY_INDEX) . RELATION_TO_NEXT;
   LATER_KEY_INDEX = KEY_INDEX;
   PRIMARY_INDEX = KEY_INDEX;
   if KEY_PERSON (KEY_INDEX + 1) . RELATION_TO_NEXT ~= NULL_RELATION then
do; /* seek multi-element combination */
      ANOTHER_ELEMENT_POSSIBLE = TRUE;
      if KEY_RELATION = SPOUSE then
do;
         LATER_KEY_INDEX = LATER_KEY_INDEX + 1;
         PRIMARY_INDEX = LATER_KEY_INDEX;
         if (KEY_PERSON (LATER_KEY_INDEX) . RELATION_TO_NEXT = SIBLING) | (KEY_PERSON (LATER_KEY_INDEX) . RELATION_TO_NEXT = COUSIN) then /* Nothing can follow SPOUSE-SIBLING or SPOUSE-COUSIN */
            ANOTHER_ELEMENT_POSSIBLE = FALSE;
         end;
      end;
      /* PRIMARY_INDEX is now correctly set. Next if-statement determines if a following SPOUSE relation should be appended to this combination or left for the next combination. */
      if ANOTHER_ELEMENT_POSSIBLE & (KEY_PERSON (PRIMARY_INDEX + 1) . RELATION_TO_NEXT = SPOUSE) /* Only a SPOUSE can follow a Primary */ then
do; /* check primary preceding and following SPOUSE. */
      PRIMARY_RELATION = KEY_PERSON (PRIMARY_INDEX) . RELATION_TO_NEXT;
      NEXT_PRIMARY_RELATION = KEY_PERSON (PRIMARY_INDEX + 2) . RELATION_TO_NEXT;
      if (NEXT_PRIMARY_RELATION = NEPHEW) | (NEXT_PRIMARY_RELATION = COUSIN) | (PRIMARy_RELATION = NEPHEW) | ( ( PRIMARY_RELATION = SIBLING | PRIMARY_RELATION = PARENT) & (NEXT_PRIMARY_RELATION ~= UNCLE ) ) then /* append following SPOUSE with this combination. */
         LATER_KEY_INDEX = LATER_KEY_INDEX + 1;
      end; /* check primary preceding and following SPOUSE */
   end; /* multi-element combination */
call DISPLAY_RELATION (KEY_INDEX, LATER_KEY_INDEX, PRIMARY_INDEX);
end CONSOLIDATE_ADJACENT_PERSONS;
put skip list (' PERSON (KEY_PERSON (KEY_INDEX) . PERSON_INDEX) . NAME);
/* End execution of RESOLVE_PATH_TO_ENGLISH. */
Begin utility procedures for RESOLVE_PATH_TO_ENGLISH. */

FULL_SIBLING: procedure (INDEX1, INDEX2)
    returns (bit(1));
    /* Determines whether two PERSONs are full siblings, i.e.,
    have the same two parents. */

declare
    (INDEX1, INDEX2) fixed binary (10,0);

return
    ((PERSON (INDEX1) . RELATIVE_IDENTIFIER (FATHER_IDENT) ^= NULL_IDENT) &
    (PERSON (INDEX1) . RELATIVE_IDENTIFIER (MOTHER_IDENT) ^= NULL_IDENT) &
    (PERSON (INDEX1) . RELATIVE_IDENTIFIER (FATHER_IDENT) =
        PERSON (INDEX2) . RELATIVE_IDENTIFIER (FATHER_IDENT) ) &
    (PERSON (INDEX1) . RELATIVE_IDENTIFIER (MOTHER_IDENT) =
        PERSON (INDEX2) . RELATIVE_IDENTIFIER (MOTHER_IDENT) ) );
end FULL_SIBLING;

CONDENSE_KEY_PERSONS: procedure (AT_INDEX, GAP_SIZE);
    /* CONDENSE_KEY_PERSONS condenses superfluous entries from the
    KEY_PERSON array, starting at AT_INDEX. */

declare
    AT_INDEX fixed binary (10,0),
    GAP_SIZE fixed binary (10,0);

declare
    (RECEIVE_INDEX, SEND_INDEX) fixed binary (10,0);

    /* begin execution of CONDENSE_KEY_PERSONS */
    RECEIVED_INDEX  = AT_INDEX + 1;
    SEND_INDEX      = RECEIVE_INDEX + GAP_SIZE;
    KEY_PERSON (RECEIVE_INDEX)  = KEY_PERSON (SEND_INDEX);
    do while (KEY_PERSON (SEND_INDEX) . RELATION_TO_NEXT ^= NULL_RELATION);
        RECEIVE_INDEX = RECEIVE_INDEX + 1;
        SEND_INDEX    = RECEIVE_INDEX + GAP_SIZE;
        KEY_PERSON (RECEIVE_INDEX)  = KEY_PERSON (SEND_INDEX);
    end;
end CONDENSE_KEY_PERSONS;

    /* End utility procedures. */
Begin DISPLAY_RELATION, which does major work of displaying
under RESOLVE_PATH_TO_ENGLISH. */

DISPLAY_RELATION: procedure (FIRST_INDEX, LAST_INDEX, PRIMARY_INDEX);

/* DISPLAY_RELATION takes 1, 2, or 3 adjacent elements in the
condensed table and generates the English description of
the relation between the first and last + 1 elements. */
declare
(FIRST_INDEX, LAST_INDEX, PRIMARY_INDEX) fixed binary (10,0);
declare
DISPLAY_BUFFER character (80) varying,
INLAW bit (1),
THIS_PROXIMITY character (1),
THIS_GENDER character (1),
SUFFIX_INDICATOR fixed binary (6,0),
(FIRST_RELATION, LAST_RELATION, PRIMARY_RELATION)
fixed binary (4,0),
(THE_GENERATION_GAP, THIS_CUSIN_RANK)
fixed binary (10,0);

/* begin execution of DISPLAY_RELATION */
FIRST_RELATION = KEY_PERSON (FIRST_INDEX) . RELATION_TO_NEXT;
LAST_RELATION = KEY_PERSON (LAST_INDEX) . RELATION_TO_NEXT;
PRIMARY_RELATION = KEY_PERSON (PRIMARY_INDEX) . RELATION_TO_NEXT;
/* set THIS_PROXIMITY */
if ((PRIMARY_RELATION = PARENT) & (FIRST_RELATION = SPOUSE)) |
((PRIMARY_RELATION = CHILD) & (LAST_RELATION = SPOUSE))
then
   THIS_PROXIMITY = STEP;
else
   if PRIMARY_RELATION = SIBLING |
      PRIMARY_RELATION = UNCLE |
      PRIMARY_RELATION = NEPHEW |
      PRIMARY_RELATION = COUSIN
then
   THIS_PROXIMITY = KEY_PERSON (PRIMARY_INDEX) . PROXIMITY;
else
   THIS_PROXIMITY = FULL;
/* set THIS_GENERATION_GAP */
if PRIMARY_RELATION = PARENT |
   PRIMARY_RELATION = CHILD |
   PRIMARY_RELATION = UNCLE |
   PRIMARY_RELATION = NEPHEW |
   PRIMARY_RELATION = COUSIN
then
   THIS_GENERATION_GAP = KEY_PERSON (PRIMARY_INDEX) . GENERATION_GAP;
else
   THIS_GENERATION_GAP = 0;
/* set INLAW */

INLAW = FALSE;
if (FIRST_RELATION = SPOUSE) &
    (PRIMARY_RELATION = SIBLING |
     PRIMARY_RELATION = CHILD |
     PRIMARY_RELATION = NEPHEW |
     PRIMARY_RELATION = COUSIN)
then
    INLAW = TRUE;
if (LAST_RELATION = SPOUSE) &
    (PRIMARY_RELATION = SIBLING |
     PRIMARY_RELATION = PARENT |
     PRIMARY_RELATION = UNCLE |
     PRIMARY_RELATION = COUSIN)
then
    INLAW = TRUE;

/* set THIS_COUSIN_RANK */
if PRIMARY_RELATION = COUSIN then
    THIS_COUSIN_RANK = KEY_PERSON (PRIMARY_INDEX) . COUSIN_RANK;
else
    THIS_COUSIN_RANK = 0;

/* parameters are set - now generate display. */

DISPLAY BUFFER =
    " | PERSON (KEY_PERSON (FIRST_INDEX) . PERSON_INDEX) . NAME | is ";
if PRIMARY_RELATION = PARENT |
    PRIMARY_RELATION = CHILD |
    PRIMARY_RELATION = UNCLE |
    PRIMARY_RELATION = NEPHEW
then
do; /* write generation-qualifier */
if THIS_GENERATION_GAP >= 3 then
do;
    DISPLAY BUFFER = DISPLAY BUFFER || "great";
if THIS_GENERATION_GAP > 3 then
    DISPLAY BUFFER = DISPLAY BUFFER || "*" ||
        TRIM (THIS_GENERATION_GAP - 2);
    DISPLAY BUFFER = DISPLAY BUFFER || "-";
end;
if THIS_GENERATION_GAP >= 2 then
    DISPLAY BUFFER = DISPLAY BUFFER || "grand-";
end;
else
    if (PRIMARY_RELATION = COUSIN) & (THIS_COUSIN_RANK > 1) then
do;
    DISPLAY BUFFER = DISPLAY BUFFER || TRIM (THIS_COUSIN_RANK);
    SUFFIX_INDICATOR = mod (THIS_COUSIN_RANK, 10);
    if SUFFIX_INDICATOR > 3 then
        SUFFIX_INDICATOR = 0;
    DISPLAY BUFFER = DISPLAY BUFFER ||
        substr ("th st nd rd ", 3 * SUFFIX_INDICATOR + 1, 3);
end;
if THIS_PROXIMITY = STEP then
    DISPLAY_BUFFER = DISPLAY_BUFFER || 'step-';
else
    if THIS_PROXIMITY = HALF then
        DISPLAY_BUFFER = DISPLAY_BUFFER || 'half-';

THIS_GENDER = PERSON (KEY_PERSON (FIRST_INDEX) . PERSON_INDEX) . GENDER;
if PRIMARY_RELATION = PARENT then
    if THIS_GENDER = MALE then DISPLAY_BUFFER = DISPLAY_BUFFER || 'father';
    else
        DISPLAY_BUFFER = DISPLAY_BUFFER || 'mother';
else if PRIMARY_RELATION = CHILD then
    if THIS_GENDER = MALE then DISPLAY_BUFFER = DISPLAY_BUFFER || 'son';
    else
        DISPLAY_BUFFER = DISPLAY_BUFFER || 'daughter';
else if PRIMARY_RELATION = SPOUSE then
    if THIS_GENDER = MALE then DISPLAY_BUFFER = DISPLAY_BUFFER || 'husband';
    else
        DISPLAY_BUFFER = DISPLAY_BUFFER || 'wife';
else if PRIMARY_RELATION = SIBLING then
    if THIS_GENDER = MALE then DISPLAY_BUFFER = DISPLAY_BUFFER || 'brother';
    else
        DISPLAY_BUFFER = DISPLAY_BUFFER || 'sister';
else if PRIMARY_RELATION = UNCLE then
    if THIS_GENDER = MALE then DISPLAY_BUFFER = DISPLAY_BUFFER || 'uncle';
    else
        DISPLAY_BUFFER = DISPLAY_BUFFER || 'aunt';
else if PRIMARY_RELATION = NEPHEW then
    if THIS_GENDER = MALE then DISPLAY_BUFFER = DISPLAY_BUFFER || 'nephew';
    else
        DISPLAY_BUFFER = DISPLAY_BUFFER || 'niece';
else if PRIMARY_RELATION = COUSIN then
    DISPLAY_BUFFER = DISPLAY_BUFFER || 'cousin';
else
    DISPLAY_BUFFER = DISPLAY_BUFFER || 'null';

if INLAW then
    DISPLAY_BUFFER = DISPLAY_BUFFER || '-in-law';

if (PRIMARY_RELATION = COUSIN) & (THIS_GENERATION_GAP > 0) then
    if THIS_GENERATION_GAP > 1 then
        DISPLAY_BUFFER = DISPLAY_BUFFER || 'times removed';
    else
        DISPLAY_BUFFER = DISPLAY_BUFFER || 'once removed';

DISPLAY_BUFFER = DISPLAY_BUFFER || 'of';
put skip list (DISPLAY_BUFFER);
/* Begin utility procedure for DISPLAY_RELATION */

TRIM: procedure (NUMERIC_VALUE) returns (character (20) varying);
/* Returns character representation of numeric values with no leading or trailing spaces. */
declare 
  NUMERIC_VALUE   fixed binary (10,0);
declare 
  STRING_REPRESENTATION character (20),
    (START_LOCATION, STOP_LOCATION)
      fixed binary (10,0);
/* Begin execution of TRIM */
STRING_REPRESENTATION    = NUMERIC_VALUE;
do 
  START_LOCATION   = 1 to 20 
    while (substr (STRING_REPRESENTATION, START_LOCATION, 1) = ' ');
end;
do 
  STOP_LOCATION   = 20 to 1 by -1
    while (substr (STRING_REPRESENTATION, STOP_LOCATION, 1) = ' ');
end;
return (substr (STRING_REPRESENTATION, 
    STOP_LOCATION - START_LOCATION + 1));
end TRIM;
end DISPLAY_RELATION;
end RESOLVE_PATH_TO_ENGLISH;

/* COMPUTE_COMMON_GENES is second major procedure (after RESOLVE_PATH_TO_ENGLISH) under FIND_RELATIONSHIP. */

COMPUTE_COMMON_GENES: procedure (INDEX1, INDEX2);
/* COMPUTE_COMMON_GENES assumes that each ancestor contributes half of the genetic material to a PERSON. It finds common ancestors between two PERSONs and computes the expected value of the PROPORTION of common material. */
declare 
  (INDEX1, INDEX2) fixed binary (10,0);
declare 
  COMMON_PROPORTION float decimal (6);
/* begin execution of COMPUTE_COMMON_GENES */
/* First zero out all ancestors to allow adding. This is necessary because there might be two paths to an ancestor. */
call ZERO_PROPORTION (INDEX1);
/* now mark with shared PROPORTION */
call MARK_PROPORTION (PERSON (INDEX1) . IDENTIFIER, 1.0, INDEX1);
COMMON_PROPORTION = 0.0;
call CHECK_COMMON_PROPORTION (COMMON_PROPORTION, 
    PERSON (INDEX1) . IDENTIFIER, 1.0, 0.0, INDEX2);
put skip list ('Proportion of common genetic material = ');
put edit (COMMON_PROPORTION) (e(13,5,6));
/* End execution of COMPUTE_COMMON_GENES. */
Begin utility procedures. */

ZERO_PROPORTION: procedure (ZERO_INDEX) recursive;
    /* ZERO_PROPORTION recursively seeks out all ancestors and zeros them out. */

declare
    ZERO_INDEX fixed binary (10,0),
    THIS_NEIGHBOR pointer;
/* begin execution of ZERO_PROPORTION */
    PERSON (ZERO_INDEX). DESCENDANT_GENES = 0.0;
    THIS_NEIGHBOR = PERSON (ZERO_INDEX). NEIGHBOR_LIST_HEADER;
    do while (THIS_NEIGHBOR ^= null());
        if THIS_NEIGHBOR -> NEIGHBOR_EDGE = PARENT then
            call ZERO_PROPORTION (THIS_NEIGHBOR -> NEIGHBOR_INDEX);
        THIS_NEIGHBOR = THIS_NEIGHBOR -> NEXT_NEIGHBOR;
    end;
end ZERO_PROPORTION;

MARK_PROPORTION: procedure (MARKER, PROPORTION, MARKED_INDEX) recursive;
    /* MARK_PROPORTION recursively seeks out all ancestors and marks them with the sender's PROPORTION of shared genetic material. This PROPORTION is diluted by one-half for each generation. */

declare
    MARKER picture '999',
    PROPORTION float decimal (6),
    MARKED_INDEX fixed binary (10,0),
    THIS_NEIGHBOR pointer;
/* begin execution of MARK_PROPORTION */
    PERSON (MARKED_INDEX). DESCENDANT_IDENTIFIER = MARKER;
    PERSON (MARKED_INDEX). DESCENDANT_GENES =
        PERSON (MARKED_INDEX). DESCENDANT_GENES + PROPORTION;
    THIS_NEIGHBOR = PERSON (MARKED_INDEX). NEIGHBOR_LIST_HEADER;
    do while (THIS_NEIGHBOR ^= null());
        if THIS_NEIGHBOR -> NEIGHBOR_EDGE = PARENT then
            call MARK_PROPORTION (MARKER, PROPORTION / 2.0,
                THIS_NEIGHBOR -> NEIGHBOR_INDEX);
        THIS_NEIGHBOR = THIS_NEIGHBOR -> NEXT_NEIGHBOR;
    end;
end MARK_PROPORTION;
CHECK_COMMON_PROPORTION: procedure
   (COMMON_PROPORTION, MATCH_IDENTIFIER, PROPORTION,
   ALREADY_COUNTED, CHECK_INDEX) recursive;

/* CHECK_COMMON_PROPORTION searches all the ancestors of
   CHECK_INDEX to see if any have been marked, and if so
   adds the appropriate amount to COMMON_PROPORTION. */

declare
   COMMON_PROPORTION float decimal (6),
   MATCH_IDENTIFIER picture '999',
   PROPORTION float decimal (6),
   ALREADY_COUNTED float decimal (6),
   CHECK_INDEX fixed binary (10,0),
   THIS_NEIGHBOR pointer,
   THIS_CONTRIBUTION float decimal (6);

/* begin execution of CHECK_COMMON_PROPORTION */
if PERSON (CHECK_INDEX) . DESCENDANT_IDENTIFIER = MATCH_IDENTIFIER then
   /* Increment COMMON_PROPORTION by the contribution of
      this common ancestor, but discount for the contribution
      of less remote ancestors already counted. */
   do;
      THIS_CONTRIBUTION = PERSON (CHECK_INDEX) . DESCENDANT_GENES
         * PROPORTION;
      COMMON_PROPORTION = COMMON_PROPORTION
         + THIS_CONTRIBUTION - ALREADY_COUNTED;
   end;
else
   THIS_CONTRIBUTION = 0.0;
   THIS_NEIGHBOR = PERSON (CHECK_INDEX) . NEIGHBOR_LIST_HEADER;
   do while (THIS_NEIGHBOR ~= null());
      if THIS_NEIGHBOR -> NEIGHBOR_EDGE = PARENT then
         call CHECK_COMMON_PROPORTION (COMMON_PROPORTION,
            MATCH_IDENTIFIER, PROPORTION / 2.0,
            THIS_CONTRIBUTION / 4.0,
            THIS_NEIGHBOR -> NEIGHBOR_INDEX);
      end;
      THIS_NEIGHBOR = THIS_NEIGHBOR -> NEXT_NEIGHBOR;
   end;
end CHECK_COMMON_PROPORTION;
end COMPUTE_COMMON_GENES;
end FIND_RELATIONSHIP;
end RELATE;
**Selection and Use of General-Purpose Programming Languages--Program Examples**

**Author(s):** John V. Cugini

**Performing Organization:**

**NATIONAL BUREAU OF STANDARDS**
**DEPARTMENT OF COMMERCE**
**GAITHERSBURG, MD 20899**

**SPONSORING ORGANIZATION NAME AND COMPLETE ADDRESS:**

Same as in item 6 above.

**Library of Congress Catalog Card Number:** 84-601120

Document describes a computer program; SF-185, FIPS Software Summary, is attached.

**Abstract:**

Programming languages have been and will continue to be an important instrument for the automation of a wide variety of functions within industry and the Federal Government. Other instruments, such as program generators, application packages, query languages, and the like, are also available and their use is preferable in some circumstances.

Given that conventional programming is the appropriate technique for a particular application, the choice among the various languages becomes an important issue. There are a great number of selection criteria, not all of which depend directly on the language itself. Broadly speaking, the criteria are based on 1) the language and its implementation, 2) the application to be programmed, and 3) the user's existing facilities and software.

This study presents a survey of selection factors for the major general-purpose languages: Ada, BASIC, C, COBOL, FORTRAN, Pascal, and PL/I. The factors covered include not only the logical operations within each language, but also the advantages and disadvantages stemming from the current computing environment, e.g., software packages, microcomputers, and standards. The criteria associated with the application and the user's facilities are explained. Finally, there is a set of program examples to illustrate the features of the various languages.

This volume includes the program examples. Volume 1 contains the discussion of language selection criteria.

**Key Words:**

Ada; alternatives to programming; BASIC; C; COBOL; FORTRAN; Pascal; PL/I; programming language features; programming languages; selection of programming language.

**Availability:**

- Unlimited
- For Official Distribution. Do Not Release to NTIS
- Order From National Technical Information Service (NTIS), Springfield, VA. 22161

**Printed Pages:** 178

**Price:**
ANNOUNCEMENT OF NEW PUBLICATIONS ON
COMPUTER SCIENCE & TECHNOLOGY

Superintendent of Documents,
Government Printing Office,
Washington, DC 20402

Dear Sir:

Please add my name to the announcement list of new publications to be issued in the series: National Bureau of Standards Special Publication 500-.

Name ____________________________________________________________

Company _________________________________________________________

Address _________________________________________________________

City __________________________ State _______________ Zip Code _______

(Notification key N-503)
**Technical Publications**

**Periodicals**

**Journal of Research**—The Journal of Research of the National Bureau of Standards reports NBS research and development in those disciplines of the physical and engineering sciences in which the Bureau is active. These include physics, chemistry, engineering, mathematics, and computer sciences. Papers cover a broad range of subjects, with major emphasis on measurement methodology and the basic technology underlying standardization. Also included from time to time are survey articles on topics closely related to the Bureau's technical and scientific programs. As a special service to subscribers each issue contains complete citations to all recent Bureau publications in both NBS and non-NBS media. Issued six times a year.

**Nonperiodicals**

**Monographs**—Major contributions to the technical literature on various subjects related to the Bureau's scientific and technical activities.

**Handbooks**—Recommended codes of engineering and industrial practice (including safety codes) developed in cooperation with interested industries, professional organizations, and regulatory bodies.

**Special Publications**—Include proceedings of conferences sponsored by NBS, NBS annual reports, and other special publications appropriate to this grouping such as wall charts, pocket cards, and bibliographies.

**Applied Mathematics Series**—Mathematical tables, manuals, and studies of special interest to physicists, engineers, chemists, biologists, mathematicians, computer programmers, and others engaged in scientific and technical work.

**National Standard Reference Data Series**—Provides quantitative data on the physical and chemical properties of materials, compiled from the world's literature and critically evaluated. Developed under a worldwide program coordinated by NBS under the authority of the National Standard Data Act (Public Law 90-396). NOTE: The Journal of Physical and Chemical Reference Data (JPCRD) is published quarterly for NBS by the American Chemical Society (ACS) and the American Institute of Physics (AIP). Subscriptions, reprints, and supplements are available from ACS, 1155 Sixteenth St., NW, Washington, DC 20036.

**Building Science Series**—Disseminates technical information developed at the Bureau on building materials, components, systems, and whole structures. The series presents research results, test methods, and performance criteria related to the structural and environmental functions and the durability and safety characteristics of building elements and systems.

**Technical Notes**—Studies or reports which are complete in themselves but restrictive in their treatment of a subject. Analogous to monographs but not so comprehensive in scope or definitive in treatment of the subject area. Often serve as a vehicle for final reports of work performed at NBS under the sponsorship of other government agencies.

**Voluntary Product Standards**—Developed under procedures published by the Department of Commerce in Part 10, Title 15, of the Code of Federal Regulations. The standards establish nationally recognized requirements for products, and provide all concerned interests with a basis for common understanding of the characteristics of the products. NBS administers this program as a supplement to the activities of the private sector standardizing organizations.

**Consumer Information Series**—Practical information, based on NBS research and experience, covering areas of interest to the consumer. Easily understandable language and illustrations provide useful background knowledge for shopping in today's technological marketplace.


*Order the following NBS publications—FIPS and NBSIR's—from the National Technical Information Service, Springfield, VA 22161.*


**NBS Interagency Reports (NBSIR)**—A special series of interim or final reports on work performed by NBS for outside sponsors (both government and non-government). In general, initial distribution is handled by the sponsor; public distribution is by the National Technical Information Service, Springfield, VA 22161, in paper copy or microfiche form.
Official Business
Penalty for Private Use $300