Programming language and Compiler

Programming Assignment 2:

department: CSIE

grade: sphomore

group: 胡師睿、劉家均

student ID: 411221301、411221426

professor: Chung Yung

Content

I. Problem Description

This assignment contains two distinct but complementary exercises:

- **Exercise 1**: Simulate and infer family relationships (e.g., parent, sibling, cousin) based on factual data such as gender, marriage, and parenthood.

  Fact #1:

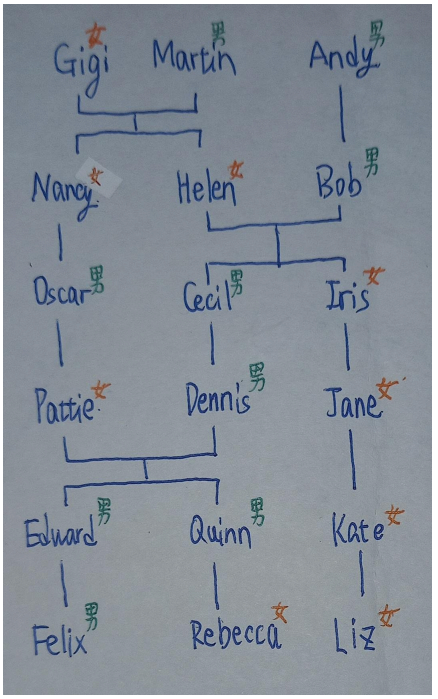
  - Males: Andy, Bob, Cecil, Dennis, Edward, Felix, Martin, Oscar, Quinn
  - Females: Gigi, Helen, Iris, Jane, Kate, Liz, Nancy, Pattie, Rebecca

  Fact #2:

  - Married Couples:
    - Bob and Helen
    - Dennis and Pattie
    - Gigi and Martin

  Fact #3:

  - Parent-child relationships (partial):
    - Andy is the parent of Bob
    - Bob is the parent of Cecil
    - Cecil is the parent of Dennis
    - … (etc.)



- **Exercise 2**: Based on the folder **“HW2-PL-Data”**, which contains student data, perform a payment record analysis to calculate the **total amount received** and identify **students who have underpaid** according to their payment type.

Each problem is implemented using a different language suited to its nature:

- **Java, ML, Prolog**: for logic inference (Exercise 1)
- **COBOL, R**: for structured financial data processing (Exercise 2)

II. Highlights of Implementation

Exercise 1: Family Relationship Inference

- **Java**:

  Common implementation highlight:

  - **Data Representation:**
    1. Hash-based structures are generally used to store and access information efficiently.
    2. Set is used to store gender groups to ensure uniqueness and avoid duplicate entries.
    3. List is used for child lists to allow ordered and easy traversal.
    4. Map is used to establish and quickly access relationships such as parents, children, and spouses.
  - **Fact Processing:**
    The given facts are parsed and structured during initialization, ensuring all relationships are properly constructed.
  - **Relational Rules:**
    Both versions implement five defined rules (parenthood, sibling, brothers/sisters, and cousins) through dedicated functions.

  Difference between two versions:

  - **Version 1: Interactive Relationship Inference**
    Use a function to input two names; the program will then automatically determine the relationship between them.
  - **Version 2: Manual Function Invocation for Relationship Checks**
    We can manually call functions like areParent or areSibling in the main function to check the relationship between two names.

- **Prolog**:
  Implemented family facts and rules declaratively using logical inference. Relations such as **parent**, **married**, **sibling**, **brother**, **sister**, and **cousin** were defined based on the provided logic. Prolog's built-in pattern matching made family relationship reasoning elegant and concise.

- **ML (Standard ML)**:
  The program uses lists to represent gender, marriage, and parent relationships. Predicate functions like is_male, married, and parent are defined using List.exists for simple fact matching. Core relationships—such as sibling, brother, sister, and cousin—are implemented through recursive and nested list checks.

Exercise 2: Student Payment Analysis

- **R**:

  Common implementation highlight:

  - Used data frames and vectorized operations to:
    - Merge tables.
    - Compute total payments.
    - Identify underpaid students.
  - Applied core R functions for efficient tabular analysis:
    - merge() – to combine datasets.
    - subset() – to filter specific conditions (e.g., underpayments).
    - sum() – to calculate totals.
  - Handled missing data (NA values) by:
    - Detecting empty entries using is.na().
    - Replacing them with zeros using assignment or ifelse() logic.

  Difference between two versions:

  - **Version 1: Manual Data Entry**
    - Student, fee, and payment records are manually constructed within the program using data frames.
  - **Version 2: CSV-Based Data Import**
    - Read three CSV files (**Student_Main.csv**, **Fees.csv, Student_Payment.csv**) into data frames using **read.csv()**.
    - The imported data is assigned to predefined variables for structured access.
    - Numeric values stored as strings are converted to proper numeric types, with commas removed to ensure accurate computations.

- **COBOL**:

  Used **WORKING-STORAGE tables** to hold student data. Implemented nested loops to compute total received and to identify underpaid students based on matching student IDs and payment types. Emphasized line-by-line, record-based processing.

III. Program Listing

In all implementations, clarity and modularity were prioritized. Each language was used in a way that aligns with its paradigm.

Exercise 1:

- **Java**: **(Two Versions)**:

  Common implementation:

  - **Data Representation:**

    In general, a combination of HashSet, Map, and List is used to initialize the core variables—such as gender, spouse, parents, and children.

```java
// Define sets of male and female names
//Set<String> is a data structure that does not repeat elements
static Set<String> males = new HashSet<>(List.of(e1:"Andy", e2:"Bob", e3:"Cecil", e4:"Dennis", e5:"Edward", e6:"Felix", e7:"Martin", e8:"Oscar", e9:"Quinn"));
static Set<String> females = new HashSet<>(List.of(e1:"Gigi", e2:"Helen", e3:"Iris", e4:"Jane", e5:"Kate", e6:"Liz", e7:"Nancy", e8:"Pattie", e9:"Rebecca"));

// Store spouse relationships (bidirectional)
static final Map<String, String> spouse = new HashMap<>();

// Map from child to their parents
static final Map<String, Set<String>> parents = new HashMap<>();

// Map from parent to list of their children
static final Map<String, List<String>> children = new HashMap<>();
```

  - **Fact Processing:**

    Based on the problem description, initial facts are created using these data structures. Spouse relationships are established using spouse.put(), and parent-child relationships are constructed by calling a our defined function addParent()

```java
static {
    // Define married couples
    spouse.put(key:"Bob", value:"Helen");
    spouse.put(key:"Helen", value:"Bob");

    spouse.put(key:"Dennis", value:"Pattie");
    spouse.put(key:"Pattie", value:"Dennis");

    spouse.put(key:"Gigi", value:"Martin");
    spouse.put(key:"Martin", value:"Gigi");

    // Add known parent relationships
    // Relation #1: If one parent is known and married, derive the second parent
    addParent(child:"Bob", parentName:"Andy");
    addParent(child:"Cecil", parentName:"Bob");
    addParent(child:"Dennis", parentName:"Cecil");
    addParent(child:"Edward", parentName:"Dennis");
    addParent(child:"Felix", parentName:"Edward");

    addParent(child:"Helen", parentName:"Gigi");
    addParent(child:"Iris", parentName:"Helen");
    addParent(child:"Jane", parentName:"Iris");
    addParent(child:"Kate", parentName:"Jane");
    addParent(child:"Liz", parentName:"Kate");

    addParent(child:"Nancy", parentName:"Martin");
    addParent(child:"Oscar", parentName:"Nancy");
    addParent(child:"Pattie", parentName:"Oscar");
    addParent(child:"Quinn", parentName:"Pattie");
    addParent(child:"Rebecca", parentName:"Quinn");
}
```

- **Function helper:**
  - **addParent() :**

    Establishes a parent-child relationship and automatically infers the second parent via marriage (Relation #1).

```
// Add parent-child relation and derive second parent through marriage
static void addParent(String child, String parentName) {
    //computeIfAbsent: a special method of Map, used to ensure that a key must have
    //If there is no key in the map, the following lambda (creating an empty colle
    parents.computeIfAbsent(child, k -> new HashSet<>()).add(parentName);
    children.computeIfAbsent(parentName, k -> new ArrayList<>()).add(child);

    // Derive second parent from marriage (Relation #1)
    String spouseName = spouse.get(parentName);
    if (spouseName != null) {
        parents.get(child).add(spouseName);
        children.computeIfAbsent(spouseName, k -> new ArrayList<>()).add(child);
    }
}
```

  - **getSiblings() :** Returns a list of all siblings (individuals who share at least one parent) of the given person, excluding the person themself (Relation #2).

```
// Get siblings of a person (Relation #2)
static List<String> getSiblings(String person) {
    Set<String> parentSet = parents.get(person);

    // parentSet == null means the key "person" is not registered at all
    // parentSet.isEmpty() means the person has no known parents
    if (parentSet == null || parentSet.isEmpty()) return List.of();

    // Get the first parent
    //iterator(): returns iterator<string> object.
    //next() → gets the next element
    String firstParent = parentSet.iterator().next();
    List<String> siblings = new ArrayList<>(children.get(firstParent));
    siblings.remove(person);
    return siblings;

}
```

- **Relational Rules:**
  - **areCousins():**

    Determines whether two individuals are cousins (Relation #5), i.e., whether their parents are siblings.

```
// Check if two people are cousins (Relation #5)
static boolean areCousins(String a, String b) {
    Set<String> aParents = parents.getOrDefault(a, Set.of());
    Set<String> bParents = parents.getOrDefault(b, Set.of());

    for (String ap : aParents) {
        for (String bp : bParents) {
            if (getSiblings(ap).contains(bp)) return true;
        }
    }
    return false;
}
```

  - **areParent():**

    Determines whether two individuals are parents, noted that we designed that the position of parent and child names can't be changed

```
// Check if X is the parent of Y
static boolean areParent(String parentName, String childName) {
    return parents.getOrDefault(childName, Set.of()).contains(parentName);
}
```

Version 1: FamilyRelation1.java

We implemented the “printRelation()” function to determine whether two given names share a familial relationship. To handle relationship types not explicitly covered by existing logic, we introduced conditional checks using if-else structures and boolean gates. Specifically, we defined custom rules for:

- areSiblings()
- areBrothers()
- areSisters()

These rules utilize the helper function “getSiblings()”, combined with the “.contains()” method, to check whether the two names are siblings. If they are, we then verify the genders of both individuals. If both are male, they are identified as brothers; if both are female, they are classified as sisters. Otherwise, the relationship is reported as siblings.

```
// Print relationship between two people based on rules
static void printRelation(String a, String b) {
    if (areParent(a, b)) {
        System.out.println(a + " is the parent of " + b + ".");
    }
    else if (getSiblings(a).contains(b)) {
        // Relation #3 and #4: brothers or sisters
        if (males.contains(a) && males.contains(b)) System.out.println(a + " and " + b + " are brothers.");
        else if (females.contains(a) && females.contains(b)) System.out.println(a + " and " + b + " are sisters.");
        else System.out.println(a + " and " + b + " are siblings.");
    } else if (areCousins(a, b)) {
        System.out.println(a + " and " + b + " are cousins.");
    } else {
        System.out.println("No direct relation found between " + a + " and " + b + ".");
    }
}
```

▲printRelation()

```
// Main function to test relationships
Run | Debug | Run main | Debug main
public static void main(String[] args) {
    printRelation(a:"Helen", b:"Cecil");       //parent
    printRelation(a:"Cecil", b:"Helen");       //parent
    printRelation(a:"Cecil", b:"Iris");        //siblings
    printRelation(a:"Edward", b:"Quinn");      //brothers
    printRelation(a:"Nancy", b:"Helen");       //sisters
    printRelation(a:"Iris", b:"Oscar");        //cousins
}
```

▲main funtion

Version 2: FamilyRelation2.java

In this version, the logic for determining relationships is encapsulated in individual functions: areSiblings(), areBrothers(), and areSisters(). This modular design allows these functions to be directly invoked within the main function to test specific relationships.

```
public static boolean areSibling(String a, String b) {
    return getSiblings(a).contains(b);
}

public static boolean areBrother(String a, String b) {
    return areSibling(a,b) && males.contains(a) && males.contains(b);
}

public static boolean areSister(String a, String b) {
    return areSibling(a,b) && females.contains(a) && females.contains(b);
}
```

▲other relation rules

```
// Main function to test relationships
Run | Debug | Run main | Debug main
public static void main(String[] args) {
    System.out.println("Is Helen Cecil's parent? " + areParent(parentName:"Helen", childName:"Cecil"));        //true
    System.out.println("Is Cecil Helen's parent? " + areParent(parentName:"Cecil", childName:"Helen"));        //false

    System.out.println("Are Cecil and Iris sibling? " + areSibling(a:"Cecil", b:"Iris"));                      //true

    System.out.println("Are Edward and Quinn brothers? " + areBrother(a:"Edward", b:"Quinn"));                 //true

    System.out.println("Are Helen and Nancy sisters? " + areSister(a:"Helen", b:"Nancy"));                     //true

    System.out.println("Are Iris and Oscar cousins? " + areCousins(a:"Iris", b:"Oscar"));                      //true
}
```

▲main function

- **Standard ML:**
  Used pure functions and immutable list structures to model logic declaratively. Relationship rules were expressed through simple boolean functions, using List.exists and nested function calls. The overall design prioritizes readability and logical clarity.
  - **Define facts:**

```
(* --- Define gender facts --- *)
val males = ["Andy", "Bob", "Cecil", "Dennis", "Edward", "Felix", "Martin", "Oscar", "Quinn"];
val females = ["Gigi", "Helen", "Iris", "Jane", "Kate", "Liz", "Nancy", "Pattie", "Rebecca"];

fun is_male x = List.exists (fn y => y = x) males;
fun is_female x = List.exists (fn y => y = x) females;

(* --- Define marriages (Relation #1 support) --- *)
val marriages = [
  ("Bob", "Helen"), ("Helen", "Bob"),
  ("Dennis", "Pattie"), ("Pattie", "Dennis"),
  ("Gigi", "Martin"), ("Martin", "Gigi")
];

fun married x y = List.exists (fn (a, b) => a = x andalso b = y) marriages;

(* --- Define direct parent relationships --- *)
val parent0 = [
  ("Andy", "Bob"),
  ("Bob", "Cecil"),
  ("Cecil", "Dennis"),
  ("Dennis", "Edward"),
  ("Edward", "Felix"),
  ("Gigi", "Helen"),
  ("Helen", "Iris"),
  ("Iris", "Jane"),
  ("Jane", "Kate"),
  ("Kate", "Liz"),
  ("Martin", "Nancy"),
  ("Nancy", "Oscar"),
  ("Oscar", "Pattie"),
  ("Pattie", "Quinn"),
  ("Quinn", "Rebecca")
];
```

  - **parent x y:**
    Determines if x is a parent of y. Checks for a direct (x, y) entry in parent0. If x is married to someone who is the recorded parent, that also qualifies x as a parent (Relation #1)

```
(* --- Relation #1: Parent relationship--- *)
fun parent x y =
  List.exists (fn (a, b) => a = x andalso b = y) parent0
  orelse List.exists (fn (a, b) => married x a andalso b = y) parent0;
```

  - **sibling x y:**
    Determines if x and y share at least one parent (Relation #2). Ensures x and y are not the same person. Iterates over all individuals to check if someone is a parent of both.

```
(* --- Relation #2: Siblings --- *)
fun sibling x y =
  x <> y andalso
  List.exists (fn p => parent p x andalso parent p y) (males @ females);
```

- **brother x y:**

  Returns true if x and y are siblings and both are male (Relation #3). Uses sibling x y and is_male x, is_male y.

```
(* --- Relation #3: Brothers --- *)
fun brother x y = sibling x y andalso is_male x andalso is_male y;
```

- **sister x y:**

  Returns true if x and y are siblings and both are female (Relation #4). Uses sibling x y and is_female x, is_female y.

```
(* --- Relation #4: Sisters --- *)
fun sister x y = sibling x y andalso is_female x andalso is_female y;
```

- **cousin x y:**

  Determines if x and y are cousins (Relation #5). Checks if any parent of x has a sibling who is also a parent of y. Uses nested List.exists to express the condition cleanly.

```
(* --- Relation #5: Cousins --- *)
fun cousin x y =
  List.exists (fn w =>
    parent w x andalso
    List.exists (fn z =>
      parent z y andalso
      sibling w z
    ) (males @ females)
  ) (males @ females);
```

- **main function:**

  Relationship checks were printed using a test function for quick verification of parent, sibling, and cousin logic.

```
fun boolToString true = "true"
  | boolToString false = "false";

(* --- Testfunction --- *)
fun test msg cond =
  print (msg ^ (boolToString cond) ^ "\n");

(* --- Example test function --- *)
val _ = (
  test "Is Helen Cecil's parent? " (parent "Helen" "Cecil");

  test "Is Cecil Helen's parent? " (parent "Cecil" "Helen");

  test "Are Cecil and Iris siblings? " (sibling "Cecil" "Iris");

  test "Are Edward and Quinn brothers? " (brother "Edward" "Quinn");

  test "Are Nancy and Helen sisters? " (sister "Nancy" "Helen");

  test "Are Iris and Oscar cousins? " (cousin "Iris" "Oscar")

);
```

- **Prolog:**
  Expressed all relationships using declarative rules and base facts. Prolog's inference engine handles logical backtracking and unification automatically, allowing concise definitions for complex relationships.
  - **Define facts:**

```
% Define gender facts
male(andy). male(bob). male(cecil). male(dennis). male(edward).
male(felix). male(martin). male(oscar). male(quinn).

female(gigi). female(helen). female(iris). female(jane). female(kate).
female(liz). female(nancy). female(pattie). female(rebecca).

% Define marriages (Relation #1 support)
married(bob, helen). married(helen, bob).
married(dennis, pattie). married(pattie, dennis).
married(gigi, martin). married(martin, gigi).

% Define direct parent relationships
parent0(andy, bob).
parent0(bob, cecil).
parent0(cecil, dennis).
parent0(dennis, edward).
parent0(edward, felix).

parent0(gigi, helen).
parent0(helen, iris).
parent0(iris, jane).
parent0(jane, kate).
parent0(kate, liz).

parent0(martin, nancy).
parent0(nancy, oscar).
parent0(oscar, pattie).
parent0(pattie, quinn).
parent0(quinn, rebecca).
```

  - **parent(X, Z):**
    True if Y and Z share at least one parent and are not the same individual (Relation #2). Uses existential pattern matching on parent(X, Y) and parent(X, Z).

```
parent(X, Z) :- parent0(X, Z).
parent(Y, Z) :- married(X, Y), parent0(X, Z).
```

  - **sibling(Y, Z):**
    True if Y and Z share at least one parent and are not the same individual (Relation #2). Uses existential pattern matching on parent(X, Y) and parent(X, Z).

```
sibling(Y, Z) :-
    parent(X, Y), parent(X, Z),
    Y \= Z.
```

- **brother(X, Y):**
  True if X and Y are siblings and both are male (Relation #3). Combines sibling/2 with male/1.

```
brother(X, Y) :-
    sibling(X, Y),
    male(X),
    male(Y).
```

- **sister(X, Y):**
  True if X and Y are siblings and both are female (Relation #4). Combines sibling/2 with female/1.

```
sister(X, Y) :-
    sibling(X, Y),
    female(X),
    female(Y).
```

- **cousin(Y, Z):**
  Determines if Y and Z are cousins (Relation #5). True if the parent of Y and the parent of Z are siblings.

```
cousin(Y, Z) :-
    sibling(W, X),
    parent(W, Y),
    parent(X, Z),
    Y \= Z.
```

- **main function:**

```
main :-
    test_parent(helen, cecil),

    test_parent(cecil, helen),

    test_sibling(iris, cecil),

    test_brother(edward, quinn),

    test_sister(helen, nancy),

    test_cousins(iris, oscar),

    halt.

% "nl" means newline
test_parent(X, Y) :-
    format('Is ~w ~w''s parent? ', [X, Y]),
    ( parent(X, Y) ->
        write('true'), nl
    ;
        write('false'), nl
    ).

test_sibling(X, Y) :-
    format('Are ~w and ~w siblings? ', [X, Y]),
    ( sibling(X, Y) ->
        write('true'), nl
    ;
        write('false'), nl
    ).
```

```
test_sister(X, Y) :-
    format('Are ~w and ~w sisters? ', [X, Y]),
    ( sister(X, Y) ->
        write('true'), nl
    ;
        write('false'), nl
    ).

test_brother(X, Y) :-
    format('Are ~w and ~w brothers? ', [X, Y]),
    ( brother(X, Y) ->
        write('true'), nl
    ;
        write('false'), nl
    ).

test_cousins(X, Y) :-
    format('Are ~w and ~w cousins? ', [X, Y]),
    (cousin(X, Y) ->
        write('true'), nl
    ;
        write('false'), nl
    ).
```

Exercise 2:

- **COBOL**:

  COBOL does not use functions in the traditional sense but instead separates logic into labeled paragraphs under the **PROCEDURE DIVISION**. Below are the key functional blocks with descriptions and representative code.

  - **Index Variables & Temporary Working Fields:**
    - These counters are used for traversing various arrays (tables) in the program logic via PERFORM VARYING loops.
    - These fields temporarily store individual student records and intermediate computation results during processing.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. STUDENT-FEE-CHECK.


DATA DIVISION.           *> declare the variable use in the program
WORKING-STORAGE SECTION.  *> Define variables, arrays reside in memory retained from beginning of program to end

*> PIC 9(n): integer
*> PIC X(n): string
*> OCCURS n TIMES: array
*> PIC 99 VALUE : maximum = 99 and initialize to 1
*> V99: floating point
*> WS: Working-Storage(local variable)
01 IDX                   PIC 99 VALUE 1.
01 FIDX                  PIC 99 VALUE 1.
01 PIDX                  PIC 99 VALUE 1.
01 WS-STUDENT-ID         PIC 9(9).
01 WS-NAME               PIC X(20).
01 WS-PAYMENT-TYPE       PIC X.
01 WS-PAID-AMOUNT        PIC 9(5)V99.
01 WS-REQUIRED-AMOUNT    PIC 9(5)V99.
01 WS-SHORT-AMOUNT       PIC 9(5)V99.
01 WS-TOTAL-RECEIVED     PIC 9(7)V99 VALUE 0.
```

  - **INITIALIZE-DATA:**

    Initializes all payment types, student payment records, and student information.

```
INITIALIZE-DATA.
    *> move a to b: assign a to b
    MOVE "A"       TO FEE-TYPE(1)
    MOVE 0         TO FEE-AMOUNT(1)
    MOVE "B"       TO FEE-TYPE(2)
    MOVE 21345.00  TO FEE-AMOUNT(2)
    MOVE "C"       TO FEE-TYPE(3)
    MOVE 42690.00  TO FEE-AMOUNT(3)

    MOVE 920121005 TO PAID-STUDENT-ID(1)
    MOVE 21345.00  TO PAID-AMOUNT(1)
    MOVE 920121009 TO PAID-STUDENT-ID(2)
    MOVE 21345.00  TO PAID-AMOUNT(2)
    MOVE 920121003 TO PAID-STUDENT-ID(3)
    MOVE 42690.00  TO PAID-AMOUNT(3)
    MOVE 920121017 TO PAID-STUDENT-ID(4)
    MOVE 21345.00  TO PAID-AMOUNT(4)
    MOVE 920121012 TO PAID-STUDENT-ID(5)
    MOVE 21345.00  TO PAID-AMOUNT(5)
    MOVE 920121002 TO PAID-STUDENT-ID(6)
    MOVE 21345.00  TO PAID-AMOUNT(6)
    MOVE 920121014 TO PAID-STUDENT-ID(7)
    MOVE 15000.00  TO PAID-AMOUNT(7)
    MOVE 920121018 TO PAID-STUDENT-ID(8)
    MOVE 21345.00  TO PAID-AMOUNT(8)
    MOVE 920121011 TO PAID-STUDENT-ID(9)
    MOVE 20000.00  TO PAID-AMOUNT(9)
    MOVE 920121006 TO PAID-STUDENT-ID(10)
    MOVE 42690.00  TO PAID-AMOUNT(10)
    MOVE 920121015 TO PAID-STUDENT-ID(11)
    MOVE 21345.00  TO PAID-AMOUNT(11)
    MOVE 920121008 TO PAID-STUDENT-ID(12)
    MOVE 10000.00  TO PAID-AMOUNT(12)

    MOVE 920121001 TO INFO-STUDENT-ID(1)
    MOVE "Andy"    TO INFO-NAME(1)
    MOVE "A"       TO INFO-PAY-TYPE(1)
    MOVE 920121002 TO INFO-STUDENT-ID(2)
    MOVE "Bob"     TO INFO-NAME(2)
    MOVE "B"       TO INFO-PAY-TYPE(2)
    MOVE 920121003 TO INFO-STUDENT-ID(3)
    MOVE "Cecil"   TO INFO-NAME(3)
    MOVE "C"       TO INFO-PAY-TYPE(3)
    MOVE 920121004 TO INFO-STUDENT-ID(4)
    MOVE "Dennis"  TO INFO-NAME(4)
    MOVE "A"       TO INFO-PAY-TYPE(4)
    MOVE 920121005 TO INFO-STUDENT-ID(5)
    MOVE "Edward"  TO INFO-NAME(5)
    MOVE "B"       TO INFO-PAY-TYPE(5)
    MOVE 920121006 TO INFO-STUDENT-ID(6)
    MOVE "Felix"   TO INFO-NAME(6)
    MOVE "C"       TO INFO-PAY-TYPE(6)
    MOVE 920121007 TO INFO-STUDENT-ID(7)
    MOVE "Gigi"    TO INFO-NAME(7)
    MOVE "B"       TO INFO-PAY-TYPE(7)
    MOVE 920121008 TO INFO-STUDENT-ID(8)
    MOVE "Helen"   TO INFO-NAME(8)
    MOVE "B"       TO INFO-PAY-TYPE(8)
    MOVE 920121009 TO INFO-STUDENT-ID(9)
    MOVE "Iris"    TO INFO-NAME(9)
    MOVE "B"       TO INFO-PAY-TYPE(9)
    MOVE 920121010 TO INFO-STUDENT-ID(10)
    MOVE "Jane"    TO INFO-NAME(10)
    MOVE "A"       TO INFO-PAY-TYPE(10)
    MOVE 920121011 TO INFO-STUDENT-ID(11)
    MOVE "Kate"    TO INFO-NAME(11)
    MOVE "B"       TO INFO-PAY-TYPE(11)
    MOVE 920121012 TO INFO-STUDENT-ID(12)
    MOVE "Liz"     TO INFO-NAME(12)
    MOVE "C"       TO INFO-PAY-TYPE(12)
    MOVE 920121013 TO INFO-STUDENT-ID(13)
    MOVE "Martin"  TO INFO-NAME(13)
    MOVE "A"       TO INFO-PAY-TYPE(13)
    MOVE 920121014 TO INFO-STUDENT-ID(14)
    MOVE "Nancy"   TO INFO-NAME(14)
    MOVE "B"       TO INFO-PAY-TYPE(14)
    MOVE 920121015 TO INFO-STUDENT-ID(15)
    MOVE "Oscar"   TO INFO-NAME(15)
    MOVE "C"       TO INFO-PAY-TYPE(15)
    MOVE 920121016 TO INFO-STUDENT-ID(16)
    MOVE "Pattie"  TO INFO-NAME(16)
    MOVE "B"       TO INFO-PAY-TYPE(16)
    MOVE 920121017 TO INFO-STUDENT-ID(17)
    MOVE "Quinn"   TO INFO-NAME(17)
    MOVE "B"       TO INFO-PAY-TYPE(17)
    MOVE 920121018 TO INFO-STUDENT-ID(18)
    MOVE "Rebecca" TO INFO-NAME(18)
    MOVE "B"       TO INFO-PAY-TYPE(18).
```

  - **COMPUTE-TOTAL-RECEIVED:**

    Iterates through the payment table and calculates the total amount paid

```
COMPUTE-TOTAL-RECEIVED.
    *> for loop
    PERFORM VARYING IDX FROM 1 BY 1 UNTIL IDX > 12
        ADD PAID-AMOUNT(IDX) TO WS-TOTAL-RECEIVED
    END-PERFORM.
```

- **FIND-REQUIRED-AMOUNT:**

  Searches the **FEES-TABLE** for the fee amount matching the student's payment type.

```
FIND-REQUIRED-AMOUNT.
    PERFORM VARYING FIDX FROM 1 BY 1 UNTIL FIDX > 3
        IF WS-PAYMENT-TYPE = FEE-TYPE(FIDX)
            MOVE FEE-AMOUNT(FIDX) TO WS-REQUIRED-AMOUNT
        END-IF
    END-PERFORM.
```

- **FIND-PAID-AMOUNT:**

  Adds all matching payments for the student by Student ID.

```
FIND-PAID-AMOUNT.
    MOVE 0 TO WS-PAID-AMOUNT
    PERFORM VARYING PIDX FROM 1 BY 1 UNTIL PIDX > 12
        IF WS-STUDENT-ID = PAID-STUDENT-ID(PIDX)
            ADD PAID-AMOUNT(PIDX) TO WS-PAID-AMOUNT
        END-IF
    END-PERFORM.
```

- **LIST-UNPAID-STUDENTS:**

  Main control structure that ties all logic together: reads each student, calculates difference between required and paid, and prints if unpaid.

```
LIST-UNPAID-STUDENTS.
    PERFORM VARYING IDX FROM 1 BY 1 UNTIL IDX > 18
        MOVE INFO-STUDENT-ID(IDX) TO WS-STUDENT-ID
        MOVE INFO-NAME(IDX)       TO WS-NAME
        MOVE INFO-PAY-TYPE(IDX)   TO WS-PAYMENT-TYPE
        PERFORM FIND-REQUIRED-AMOUNT
        PERFORM FIND-PAID-AMOUNT
        COMPUTE WS-SHORT-AMOUNT = WS-REQUIRED-AMOUNT -
                                  WS-PAID-AMOUNT
        IF WS-SHORT-AMOUNT > 0
            DISPLAY WS-STUDENT-ID " " WS-NAME
                    " Unpaid by: " WS-SHORT-AMOUNT
        END-IF
    END-PERFORM.
```

- **Main function:**

```
*> Main program logic area
PROCEDURE DIVISION.
MAIN-PARA.
    PERFORM INITIALIZE-DATA
    PERFORM COMPUTE-TOTAL-RECEIVED
    DISPLAY "Total amount received: " WS-TOTAL-RECEIVED
    DISPLAY "Students with unpaid balances:"
    PERFORM LIST-UNPAID-STUDENTS
    STOP RUN.
```

- **R (Two Versions)**:

  The R version was implemented in two ways to demonstrate both controlled testing and real-world data analysis:

  1. Manual Data Version

  - Constructed mock datasets (student_main, fees, student_payment) directly using data.frame().
  - Demonstrated core operations such as:
    - merge() to join tables,
    - Arithmetic computation to calculate ShortAmount (Required – Paid),
    - Conditional filtering using subset() to identify underpaid students.

```r
student_main <- data.frame(
  StudentID = c(920121001, 920121002, 920121003, 920121004, 920121005,
                920121006, 920121007, 920121008, 920121009, 920121010,
                920121011, 920121012, 920121013, 920121014, 920121015,
                920121016, 920121017, 920121018),
  Name = c("Andy", "Bob", "Cecil", "Dennis", "Edward", "Felix", "Gigi", "Helen",
           "Iris", "Jane", "Kate", "Liz", "Martin", "Nancy", "Oscar",
           "Pattie", "Quinn", "Rebecca"),
  PaymentType = c(
    "A", "B", "C", "A", "B", "C", "B", "B", "B", "A", "B", "C",
    "A", "B", "C", "B", "B", "B"
  ),
  stringsAsFactors = FALSE
)

# Create Table 2: Fees
fees <- data.frame(
  PaymentType = c("A", "B", "C"),
  AmountRequired = c(0, 21345.00, 42690.00)
)

# Create Table 3: Student-Payment
student_payment <- data.frame(
  StudentID = c(920121005, 920121009, 920121003, 920121017, 920121012,
                920121002, 920121014, 920121018, 920121011, 920121006,
                920121015, 920121008),
  AmountPaid = c(21345.00, 21345.00, 42690.00, 21345.00, 21345.00,
                 21345.00, 15000.00, 21345.00, 20000.00, 42690.00,
                 21345.00, 10000.00)
)
```

  2. CSV Import Version

  - Loaded external data files (.csv) using read.csv() with stringsAsFactors = FALSE.
  - Cleaned numeric fields by removing comma separators using gsub() and converting them to numbers via as.numeric().
  - Renamed columns using names() to standardize field labels.
  - Merged student data with fee schedules and payment history using merge().
  - Used is.na() to detect missing payments and replaced them with 0.
  - Calculated total received using sum() and filtered underpaid records using subset().
  - Displayed results using cat() and print().

Key Code Sections in R:

1. **Load and prepare data:**

   The CSV files containing student information, fee schedules, and payment history are imported. To ensure efficient variable usage, column names are standardized by renaming them to remove any spaces present in the original files.

```r
student_main <- read.csv("HW2-PL-Data/HW2-Student-Main.csv", stringsAsFactors = FALSE) # nolint
fees <- read.csv("HW2-PL-Data/HW2-Fees.csv", stringsAsFactors = FALSE)
student_payment <- read.csv("HW2-PL-Data/HW2-Student-Payment.csv", stringsAsFactors = FALSE) # nolint
```

```r
names(student_payment)[names(student_payment) == "Amount"] <- "AmountPaid"
names(fees)[names(fees) == "Amount"] <- "AmountRequired"
```

2. **Clean and convert numeric fields:**

   Remove commas and convert fee/payment strings to numeric for computation

```r
# convert the amount string(there is a comma at thousands digit) into numbers
student_payment$AmountPaid <- as.numeric(gsub(",", "", student_payment$AmountPaid)) # nolint
fees$AmountRequired <- as.numeric(gsub(",", "", fees$AmountRequired))
```

3. **Merge all data into a unified table:**

   Join student records with fee and payment data using common keys.

```r
# Merge to include PaymentType and required Amount
# merge(x = tableA, y = tableB, by = "column_name", all.x = ..., all.y = ..., all = ...) # nolint
merged_data <- merge(student_main, fees, by = "Payment.Type")
merged_data <- merge(merged_data, student_payment, by = "Student.ID", all.x = TRUE) # nolint
```

4. **Replace missing payment data and compute shortfalls:**

   Ensure no payment is treated as 0, and compute how much each student underpaid.

```r
# Replace NA in AmountPaid with 0 (means no payment)
# $: is the way R selects columns
merged_data$AmountPaid[is.na(merged_data$AmountPaid)] <- 0
```

```r
# Computation #2: Students who underpaid
# create a new column ShortAmount to find out how much each student is short
merged_data$ShortAmount <- merged_data$AmountRequired - merged_data$AmountPaid
```

5. **Filter and print results:**

   After proper computation, we get:

   - Total amount recieved
   - Students who underpaid

```r
# Computation #1: Total amount received
total_received <- sum(merged_data$AmountPaid)
cat("Total amount received before due:", total_received, "\n\n")
```

▲compute and display the question1

```r
# subset(x, subset, select), x is the data frame, subset is a logical condition, select is the columns to keep # nolint
underpaid <- subset(merged_data, ShortAmount > 0, select = c("Student.ID", "Name", "ShortAmount")) # nolint
# cat() is a function in R that outputs text to the console, similar to print,
# but more concise, without the extra quotes or print constructs.
cat("Students who did not pay the required fees with the short amount:\n")
print(underpaid, row.names = FALSE)
```

▲select data column and display the question2

IV. Test Run Results

4.1 Programming Exercise #1 – Family Relationship Evaluation

These are the logic questions designed to evaluate the relationship-detection programs implemented in various languages:

1. Is Helen Cecil's parents?
2. Is Cecil Helen's parents?
3. Are Cecil and Iris siblings?
4. Are Edward and Quinn brothers?
5. Are Nancy and Helen sisters?
6. Are Iris and Oscar cousins?

4.1.2 Language Implementations and Results

The output from each programming language is presented under separate subheadings. It is important to note that the order of names in parent-child relationships is fixed (i.e., the parent must appear first), whereas the order of names in other relationships—such as siblings, brothers, sisters, and cousins—does not affect the result:

(a) Java - version1

```
Is Helen Cecil's parent? true
Is Cecil Helen's parent? false
Are Cecil and Iris sibling? true
Are Edward and Quinn brothers? true
Are Helen and Nancy sisters? true
Are Iris and Oscar cousins? true
```

(b) Java - version2

```
Helen is the parent of Cecil.
No direct relation found between Cecil and Helen.
Cecil and Iris are siblings.
Edward and Quinn are brothers.
Nancy and Helen are sisters.
Iris and Oscar are cousins.
```

(c) ML

```
Output
> val males = ["Andy", "Bob", "Cecil", "Dennis", "Edward", "Felix", "Martin", "Oscar", "Quinn"]: string list;
> val females = ["Gigi", "Helen", "Iris", "Jane", "Kate", "Liz", "Nancy", "Pattie", "Rebecca"]: string list;
> val is_male = fn: string → bool;
> val is_female = fn: string → bool;
> val marriages = [("Bob", "Helen"), ("Helen", "Bob"), ("Dennis", "Pattie"), ("Pattie", "Dennis"), ("Gigi", "Martin"), ("Martin", "Gigi")]: (string * string) list;
> val married = fn: string → string → bool;
> val parent0 = [("Andy", "Bob"), ("Bob", "Cecil"), ("Cecil", "Dennis"), ("Dennis", "Edward"), ("Edward", "Felix"), ("Gig…", "…"), ("…", …), …]: (string * string) list;
> val parent = fn: string → string → bool;
> val sibling = fn: string → string → bool;
> val brother = fn: string → string → bool;
> val sister = fn: string → string → bool;
> val cousin = fn: string → string → bool;
> val boolToString = fn: bool → string;
> val test = fn: string → bool → unit;
Printed: Is Helen Cecil's parent? true
Printed: Is Cecil Helen's parent? false
Printed: Are Cecil and Iris siblings? true
Printed: Are Edward and Quinn brothers? true
Printed: Are Nancy and Helen sisters? true
Printed: Are Iris and Oscar cousins? true
```

(d) Prolog

```
Is helen cecil's parent? true
Is cecil helen's parent? false
Are iris and cecil siblings? true
Are edward and quinn brothers? true
Are helen and nancy sisters? true
Are iris and oscar cousins? true
```

4.1.3 Summary of Result:

Regardless of the programming language used, the problem was solved consistently across all implementations, and the results matched the expected correct answers.

Language	**Q1**	**Q2**	**Q3**	**Q4**	**Q5**	**Q6**
Java	true	false	true	true	true	true
ML	true	false	true	true	true	true
Prolog	true	false	true	true	true	true

4.2 Programming Exercise #2 – Target Value Computation

This part describes results from Exercise #2 using the provided dataset folder HW2-PL-Data.

We can present results by language:

4.2.1 Language Implementations and Results

(a) R

The result of version1 is as same as version2

```
Total amount received before due: 279795

Students who did not pay the required fees with the short amount:
 StudentID   Name ShortAmount
 920121007   Gigi       21345
 920121008  Helen       11345
 920121011   Kate        1345
 920121012    Liz       21345
 920121014  Nancy        6345
 920121015  Oscar       21345
 920121016 Pattie       21345
```

(b) COBOL

```
Total amount received: 0279795.00
Students with unpaid balances:
920121007 Gigi                 Unpaid by: 21345.00
920121008 Helen                Unpaid by: 11345.00
920121011 Kate                 Unpaid by: 01345.00
920121012 Liz                  Unpaid by: 21345.00
920121014 Nancy                Unpaid by: 06345.00
920121015 Oscar                Unpaid by: 21345.00
920121016 Pattie               Unpaid by: 21345.00
```

4.2.2 Summary of Result:

- **Total amount received before the due date:**

Language	Target Value
R	237.54
COBOL	237.54

- **Students with outstanding balances (underpaid fees):**

Student ID	Name	Short Amount
920121007	Gigi	21,345
920121008	Helen	11,345
920121011	Kate	1,345
920121012	Liz	21,345
920121014	Nancy	6,345
920121015	Oscar	21,345
920121016	Pattie	21,345

V. Discussion

1. Cross-Language Practice and Conceptual Understanding

In this assignment, we practiced implementing logic-based and data-processing tasks across five different programming languages: Java, ML, Prolog, COBOL, and R. Each language required us to adapt our thinking and syntax to meet the problem requirements, reinforcing the importance of both language semantics and core programming concepts.

For **Exercise #1**, which involved defining family relationships based on logical rules, we observed that **Prolog** was especially suited for this kind of task due to its built-in support for logical inference and pattern matching. On the other hand, **Java** and **ML** required us to manually implement relationship inference logic through data structures, functions, and iterations which was more complex but offered greater control.

2. Language-Specific Observations

- **Java**: Provided a structured and object-oriented approach, but required more tedious implementation for relationship checking.
- **ML**: Demonstrated functional programming thinking through use of high-order functions like List.exists.
- **Prolog**: Allowed us to define facts and rules very naturally. The querying system made it easier to answer relationship questions like "Are Liz and Rebecca cousins?"

For **Exercise #2**, the task shifted to data computation. The difference in paradigms became more pronounced:

- **COBOL**: Was difficult but reliable for fixed-format, line-by-line data processing. It required tedious declarations and manual looping constructs (PERFORM VARYING) for even basic tasks, such as summing payments or matching records. While powerful for legacy business applications, it made logic structuring and debugging more time-consuming compared to modern languages.
- **R**: Was highly expressive for statistical and tabular data processing. It enabled concise operations such as data merging (joins), aggregate computations (like total payments), and conditional filtering (e.g., underpaid students) with minimal code. The use of data.frame, merge, and vectorized operations made it ideal for scenarios involving structured datasets.

3. Common Challenges

Across both exercises, some common challenges emerged:

- Designing reusable logic (especially in Java and ML) for checking relationships.
- Understanding each language's input/output mechanism, particularly in Prolog and COBOL.
- Ensuring that computation logic matched expected output formats (e.g., listing short payments, verifying relationships).
- Debugging logic errors when the rule structure or conditions were incorrect (especially in Prolog and ML).

4. Reflection on Multi-Language Programming

This assignment highlighted the strengths and limitations of each language. While modern languages like R and Java are well-supported and versatile, older languages like COBOL remain relevant in legacy data systems. Prolog stood out as a powerful tool for declarative logic problems.

The practice improved our ability to **translate the same problem across different approaches**, reinforcing both algorithmic thinking and adaptability. It also demonstrated that while syntax differs, **the core programming logic remains transferable** once understood the ideals for each problem.

VI. Appendix(complete code)

FamilyRelation1.java :

```java
import java.util.*;

// Main class to represent family relationships
public class FamilyRelations1 {
    // Define sets of male and female names
    //Set<String> is a data structure that does not repeat elements
    static Set<String> males = new HashSet<>(List.of("Andy", "Bob", "Cecil", "Dennis", "Edward", "Felix", "Martin", "Oscar", "Quinn"));
    static Set<String> females = new HashSet<>(List.of("Gigi", "Helen", "Iris", "Jane", "Kate", "Liz", "Nancy", "Pattie", "Rebecca"));

    // Store spouse relationships (bidirectional)
    static final Map<String, String> spouse = new HashMap<>();

    // Map from child to their parents
    static final Map<String, Set<String>> parents = new HashMap<>();

    // Map from parent to list of their children
    static final Map<String, List<String>> children = new HashMap<>();

    static {
        // Define married couples
        spouse.put("Bob", "Helen");
        spouse.put("Helen", "Bob");

        spouse.put("Dennis", "Pattie");
        spouse.put("Pattie", "Dennis");

        spouse.put("Gigi", "Martin");
        spouse.put("Martin", "Gigi");

        // Add known parent relationships
        // Relation #1: If one parent is known and married, derive the second parent
        addParent("Bob", "Andy");
        addParent("Cecil", "Bob");
        addParent("Dennis", "Cecil");
        addParent("Edward", "Dennis");
        addParent("Felix", "Edward");

        addParent("Helen", "Gigi");
        addParent("Iris", "Helen");
        addParent("Jane", "Iris");
        addParent("Kate", "Jane");
        addParent("Liz", "Kate");

        addParent("Nancy", "Martin");
        addParent("Oscar", "Nancy");
        addParent("Pattie", "Oscar");
        addParent("Quinn", "Pattie");
        addParent("Rebecca", "Quinn");
    }

    // Add parent-child relation and derive second parent through marriage
    static void addParent(String child, String parentName) {
        //computeIfAbsent: a special method of Map, used to ensure that a key must have a corresponding value
        //If there is no key in the map, the following lambda (creating an empty collection) is executed to put it in.
        parents.computeIfAbsent(child, k -> new HashSet<>()).add(parentName);
        children.computeIfAbsent(parentName, k -> new ArrayList<>()).add(child);

        // Derive second parent from marriage (Relation #1)
        String spouseName = spouse.get(parentName);
        if (spouseName != null) {
            parents.get(child).add(spouseName);
            children.computeIfAbsent(spouseName, k -> new ArrayList<>()).add(child);
        }
    }
```

```java
    static List<String> getSiblings(String person) {
        Set<String> parentSet = parents.get(person);

        // parentSet == null means the key "person" is not registered at all
        // parentSet.isEmpty() means the person has no known parents
        if (parentSet == null || parentSet.isEmpty()) return List.of();

        // Get the first parent
        //iterator(): returns iterator<string> object.
        //next() → gets the next element
        String firstParent = parentSet.iterator().next();
        List<String> siblings = new ArrayList<>(children.get(firstParent));
        siblings.remove(person);
        return siblings;

    }

    // Check if two people are cousins (Relation #5)
    static boolean areCousins(String a, String b) {
        Set<String> aParents = parents.getOrDefault(a, Set.of());
        Set<String> bParents = parents.getOrDefault(b, Set.of());

        for (String ap : aParents) {
            for (String bp : bParents) {
                if (getSiblings(ap).contains(bp)) return true;
            }
        }
        return false;
    }

    // Check if X is the parent of Y
    static boolean isParent(String parentName, String childName) {
        return parents.getOrDefault(childName, Set.of()).contains(parentName);
    }

    // Print relationship between two people based on rules
    static void printRelation(String a, String b) {
        if (isParent(a, b)) {
            System.out.println(a + " is the parent of " + b + ".");
        }
        else if (getSiblings(a).contains(b)) {
            // Relation #3 and #4: brothers or sisters
            if (males.contains(a) && males.contains(b)) System.out.println(a + " and " + b + " are brothers.");
            else if (females.contains(a) && females.contains(b)) System.out.println(a + " and " + b + " are sisters.");
            else System.out.println(a + " and " + b + " are siblings.");
        } else if (areCousins(a, b)) {
            System.out.println(a + " and " + b + " are cousins.");
        } else {
            System.out.println("No direct relation found between " + a + " and " + b + ".");
        }
    }

    // Main function to test relationships
    public static void main(String[] args) {
        printRelation("Iris", "Jane");       //parent
        printRelation("Jane", "Iris");       //no relation
        printRelation("Iris", "Kate");       //no relation
        printRelation("Iris", "Oscar");      //cousins
        printRelation("Oscar", "Iris");      //cousins
        printRelation("Oscar", "Cecil");     //cousins
        printRelation("Dennis", "Jane");     //cousins
        printRelation("Felix", "Rebecca");   //cousins
        printRelation("Cecil", "Iris");      //siblings
        printRelation("Edward", "Quinn");    //brothers
        printRelation("Nancy", "Helen");     //sisters
    }
}
```

FamilyRelation2.java :

```java
// Import all classes and interfaces from the java.util package of the Java Standard Library
import java.util.*;

// Main class to represent family relationships
public class FamilyRelations2 {
    // Define sets of male and female names
    //Set<String> is a data structure that does not repeat elements
    static Set<String> males = new HashSet<>(List.of("Andy", "Bob", "Cecil", "Dennis", "Edward", "Felix", "Martin", "Oscar", "Quinn"));
    static Set<String> females = new HashSet<>(List.of("Gigi", "Helen", "Iris", "Jane", "Kate", "Liz", "Nancy", "Pattie", "Rebecca"));

    // Store spouse relationships (bidirectional)
    static final Map<String, String> spouse = new HashMap<>();

    // Map from child to their parents
    static final Map<String, Set<String>> parents = new HashMap<>();

    // Map from parent to list of their children
    static final Map<String, List<String>> children = new HashMap<>();

    static {
        // Define married couples
        spouse.put("Bob", "Helen");
        spouse.put("Helen", "Bob");

        spouse.put("Dennis", "Pattie");
        spouse.put("Pattie", "Dennis");

        spouse.put("Gigi", "Martin");
        spouse.put("Martin", "Gigi");

        // Add known parent relationships
        // Relation #1: If one parent is known and married, derive the second parent
        addParent("Bob", "Andy");
        addParent("Cecil", "Bob");
        addParent("Dennis", "Cecil");
        addParent("Edward", "Dennis");
        addParent("Felix", "Edward");

        addParent("Helen", "Gigi");
        addParent("Iris", "Helen");
        addParent("Jane", "Iris");
        addParent("Kate", "Jane");
        addParent("Liz", "Kate");

        addParent("Nancy", "Martin");
        addParent("Oscar", "Nancy");
        addParent("Pattie", "Oscar");
        addParent("Quinn", "Pattie");
        addParent("Rebecca", "Quinn");
    }

    // Add parent-child relation and derive second parent through marriage
    static void addParent(String child, String parentName) {
        //computeIfAbsent: a special method of Map, used to ensure that a key must have a corresponding value
        //If there is no key in the map, the following lambda (creating an empty collection) is executed to put it in.
        parents.computeIfAbsent(child, k -> new HashSet<>()).add(parentName);
        children.computeIfAbsent(parentName, k -> new ArrayList<>()).add(child);

        // Derive second parent from marriage (Relation #1)
        String spouseName = spouse.get(parentName);
        if (spouseName != null) {
            parents.get(child).add(spouseName);
            children.computeIfAbsent(spouseName, k -> new ArrayList<>()).add(child);
        }
    }
```

```java
    // Get siblings of a person (Relation #2)
    static List<String> getSiblings(String person) {
        Set<String> parentSet = parents.get(person);

        // parentSet == null means the key "person" is not registered at all
        // parentSet.isEmpty() means the person has no known parents
        if (parentSet == null || parentSet.isEmpty()) return List.of();

        // Get the first parent
        //iterator(): returns iterator<string> object.
        //next() → gets the next element
        String firstParent = parentSet.iterator().next();
        List<String> siblings = new ArrayList<>(children.get(firstParent));
        siblings.remove(person);
        return siblings;
    }

    // Check if two people are cousins (Relation #5)
    public static boolean areCousins(String a, String b) {
        Set<String> aParents = parents.getOrDefault(a, Set.of());
        Set<String> bParents = parents.getOrDefault(b, Set.of());

        for (String ap : aParents) {
            for (String bp : bParents) {
                if (getSiblings(ap).contains(bp)) return true;
            }
        }
        return false;
    }

    // Check if X is the parent of Y
    public static boolean areParent(String parentName, String childName) {
        return parents.getOrDefault(childName, Set.of()).contains(parentName);
    }

    public static boolean areSibling(String a, String b) {
        return getSiblings(a).contains(b);
    }

    public static boolean areBrother(String a, String b) {
        return areSibling(a,b) && males.contains(a) && males.contains(b);
    }

    public static boolean areSister(String a, String b) {
        return areSibling(a,b) && females.contains(a) && females.contains(b);
    }

    // Main function to test relationships
    public static void main(String[] args) {
        System.out.println(areParent("Iris", "Jane"));           //true
        System.out.println(areParent("Jane", "Iris"));           //false
        System.out.println(areParent("Iris", "Kate"));           //false

        System.out.println(areCousins("Iris", "Oscar"));                            //true
        System.out.println(areCousins("Oscar", "Iris"));                            //true
        System.out.println(areCousins("Oscar", "Cecil"));                           //true
        System.out.println(areCousins("Dennis", "Jane"));                           //true
        System.out.println(areCousins("Felix", "Rebecca"));                         //true

        System.out.println(areSibling("Cecil", "Iris"));                            //true
        System.out.println(areSibling("Edward", "Quinn"));                          //true
        System.out.println(areBrother("Edward", "Quinn"));                          //true
        System.out.println(areSister("Nancy", "Helen"));                            //true
    }
}
```

ML :

```
(* --- Define gender facts --- *)
val males = ["Andy", "Bob", "Cecil", "Dennis", "Edward", "Felix", "Martin", "Oscar", "Quinn"];
val females = ["Gigi", "Helen", "Iris", "Jane", "Kate", "Liz", "Nancy", "Pattie", "Rebecca"];

fun is_male x = List.exists (fn y => y = x) males;
fun is_female x = List.exists (fn y => y = x) females;

(* --- Define marriages (Relation #1 support) --- *)
val marriages = [
  ("Bob", "Helen"), ("Helen", "Bob"),
  ("Dennis", "Pattie"), ("Pattie", "Dennis"),
  ("Gigi", "Martin"), ("Martin", "Gigi")
];

fun married x y = List.exists (fn (a, b) => a = x andalso b = y) marriages;

(* --- Define direct parent relationships --- *)
val parent0 = [
  ("Andy", "Bob"),
  ("Bob", "Cecil"),
  ("Cecil", "Dennis"),
  ("Dennis", "Edward"),
  ("Edward", "Felix"),
  ("Gigi", "Helen"),
  ("Helen", "Iris"),
  ("Iris", "Jane"),
  ("Jane", "Kate"),
  ("Kate", "Liz"),
  ("Martin", "Nancy"),
  ("Nancy", "Oscar"),
  ("Oscar", "Pattie"),
  ("Pattie", "Quinn"),
  ("Quinn", "Rebecca")
];

(* --- Relation #1: Parent relationship--- *)
fun parent x y =
  List.exists (fn (a, b) => a = x andalso b = y) parent0
  orelse List.exists (fn (a, b) => married x a andalso b = y) parent0;

(* --- Relation #2: Siblings --- *)
fun sibling x y =
  x <> y andalso
  List.exists (fn p => parent p x andalso parent p y) (males @ females);

(* --- Relation #3: Brothers --- *)
fun brother x y = sibling x y andalso is_male x andalso is_male y;

(* --- Relation #4: Sisters --- *)
fun sister x y = sibling x y andalso is_female x andalso is_female y;

(* --- Relation #5: Cousins --- *)
fun cousin x y =
  List.exists (fn w =>
    parent w x andalso
    List.exists (fn z =>
      parent z y andalso
      sibling w z
    ) (males @ females)
  ) (males @ females);

fun boolToString true = "true"
  | boolToString false = "false";

(* --- Testfunction --- *)
fun test msg cond =
  print (msg ^ (boolToString cond) ^ "\n");

(* --- Example test function --- *)
val _ = (
  test "Is Iris the parent of Jane? " (parent "Iris" "Jane");
  test "Is Jane the parent of Iris? " (parent "Jane" "Iris");
  test "Is Iris the parent of Kate? " (parent "Iris" "Kate");

  test "Are Iris and Oscar cousins? " (cousin "Iris" "Oscar");
  test "Are Oscar and Iris cousins? " (cousin "Oscar" "Iris");
  test "Are Oscar and Cecil cousins? " (cousin "Oscar" "Cecil");
  test "Are Dennis and Jane cousins? " (cousin "Dennis" "Jane");
  test "Are Felix and Rebecca cousins? " (cousin "Felix" "Rebecca");

  test "Are Cecil and Iris siblings? " (sibling "Cecil" "Iris");
  test "Are Edward and Quinn siblings? " (sibling "Edward" "Quinn");
  test "Are Edward and Quinn brothers? " (brother "Edward" "Quinn");
  test "Are Nancy and Helen sisters? " (sister "Nancy" "Helen")
);
```

COBOL:

```
            MOVE 920121013 TO INFO-STUDENT-ID(13)
            MOVE "Martin"  TO INFO-NAME(13)
            MOVE "A"       TO INFO-PAY-TYPE(13)
            MOVE 920121014 TO INFO-STUDENT-ID(14)
            MOVE "Nancy"   TO INFO-NAME(14)
            MOVE "B"       TO INFO-PAY-TYPE(14)
            MOVE 920121015 TO INFO-STUDENT-ID(15)
            MOVE "Oscar"   TO INFO-NAME(15)
            MOVE "C"       TO INFO-PAY-TYPE(15)
            MOVE 920121016 TO INFO-STUDENT-ID(16)
            MOVE "Pattie"  TO INFO-NAME(16)
            MOVE "B"       TO INFO-PAY-TYPE(16)
            MOVE 920121017 TO INFO-STUDENT-ID(17)
            MOVE "Quinn"   TO INFO-NAME(17)
            MOVE "B"       TO INFO-PAY-TYPE(17)
            MOVE 920121018 TO INFO-STUDENT-ID(18)
            MOVE "Rebecca" TO INFO-NAME(18)
            MOVE "B"       TO INFO-PAY-TYPE(18).

        COMPUTE-TOTAL-RECEIVED.
            *> for loop
            PERFORM VARYING IDX FROM 1 BY 1 UNTIL IDX > 12
                ADD PAID-AMOUNT(IDX) TO WS-TOTAL-RECEIVED
            END-PERFORM.

        LIST-UNPAID-STUDENTS.
            PERFORM VARYING IDX FROM 1 BY 1 UNTIL IDX > 18
                MOVE INFO-STUDENT-ID(IDX) TO WS-STUDENT-ID
                MOVE INFO-NAME(IDX)       TO WS-NAME
                MOVE INFO-PAY-TYPE(IDX)   TO WS-PAYMENT-TYPE
                PERFORM FIND-REQUIRED-AMOUNT
                PERFORM FIND-PAID-AMOUNT
                COMPUTE WS-SHORT-AMOUNT = WS-REQUIRED-AMOUNT -
                                          WS-PAID-AMOUNT
                IF WS-SHORT-AMOUNT > 0
                    DISPLAY WS-STUDENT-ID " " WS-NAME
                            " Unpaid by: " WS-SHORT-AMOUNT
                END-IF
            END-PERFORM.


        FIND-REQUIRED-AMOUNT.
            PERFORM VARYING FIDX FROM 1 BY 1 UNTIL FIDX > 3
                IF WS-PAYMENT-TYPE = FEE-TYPE(FIDX)
                    MOVE FEE-AMOUNT(FIDX) TO WS-REQUIRED-AMOUNT
                END-IF
            END-PERFORM.

        FIND-PAID-AMOUNT.
            MOVE 0 TO WS-PAID-AMOUNT
            PERFORM VARYING PIDX FROM 1 BY 1 UNTIL PIDX > 12
                IF WS-STUDENT-ID = PAID-STUDENT-ID(PIDX)
                    ADD PAID-AMOUNT(PIDX) TO WS-PAID-AMOUNT
                END-IF
            END-PERFORM.

```

```
            MOVE 920121005 TO PAID-STUDENT-ID(1)
            MOVE 21345.00  TO PAID-AMOUNT(1)
            MOVE 920121009 TO PAID-STUDENT-ID(2)
            MOVE 21345.00  TO PAID-AMOUNT(2)
            MOVE 920121003 TO PAID-STUDENT-ID(3)
            MOVE 42690.00  TO PAID-AMOUNT(3)
            MOVE 920121017 TO PAID-STUDENT-ID(4)
            MOVE 21345.00  TO PAID-AMOUNT(4)
            MOVE 920121012 TO PAID-STUDENT-ID(5)
            MOVE 21345.00  TO PAID-AMOUNT(5)
            MOVE 920121002 TO PAID-STUDENT-ID(6)
            MOVE 21345.00  TO PAID-AMOUNT(6)
            MOVE 920121014 TO PAID-STUDENT-ID(7)
            MOVE 15000.00  TO PAID-AMOUNT(7)
            MOVE 920121018 TO PAID-STUDENT-ID(8)
            MOVE 21345.00  TO PAID-AMOUNT(8)
            MOVE 920121011 TO PAID-STUDENT-ID(9)
            MOVE 20000.00  TO PAID-AMOUNT(9)
            MOVE 920121006 TO PAID-STUDENT-ID(10)
            MOVE 42690.00  TO PAID-AMOUNT(10)
            MOVE 920121015 TO PAID-STUDENT-ID(11)
            MOVE 21345.00  TO PAID-AMOUNT(11)
            MOVE 920121008 TO PAID-STUDENT-ID(12)
            MOVE 10000.00  TO PAID-AMOUNT(12)

            MOVE 920121001 TO INFO-STUDENT-ID(1)
            MOVE "Andy"    TO INFO-NAME(1)
            MOVE "A"       TO INFO-PAY-TYPE(1)
            MOVE 920121002 TO INFO-STUDENT-ID(2)
            MOVE "Bob"     TO INFO-NAME(2)
            MOVE "B"       TO INFO-PAY-TYPE(2)
            MOVE 920121003 TO INFO-STUDENT-ID(3)
            MOVE "Cecil"   TO INFO-NAME(3)
            MOVE "C"       TO INFO-PAY-TYPE(3)
            MOVE 920121004 TO INFO-STUDENT-ID(4)
            MOVE "Dennis"  TO INFO-NAME(4)
            MOVE "A"       TO INFO-PAY-TYPE(4)
            MOVE 920121005 TO INFO-STUDENT-ID(5)
            MOVE "Edward"  TO INFO-NAME(5)
            MOVE "B"       TO INFO-PAY-TYPE(5)
            MOVE 920121006 TO INFO-STUDENT-ID(6)
            MOVE "Felix"   TO INFO-NAME(6)
            MOVE "C"       TO INFO-PAY-TYPE(6)
            MOVE 920121007 TO INFO-STUDENT-ID(7)
            MOVE "Gigi"    TO INFO-NAME(7)
            MOVE "B"       TO INFO-PAY-TYPE(7)
            MOVE 920121008 TO INFO-STUDENT-ID(8)
            MOVE "Helen"   TO INFO-NAME(8)
            MOVE "B"       TO INFO-PAY-TYPE(8)
            MOVE 920121009 TO INFO-STUDENT-ID(9)
            MOVE "Iris"    TO INFO-NAME(9)
            MOVE "B"       TO INFO-PAY-TYPE(9)
            MOVE 920121010 TO INFO-STUDENT-ID(10)
            MOVE "Jane"    TO INFO-NAME(10)
            MOVE "A"       TO INFO-PAY-TYPE(10)
            MOVE 920121011 TO INFO-STUDENT-ID(11)
            MOVE "Kate"    TO INFO-NAME(11)
            MOVE "B"       TO INFO-PAY-TYPE(11)
            MOVE 920121012 TO INFO-STUDENT-ID(12)
            MOVE "Liz"     TO INFO-NAME(12)
            MOVE "C"       TO INFO-PAY-TYPE(12)
```

```
    IDENTIFICATION DIVISION.
    PROGRAM-ID. STUDENT-FEE-CHECK.


    DATA DIVISION.          => declare the variable use in the program
    WORKING-STORAGE SECTION.  => Define variables, arrays reside in memory retained from beginning of program to end

    => PIC 9(n): integer
    => PIC X(n): string
    => OCCURS n TIMES: array
    => PIC 99 VALUE : maximum = 99 and initialize to 1
    => V99: floating point
    => WS: Working-Storage(local variable)
    01 IDX                  PIC 99 VALUE 1.
    01 FIDX                 PIC 99 VALUE 1.
    01 PIDX                 PIC 99 VALUE 1.
    01 WS-STUDENT-ID        PIC 9(9).
    01 WS-NAME              PIC X(20).
    01 WS-PAYMENT-TYPE      PIC X.
    01 WS-PAID-AMOUNT       PIC 9(5)V99.
    01 WS-REQUIRED-AMOUNT   PIC 9(5)V99.
    01 WS-SHORT-AMOUNT      PIC 9(5)V99.
    01 WS-TOTAL-RECEIVED    PIC 9(7)V99 VALUE 0.

    01 STUDENT-PAID-TABLE.
        05 STUDENT-PAID-ENTRY OCCURS 12 TIMES.
            10 PAID-STUDENT-ID   PIC 9(9).
            10 PAID-AMOUNT       PIC 9(5)V99.

    01 STUDENT-INFO-TABLE.
        05 STUDENT-INFO-ENTRY OCCURS 18 TIMES.
            10 INFO-STUDENT-ID   PIC 9(9).
            10 INFO-NAME         PIC X(20).
            10 INFO-PAY-TYPE     PIC X.

    01 FEES-TABLE.
        05 FEE-ENTRY OCCURS 3 TIMES.
            10 FEE-TYPE          PIC X.
            10 FEE-AMOUNT        PIC 9(5)V99.

    => Main program logic area
    PROCEDURE DIVISION.
    MAIN-PARA.
        PERFORM INITIALIZE-DATA
        PERFORM COMPUTE-TOTAL-RECEIVED
        DISPLAY "Total amount received: " WS-TOTAL-RECEIVED
        DISPLAY "Students with unpaid balances:"
        PERFORM LIST-UNPAID-STUDENTS
        STOP RUN.

    INITIALIZE-DATA.
        => move a to b: assign a to b
        MOVE "A"        TO FEE-TYPE(1)
        MOVE 0          TO FEE-AMOUNT(1)
        MOVE "B"        TO FEE-TYPE(2)
        MOVE 21345.00   TO FEE-AMOUNT(2)
        MOVE "C"        TO FEE-TYPE(3)
        MOVE 42690.00   TO FEE-AMOUNT(3)

        MOVE 920121005 TO PAID-STUDENT-ID(1)
```

R:(version1)

```r
# Create Table 1: Student-Main
# <-: meaning assignment operator in R
# c(): concatenate function to create vectors
# stringsAsFactors: Don't convert the string into factor, keep the character
student_main <- data.frame(
  StudentID = c(920121001, 920121002, 920121003, 920121004, 920121005,
                920121006, 920121007, 920121008, 920121009, 920121010,
                920121011, 920121012, 920121013, 920121014, 920121015,
                920121016, 920121017, 920121018),
  Name = c("Andy", "Bob", "Cecil", "Dennis", "Edward", "Felix", "Gigi", "Helen",
           "Iris", "Jane", "Kate", "Liz", "Martin", "Nancy", "Oscar",
           "Pattie", "Quinn", "Rebecca"),
  PaymentType = c(
    "A", "B", "C", "A", "B", "C", "B", "B", "B", "A", "B", "C",
    "A", "B", "C", "B", "B", "B"
  ),
  stringsAsFactors = FALSE
)

# Create Table 2: Fees
fees <- data.frame(
  PaymentType = c("A", "B", "C"),
  AmountRequired = c(0, 21345.00, 42690.00)
)

# Create Table 3: Student-Payment
student_payment <- data.frame(
  StudentID = c(920121005, 920121009, 920121003, 920121017, 920121012,
                920121002, 920121014, 920121018, 920121011, 920121006,
                920121015, 920121008),
  AmountPaid = c(21345.00, 21345.00, 42690.00, 21345.00, 21345.00,
                 21345.00, 15000.00, 21345.00, 20000.00, 42690.00,
                 21345.00, 10000.00)
)

```

```r
# Merge to include PaymentType and required Amount
# merge(x = tableA, y = tableB, by = "column_name", all.x =, all.y =, all =
merged_data <- merge(
  student_main, fees,
  by = "PaymentType"
)
merged_data <- merge(
  merged_data, student_payment,
  by = "StudentID", all.x = TRUE
)

# Replace NA in AmountPaid with 0 (means no payment)
# $: is the way R selects columns
merged_data$AmountPaid[is.na(merged_data$AmountPaid)] <- 0

# Computation #1: Total amount received
total_received <- sum(merged_data$AmountPaid)
cat("Total amount received before due:", total_received, "\n\n")

# Computation #2: Students who underpaid
# create a new column ShortAmount to find out how much each student is short
# subset(x, subset, select) # nolint: commented_code_linter.
# x is data frame, subset is a logical condition, select is the columns to keep
merged_data$ShortAmount <- merged_data$AmountRequired - merged_data$AmountPaid
underpaid <- subset(
  merged_data,
  ShortAmount > 0,
  select = c("StudentID", "Name", "ShortAmount")
)

# cat() is a function in R that outputs text to the console, similar to print,
# but more concise, without the extra quotes or print constructs.
cat("Students who did not pay the required fees with the short amount:\n")
print(underpaid, row.names = FALSE)
```

R:(version2)

```r
# Create Table 1: Student-Main
# <-: meaning assignment operator in R
# c(): concatenate function to create vectors
# stringsAsFactors: Do Not automatically convert the string into factor, keep the character # nolint
# read CSV files
student_main <- read.csv("HW2-PL-Data/HW2-Student-Main.csv", stringsAsFactors = FALSE) # nolint
fees <- read.csv("HW2-PL-Data/HW2-Fees.csv", stringsAsFactors = FALSE)
student_payment <- read.csv("HW2-PL-Data/HW2-Student-Payment.csv", stringsAsFactors = FALSE) # nolint

names(student_payment)[names(student_payment) == "Amount"] <- "AmountPaid"
names(fees)[names(fees) == "Amount"] <- "AmountRequired"

# convert the amount string(there is a comma at thousands digit) into numbers
student_payment$AmountPaid <- as.numeric(gsub(",", "", student_payment$AmountPaid)) # nolint
fees$AmountRequired <- as.numeric(gsub(",", "", fees$AmountRequired))

# Merge to include PaymentType and required Amount
# merge(x = tableA, y = tableB, by = "column_name", all.x = ..., all.y = ..., all = ...) # nolint
merged_data <- merge(student_main, fees, by = "Payment.Type")
merged_data <- merge(merged_data, student_payment, by = "Student.ID", all.x = TRUE) # nolint

# Replace NA in AmountPaid with 0 (means no payment)
# $: is the way R selects columns
merged_data$AmountPaid[is.na(merged_data$AmountPaid)] <- 0

# Computation #1: Total amount received
total_received <- sum(merged_data$AmountPaid)
cat("Total amount received before due:", total_received, "\n\n")

# Computation #2: Students who underpaid
# create a new column ShortAmount to find out how much each student is short
# subset(x, subset, select), x is the data frame, subset is a logical condition, select is the columns to keep # nolint
merged_data$ShortAmount <- merged_data$AmountRequired - merged_data$AmountPaid
underpaid <- subset(merged_data, ShortAmount > 0, select = c("Student.ID", "Name", "ShortAmount")) # nolint

# cat() is a function in R that outputs text to the console, similar to print,
# but more concise, without the extra quotes or print constructs.
cat("Students who did not pay the required fees with the short amount:\n")
print(underpaid, row.names = FALSE)
```