Homework 3
TK2ICM: *Logic Programming* (CSH4Y3)
Midterm Exam Preparation
Second Term 2019-2020

Due date : Saturday, March 7, 2020 at 8:00 p.m. CeLoE time
Type : ***open all***, *individual, cooperation is allowed*

1. Please submit your homework through the submission slot at CeLoE, contact your teaching assistant if you encounter any difficulty.
2. You are allowed to discuss these problems with other class participants, but make sure that you solve the problem individually. Copying answers from elsewhere without understanding them will not enhance your knowledge.
3. You may use any reference (books, slides, internet) as well as ask other students who are not enrolled to this class.
4. Try to solve all problem in a time constraint to make you accustomed to the exam condition.
5. Use the predicate name as described in each of the problem. **The name of the predicate must be precisely identical**. Typographical error may lead to the deduction or cancellation of your points.
6. Submit your work to the provided slot at CeLoE under the file name `Hw3-<iGracias_ID>.pl`. For example: `Hw3-albert.pl` if your iGracias ID is `albert`.

Remark 1 Some of the exam problems will be related to the problems in this homework.

1 *Pakde*, *Bude*, *Paklik*, and *Bulik*

Remark 2 This problem is worth **15 points**.

Suppose we have a family tree as in Figure 1. This family tree is identical to that appeared in Problem Set 1 and Homework 1.

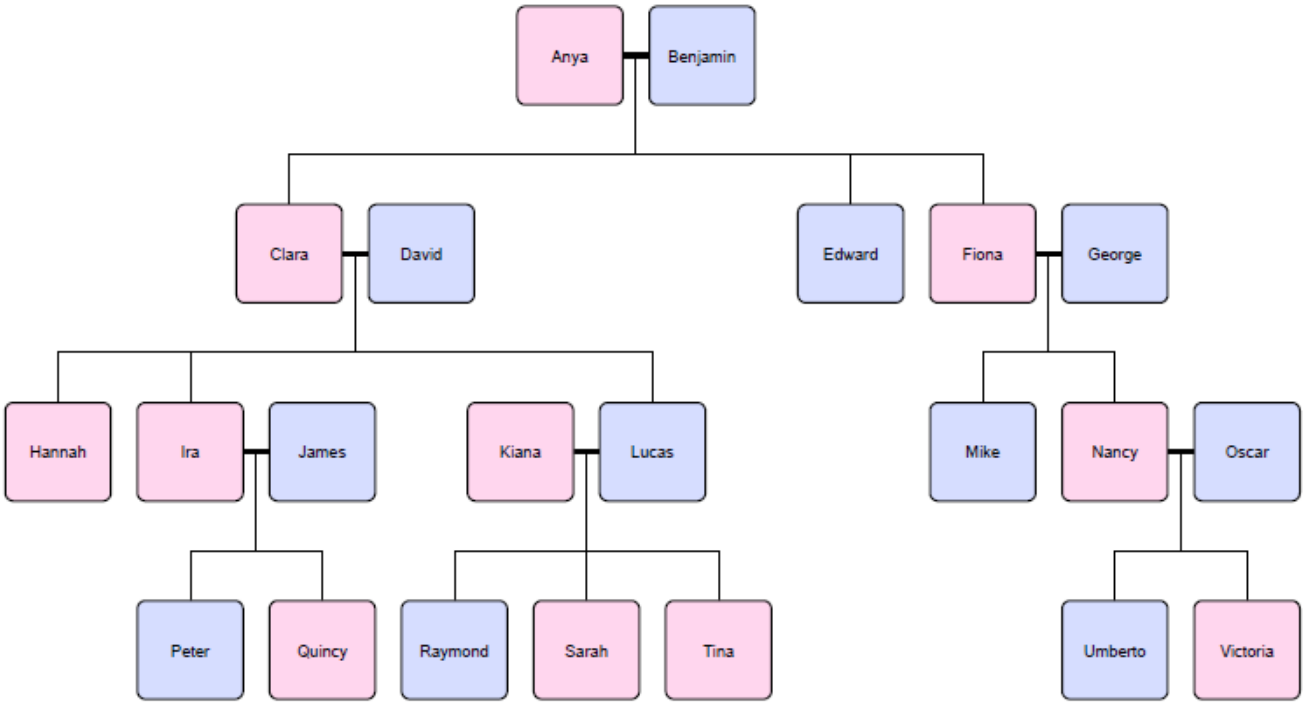

Figure 1: Benjamin's family tree.

Suppose we have the Knowledge Base 1 regarding the facts about parental relationships, genders, and birth years of the people.

Knowledgebase 1 (Note: copy the following knowledge base to your Prolog editor).

```
% parent(X,Y) denotes that X is one of Y's parent
parent(anya,clara).  parent(anya,edward).  parent(anya,fiona).
parent(benjamin,clara).  parent(benjamin,edward).  parent(benjamin,fiona).

parent(clara,hannah).  parent(clara,ira).  parent(clara,lucas).
parent(david,hannah).  parent(david,ira).  parent(david,lucas).

parent(fiona,mike).  parent(fiona,nancy).
parent(george,mike).  parent(george,nancy).

parent(ira,peter).  parent(ira,quincy).
parent(james,peter).  parent(james,quincy).

parent(kiana,raymond).  parent(kiana,sarah).  parent(kiana,tina).
parent(lucas,raymond).  parent(lucas,sarah).  parent(lucas,tina).

parent(nancy,umberto).  parent(nancy,victoria).
```

```
parent(oscar,umberto).  parent(oscar,victoria).

/* person(Person,Gender,Birthyear) explains that the person Person is
of gender Gender and born at Birthyear */

person(anya,female,1938).  person(benjamin,male,1929).

person(clara,female,1959).  person(david,male,1950).
person(edward,male,1963).
person(fiona,female,1965).  person(george,male,1955).

person(hannah,female,1980).
person(ira,female,1982).  person(james,male,1979).
person(kiana,female,1990).  person(lucas,male,1989).

person(mike,male,1991).
person(nancy,female,1994).  person(oscar,male,1992).

person(peter,male,2005).  person(quincy,female,2008).
person(raymond,male,2013).  person(sarah,female,2015).
person(tina,female,2018).

person(umberto,male,2016).  person(victoria,female,2019).
```

Unlike English or Bahasa Indonesia, Javanese language has unique vocabularies to express kinship in an extended family. In Javanese, seniority (or birth order) of children is important. Javanese has distinct terminologies to address the brother or sister of one's parent depending on his/her seniority. In English, we say that X is an uncle of Y if X is a brother of Y's parent (X is Y's uncle by blood, i.e., both X and Y share some genetic overlap) or X is married to the sister of Y's parent (X is Y's uncle by marriage, i.e., they do not share any genetic overlap). Javanese people distinguish the term uncle by considering the seniority of this uncle. There are two separate words for uncle in Javanese, i.e.: *pakde* (an acronym of *ba***Pak** *ge***De**, which literally means "big father") and *paklik* (a short of *ba***Pak** *ci***Lik**, which literally means "little father").

In Javanese, X is a *pakde* of Y if one of the following conditions is satisfied:

- X is the older brother of Y's parent (*pakde* by blood), or
- X is married to the older sister of Y's parent (*pakde* by marriage).

Observe that the birth year of X in the second case is irrelevant, the only important thing is that X is married to someone older than one of Y's parent. Similar definition applies to *paklik*, namely, X is a *paklik* of Y if one of the following conditions is satisfied:

- X is the younger brother of Y's parent (*paklik* by blood), or
- X is married to the younger sister of Y's parent (*paklik* by marriage).

Similar notion applies to the specification of the term aunt. The word *bude* (a short of *i***Bu** *ge***De**, literally means "big mother") is the female counterpart of *pakde*, while the word *bulik* (a short of *i***Bu** *ci***Lik**, literally means "little mother") is the female counterpart of *paklik*.

In this problem your task is to create the predicates `pakde/2`, `bude/2`, `paklik/2`, and `bulik/2`. The predicate `pakde(X,Y)` is **true** if X is a *pakde* of Y. Similar outcomes apply to the predicate `bude/2`, `paklik/2`, and `bulik/2`. Some of the test cases are:

- `?- pakde(edward,nancy).` returns **true**.
- `?- paklik(george,lucas).` returns **true**.
- `?- bude(hannah,raymond).` returns **true**.
- `?- bulik(fiona,lucas).` returns **true**.
- `?- paklik(X,kiana).` returns **false**.
- `?- pakde(X,oscar).` returns **false**.

Hint 1 First, we need to define the predicate `older(X,Y)` that holds whenever `X` is older than `Y`, this predicate can be defined using the predicate `person/3`. In addition, it is more convenient if we define the predicate `male(X)` and `female(X)` using only the predicate `person/3`. Afterward, we can use the similar idea as in Problem Set 1 and Homework 1.

2 *Pasaran* Days

Remark 3 This problem is worth **15 points**.

Javanese culture is a culture of Javanese people in Indonesia. Javanese calendar is one of the scientific product of this culture. This calendrical system is used concurrently with the Gregorian and Islamic calendar. The current Javanese calendrical system was initiated by Sultan Agung of Mataram in the Gregorian year 1633 CE. This system adopts the Islamic lunar calendar yet it retains the Hindu Saka calendar system of counting.

Javanese calendar has several native ways for the measurement of times, called *cycles*. In addition to the common Gregorian and Islamic seven-day week, Javanese calendar observes a five-day week, called *pasaran* days. The name *pasaran* likely derives from the word *pasar*, which means market in Javanese. According to historian, these *pasaran* cycle was used extensively prior to the introduction of Abrahamic religion in Java. Javanese villagers, merchants, and peasants used this cycle to gather communally at a local market to engage in commerce and socialize.

The days' names of a cycle of a *pasaran* days consecutively are: *legi*, *pahing*, *pon*, *wagé*, and *kliwon*. The cycle continues indefinitely, this means if today is *legi*, then two days from now is *pon*; and if today is *pon*, then three days from now is *legi*. Nowadays, the *pasaran* days are used synchronously with the usual seven-day week cycle to create a *wetonan cycle*—a cycle used in the native Javanese astrological belief.

In this problem a Javanese astrologer ask you to help her in determining the N-th *pasaran* day of a particular period if the first *pasaran* day is known. For example, if the first *pasaran* day is *wagé*, then the third *pasaran* day is *legi*; if the first *pasaran* day is *kliwon*, then the eighth *pasaran* day is *pahing*. To help the astrologer, we create a predicate `day/3` with three arguments, the first argument is the first *pasaran* day, the second argument is a an integer $N \geq 1$, and the last argument is the N-th *pasaran* day. In the previous example, we have `day(wage,3,P)` returns `P = legi` and `day(kliwon,8,P)` returns `P = pahing`. Your script must not yield an infinite recursive call. The predicate `day/3` fails if the first argument is not one of the *pasaran* days or the second argument is not a positive integer. Some test cases:

- `?- day(wage,3,P).` returns `P = legi`.
- `?- day(kliwon,3,P).` returns `P = pahing`.
- `?- day(pon,17,P).` returns `P = wage`.
- `?- day(P,19,legi).` returns `P = pon`.
- `?- day(P,28,pahing).` returns `P = kliwon`.
- `?- day(P,-1,kliwon).` returns **false**.

Side effects such as true or false are admissible. You may ignore the occurrence of singleton variables.

3 The Long Travels

Remark 4 This problem is worth **15 points**.

We are given the following knowledge base of travel information:

```
byCar(auckland,hamilton).
byCar(hamilton,raglan).
byCar(valmont,saarbruecken).
byCar(valmont,metz).

byTrain(metz,frankfurt).
byTrain(saarbruecken,frankfurt).
byTrain(metz,paris).
byTrain(saarbruecken,paris).

byPlane(frankfurt,bangkok).
byPlane(frankfurt,singapore).
byPlane(paris,losAngeles).
byPlane(bangkok,auckland).
byPlane(losAngeles,auckland).
```

(a). **[5 points]** Write a predicate `travelable/2` which determines whether it is possible to travel from one place to another by "chaining together" car, train, and plane journeys. For example:

(i) `travelable(valmont,raglan).` returns

true;
false. (Try to avoid infinite recursive call.)

(ii) `travelable(paris,X).` returns

```
X = losAngeles;
X = auckland;
X = hamilton;
X = raglan;
```

false. (Try to avoid infinite recursive call.)

(iii)
```
travelable(X,bangkok).
X = frankfurt ;
X = valmont ;
X = metz ;
X = saarbruecken ;
```

false. (Try to avoid infinite recursive call.)

(iv) `travelable(X,X).` returns

false. (Try to avoid infinite recursive call.)

(v) `travelable(X,valmont).` returns

false. (Try to avoid infinite recursive call.)

(vi) `travelable(raglan,X).` returns

false. (Try to avoid infinite recursive call.)

(b). **[5 points]** By using `travelable/2` and the above knowledge base, we can find out the possibility to go from a city to another city. In case we are planning a travel, sometimes we really want to know how exactly to get from a city to another city. Write a predicate `travelwhere/3` which tells us how to travel from one place to another. Use the functor `go` to describe the path of the travel. For example:

(i) `travelwhere(hamilton,raglan,X).` returns
`X = go(hamilton,raglan).`
false. (Try to avoid infinite recursive call.)

(ii) `travelwhere(auckland,raglan,X).` returns
`X = go(auckland,hamilton, go(hamilton,raglan));`
false. (Try to avoid infinite recursive call.)

(iii) `travelwhere(losAngeles,raglan,X).` returns
`X = go(losAngeles,auckland, go(auckland,hamilton, go(hamilton,raglan)));`
false. (Try to avoid infinite recursive call.)

(iv) `travelwhere(paris,raglan,X).` returns
`X = go(paris,losAngeles, go(losAngeles,auckland, go(auckland,hamilton, go(hamilton,raglan))));`
false. (Try to avoid infinite recursive call.)

(v) `travelwhere(valmont,paris,X).` returns
`X = go(valmont,saarbruecken, go(saarbruecken,paris));`
`X = go(valmont,metz, go(metz,paris));`
false. (Try to avoid infinite recursive call.)

(c). **[5 points]** Write a predicate `travelhow/3` that not only tells us via which other cities we have to go to get from one place to another, but also *how* we get from one city to the next, i.e., the mode of transport (by car, train, or plane). For example:

(i) `travelhow(hamilton,raglan,X).` returns
`X = go(hamilton,raglan,car);`
false. (Try to avoid infinite recursive call.)

(ii) `travelhow(auckland,raglan,X).` returns
`X = go(auckland,hamilton,car, go(hamilton,raglan,car));`
false. (Try to avoid infinite recursive call.)

(iii) `travelhow(losAngeles,raglan,X).` returns
`X = go(losAngeles,auckland,plane, go(auckland,hamilton,car, go(hamilton,raglan,car)));`
false. (Try to avoid infinite recursive call.)

(iv) `travelhow(paris,raglan,X).` returns
`X = go(paris,losAngeles,plane, go(losAngeles,auckland,plane, go(auckland,hamilton,car, go(hamilton,raglan,car))));`
false. (Try to avoid infinite recursive call.)

(v) `travelhow(valmont,paris,X).` returns
`X = go(valmont,saarbruecken,car, go(saarbruecken,paris,train));`
`X = go(valmont,metz,car, go(metz,paris,train));`
false. (Try to avoid infinite recursive call.)

4 Greatest Common Divisor ($\gcd$)

Remark 5 This problem is worth **20 points**.

Given two nonnegative integers a and b, not both zero, the largest integer d that divides a and b is called the greatest common divisor of a and b. Furthermore, the number d is denoted by $\gcd(a, b)$. In elementary school, we learnt how to determine the $\gcd$ of two or more numbers by listing all of their possible positive divisors. For example, to determine the $\gcd$ of 24 and 36, we see that the positive divisors of 24 are 1, 2, 3, 4, 6, 12, and 24; whereas the positive divisors of 36 are 1, 2, 3, 4, 6, 9, 12, 18, 36. Hence, we get $\gcd(24, 36) = 12$.

In high school and Discrete Mathematics A class, we have seen that listing all possible divisors is laborious. Using the fundamental theorem of arithmetic (See: Discrete Mathematics and Its Application by K. H. Rosen), if the prime factorization of a is $p_1^{a_1} p_2^{a_n} \cdots p_n^{a_n}$ and the prime factorization of b is $p_1^{b_1} p_2^{b_n} \cdots p_n^{b_n}$, we have

$$\gcd(a, b) = p_1^{\min(a_1,b_1)} p_2^{\min(a_2,b_2)} \cdots p_n^{\min(a_n,b_n)}.$$

However, finding the $\gcd$ of two integers using prime factorization is also inefficient. Using the fact that $\gcd(a, b) = \gcd(b, a \bmod b)$, the $\gcd$ of two numbers can be computed efficiently using Euclidean algorithm as follows (imperatively implemented in Python):

```
def gcd(a, b):
    if b == 0:
        return a
    else:
        return gcd(b, a mod b)
```

Alternatively, finding $\gcd(a, b)$ can also be performed in the following way:

1. If $a \neq 0$ and $b = 0$, then $\gcd(a, b) = \gcd(a, 0) = a$.
2. If $b \neq 0$ and $a = 0$, then $\gcd(a, b) = \gcd(0, b) = b$.
3. If $a < b$, then $\gcd(a, b) = \gcd(a, b - a)$.
4. If $a > b$, then $\gcd(a, b) = \gcd(a - b, b)$.

The second description is easier to be implemented in Prolog. In this problem you have to write the predicate `gcd(+A,+B,D)` which returns true whenever `+A` and `+B` are instantiated and `D` is the $\gcd$ of `+A` and `+B`. The value of `+A` and `+B` are restricted to **nonnegative integers**. In addition, the variable `+A` and `+B` must be instantiated and not both zero. If both `+A` and `+B` are zero, then `gcd(0,0,D)` returns the string "`gcd error`" (without quotation mark). For example:

(a). `gcd(0,0,D)` returns
`gcd error`
true;
false. (Try to avoid infinite recursive call.)

(b). `gcd(3,0,D)` returns
`D = 3;`
false. (Try to avoid infinite recursive call.)

(c). `gcd(0,3,D)` returns
`D = 3;`
false. (Try to avoid infinite recursive call.)

(d). `gcd(3,3,D)` returns
`D = 3;`
false. (Try to avoid infinite recursive call.)

(e). `gcd(720,900,D)` returns
`D = 180;`
false. (Try to avoid infinite recursive call.)

(f). `gcd(900,720,D)` returns
`D = 180;`
false. (Try to avoid infinite recursive call.)

(g). `gcd(30,120,D)` returns
`D = 30;`
false. (Try to avoid infinite recursive call.)

(h). `gcd(120,30,D)` returns
`D = 30;`
false. (Try to avoid infinite recursive call.)

(i). `gcd(31,33,D)` returns
`D = 1;`
false. (Try to avoid infinite recursive call.)

(j). `gcd(33,31,D)` returns
`D = 1;`
false. (Try to avoid infinite recursive call.)

5 Favorite Meals

Remark 6 This problem is worth **15 points**.

Suppose we have the following knowledge base of favorite meals information:

```
:- op(650, xfx, suka).
alia suka mie.
alia suka bakso.
alia suka rendang.
alia suka eskrim.

bambang suka bakso.
bambang suka sate.
bambang suka coklat.
bambang suka eskrim.

caca suka sate.
caca suka mie.
caca suka bakso.
caca suka coklat.

dani suka bakso.
dani suka sate.
dani suka rendang.
dani suka eskrim.
```

Your task is to write a predicate `dan` in the form of an infix operator (`dan` means "and" in English) so that the following queries are possible:

(a). `?- Siapa suka bakso dan mie.` returns
```
Siapa = alia;
Siapa = caca.
```
(Try to avoid infinite recursive call, side effects such as **true** or **false** is admissible.)

(b). `?- Siapa suka bakso dan coklat dan eskrim.` returns
```
Siapa = bambang.
```
(Try to avoid infinite recursive call, side effects such as **true** or **false** is admissible.)

(c). `?- Siapa suka bakso dan sate dan sate dan bakso.` returns
```
Siapa = bambang;
Siapa = dani;
Siapa = caca.
```
(Try to avoid infinite recursive call, side effects such as **true** or **false** is admissible.)

(d). `?- Siapa suka mie dan rendang dan coklat.` returns
false.
(Try to avoid infinite recursive call, side effects such as **true** or **false** is admissible.)

(e). `?- dani suka Apa.` returns all combinations of meals that are liked by `dani`, i.e.,
```
Apa = bakso ;
Apa = sate ;
Apa = rendang ;
```

```
Apa = eskrim ;
Apa = bakso dan bakso ;
Apa = bakso dan sate ;
Apa = bakso dan rendang ;
Apa = bakso dan eskrim ;
Apa = bakso dan bakso dan bakso ;
Apa = bakso dan bakso dan sate ;
Apa = bakso dan bakso dan rendang ;
Apa = bakso dan bakso dan eskrim ;
Apa = bakso dan bakso dan bakso dan bakso ;
Apa = bakso dan bakso dan bakso dan sate ;
Apa = bakso dan bakso dan bakso dan rendang ;
Apa = bakso dan bakso dan bakso dan eskrim ;
⋮
```

(There are infinitely many combinations.)

6 Counting Derangements

Remark 7 This problem is worth **20 points**.

A derangement is a permutation of objects that leaves no object in its original position. The number of derangement of n objects is denoted by $!n$ and read as n *subfactorial*. We have the following example.

Example 1 We define $!0 = 1$ (i.e., there is one permutation of zero object that leaves zero object in its original position) and $!1 = 0$ (i.e., there is no permutation of one object that leaves one object in its original position). The value $!2$, $!3$, and $!4$ can be determined as follows:

- Given a pair $(1, 2)$, the derangement of this pair is $(2, 1)$, so $!2 = 1$.
- Given a tripe $(1, 2, 3)$, the derangement of this triple are $(2, 3, 1)$ and $(3, 1, 2)$, so $!3 = 2$.
- Given a 4-tuple $(1, 2, 3, 4)$, the derangement of this tuple are:

$$(2,1,4,3)\,,\ (2,3,4,1)\,,\ (2,4,1,3)\,,$$
$$(3,1,4,2)\,,\ (3,4,1,2)\,,\ (3,4,2,1)\,,$$
$$(4,1,3,2)\,,\ (4,3,1,2)\,,\ (4,3,2,1)\,,$$

  thus $!4 = 9$.

Using inclusion-exclusion principle and mathematical induction, the number of derangement of a set with n elements (i.e., the value of $!n$) is given by

$$\begin{aligned} !n &= n! - \frac{n!}{1!} + \frac{n!}{2!} - \frac{n!}{3!} + \cdots + (-1)^n \frac{n!}{n!}. \\ &= n! \left(1 - \frac{1}{1!} + \frac{1}{2!} - \frac{1}{3!} + \cdots + (-1)^n \frac{1}{n!}\right). \end{aligned}$$

Write a predicate `subfactorial(+N,S)` that returns **true** whenever `S` is equal to the value of `!+N`, that is, `S` is the number of derangement of `+N` objects. The variable `+N` must be instantiated at **it is assumed to be nonnegative**. You may use the predicate `factorial/2` as a subroutine. However, it is easier if we can find the recursive formula for $!n$. (Hint: find a recursive formula for $!n$ first. **Use algebraic manipulation to obtain it**. It involves a lot of trick though.) Test case example:

- `subfactorial(0,X).` returns
  `X = 1`;
  false. (This is the base case, try to avoid infinite recursion.)
- `subfactorial(1,X).` returns
  `X = 0`;
  false. (This is the base case, try to avoid infinite recursion.)
- `subfactorial(2,X).` returns
  `X = 1`;
  false. (Try to avoid infinite recursion.)
- `subfactorial(3,X).` returns
  `X = 2`;
  false. (Try to avoid infinite recursion.)

- `subfactorial(4,X).` returns
  `X = 9`;
  false. (Try to avoid infinite recursion.)
- `subfactorial(5,X).` returns
  `X = 44`;
  false. (Try to avoid infinite recursion.)
- `subfactorial(10,X).` returns
  `X = 1334961`;
  false. (Try to avoid infinite recursion.)
- `subfactorial(20,X).` returns
  `X = 895014631192902121`;
  false. (Try to avoid infinite recursion.)

(Hint: some test cases can be seen at Number of Derangements.)