Name: NIM:

Homework 3
TK2ICM: *Logic Programming* (CSH4Y3)
Midterm Exam Preparation
Second Term 2018-2019

Due date : Monday, March 11, 2019 at 05:00 a.m. CeLoE or Google Classroom time
Type : ***open all***, *individual, cooperation is allowed*

Instruction:

1. This homework is due **Monday March 11 at 05:00 a.m. CeLoE or Google Classroom time**. Please submit your homework through the submission slot at CeLoE and Google Classroom (*to make you familiar with the submission during the exam, you have to submit your script to both submission slots*). You are allowed to discuss these problems with other class participants, but make sure that you solve the problems individually. Copying answers from elsewhere without understanding them will not enhance your knowledge.
2. You may use any reference (books, slides, internet) as well as ask other students who are not enrolled to this class.
3. Discussion with fellow participants is encourage as long as you do not copy or rewrite the codes without understanding them. Try solving the problem individually before consulting other students.
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 cancellation of your points.
6. Submit your work to the provided slot at CeLoE and Google Classroom under the file name `Hw3-<your_name>.pl`. For example: `Hw3-Albert.pl`. Please see an information regarding your nickname at Google Classroom.

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

1 *Pasaran* Days

Remark 2 This problem is worth **20 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`.

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

2 Triangles

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

This problem consist of three parts. All parts related to triangles, yet each part is unique to one another.

2.1 Inverted Triangle

(**This subproblem is worth 5 points.**)

Write a predicate `inverttri/1`, the predicate `inverttri(N)` yields an "inverted triangle" as in the following test cases:

- `?- inverttri(4).` returns

```
* * * *
* * *
* *
*
```

- `?- inverttri(N).` returns false for any integer `N` $<$ `1`.

2.2 Triangular Stars – Version 1

(**This subproblem is worth 5 points.**)

Write a predicate `startri/1`, the predicate `startri(N)` returns a "triangle" as in the following test cases:

- `?- startri(4).` returns

```
*
* *
* * *
* * * *
```

- `?- startri(N).` returns false for any integer `N` $<$ `1`.

2.3 Triangular Stars – Version 2

(**This subproblem is worth 5 points.**)

Write a predicate `tristar/2`, the predicate `tristar(N)` returns a "triangle" as in the following test cases:

- `?- tristar(4).` returns

```
      *
    * *
  * * *
* * * *
```

- `?- tristar(N).` returns false for any integer `N` $<$ `1`.

(Hint: observe that the stars in line i for $1 \leq i \leq N$ is preceded by $N - i$ blank spaces.)

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 Happy Pi Day!

Remark 7 This problem is worth **15 points**.

Pi Day is an annual celebration of the mathematical constant pi (π). It is observed on March 14 (3/14 in the month/day date format) since 3, 1, and 4 are the first three digits of π. The number π is an *irrational number*, which means that π cannot be expressed as a ratio of two integers (i.e., there are no $a, b \in \mathbb{Z}$ such that $\frac{a}{b} = \pi$). Nevertheless, in elementary school, we learnt that π can be approximated by the number $\frac{22}{7}$ (although it is only accurate to two decimal digits). In addition to this fact, mathematician in 19th century had proved that π is also a *transcendental number*. This means π is not a root of any nonzero polynomial equation with rational coefficients.

To clarify the transcendence of π, consider the following explanation. In Mathematical Logic A, we've seen the prove of the irrationality of $\sqrt{2}$, i.e., there are no $a, b \in \mathbb{Z}$ such that $\frac{a}{b} = \sqrt{2}$. This fact can be easily proven using an indirect proof by contradiction. However, the number $\sqrt{2}$ is not a transcendental number since $\sqrt{2}$ is a root of the polynomial equation $x^2 - 2 = 0$. The polynomial $x^2 - 2$ has integer (and therefore rational) coefficients. This condition does not apply to π, i.e., there is no polynomial equation

$$a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n = 0$$

of degree n for any $n \in \mathbb{N}$ where $a_i \in \mathbb{Q}$ for any $1 \leq i \leq n$ such that

$$a_0 + a_1 \pi + a_2 \pi^2 + \cdots + a_n \pi^n = 0.$$

Despite the irrationality and transcendence of π, there is an interesting connection between π and the summation of the reciprocals of squares of natural numbers. This was first formally posed by Pietro Mengoli in 1644 and solved by Leonhard Euler in 1734. This problem asks for the precise summation of the infinite series:

$$\sum_{n=1}^{\infty} \frac{1}{n^2} = \frac{1}{1^2} + \frac{1}{2^2} + \frac{1}{3^2} + \cdots, \tag{1}$$

Euler proved that the exact value of (1) is equal to $\frac{\pi^2}{6}$. This breakthrough gives us one way to approximate π by using the following identity

$$\begin{aligned} \sum_{n=1}^{\infty} \frac{1}{n^2} &= \frac{\pi^2}{6} \\ \pi^2 &= 6 \sum_{n=1}^{\infty} \frac{1}{n^2} \\ \pi &= \sqrt{6 \sum_{n=1}^{\infty} \frac{1}{n^2}}. \end{aligned} \tag{2}$$

To approximate π, we can use (2) and define the following sum

$$approxpi\,(N) = \sqrt{6 \sum_{n=1}^{N} \frac{1}{n^2}}. \tag{3}$$

Your task is to create a predicate `approxpi(+N,Approx)` that returns **true** whenever `Approx` is equal to the approximation of π using the summation (3). The variable `+N` must be instantiated and **it is assumed to be positive integers**. Several test case examples are:

- `?- approxpi(1,X)` returns

`X = 2.449489742783178;`
false. (Try to avoid infinite recursion.)

- `?- approxpi(10,X)` returns
  `X = 3.04936163598207;`
  false. (Try to avoid infinite recursion.)
- `?- approxpi(100,X)` returns
  `X = 3.1320765318091053;`
  false. (Try to avoid infinite recursion.)
- `?- approxpi(1000,X)` returns
  `X = 3.1320765318091053;`
  false. (Try to avoid infinite recursion.)
- `?- approxpi(10000,X)` returns
  `X = 3.1414971639472147;`
  false. (Try to avoid infinite recursion.)
- `?- approxpi(100000,X)` returns
  `X = 3.141583104326456;`
  false. (Try to avoid infinite recursion.)