2. Simple arithmetics. Recursion

2.1 Greatest Common Divisor (GCD)

Let us write a predicate which computes the *greatest common divisor* of two natural numbers. We will apply *Euclid's algorithm*, for which you have the pseudocode below:

gcd(a,a) = a
gcd(a,b) = gcd(a – b, b), if b<a
gcd(a,b) = gcd(a, b – a), if a<b

The above algorithm is a mathematical recurrence, which means that we will need to write a recursive predicate. Since a Prolog predicate does not return a value other than *yes/no (T/F)*, we need to add the result to the predicate parameter list. Therefore, our predicate will be gcd/3, or gcd(X,Y,Z), where X and Y are the two natural numbers, and Z is their *gcd*. The first clause is a fact, stating that the *gcd* of two equal numbers is their value:

```
gcd(X,X,X). % clause 1
```

One more thing you need to know before writing clauses 2 and 3 is that, in Prolog, mathematical expressions are not evaluated implicitly. Therefore, you need to force their evaluation, by using the ***is*** operator (X is <expression>). Therefore:

```
gcd(X,Y,Z):- X>Y, R is X-Y, gcd(R,Y,Z). % Y<X, clause 2
gcd(X,Y,Z):- X<Y, R is Y-X, gcd(X,R,Z). % X<Y, clause 3
```

Let us follow the execution of several queries for the *gcd/3* predicate (use the trace command):

```
?- gcd(3, 3, X).
      1    1 Call: gcd(3,3,_407) ?
?     1    1 Exit: gcd(3,3,3) ?  % unifies with clause 1, stop
X = 3 ? ;                       % solution, repeat the question
      1    1 Redo: gcd(3,3,3) ? % attempt to unify query with following clause
      2    2 Call: 3>3 ?  % first call in body of clause 2
      2    2 Fail: 3>3 ?  % fail, attempt to unify with following clause
      3    2 Call: 3<3 ?   % first call in body of clause 3
      3    2 Fail: 3<3 ?  % fail, no clauses left
      1    1 Fail: gcd(3,3,_407) ? %fail
no
?- gcd(3, 7, X).
      1    1 Call: gcd(3,7,_407) ?  % initial call
      2    2 Call: 3>7 ?  % unify with head of the second clause, first call in body
      2    2 Fail: 3>7 ?  % fail
      3    2 Call: 3<7 ?  % unify with clause 3, first call in body
```

```
     3      2 Exit: 3<7 ? % success
     4      2 Call: _861 is 7-3 ? % second call in body of clause 3
     4      2 Exit: 4 is 7-3 ? % success
     5      2 Call: gcd(3,4,_407) ? % third call in body of clause 3
     6      3 Call: 3>4 ? % unify with clause 2, first call in body
     6      3 Fail: 3>4 ? % fail
     7      3 Call: 3<4 ? % unify with clause 3, first call in body
     7      3 Exit: 3<4 ? % success
     8      3 Call: _3317 is 4-3 ? % second call in body of clause 3
     8      3 Exit: 1 is 4-3 ? % success
     9      3 Call: gcd(3,1,_407) ? % … and so on…
    10      4 Call: 3>1 ?
    10      4 Exit: 3>1 ?
    11      4 Call: _5773 is 3-1 ?
    11      4 Exit: 2 is 3-1 ?
    12      4 Call: gcd(2,1,_407) ?
    13      5 Call: 2>1 ?
    13      5 Exit: 2>1 ?
    14      5 Call: _8229 is 2-1 ?
    14      5 Exit: 1 is 2-1 ?
    15      5 Call: gcd(1,1,_407) ?
?   15      5 Exit: gcd(1,1,1) ?
?   12      4 Exit: gcd(2,1,1) ?
?    9      3 Exit: gcd(3,1,1) ?
?    5      2 Exit: gcd(3,4,1) ?
?    1      1 Exit: gcd(3,7,1) ?
X = 1 ? ;
     1      1 Redo: gcd(3,7,1) ?
     5      2 Redo: gcd(3,4,1) ?
     9      3 Redo: gcd(3,1,1) ?
    12      4 Redo: gcd(2,1,1) ?
    15      5 Redo: gcd(1,1,1) ?
    16      6 Call: 1>1 ?
    16      6 Fail: 1>1 ?
    17      6 Call: 1<1 ?
    17      6 Fail: 1<1 ?
    15      5 Fail: gcd(1,1,_407) ?
    18      5 Call: 2<1 ?
    18      5 Fail: 2<1 ?
```

```
   12      4 Fail: gcd(2,1,_407) ?
   19      4 Call: 3<1 ?
   19      4 Fail: 3<1 ?
    9     3 Fail: gcd(3,1,_407) ?
    5     2 Fail: gcd(3,4,_407) ?
    1    1 Fail: gcd(3,7,_407) ?
no
```

Exercise 2.1: Trace the execution of the following queries, repeating the question:

1. ?- gcd(30, 24,X).
2. ?- gcd(15, 2, X).
3. ?- gcd(4, 1, X).

2.2 Factorial

The *factorial* of a number is again defined as a recurrent mathematical relation:

fact(0)=1

*fact(n) =n*fact(n-1), n>0*

Let's write a predicate which computes the factorial of a natural number (below). Note that again you need to use the *is* operator to force the evaluation of a mathematical expression:

```
fact(0,1).
fact(N,F):-N1 is N-1, fact(N1,F1), F is F1*N.
```

Exercise 2.2: Follow the execution of the following queries, repeating the question:

1. ?- fact(6, 720).
2. ?- N=6, fact(N, 120).
3. ?- fact(6, F).
4. ?- fact(N,720).
5. ?- fact(N,F).

Questions 2.1: Why do you think the execution enters an infinite loop when repeating the question? Why do you get an error for queries 3 and 5?

Answers: a. The last call in the deduction tree is fact(0,_someInternalFreeVariable), which has been matched with the first clause. When repeating the question, this call is matched with the second clause, N reaches -1, in the next call -2, ...a.s.o. To prevent this, we should add at the beginning of the second clause: N>0; *therefore, the body of the second clause is:*

```
N>0, N1 is N-1, fact(N1,F1), F is F1*N.
```

b. this predicate is not reversible; i.e. you cannot change the direction of the input/output parameters.

The above version for the factorial predicates builds the solution as recursion returns, i.e. for the factorial of *n*, it assumes that we have already computed the factorial of *n-1* (just like in the recurrence formula). Is there another way to write the predicate which computes the factorial of a number? The answer is, of course, ***yes***: assume we start the computation from *n*, at each step multiply the partial result with the current natural number and get to the previous natural number; stop when we reach *0*. Let's see how such a predicate looks like:

```
fact1(0, FF, FF).
fact1(N, FP, FF):-N>0, N1 is N-1, FP1 is FP*N, fact1(N1, FP1, FF).
```

Question 2.2: How do you call/query the fact1/3 predicate, to get the factorial of 6, for example?

Answer: ?- fact1(6, 1, F)*, i.e. you must initialize the accumulation parameter with the neutral (default) element, which for "*" is 1.*

Exercise 2.2: Follow the execution of the following queries, repeating the question:

1. ?- fact1(6, 1, F).
2. ?- fact1(2, 0, F).

For such predicates in which you employ an accumulation parameter, which has to be initialized at call time, you may write a pretty call, which hides this initialization. For example, for the fact1/3 predicate, you may write:

```
fact1_pretty(N,F):-fact1(N,1,F).
```

This way, you no longer have to worry about the correct initialization value for the accumulator.

2.3 FOR loop

Even if repetitive control structures are not specific to Prolog programming, they can be easily implemented. Let us take a look at an example for the *for* loop:

for(int i=n; i>0; i--) {….}

In Prolog, this would look like:

```
for(In,In,0):-!.
for(In,Out,I):-
        NewI is I-1,
        <do_something_to_In_to_get_Intermediate>,
        for(Intermediate,Out,NewI).
```

Exercise 2.3: Write a predicate forLoop/3 which computes the sum of all integers smaller than some integer (e.g. forLoop(0, Sum , 9) should output: 45). Trace the execution on several queries on your predicate.

2.4 Quiz Exercices

2.4.1 ***Least Common Multiplier****:* write a predicate which computes the least common multiplier of two natural numbers (*Hint: the least common multiplier of two natural numbers is equal to the ratio between their product and their gcd*).

2.4.2 ***Fibonacci Sequence****:* write a predicate which computes the nth number in the Fibonacci sequence. The recurrence formula for the Fibonacci sequence is:

fib(0)=1
fib(1)=1
fib(n) =fib(n-1)+fib(n-2), n>1

2.4.3 **Repeat….until:** write a predicate which simulates a *repeat…until* loop and prints all integers between *Low* and *High*.

Hint: the structure of such a loop is:
repeat
<do something>
until <some condition>

2.4.4 **While:** write a predicate which simulates a *while* loop and prints all integers between *Low* and *High*.

Hint: the structure of such a loop is:
while <some condition>
<do something>
end while

2.5 Problems

2.5.1. ***Triangle Inequality:*** the triangle inequality states that for any triangle, the sum of the lengths of any two sides must be greater than the length of the remaining side. Write a predicate triangle/3, which verifies if the arguments can form the sides of a triangle.

2.5.2. ***2nd order equation***: write a predicate solve_eq2/4 which solves a second order equation of the form $ax^2+bx+c=0$.