Arithmetic Operations

1 Theoretic Considerations

During this laboratory you will become familiar with the principal mathematical recurrences and with the types of recursion.

1.1 The „is" operator

In Prolog, mathematical expressions are **only** evaluated through the „is" operator.

Examples:

Query
?- X = 1+2. X = 1+2. ?- X is 1+2. X = 3.

The mathematical expression is written on the right of „is" and must not contain any uninstantiated variables.

Examples:

Query
?- X is (1+2*3)/7-1. X = 0. ?- X is sqrt(4). X = 2.0. ?- Y = 10, X is Y mod 2. X = 0. ?- X is Y+1. Arguments are not sufficiently instantiated ?- X is 1+2, X is 3+4. false.

1.2 Greatest Common Divisor (GCD)

To calculate the greatest common divisor we will use Euclid's algorithm. The first variant of the pseudocode:

gcd(a,a) = a
gcd(a,b) = gcd(a-b, b), dacă a>b
gcd(a,b) = gcd(a, b-a), dacă b>a

The second variant (more efficient for big numbers, nonzero numbers):

gcd(a,0) = a
gcd(a,b) = gcd(b, a mod b)

Because in Prolog, predicates return only true/false, we must add the result in the list of parameters.

Code

```
% Variant 1
gcd(X,X,X). % The third parameter is the result of GCD
gcd(X,Y,Z) :- X>Y, Diff is X-Y, gcd(Diff,Y,Z).
gcd(X,Y,Z) :- X<Y, Diff is Y-X, gcd(X,Diff,Z).

% Variant 2
gcd2(X,0,X). % The third parameter is the result of GCD
gcd2(X,Y,Z) :- Y\=0, Rest is X mod Y, gcd2(Y,Rest,Z).
```

Example of tracing the execution (see Laboratory 1 for **trace**):

Query

```
[trace] 3 ?- gcd(3,3,R).
   Call: (7) gcd(3, 3, _G1614) ?          % unifies with first clause
   Exit: (7) gcd(3, 3, 3) ?               % succeeds
R = 3 ;                                   % repeat query
   Redo: (7) gcd(3, 3, _G1614) ?          % unifies with clause 2
   Call: (8) 3>3 ?                        % first call of clause 2
   Fail: (8) 3>3 ?                        % fails and moves to
% the next clause
   Redo: (7) gcd(3, 3, _G1614) ?          % unifies with clause 3
   Call: (8) 3<3 ?                        % first call of clause 3
   Fail: (8) 3<3 ?                        % fails
   Fail: (7) gcd(3, 3, _G1614) ?          % fails (there is no
% other clause to unify with)
```

```
false.
```

Query

```
[trace] 6 ?- gcd(3,5,R).
   Call: (7) gcd(3, 5, _G1614) ?          % unifies with clause 2
   Call: (8) 3>5 ?                        % first call of clause 2
   Fail: (8) 3>5 ?                        % fails and moves to
% the next clause
   Redo: (7) gcd(3, 5, _G1614) ?          % unifies with clause 3
   Call: (8) 3<5 ?                        % first call of clause 3
   Exit: (8) 3<5 ?                        % succes
   Call: (8) _G1693 is 5-3 ?              % second call of clause 3
   Exit: (8) 2 is 5-3 ?                   % succes
   Call: (8) gcd(3, 2, _G1614) ?          % recursive call from clause 3
   Call: (9) 3>2 ?                        % first call of clause 2
   Exit: (9) 3>2 ?                        % succes
   Call: (9) _G1696 is 3-2 ?              % second call of clause 2
   Exit: (9) 1 is 3-2 ?                   % succes
   Call: (9) gcd(1, 2, _G1614) ?          % recursive call from clause 3
   Call: (10) 1>2 ?                       % etc.
   Fail: (10) 1>2 ?
   Redo: (9) gcd(1, 2, _G1614) ?
   Call: (10) 1<2 ?
   Exit: (10) 1<2 ?
   Call: (10) _G1699 is 2-1 ?
   Exit: (10) 1 is 2-1 ?
   Call: (10) gcd(1, 1, _G1614) ?
   Exit: (10) gcd(1, 1, 1) ?
   Exit: (9) gcd(1, 2, 1) ?
   Exit: (8) gcd(3, 2, 1) ?
   Exit: (7) gcd(3, 5, 1) ?
R = 1
```

Follow the execution of the following:

Query

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

It is important to remember the following:

- Prolog predicates have a boolean (*true/false*) return type, this return is used by the prolog interpretor in order for sequential processing of commands. Think of a *logical and* – that is the "**,**" operator in Prolog – in order for a predicate to return *true* means that *all* its called subpredicates need to return *true*. If *one* returns *false*, it results in *false*.
- We don't always want prolog to return a true or false, we might want it to calculate something for us. In this case, think of C as an analogy. C allows one return, when we want to have access to more -> we can use the parameters of a function through the address(&) operator. In a similar fashion, the output of prolog predicates is one of the parameters, in the cases present above of the gcd/predicate, it's the third argument.
    - In consequence, we cannot use any of the following to receive the output of a prolog predicate:
        - gcd(...) = R
        - R = gcd(...)
        - gcd(...) is R
        - R is gcd(...)
    - The correct way is just to use the parameters *gcd(X,Y,R)*, where X and Y are instantiated variables and R is an uninstantiated variable that becomes instantiated once the call is finished.

1.3 Factorial

The mathematical recurrence of the factorial is:

factorial(0) = 1
factorial(n) = n * factorial(n-1), pentru n>0

1.3.1 Backwards Recursion

Within Backwards Recursion, the result is constructed on the return from the recursion. It means that the result is initialized when returning from the recursive call.

Code

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

Follow the execution of:

Query

```
?- trace, fact_bwd(6, 720).
?- trace, N=6, fact_bwd(N, 120).
?- trace, fact_bwd(6, F).
?- trace, fact_bwd(N,720).
?- trace, fact_bwd(N,F).
```

If the query is repeated (through **;** in *SWI* or **Next** in *SWISH*), it would result in an infinite loop – as the stopping condition would be ignored. In order to receive only one result, we will need to ensure that N does not become negative in the second clause (as the first is ignored in subsequent queries). The second clause of the predicate must be:

Code

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

1.3.2 Forwards Recursion

Within Forwards Recursion, the result is constructed in an **accumulator.** An accumulator is an *additional* argument (in this case, the 2nd) required for forward recursion. The value of the accumulator must be assigned to the result at the last step of recursion (the stopping clause). The value of the new accumulator will be calculated before the recursive call.

Code

```
fact_fwd(0,Acc,F) :- F = Acc.
fact_fwd(N,Acc,F) :- N > 0, N1 is N-1, Acc1 is Acc*N, fact_fwd(N1,Acc1,F).

% the wrapper
fact_fwd(N,F) :- fact_fwd(N,1,F). % the accumulator is initialized with 1
```

The accumulator must be instantiated at every query with the same value. Being a factorial, we will use the neutral element of multiplication (**1**). Having to write an additional parameter on every query can become bothersome, to ease the process a *wrapper* can be implemented.

Note. The signature of a Prolog predicate is determined by the name and the number of arguments, as such the *fact_fwd/2* and *fact_fwd/3* can be distinguished as different predicates by Prolog.

1.3.3 Backwards vs Forwards recursion

We will make a rudimentary trace over the two types of recursion, using a query of factorial of 3:

- Backwards:

Query

```
?- fact_bwd(3, F).

fact_bwd(3, Fa).
        N1(2) is N(3) - 1.
        fact_bwd(2, Fb).
                N1(1) is N(2) - 1.
                fact_bwd(1, Fc).
```

```
                        N1(0) is N(1) - 1.
                        fact_bwd(0, Fd).

% at this point, the predicate is able to unify with the stopping condition (clause
1);
% DO NOTICE that it uses only the 0 to stop
% In this case only the first argument is used to reach
% the stopping condition, we could call it the stopping argument
% While the second argument (1) is actually used to instantiate the second
argument

% DO NOTICE that no operations were made yet in order to calculate the
% factorial, the operations are made on the return calls from the
% recursion, hence the name: backward recursion

                        fact_bwd(0, 1).
                        F is F1(1) * N(1)
                fact_bwd(1, 1) % second argument is the F from previous
                F is F1(1) * N(2)
        fact_bwd(2, 2) % second argument is the F from previous
        F is F1(2) * N(3)
fact_bwd(3, 6).
```

- Forwards:

As mentioned above the accumulator (second argument) must be instantiated on each query:

Query

```
?- fact_fwd(3, 1, F).

fact_fwd(3, 1, F).
        N1 is N(3) - 1
        Acc1 is Acc(1) * N

        fact(2, 3, F)
                N1 is N(2) - 1
                Acc1 is Acc(3) * N

fact(1, 6, F).
                        N1 is N(1) - 1
                        Acc1 is Acc(6) * N
```

```
                        fact_fwd(0, 6, F).

% at this point, the predicate unifies with the stopping condition
% (first clause), through the use of the first argument.

% The second argument being an instantiated variable and
% the third argument an uninstanted variable – that was transfered until
% this point from one recursive call to another – are unified and
% will both contain the same value, in this case 6.

fact_fwd(0, 6, 6).
                fact_fwd(1, 6, 6).
        fact_fwd(2, 3, 6).
fact_fwd(3, 1, 6).

% As you can see at the return of the recursive calls there are no calculations
made,
% all the operations were done before each recursive call, hence the name:
% forwards recursion
```

There are a few important things to note:

- Firstly, watch what happens to the accumulator at the return of each recursive call: it goes back to the state it was in that call. This is the first reason, we must use an output (thrd argument) in order to save the final value.
- The second reason is that without the uninstantiated variable (the third argument) it would be unable to return a value – it would return only true or false. Try this, remove the last argument from this predicate and use the following query:

?- fact(3, 1).

- The third reason that justifies the use of an accumulator is that we would not be able to do mathematical operations otherwise. Try to remove the second argument and replace it with the third in the operations. What do you think will happen? You would be multiplying a number(**N**) with an uninstanted variable (**F**) which would result in an error.

Note. In the backward trace the output (last argument in both) changes on each recursive call, while in the forwards it remains the same; this is a inherent feature of how the two types of recursion differ in processing.

1.4 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:

Code

```
for(Inter,Inter,0).
for(Inter,Out,In):-
  In>0,
  NewIn is In-1,
  <do_something_to_Inter_to_get_Intermediate>
  for(Intermediate,Out,NewIn).
```

Note1. Observe what happens in the stopping condition (clause 1 of the *for/3* predicate). What kind of recursion is that? If you thought of forward recursion, you would be right. The first argument behaves like an accumulator.

Note2. Observe that for this predicate, the classic output is given by the second argument (not the last). In general, the wanted output will be the last argument because of convenience and readability, not obligation.

Note3. The bolded line of the second clause (I>0), what does it do? The best way to find out is to trace the predicate one without it and once with it. Remember: operations like a *sum*, have a singular mathematical answer.

2 Exercises

1. Write the predicate *lcm/3* which calculates the Least Common Multiplier (LCM).

Suggestion: LCM between two natural numbers is the division of their product by their GCD.

Query

```
% lcm(X, Y, Z). – will calculate the least common
% multiplier between X and Y and return it in Z
?- lcm(12, 24, Z).
Z=24;
false.
```

2. Write the *triangle/3* predicate that verifies if the three parameters can represent the lengths of the sides of a triangle. The sum of any two edges must be bigger than the third.

Query

```
% triangle(A, B, C). – will check whether A,B,C could be the sides
% of a triangle and will return true or false
?- triangle(3, 4, 5).
true.

?- triangle(3, 4, 8).
false.
```

3. Write the *solve/4* predicate that solves the quadratic equation ($a*x^2+b*x+c = 0$). The predicate must have 3 input parameters (A,B,C) and one ouput parameters (X). When repeating the query it must return the second (distinct) solution if it exists.

Query

```
% solve(A, B, C, X). – will solve the quadratic equation A*x2+B*x+C = 0
% and will return the result(s) in X or false otherwise
?- solve(1, 3, 2, X).
X=-1;
X=-2;
false.

?- solve(1, 2, 1, X).
X=-1;
false.
```

```
?- solve(1, 2, 3, X).
false.
```

4. Write the predicates for raising a number to a chosen power using:
   - forward recursion: *power_fwd/3*
   - backword recusion: *power_bwd/3*

Query

```
% power(X, Y, Z). – will calculate X to the power of Y and return it in Z
?- power_fwd(2, 3, Z).
Z=8;
false.

?- power_bwd(2, 3, Z).
Z=8;
false.
```

5. Write the predicate **fib/2** which calculates the n-th number in the Fibonacci sequence.

The recurrence formula is:

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

Query

```
% fib(N, X). – will calculate N-th number of the Fibonacci sequence
% and return it in X
?- fib(5, X).
X=5;
false.

?- fib(8, X).
X=21;
false.
```

6. Write the previous predicate using only 1 recursion call.

Query

```
% fib1(N, X). – will calculate N-th number of the Fibonacci sequence
% and return it in X
?- fib1(5, X).
X=5;
false.

?- fib1(8, X).
X=21;
false.
```

7. **For:** Write the *for/3* predicate after the specifications given in section 2.4 that calculates the sum of all numbers smaller than a given number.

Query

```
?- for(0, S, 9).
S=45;
false.
```

8. **For:** Write the *for_bwd/2* predicate using backwards recursion that calculates the sum of all numbers smaller than a given number.

Query

```
% for_bwd(In,Out). – will compute the sum of all values between 0 and In
% and return the result in Out
?- for_bwd(9, S).
S=45;
false.
```

9. **While:** Write the *while/3* predicate which simulates a *while* loop and returns the sum of all values between *Low* and *High*.

   Hint. The structure of such a loop is:

```
while <some condition>
        <do something>
end while
```

Query

```
% while(Low,High,Sum). – will compute the sum of all values between
% Low and High and return the result in Sum
?- while(0, 5, S).
S=10;
```

```
false.

?- while(2, 2, S).
S=0;
false.
```

10. **Repeat….until:** Write the *dowhile/3* predicate which simulates a *repeat…until* loop and returns the sum of all values between two given numbers.

    Hint. The structure of such a loop is:

    ```
    repeat
            <do something>
    until <some condition>
    ```

Query

```
% dowhile(Low,High,Sum). – will compute the sum of all values between
% Low and High and return the result in Sum
?- dowhile(0, 5, S).
S=10;
false.

?- dowhile(2, 2, S).
S=2;
false.
```