Cairo University
Faculty of Computers & Information

Artificial Intelligence (AI)
Lab#2
Recursion

Lab objective:

- To introduce how to solve problems recursively.
- Solve problems in Prolog using recursion

Lab process:

Part1-> Explanation by TAs
Part2 -> Explanation by TAs
Part3 -> Student activities
Part4 -> Explanation by TAs

Recursion Exercises in Prolog

Part1:

Recursion is an algorithmic technique where a function, in order to accomplish a task, calls itself with some part of the task.

Every recursive solution involves two major parts or cases.

- Base case(s), in which the problem is simple enough to be solved directly, and
- Recursive case(s). A recursive case has three components:
    1. Divide the problem into one or more simpler or smaller parts of the problem,
    2. Call the function (recursively) on each part, and
    3. Combine the solutions of the parts into a solution for the problem.

In Prolog, recursion appears when a predicate contain a goal that refers to itself. Recursive definition always has at least two parts:

- A first fact that act like a stopping condition. "***Base Case***"
- A rule that calls itself. "***Recursive Case***"

At each level the first fact is checked. If the fact is true, the recursion ends. If not, the recursion continues.

A recursive rule must never call itself with the same arguments! If that happens, the program will never end.

Recursion Solution vs. Iterative Solution

Now let's think about when it is a good idea to use recursion and why. In many cases there will be a choice: many methods can be written either with or without using recursion.

Q: Is the recursive version usually faster?
A: No -- it's usually slower (due to the overhead of maintaining the stack)

Q: Does the recursive version usually use less memory?
A: No -- it usually uses more memory (for the stack).

Q: Then why use recursion??
A: Sometimes it is much simpler to write the recursive version.

Part2:

Problem#1: **Power**

Define the predicate **power (X, N, P)** that sets the value of P to X^N

The power function, $p(X, N) = X^N$, can be defined recursively:

$$P(X, N) = \begin{cases} X & \text{if } N = 1 \\ X * P(X, N-1) & \text{else} \end{cases}$$

Here is the pseudo-code of the function **power**:

```
Algorithm Power(x, n):
Input: A number x and integer n
Output: The value x^n
if n = 0 then
      return 1
else
      return x * Power(x, n-1)
```

```
% Compute the power of N-1 then P will be this power * X

power(X,1,X).
power(X,N,P):-
      N1 is N-1,
      power(X,N1,P1),
      P is X * P1.


% another solution could be as follows: X^10 is (X^5)*(X^5) and
% X^5 is (X) * (X^2) * (X^2)
%Base cases
power(X, 1, X).

% If the power was even
power(X, Y, Output):-
      0 is Y mod 2,
      Y2 is div(Y,2),
      power(X, Y2, Output2),
      Output is Output2 * Output2.

% If the power was odd
power(X, Y, Output):-
      Y2 is div(Y,2),
      power(X, Y2, Output2),
      Output is X * Output2 * Output2.
```

Part3:

Problem#2: **Successor Arithmetic**

The following is a possible implementation of simple arithmetic using compound terms to represent integers:

0	0
1	s(0)
2	s(s(0))
3	s(s(s(0)))
4	s(s(s(s(0))))

.

Number Check:

Define predicate is_number(arg1) that takes one argument as a compound term to represent an integer number then tries to prove if it's a successor term or not.

Hint:

The recursive definition of is_number relation is as the following:

is_number(Num) is true if is_number(Num-1)

```
%is_number(+Num), where Num is a successor Term

is_number(0).
is_number(s(Number)) :-
      is_number(Number).
```

Problem#3: **Differentiation**

Define the predicate **differentiate (E1, X, E2)** that is true if expression E2 is a possible form for the derivative of expression E1 with respect to X. for example:

$$\frac{d}{dx}(A+B) = \frac{d}{dx}A + \frac{d}{dx}B$$

This predicate should handle the plus(A+B) and times(A*B) only. For example,

```
%?- diff( plus(times(x, x), times(3,x)), x, Dx).
%Produces:
%Dx  =
%plus(plus(times(x,1),times(1,x)),plus(times(3,1),times(0,x)))
```

```
%diff(+Expr, +X, -Output), where Output is the derivative of Expr
with respect to X

diff(plus(A,B), X, plus(DA, DB)):-
       diff(A, X, DA),
       diff(B, X, DB).

diff(times(A,B), X, plus(times(A, DB), times(DA, B))):-
       diff(A, X, DA),
       diff(B, X, DB).

equal(X, X).
diff(X, X, 1).
diff(Y, X, 0):-
      \+(X=Y). %Y is constant
```

Part4:

Tail Recursion

Tail recursion is a special case of call functions in which last operation of the function is the recursive call. Tail recursive functions store information while computing the recursive call that is independent of the number of recursive calls. Here's an example of the factorial function written recursively first, and using tail recursion second.

```
Algorithm Factorial(x) using normal recursion:
Input: A number x
Output: The value x!
if x = 0 then
      return 1
else
      return x * Factorial(x-1)
```

```
%Factorial program using normal recursion
factorial(1,1).
factorial(N,Result) :- NewN is N-1,
        factorial(NewN,NewResult),
         Result is N * NewResult.
```

```
Algorithm Factorial(x) using tail recursion:
Input: A number x
Output: The value x!
Factorial(x, 1, 1)

Algorithm Factorial(x, i, temp)
Input: A number x, index and a temporal result
Output: the value of x!
if x = i then
      return temp
else
      return Factorial(x, i+1, (i+1)*temp)
```

```
%Factorial program re-written using tail recursion
factorial(N,Result) :- fact_iter(N,1,Result).
fact_iter(0,Result,Result).
fact_iter(N,Acc,Result) :-
                NewN is N-1,
                NewAcc is N *Acc,
                fact_iter(NewN,NewAcc,Result).
```