TITLE
Recursion and lists in Prolog.
Weeks 5, 6

OBJECTIVES
- Using recursion in Prolog.
- Using lists in Prolog

RESOURCES
- Course Slides
- SWI Prolog official learning book

DELIVERABLES
- Prolog code files
- A document describing the observations and conclusions.

LAB

Recursion

For recursion in Prolog, there will always be at least two rules to code: at least one rule for the *base case*, or non-recursive case, and at least one rule for the *recursive case*.

```
parent(david, john).
parent(jim, david).
parent(steve, jim).
parent(nathan, steve).
grandparent(A, B) :- parent(A, X), parent(X, B).
```

```
Define when someone is an ancestor of someone else. (non-recursive way)
ancestor(A,B) :- parent(A, X), parent(X, Y), parent(Y, B).
ancestor(A,B) :- parent(A, X), parent(X, Y), parent(Y, Z), parent(Z,B).
```

```
Define when someone is an ancestor of someone else. (recursive way)
ancestor(A, B) :- parent(A, B). % base case
ancestor(A, B) :- parent(A, X), ancestor(X, B). % recursive case
```

```
% factorial of a given number
factorial(0, 1).
factorial(N, F) :-
        N > 0,
        N1 is N - 1,
        factorial(N1 ,F1),
                F is N * F1.
```

Lists

In Prolog, a list is an object that contains an arbitrary number of other objects within it.

Eg.: [1, 2, 3] ≠ [1, 3, 2]

Prolog notation: [H|T] where:

- H is the head of the list and it represents the first element of the list
- T is the tail of the list and it represents the rest of the list.

Heads and Tails of Lists

List	Head	Tail
[a,b,c,d]	*a*	[b,c,d]
[a]	*a*	[]
[[1,2],3]	[1,2]	[3]

Unification of Lists

List 1	List 2	Binding
[X,Y,Z]	[alex, eats, fruits]	X = alex, Y = eats, Z = fruits
[10]	[X|Y]	X = 10, Y = []
[a, b, c, d]	[X,Y|Z]	X = a ,Y = b, Z = [c, d]
[4, 5]	[3|X]	FAIL!

```
%parsing a list
parse([]).
parse([H|T]) :-write(H),parse(T).


%check if is a list

is_list([]).
is_list([_|T]) :- is_list(T).

%append a list to another list
append1([],L,L).
append1([H|T],L2,[H|L3])  :-  append1(T,L2,L3).
```

Reversing a list

Course 4 example

```
% naive rec reverse list
naive_rev([],[]).
naive_rev([H|T], R) :- naive_rev(T, S), append(S, [H], R).
% accumulator rec reverse list
rev(L, R) :- rev(L, [], R).
rev([], R, R).
rev([H|T], C, R) :- rev(T, [H|C], R).
```

Another accumulator example

```
list_length1([],0).
list_length1([_|Tail],Length) :-
        list_length1(Tail,TailLength),
        Length is TailLength + 1.

%accumulator based
list_length2(List,Length) :- list_length2(List,0,Length).
list_length2([],Length,Length).
list_length2([_|Tail],Accumulator,Length) :-
        NewAcc is Accumulator + 1,
        list_length2(Ta il,NewAcc,Length).
```

Adding up a list of numbers

```
% base case
addup([], 0).

% recursive case: if the base-case rule does not match, this one must:
addup([FirstNumber | RestOfList], Total) :-
        addup(RestOfList, TotalOfRest),
        Total is FirstNumber + TotalOfRest.
```

Exercises:

1. Given facts such as:

Bob is taller than Mike.
Mike is taller than Jim.
Jim is taller than George.
Write a recursive program that will determine that Bob's height is greater than George's.
2. Get elements on the n-th position in a given list.
3. Check if a list is a prefix of another list. Check if a list is a suffix of another list.
4. Add one to each element of a list.