# Prolog is declarative.

From wikipedia: "**Declarative** programming is a non-imperative style of programming in which programs describe their desired results without explicitly listing commands or steps that must be performed."
For example, To find an element X that lies in both Lists A and B, you could declare:

```
in_both( A, B, X):-
        member(X,A),
        member(X,B).
```

This can be used to check if a constant is in both *A* and *B* by passing a constant as *X*

```
        ?-  in_both( [1,2,3,4], [2,4,6,8], 2).
```

Or to generate an element that is in both A and B by passing a variable as *X* .

```
        ?- in_both( [1,2,3,4], [2,4,6,8], X).
```

Using the inbuilt *findall* predicate, All the values of a variable that satisfy a rule can be collected into a list.
Set intersection can be implemented as:

```
set_intersection( A, B, Intersection ):-
        findall( X, in_both(A,B,X), Intersection ).
```

**Implement:**

**Q1.** set_difference(A, B, Differences ):-

*A* and *B* are Lists with no repetition.
*Differences* is the result of A-B

**Q2.** suffix( Suffix, String):-

*Suffix* is a suffix of the string *String.*
( Hint: use the append function )

***Q3.*** substr( Substr, String) ):-

*Substr* is a substring of *String.*

## Recursive rules

Prolog rules can be recursive. For example, Given facts about the parent relation b/w individuals, The predecessor relation can be defined as:

```
predecessor(X,Y):-
        parent(X,Y). % Base case for recursion
predecessor(X,Y):-
        parent(X,Z),
        predecessor( Z, Y).    % Recursive rule
```

Lists are recursively defined as a head element appended to a list. The member function can be implemented as:

```
        member(X, [X|Tail] ).
        member(X, [Head|Tail] ):-
                member(X, Tail ).
```

**Implement:**

**Q1.** reverse_list( List, Reversed):-

*Reversed* is *List* with elements in reverse order

( You may have to declare a rule with 3 arguments. )

**Q2.** palindrome( List ):-

True if *List* is a palindrome

**Q3.** merge( A, B, Res):-

*Res* is the list resulting from merging the two sorted lists *A* and *B*.

## Backtracking for CSPs

Prolog does backtracking in an attempt to find a model satisfying the specified rules/constraints. This makes it easy for us ( not for computers ) to frame and solve Constraint Satisfaction Problems such as sudoku.

**Q1.** Given a 2x2(x2x2) sudoku grid, come up with a state representation and formulate a goal. The rules are:

- Each row must contain each of the numbers 1,2,3,4 exactly once.
- Each column must contain each of the numbers 1,2,3,4 exactly once.
- Each of the boxes must contain each of the numbers 1,2,3,4 exactly once.

Here's an example problem and solution

| | | | |
|---|---|---|---|
| | | | |
| **1** | | **3** | |
| | **1** | | **4** |
| | | | |

With solution:

| | | | |
|---|---|---|---|
| 2 | 3 | 4 | 1 |
| **1** | 4 | **3** | **1** |
| 3 | **1** | 2 | 3 |
| 4 | 2 | 1 | 4 |

**And here's one for you to solve:**

| | | | |
|---|---|---|---|
| | | | |
| **2** | **3** | | |
| | | **4** | **2** |
| | | | |

( Hint: Use the permutation inbuilt predicate )

**If you want to test your solution on more instances, here's a problem generator:**
http://www.menneske.no/sudoku/2/eng/

## Bonus questions

**Q1.** max_element( List, MaxElement ):-

*MaxElement* holds the value of the largest element in *List*

**Q2.** bubblesort( List, Sorted ):-

*Sorted* is the sorted *List.*

*(* Your approach should implement the bubble_mintohead )

**Q3.** bubble_mintohead( List, Res):-

*Res* is a list that results after moving the minimum element of *List* to be the head.