Some useful built-in predicates

append (List1, List2, Final_List)

- This is a simple concatenation of two strings List1 and List2 into a new string List3.

```
?- append([a,b,c],[d,e,f],Ans).
Ans = [a,b,c,d,e,f].
```

append (ListofLists, Final_List)

- Pretty much intuitively the same as earlier one. This helps in concatenating a list of lists together.

```
?- append([[a,b,c],[d,e,f]],Ans).
Ans = [a,b,c,d,e,f].
```

member (Element, List)

- This gives a *true* if the *Element* is a member of the *List*. Technically, it succeeds if *Element* can be unified with one of the members of *List*.

```
?- member([a,b],[a,b,c]).
false.
?- member([a,b],[[a,b],c,d]).
true.
```

delete (List, Item, Final_List)

- *Item* is deleted from *List* and stored in *Final_List. Here, Prolog checks _List* for the element which is unifiable with *Item* and eliminates it.

```
?- delete([a,b,c,d],[a],Ans).
Ans = [a,b,c,d].
?- delete([a,b,c,d],a,Ans).
Ans = [b,c,d].
?- delete([x,[a,b],y],[a,b],Ans).
Ans = [x,y].
```

select (Elem, List, Rest)

- *select*/3 is a smart predicate. It looks for *Elem* in *List* and conveniently omits it in *Rest*.

```
?- select(a,[a,b,c,d],Ans).
Ans = [b,c,d].
```

- The smartness is to use it for inserting elements.

```
?- select(xxx,Ans,[a,b,c]).
A = [xxx, b, c, d] ;
A = [b, xxx, c, d] ;
A = [b, c, xxx, d] ;
A = [b, c, d, xxx] ;
false.
```

nth0 (Index, List, Elem)

- Gives a *true* if the *Index_th element of _List* unifies with *Elem*, starting the indexing from 0. Similarly, Prolog has *nth1* for counting from 1.

```
?- nth0(2,[a,b,c,d],c).
true.
```

last (List, Elem)

- This one unifies the *Elem* with the last element of the *List*. So, this means we can use it to check for the last element, or get it as well.

```
?- last([a,b,c,d],d).
true.
?- last([a,b,c,d],Ans).
Ans = d.
```

reverse (List, Final_List)

- Certainly a helpful list manipulation predicate. Reverses the given *List* and equates it with the _Final_List.

```
?- reverse([a,b,c,d],Ans).
Ans = [d,c,b,a].
```

permutation (List1, List2)

- It can be used to validate or enumerate the two list.

```
?- permutation([a,b,c],As).
As = [a, b, c] ;
As = [a, c, b] ;
As = [b, a, c] ;
As = [b, c, a] ;
As = [c, a, b] ;
As = [c, b, a] ;
false.
```

flatten (List1, List2)

- Recursively, it eliminates the lists in a list, by replacing them with its contents.

```
?- flatten([a, [b, [c, d], e]], X).
X = [a, b, c, d, e]
```

sumlist (List, Sum)

- Pretty straightforward, isn't it ?

```
?- sumlist([1,2,3,4,5],A).
A = 15.
?- sumlist([a,b,c,d],A).
ERROR: is/2: Arithmetic: `a/0' is not a function
```

- Look into how *is/2* is used in arithmetic in Prolog.

numismatist (Low, High, List)

- Another interesting one. It unifies the *List* to a list with content ranging from *Low* to *High*, both inclusive. Obviously, *Low* <= *High* and both of em should be integers.

```
?- numlist(2,4,[2,3,4]).
Ans = true.
?- numlist(1,9,Ans).
Ans = [1,2,3,4,5,6,7,8,9].
```

- Reminds of *range* from Python.

prefix (Part, Whole)

- Returns a *true* value if the *Part* is a prefix of the *Whole*. Look into how we can use *append* for the same.

```
?- prefix(A,[a,b]).
A = [] ;
A = [a] ;
A = [a, b] ;
false.
?- prefix([a],[a,b,c]).
true.
?- append([a,b],_,[a,b,c]).
true.
```

same_length (List1, List2)

- As the name suggests, this simple yet effective predicate returns a *true* if the lists are of same length.

```
?- same_length([a,b],[a]).
false.
?- same_length([a,b],[a,b]).
true.
```

is_set (Set)

- This is used to check if the list *Set* has duplicates or not.

```
?- is_set([a,b,v]).
true
?- is_set([a,v,v]).
false.
```

max_list (List, Max)

- True if *Max* is the largest number in *List*. Fails if *List* is empty. *max_number* is also similar.
- *min_list* and *min_number* look for the minimum in the *List*.

```
?- max_list([1,2,3,4,5],Ans).
Ans = 5.
?- min_list([1,2,3,4,5],Ans).
Ans = 1.
```

compare (Order, Item1, Item2)

- Determine or test the Order between two items in the standard order of items. *Order* is one of <, > or =, with the obvious meaning.

```
?- compare(<,1,2).
true.
?- compare(<,1,-1).
false.
?- compare(=,1,-1).
false.
?- compare(=,1,1).
true.
```

list_to_set (List, Set)

- This unifies the *Set* with the set equivalent of *List*, i.e. it omits all duplicates.

```
?- list_to_set([a,b,a], Ans).
Ans = [a,b].
?- list_to_set([a,b,c],Ans).
Ans = [a,b,c].
```

intersection (List1, List2, Intersection_List)

- *True* if the third list is the intersection of the first two lists.
- Observe how the ordering is followed in the final answer, and how duplicates are handled.

```
?- intersection([a,b,c],[a,b],Ans).
Ans = [a, b].
?- intersection([b,a,c],[a,b],Ans).
Ans = [b, a].
?- intersection([b,a,c],[],Ans).
Ans = [].
?- intersection([b,a,c],[a,b,c,d],Ans).
Ans = [b, a, c].
?- intersection([a,b,a], [b,a,b], Ans).
Ans = [a, b, a].
```

union (List1, List2, Union_List)

- *True* if the third list is the union of the first two lists.
- Again, look into the ordering of elements in the final list.

```
?- union([a,b,c],[a,b],Ans).
Ans = [c, a, b].
?- union([a,b,c],[a,c,b],Ans).
Ans = [a, c, b].
?- union([a,b,c,d,e,f],[a,c,b],Ans).
Ans = [d, e, f, a, c, b].
?- union([a,b,c,d,e,f],[a,c,b],[x,z,y]).
false.
```

subset (SubSet, Set)

- *True* if all elements of *SubSet* belong to *Set* as well. *or-subset* works in a similar fashion.

```
?- subset([a,b,c],[a,b,c,d]).
true.
?- subset([a,b,c],[a,b,d]).
false.
```

delete (Set, Delete, Result)

- Deletes all elements in *Delete* from *Set*.

```
subtract([a,b,c,d],[b,d],Ans).
Ans = [a, c].
```

sort (List, Sorted)

- Returns *true* is the *Sorted* is the sorted version of *List*. Sorting is done to the standard order of terms, with duplicates removed.

```
?- sort([1,2,3],A).
A = [1, 2, 3].
?- sort([1,3,2],A).
A = [1, 2, 3].
?- sort([B,A,1],[2,3,1]).
B = 2,
A = 3.
?- sort([1,3,2],[2,1,3]).
false.
```

Authored by Divyansh Khanna, 2015.