Prolog List Notation

In Prolog list elements are enclosed by brackets and separated by commas.

```
[1,2,3,4]
[[mary,joe],[bob,carol,ted,alice]]
[]
```

Another way to represent a list is to use the **head/tail notation** `[H|T]`. Here the head of the list, H, is separated from the tail of the list, T, by a vertical bar. The tail of a list is the original list with its first element removed. The tail of a list is always a list, even if it's the empty list.

In Prolog, the H|T notation is used together with unification to combine and break up lists. For example, suppose we have the following list:

```
[bob,carol,ted,alice]
```

Here's the various matches we would obtain using H|T:

```
[X|Y]         matches with X=bob Y=[carol,ted,alice]
[X,Y|Z]       matches with X=bob, Y=carol, Z=[ted,alice]
[X,Y,Z|W]     matches with X=bob, Y=carol, Z=ted W=[alice]
[X,Y,Z,W|V]   matches with X=bob, Y=carol, Z=ted,  W=alice and V=[]
[X,Y,Z,Y]     won't match because Y=carol and carol != alice
[X,Y,Z,W,V|U] won't match because the list does not contain 5 elements
```

We can also build lists using unification and H|T notation. Suppose L unifies with [X|Y] and X=bob and Y=[carol,ted,alice]. Then L=[bob,carol,ted,alice].

Recursive List Examples

In some Prolog environments the `member` predicate is not a built-in predicate and must be defined within your program. It takes the form `member(Element,List)` and evaluates to true if and only if Element is a member of List. The underscore (_) can be used as a **anonymous** or **don't care** variable, meaning we don't care what value it has. It's there solely for pattern-matching (unification) purposes.

```
member(X,[X|_]).          /* 1. Base case:      X is a member of the list
headed by X */
member(X,[Y|L]) :- member(X,L).       /* 2.  Recursive case: X is a member of
the list headed by Y */
              /*                     if X is a member of that list's tail (L)
*/
```

Here's a trace of the member() predicate on the query: member(c,[a,b,c]).

```
member(c,[a,b,c]).
    call 1 (base case). fails, since c != a.
    call 2 (recursive case). X=c, Y=a, L=[b,c], member(c,[b,c]) ?
      call 1 (base). fails, since c != b.
```

```
      call 2 (recursive). X=c, Y=b, L=[c], member(c,[c]) ?
         call 1. Success, c = c.
      Yes to call 2. (backing out of recursion)
    Yes to call 2.   (backing out of recursion)
Yes.                 (original query succeeds)
```

Here's a trace of the member() predicate on the query: member(c,[a,b]).

```
member(c,[a,b]).
    call 1 (base case). fails, since c != a.
    call 2 (recursive case). X=c, Y=a, L=[b], member(c,[b]) ?
      call 1 (base). fails, since c != b.
      call 2 (recursive). X=c, Y=b, L=[], member(c,[]) ?
         call 1. fails, since [] does not match [X|_].
         call 2. fails, since [] does not match [Y|L].
      No to call 2. (backing out of recursion)
    No to call 2.   (backing out of recursion)
No.                 (original query succeeds)
```

The following predicate writes each element of a list using Prolog's built-in `write()` predicate and built-in `nl` (newline) predicate:

```
writelist([]).                                  /* Base case: An empty list */
writelist([H|T]) :- write(H),nl,writelist(T).  /* Recursive case: */


writelist([1,2,3]).

H=1, T=[2,3]

H=2, T=[3]

H=3,T=[]
```

The following predicate writes a list in reverse order:

```
reverse_writelist([]).                                          /* Base case: An
empty list */
reverse_writelist([H|T]) :- reverse_writelist(T),write(H),nl.  /* Recursive
case: */



append([],List,List).



append([Head|Tail],List2,[Head|Result]):-
```

```
    append(Tail,List2,Result).
```

Let's now follow the program as it executes. Using the second rule we first reduce the query to

```
append([b,c],[one,two,three],Result)
```

and then to

```
append([c],[one,two,three],Result)
```

and finally to

```
append([],[one,two,three],Result).
```

This final clause can match against the initial fact, giving append([],[one,two,three],[one,two,three]). Since this is a fact, this terminates the recursion. As we pop out of each recursive step we then add on (respectively) c,b, and a to the list, giving the list [a,b,c.one,two,three].

Write a Prolog program to implement sumlist(L, S) so that S is the sum of a given list L.
sumlist([],0).
sumlist([H|T],S):-sumlist(T,S1),S is H+S1.

Write a Prolog program to implement two predicates evenlen(List) and oddlen(List) so that they are true if their argument is a list of even or odd length respectively
evelen([]).
evelen([_|[_|List]]):-evelen(List).
oddlen([_]).
oddlen([_|[_|List]]):-oddlen(List).

Write a Prolog program to implement nth_element (N, L, X) where N is the desired position, L is a list and X represents the Nth element of L.
nth_element(1,[H|T],H).
nth_element(N,[H|T],X):-N1 is N-1,nth_element(N1,T,X).