Finding nth element of a list

```
domains
        list = integer*

predicates
        nthelement (list, integer)

clauses
        nthelement ([Head|_], 1) :-
                write (Head), nl.
        nthelement ([_|Tail], N) :-
                NN = N-1,
                nthelement (Tail, NN).

goal : nthelement ([1,2,3], 3)
 3
 Yes

goal : nthelement ([1,2,3], 4)
 No
```

Note: This works for N to be ≥ 1.
A slight modification can be made so that it works for any integer N (Putting $N > 0$ as a condition)

Finding Maximum Of a list

Goal: maximum ([5, 2, 7, 3], X)

X = 7 1 soln

V1

```
/*C3*/ max ( [ ], Max, Max).

/*C1*/ max ( [Head | Tail], Max, R) :-
                    Head > Max, max (Tail, Head, R).

/*C2*/ max ( [Head | Tail], Max, R) :-
                    Head <= Max, max (Tail, Max, R).
```

If we put 'cut' in C1, Then Head <= Max is not required. 'cut' may be put as the last element in C1.

V2

```
max ( [X], X).

max ( [Head | Tail], Head ) :-
                    max (Tail, X),
                    Head > X.

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

Finding length of a list

goal: length ([2, 1, 6], X)

X = 3 1 soln

```
length ([], 0).

length ([_|T], N) :- length (T, NN),
                     N = NN + 1.
```

NN = N - 1 won't work
("Free variable in Expression" error)

```
N = NN + 1, length (T, NN)
```

won't work and give same error as above.

Finding Sum of first N Natural Numbers

```
sum (0, 0).

sum (N, Sum) :- N > 0, N1 = N - 1,
                sum (N1, R1), Sum = R1 + N.
```

OR

Without Recursion

```
Sum (N, Sum) :- Sum = N * (N + 1) / 2.
```

Finding odd & even elements separately and framing lists, i.e. Split the input lists into two lists, 1 containing odd elements and the other containing even elements.

goal: oe ([5, 3, 2, 1], X, Y).

X = [5, 3, 1]

Y = [2]

1 solution.

```
oe ( [], [], [] ).

oe ( [H|T], [H|T1] , T2)
        :- H mod 2 <> 0,
           oe (T, T1, T2).

oe ( [H|T], T1, [H|T2] )
        :- H mod 2 = 0,
           oe (T, T1, T2).
```

Reverse without accumulating parameter

```
reverse ( [], [] ).

reverse ( [H|T], Z) :- reverse (T, T1),
                       append (T1, [H], Z).
                                  /* usual def^n of
goal: reverse ( [1, 2, 3], R).      append */

      R = [3, 2, 1]   1 soln
```

Execution Trace for reverse ([1, 2, 3], R)

```
Call : reverse ( [1, 2, 3], R)
    Call: reverse ( [2, 3], T1)
        Call: reverse ( [3], _T1)
            Call: reverse ([], _T1)
            Ret: reverse ([], [])T1
            Call : append ( [], [3], _Z )
            Ret : append ( [], [3], [3])Z
        Ret : reverse ([3], [3]).
                          Z/T1
    Ret: reverse ([2,3],
        Call : append ([3], [2], _Z )
        Ret : append ([3], [2], [3,2])Z
    Ret : reverse ( [2,3], [3,2])
    Call : append ( [3,2], [1], _Z )
    Ret : append ( [3,2], [1], [3,2,1])Z
```

Palindrome — V1 (Using Compare)

clauses

```
reverse([], []).
reverse([H|T], Res) :-
        reverse(T, T1), append(T1, [H], Res).

append([], L, L).
append([X|L1], L2, [X|L3]) :-
        append(L1, L2, L3).

compare([], []).
compare([H|T1], [H|T2]) :-
        compare(T1, T2).

palindrome(List) :- reverse(List, List1),
        compare(List, List1).
```

goal: palindrome([r,a,d,a,r])

Yes

goal: palindrome([s,t,a,d,e,n,t,s])

NO

Palindrome — V2 — Using 2-arg Reverse

```
palindrome(L) :- reverse(L, L).
```

Palindrome — V3 — Using Reverse-2 arg.

```
palindrome([]).
palindrome([_]).
palindrome([H|T]) :-
        reverse(T, [H|T1]),
        palindrome(T1).
```

Palindrome – V4

Works for all even/odd np

```
pal([]).
pal([_]).
pal([H|T]) :- append(X, [H], T),
              pal(X).
```

Here, X is the content obtained by removing 2 elements from the list – The first and the last.

Palindrome – V5

Works for all 3 cases

```
pal([]).
pal([_]).
pal(List) :- append([Y], _, List),      /* Y is First element */
             /* c2 */ append([_|Mid], [Y], List),
                      pal(Mid).
```

---

c2 may be `append([Y|Mid], [Y], List)`