Lists in Prolog

- ✔ The list with elements `a, b, c, d` can be represented in Prolog as

```
[a,b,c,d]
```

- ✔ The empty list is represented as `[]`

- ✔ Any nonempty list can also be represented as `[ Head | Tail ]` where `Head` is the first element of the list and `Tail` is the rest of the list

```
?- [a,b,c,d] = [Head|Tail].
Head = a
Tail = [b,c,d]
```

- ✔ Thus a list can be represented in various ways

```
[a,b,c,d] == [a|[b,c,d]]

[a,b,c,d] == [a,b|[c,d]]

[a,b,c,d] == [a,b,c|[d]]

[a,b,c,d] = [a,b,c,d|[]]
```

Lists in Prolog

- These representations of lists permit defining some interesting relations on lists

- The member relation `member(X,L)` is true if the object `X` occurs in the list `L`

```
member(X,[X|Tail]).

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

- The appending relation `append(L1,L2,L3)` is true if the list `L3` is the result of appending `L1` and `L2`. So for example `append([a,b],[c,d],[a,b,c,d])` is true, but `append([a,b],[c,d],[[a,b],[c,d]])` is false.

```
append([],L,L).

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

Now

```
?- append([a,b],[c,d],L).
L = [a,b,c,d]
```

Lists in Prolog

- ✔ As defined, `append` can do more than compute the result of appending two lists:

  - ✗ `append` can find all decompositions of a list

```
?- append(L1,L2,[a,b,c]).
L1 = []
L2 = [a,b,c];
L1 = [a]
L2 = [b c];
L1 = [a b]
L2 = [c];
L1 = [a b c]
L2 = [];
no
```

  - ✗ append can find a pattern in a list

```
?- append(Before,[may|After],[jan,feb,mar,apr,may,jun,jul]).
   Before = [jan,feb,mar,apr]
   After = [jun,jul]
```

  - ✗ append can be used to define member

```
member(X,L) :- append(_,[X|_],L).
```

Relations and Functions

- ✔ In a "conventional" programming language, you define a function which, when supplied with some arguments, returns a value

```
int f(int a,b,c)
{
 ...
 return ( ... )
}
```

- ✔ Or, you can define a function that has one or more "var parameters" which can be used to return values

```
int g(int a,b,c, *val)
{
 ...
 *val = ...
}
```

- ✔ In Prolog, you define *relations*
  - ✗ for an n-ary relation, some set of n-tuples satisfy the relation
  - ✗ in answering a query to the Prolog interpreter, Prolog will attempt to find objects which, when substituted for variables in the query, satisfy the relation
  - ✗ variables can occupy any argument position in the query
  - ✗ so, any argument to a Prolog relation can act like a "var parameter" which can be used to return values

Lists in Prolog

- ✔ Adding an item to a list

  - ✗ if `X` is an object and `L` is a list, it is easy to represent the list you get from adding `X` to the front of `L`:

```
[X | L]
```

  - ✗ this could be defined in terms of a relation

```
cons(X,L,[X|L]).
```

Lists in Prolog

- ✔ Deleting an item from a list: `del(X,L1,L2)` is true if `L2` is a list obtained by deleting an occurence of `X` from `L1`
  - ✗ if `L1` is of the form `[X|Tail]`, then `L2` is `Tail`. Otherwise `L2`'s tail is the result of deleting `X` from `Tail`...

```
del(X,[X|Tail],Tail).

del(X,[Y|Tail1],[Y|Tail2]) :- del(X,Tail1,Tail2).
```

- ✔ Why does the following happen:

```
?- del(a,[a,b,a,a],L).
L = [b,a,a];
L = [a,b,a];
L = [a,b,a];
no
```

Lists in Prolog

- ✔ `del` can be used to define an insert relation:

```
insert(X,L1,L2) :- del(X,L2,L1).
```

- ✔ `del` can also be used to define the member relation, since `del(X,L1,L2)` will fail if `X` does not occur in `L1`

```
member(X,L) :- del(X,L,_).
```

Lists in Prolog

- ✔ The sublist relation: `sublist(X,Y)` is true if `X` is a sublist of `Y`

```
?- sublist([c,d,e],[a,b,c,d,e,f]).
yes

?- sublist([c,e],[a,b,c,d,e,f]).
no

?- sublist([a,b],[a,b]).
yes
```

```
sublist(X,Y) :- append(L1,L2,Y), append(X,L3,L2).
```

Lists in Prolog

- ✔ The permutation relation: `permutation(L1,L2)` is true if `L2` is a permutation of `L1`
  - ✗ the empty list is a permutation of itself
  - ✗ you can make a permutation of a nonempty list by taking the first element, and inserting it somewhere in a permutation of the tail of the list

```
permutation([],[]).

permutation([X|L],P) :-
   permutation(L,L1), insert(X,L1,P).
```

- ✔ consider the simple cases

```
?- permutation([a,b],L).
```

```
?- permutation(L,[a,b]).
```

Examples:

- ✔ Define two predicates: `evenlist(L)` which is true if `L` is a list with an even number of elements, and `oddlist(L)` which is true if `L` is a list with an odd number of elements.

- ✔ Define a predicate `reverse(L1,L2)` which is true if `L2` is `L1` with its elements in reverse order

- ✔ Define a predicate `palindrome(L)` which is true if `L` is a palindrome