Logic Programming and Prolog

Keep reading: Scott, Chapter 12

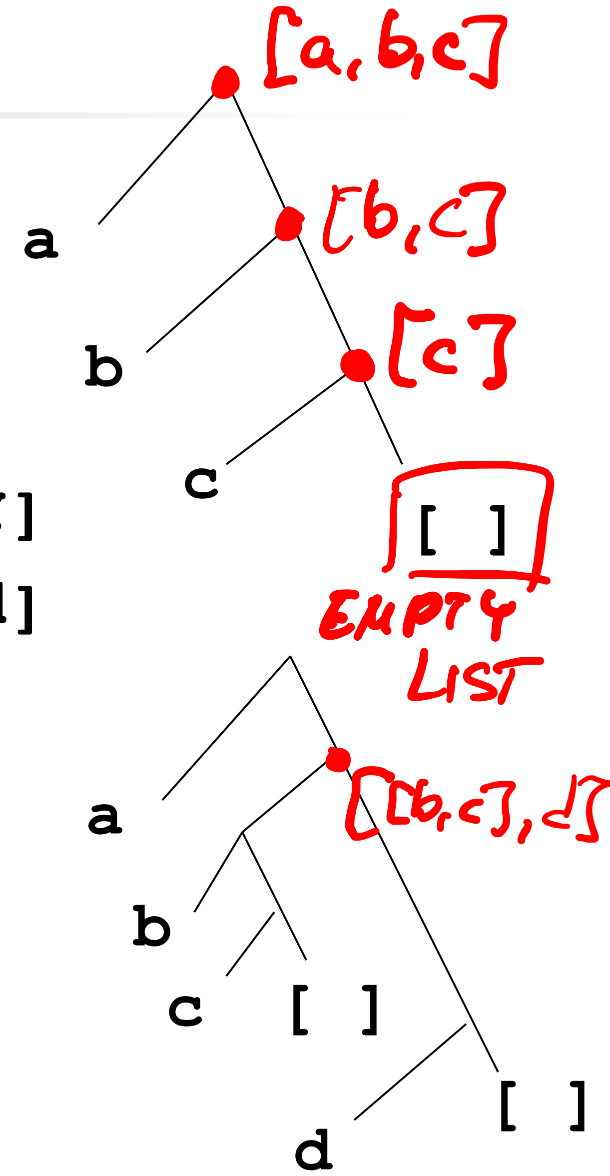
Lecture Outline

- Prolog
  - Lists
  - Programming with lists
  - Arithmetic

Lists

[a | [b,c]]

list	head	tail
`[a,b,c]`	`a`	`[b,c]`
`[X,[cat],Y]`	`X`	`[[cat],Y]`
`[a,[b,c],d]`	`a`	`[[b,c],d]`
`[X	Y]`	`X`	`Y`



Lists: Unification

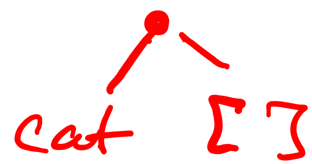
- `[ H1 | T1 ] = [ H2 | T2 ]`
  - Head `H1` unifies with `H2`, possibly recursively
  - Tail `T1` unifies with `T2`, possibly recursively

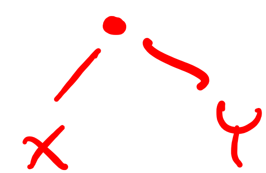
- E.g., `[ a | [b, c] ] = [ X | Y ]`
  - `X = a`
  - `Y = [b, c]`
- NOTE: In Prolog, = denotes unification, not assignment!

Question

.(a, .(b, [ ])) ≡ [a, b] ≡ [a | [b]]

- `[X,Y,Z] = [john, likes, fish]`
  - `X = john, Y = likes, Z = fish`
- `[cat] = [X | Y]`
  - `X = cat, Y = [ ]`





- `[[the, Y]|Z] = [[X, hare]|[is,here]]`
  - `X = the, Y = hare, Z = [is, here]`

[1, 2] [1 | 2]

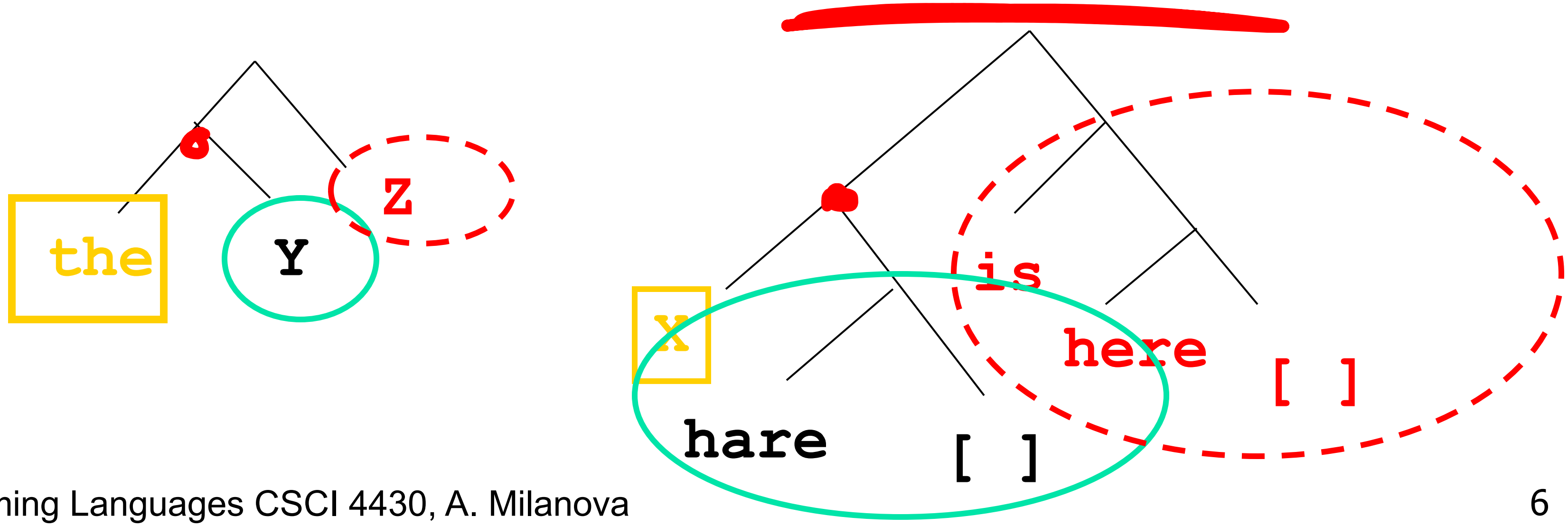
[1 | [2]]

Lists: Unification

- Sequence of comma separated terms, or
- [first term | rest_of_list]

```
[ [the | Y] | Z ]   = [ [X, hare] | [is, here] ]
```

X = the   Y = [hare]   Z = [is, here]



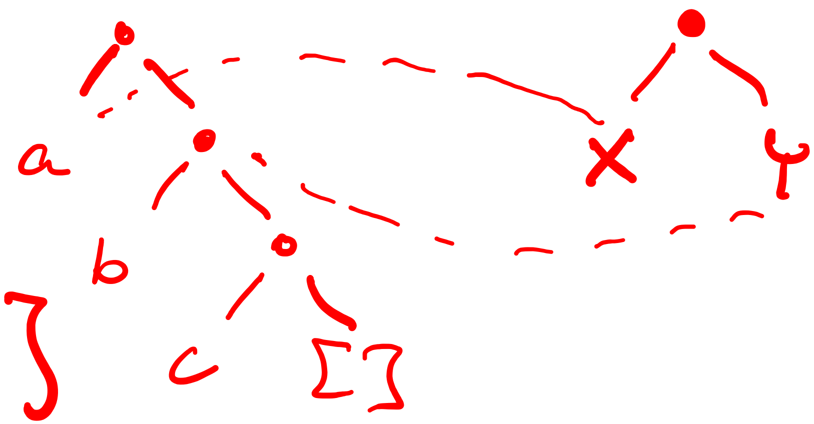
Lists Unification

- Look at the trees to see how this works!

```
[ a, b, c ] = [ X | Y ]
  X = a, Y = [b,c].
```

[b|[c]]

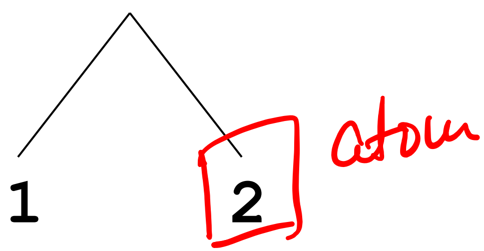
[b|[c|[]]]

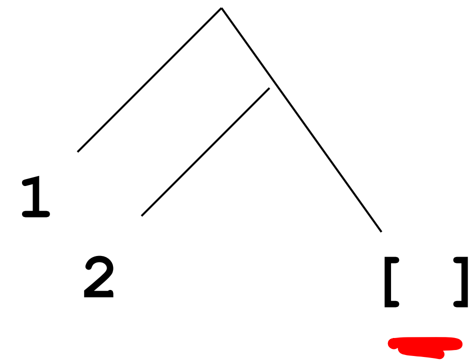



```
[a | Z ] = [ X | Y ]
  X = a, Y = Z.
```

Improper and Proper Lists

`[1 | 2]` versus `[1, 2]`





IMPROPER LIST

PROPER LIST

Question. Can we unify these lists?

[abc, Y] =? [abc | Y]

Answer: No. There is no value binding for **Y** that makes these two trees isomorphic

```
?- [abc, Y] = [abc | Y]
?- Y = [Y].
```

Unification and the Occurs check

(V, [....])

unify ((T1, T2))

// succeeds with bindings of vars, if T1 and T2 can be made equal

// fails otherwise

occurs check flag default is FALSE

OFF

- case (V, T2) → if occurs(V, T2) fail; o.w. V=T2 ✓
- case (T1, V) → if occurs(V, T1) fail; o.w. V=T1 ✓
- case (const1, const2) → if const1 ≠ const2 fail; o.w. skip ✓
- case (p(T11, T12), p(T21, T22)) → unify(T11, T21); unify(T12, T22)
- _ → fail

TURN OCCURS CHECK FLAG ON!

Lecture Outline

- Prolog
  - Lists
  - Programming with lists
  - Arithmetic

Member_of

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

?- member(X,[a,b,c]).
   X = a ;
   X = b ;
   X = c ;
   false.
```

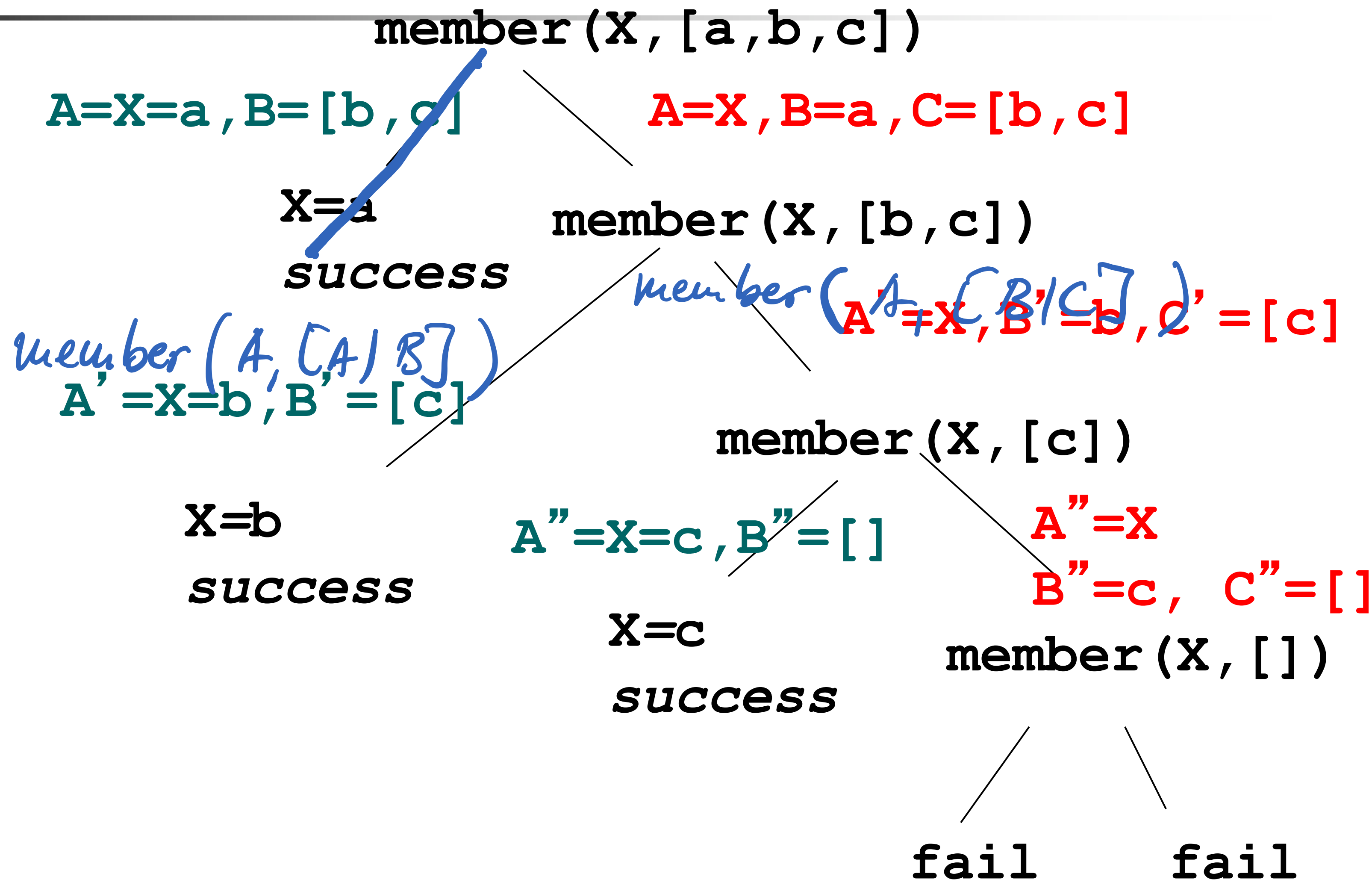
```
1. member(A, [A | B]).
2. member(A, [B | C]) :- member(A, C).
```

Member_of

```
?- member(a,[a,b]).
  true.
?- member(a,[b,c]).
  false.
?- member(X,[a,b,c]).
  X = a ;
  X = b ;
  X = c.
?- member(a,[b,c,X]).
  X = a ;
  false.
```

```
1. member(A, [A | B]).
2. member(A, [B | C]) :- member (A, C).
```

Prolog Search Tree (OR levels only)




```
1. member(A, [A | B]).
2. member(A, [B | C]) :- member (A, C).
```

[] ≠ [A|B]

Member_of

```
member(A, [A|B]).
member(A, [B|C]) :- member(A,C).
```

logical semantics: For every **A,B** and **C**

member(A,[B|C]) if **member(A,C)**

procedural semantics: Head of clause is procedure entry. Tail of clause is procedure body; subgoals correspond to calls.

"Procedural" Interpretation

```
member(A, [A|B]).
member(A, [B|C]) :- member(A,C).
```

- **`member`** is a recursive "procedure"
- **`member(A, [A|B]).`** is the base case. "Procedure" exits with true if the element we are looking for, **`A`**, is the first element in the list. It exits with false if we have reached the end of the list
- **`member(A, [B|C]) :- member(A,C).`** is the recursive case. If element **`A`** is not the first element in the list, call member recursively with arguments **`A`** and tail **`C`**

Question

```
1. member(A, [A | B]).
2. member(A, [B | C]) :- member(A, C).
```

Give all answers to the following query:

```
?- member(a,[b, a, X]).
```

true ;

X = a ;

false.

Answer:

```
true ;
X = a ;
false.
```

member(a, [b, a, c]).

Question

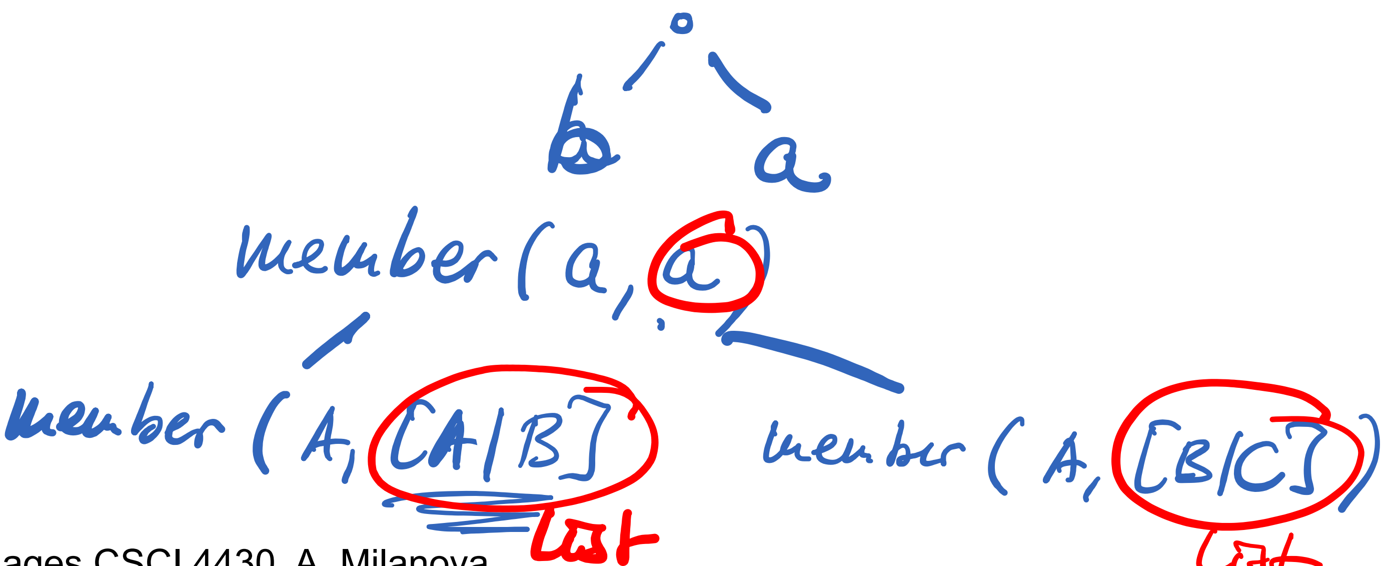
```
1. member(A, [A | B]).
2. member(A, [B | C]) :- member(A, C).
```

Give all answers to the following query:

?- member(a, [b | a]).

Answer:

false.



Append

append(X, Y, Z)
Z is the result of appending Y at the end of X.

```
append([ ], A, A).
append([A|B], C, [A|D]) :- append(B,C,D).
```

result

- Build a list:

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

- Break a list into constituent parts:

```
?- append(X,Y,[a,b]).
  X = [], Y = [a,b]; X = [a], Y = [b];
  X = [a,b], Y = []; false.
```

More Append

```
append([ ], A, A).
append([A|B], C, [A|D]) :- append(B,C,D).
```

unbound variable

- Break a list into constituent parts

```
?- append(X,[b],[a,b]).
X = [ a ]
?- append([a],Y,[a,b]).
Y = [ b ]
```

More Append

```
? - append(X,Y,[a,b]).
X = [ ],
Y = [a,b] ;
X = [a],
Y = [b]    ;
X = [a,b],
Y = [ ]    ;
false.
```

Unbounded Arguments

- Generating an unbounded number of lists

```
?- append(X,[b],Y).
X = [ ]
Y = [b]   ;
X = [ _G604]
Y = [ _G604, b]      ;
X = [ _G604, _G610]
Y = [ _G604, _G610, b]   ;
Etc.
```

An underscore, "don't care" variable.
Unifies with anything.
E.g., `bad(Dog) :- bites(Dog,_).`

- Be careful when using append with 2 unbounded arguments!

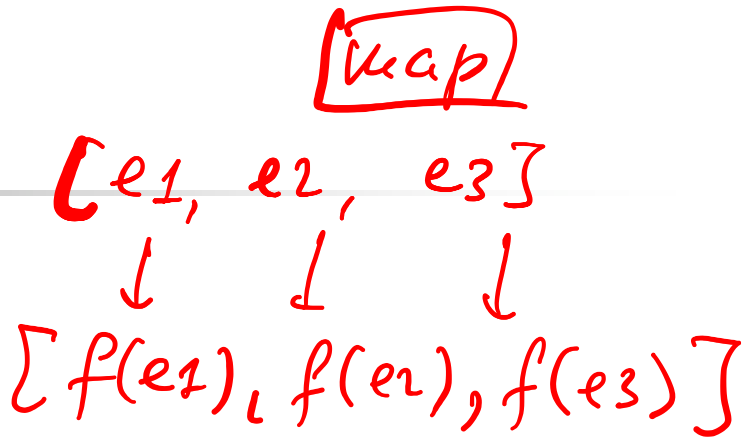
Question

- What does this "procedure" do:

```
p([],[]).
p([A|B],[[A]|Rest]) :- p(B,Rest),
                                   Result = [[A] | Rest].
        Result
```

```
?- p([a,b,c],Y).
Y = [ [a],[b],[c] ]
```

Common Structure



- "Processing" a list:

```
p([],[]).
p([H|T],[H1|T1]) :- f(H,H1),p(T,T1).
```

- Base case: we have reached the end of list. In our case, the result for `[ ]` is `[ ]`.
- Recursive case: result is `[H1|T1]`. `H1` was obtained by calling `f(H,H1)` --- processes element `H` into result `H1`. `T1` is the result of recursive call of `p` on `T`.

Lecture Outline

- Prolog
  - Lists
  - Programming with lists
  - Arithmetic

Arithmetic

- Prolog has all arithmetic operators
- Built-in predicate `is`
  - `is(X, 1+3)` or more commonly we write
  - `X is 1+3`

  `is` forces evaluation of `1+3`:

  ```
  ?- X is 1+3
  X = 4
  ```
- `=` is unification not assignment!

  ```
  ?- X = 4-1.
  X = 4-1 % unifies X with 4-1!!!
  ```

Arithmetic: Pitfalls

- `is` is not invertible! That is, arguments on the right cannot be unbound!
  - `3 is 3 - X.`

```
ERROR: is/2: Arguments are not
  sufficiently instantiated
```

- This doesn't work either:

X is 4, X1 is X+1

```
?- X is 4, X = X+1.
false.
```

✗ NO!

Why? What's going on here?

Exercise

- Write **sum**, which takes a list of integers and computes the sum of the integers. E.g.,

H=1 T=[2,3]

```
sum([1,2,3],R).
?- R = 6.
```

```
sum([],0).
sum([H|T],R) :-
    integer(H), sum(T,RT), R is RT+H.
```

- How about if the integers are arbitrarily nested? E.g.,

```
sum([[1],[[[2]],3]],R).
?- R = 6.
```

Exercise

[[[[1]]],2]

```
sum([],0).
sum(H,H) :- integer(H).
sum([H|T],R) :-
    sum(H,RH), sum(T,RT),
    R is RH+RT.
```

Exercise

- Write `plus10`, which takes a list of integers and computes another list, where all integers are shifted +10. E.g.,

```
plus10([1,2,3],R).
?- R = [11,12,13].
```

- Write `len`, which takes a list and computes the length of the list. E.g.,

```
len([1,[2],3],R).
?- R = 3.
```

Exercise

- Write **`atoms`**, which takes a list and computes the number of atoms in the list. E.g.,

  ```
  atoms([a,[b,[[c]]]],R).
  ?- R = 3.
  ```

- Hint: built-in predicate **`atom(X)`** yields true if **`X`** is an atom (i.e., symbolic constant such as **`x`**, **`abc`**, **`tom`**).

Negation

```
not(member(a,[a,b])).
```

not

```
not(member(c,[a,b])).
```

- ```
  sister_of(X,Y) :-
      female(X),parents(X,M,F),
      parents(Y,M,F).
  ```
  ```

The End