CSc 372 — Comparative Programming Languages

22 : Prolog — Lists

Christian Collberg
Department of Computer Science
University of Arizona
collberg@gmail.com

October 25, 2011

1

Introduction

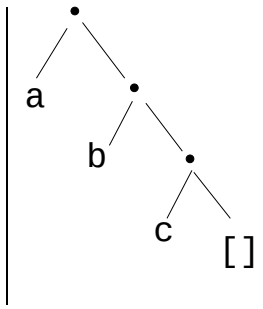
2 Prolog Lists

Haskell:

```
> 1 : 2 : 3 : []
[1,2,3]
```

Prolog:

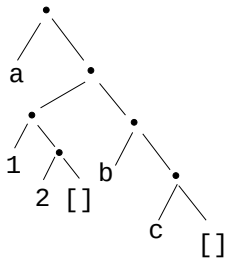
```
?- L = .(a, .(b, .(c, [])))
   L = [a, b, c]
```



- Both Haskell and Prolog build up lists using cons-cells.
- In Haskell the cons-operator is `:`, in Prolog `.`.

3 Prolog Lists...

```
?- L = .(a, .(.(1, .(2, [])), .(b, .(c, []))))
     L = [a, [1, 2], b, c]
```



- Unlike Haskell, Prolog lists can contain elements of arbitrary type.

4 Matching Lists – `[Head | Tail]`

A	F	$A \equiv F$	variable subst.
`[]`	`[]`	yes	
`[]`	`a`	no	
`[a]`	`[]`	no	
`[[]]`	`[]`	no	
`[a | [b, c]]`	`L`	yes	`L=[a,b,c]`
`[a]`	`[H | T]`	yes	`H=a, T=[]`

5 Matching Lists – `[Head | Tail]...`

A	F	$A \equiv F$	variable subst.
`[a, b, c]`	`[H | T]`	yes	`H=a,T=[b,c]`
`[a, [1, 2]]`	`[H | T]`	yes	`H=a, T=[[1, 2]]`
`[[1, 2], a]`	`[H | T]`	yes	`H=[1,2], T=[a]`
`[a, b, c]`	`[X, Y, c]`	yes	`X=a, Y=c`
`[a, Y, c]`	`[X, b, Z]`	yes	`X=a, Y=b, Z=c`
`[a, b]`	`[X, c]`	no	

6

Member

7 Prolog Lists — Member

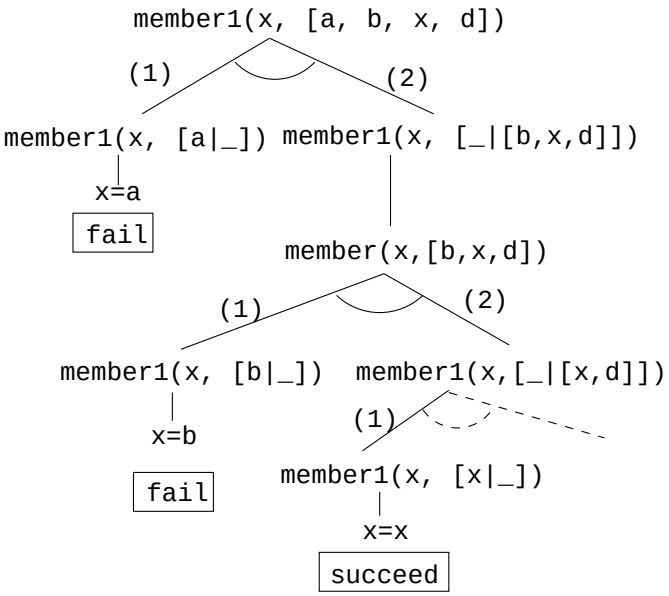
```
(1)   member1(X, [Y|_]) :- X = Y.
(2)   member1(X, [_|Y]) :- member1(X, Y).

(1)   member2(X, [X|_]).
(2)   member2(X, [_|Y]) :- member2(X, Y).

(1)   member3(X,[Y|Z]) :- X = Y; member3(X,Z).
```

8 Prolog Lists — Member...

```
?- member(x, [a, b, c, x, f]).
```

```
      yes

?- member(x, [a, b, c, f]).
      no

?- member(x, [a, [x, y], f]).
      no

?- member(Z, [a, [x, y], f]).
      Z = a
      Z = [x, y]
      Z = f
```

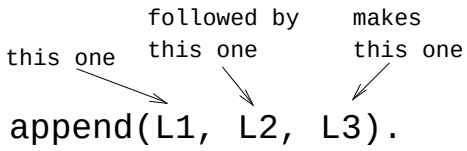
9 Prolog Lists — Member...



10

Append

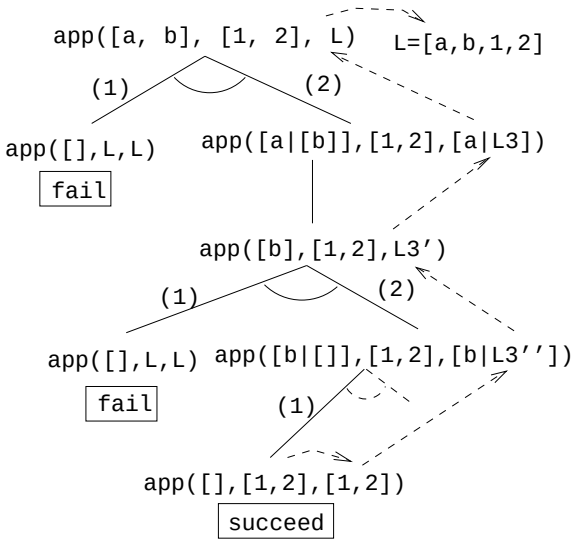
11 Prolog Lists — Append



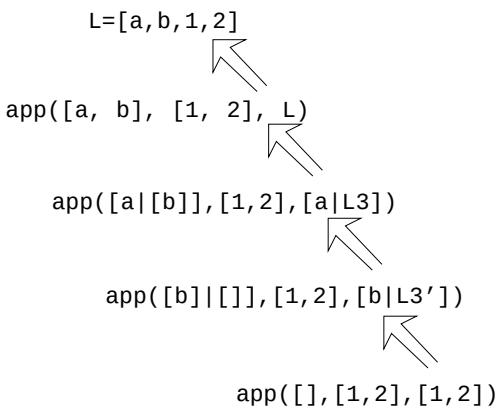

```
(1) append([], L, L)
(2) append([X|L1], L2, [X|L3]) :-
      append(L1, L2, L3).
```

1. Appending L onto an empty list, makes L.
2. To append L_2 onto L_1 to make L_3
   (a) Let the first element of L_1 be the first element of L_3.
   (b) Append L_2 onto the rest of L_1 to make the rest of L_3.

12 Prolog Lists — Append...



13 Prolog Lists — Append...




```
?- L = [a | L3], L3 = [b | L3'], L3' = [1,2].
      L = [a,b,1,2], L3 = [b,1,2], L3' = [1,2]
```

14 Prolog Lists — Using Append

1. `append([a,b], [1,2], L)`
   - What's the result of appending `[1,2]` onto `[a,b]`?
2. `append([a,b],[1,2],[a,b,1,2])`

- Is `[a,b,1,2]` the result of appending `[1,2]` onto `[a,b]`?

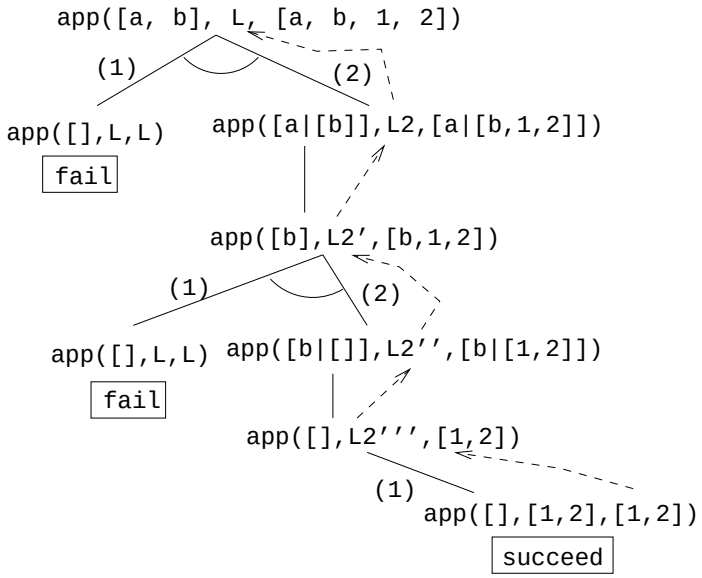
3. `append([a,b], L, [a,b,1,2])`
   - What do we need to append onto `[a,b]` to make `[a,b,1,2]`?
   - What's the result of removing the prefix `[a,b]` from `[a,b,1,2]`?

15 Prolog Lists — Using Append...

4. `append(L, [1,2], [a,b,1,2])`
   - What do we need to append `[1,2]` onto to make `[a,b,1,2]`?
   - What's the result of removing the suffix `[1,2]` from `[a,b,1,2]`?

5. `append(L1, L2, [a,b,1,2])`
   - How can the list `[a,b,1,2]` be split into two lists L1 & L2?

16 Prolog Lists — Using Append...

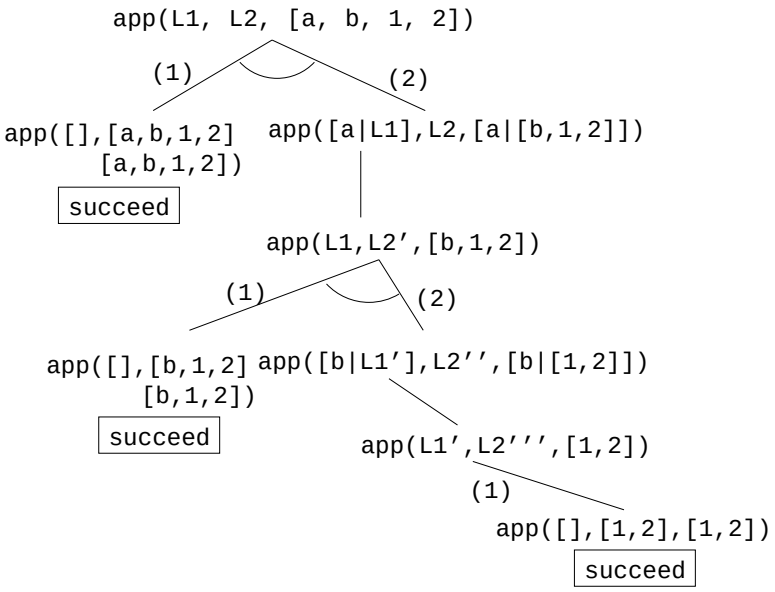


17 Prolog Lists — Using Append...

```
?- 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
```

18 Prolog Lists — Using Append...



19 Prolog Lists — Reusing Append

member Can we split the list Y into two lists such that X is at the head of the second list?

adjacent Can we split the list Z into two lists such that the two element X and Y are at the head of the second list?

last Can we split the list Y into two lists such that the first list contains all the elements except the last one, and X is the sole member of the second list?

20 Prolog Lists — Reusing Append...

```
member(X, Y) :- append(_, [X|Z], Y).
   ?- member(x,[a,b,x,d]).

adjacent(X, Y, Z) :- append(_, [X,Y|Q], Z).
   ?- adjacent(x,y,[a,b,x,y,d]).

last(X, Y) :- append(_, [X], Y).
   ?- last(x, [a,b,x]).
```

21

Reversing a List

22 Prolog Lists — Reverse

- `reverse1` is known as *naive reverse*.
- `reverse1` is *quadratic* in the number of elements in the list.
- From *The Art of Prolog*, Sterling & Shapiro pp. 12-13, 203.
- Is the basis for computing LIPS (Logical Inferences Per Second), the performance measure for logic computers and programming languages. Reversing a 30 element list (using naive reverse) requires 496 reductions. A reduction is the basic computational step in logic programming.

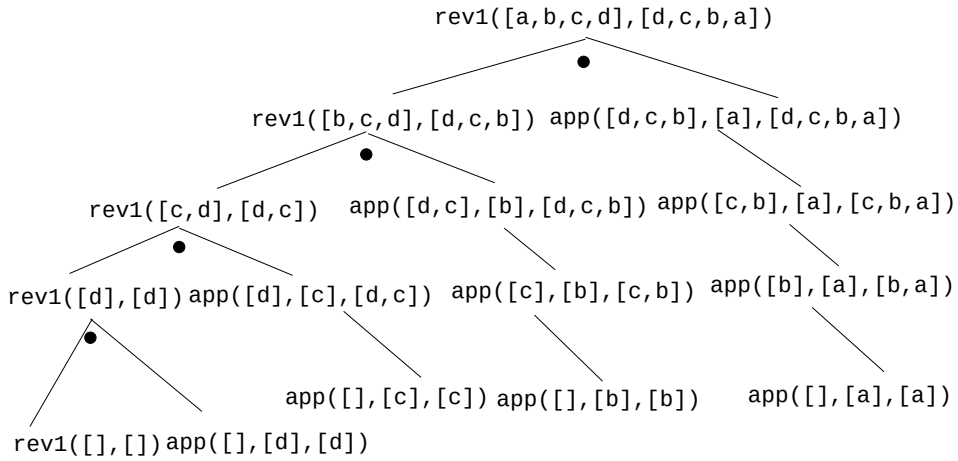
23 Prolog Lists — Reverse...

- `reverse1` works like this:
  1. Reverse the tail of the list.
  2. Append the head of the list to the reversed tail.
- `reverse2` is *linear* in the number of elements in the list.
- `reverse2` works like this:
  1. Use an accumulator pair `In` and `Out`
  2. `In` is initialized to the empty list.
  3. At each step we take one element (`X`) from the original list (`Z`) and add it to the beginning of the `In` list.
  4. When the original list (`Z`) is empty we instantiate the `Out` list to the result (the `In` list), and return this result up through the levels of recursion.

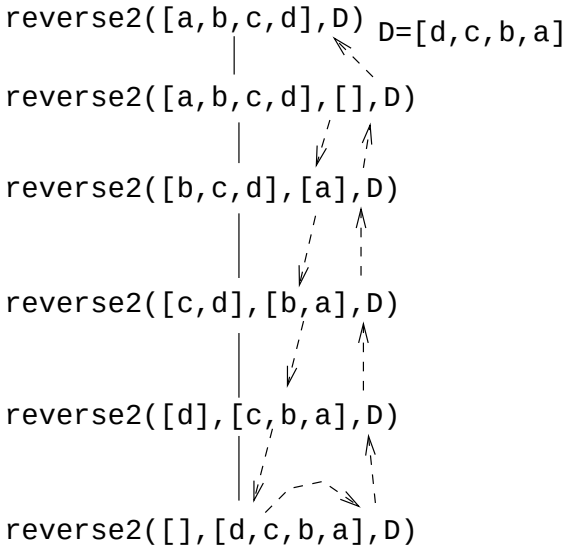
24 Prolog Lists — Reverse...

```
reverse1([], []).
reverse1([X|Q], Z) :-
   reverse1(Q, Y), append(Y, [X], Z).

reverse2(X, Y) :- reverse2(X, [], Y).
reverse2([X|Z], In, Out) :-
   reverse(Z, [X|In], Out).
reverse2([], Y, Y).
```

25 Reverse – Naive Reverse



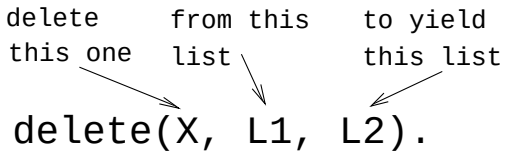
26 Reverse – Smart Reverse



27

Delete

28 Prolog Lists — Delete...



delete_one • Remove the first occurrence.

delete_all • Remove all occurrences.

delete_struct • Remove all occurrences from all levels of a list of lists.

29 Prolog Lists — Delete...

```
?- delete_one(x, [a, x, b, x], D).
      D = [a, b, x]
?- delete_all(x, [a, x, b, x], D).
      D = [a, b]
?- delete_all(x, [a, x, b, [c, x], x], D).
      D = [a, b, [c, x]]
?- delete_struct(x, [a, x, [c, x], v(x)], D).
      D = [a, b, [c], v(x)]
```

30 Prolog Lists — Delete...

delete_one

1. If X is the first element in the list then return the tail of the list.
2. Otherwise, look in the tail of the list for the first occurrence of X.

31 Prolog Lists — Delete...

delete_all

1. If the head of the list is X then remove it, and remove X from the tail of the list.
2. If X is *not* the head of the list then remove X from the tail of the list, and add the head to the resulting tail.
3. When we're trying to remove X from the empty list, just return the empty list.

32 Prolog Lists — Delete...

- Why do we test for the recursive boundary case (`delete_all(X,[],[])`) last? Well, it only happens once so we should perform the test as few times as possible.
- The reason that it works is that when the original list (the second argument) is `[]`, the first two rules of `delete_all` won't trigger. Why? Because, `[]` does not match `[H|T]`, that's why!

33 Prolog Lists — Delete...

delete_struct

1. The first rule is the same as the first rule in `delete_all`.
2. The second rule is also similar, only that we descend into the head of the list (in case it should be a list), as well as the tail.
3. The third rule is the catch-all for lists.
4. The last rule is the catch-all for non-lists. It states that all objects which are not lists (atoms, integers, structures) should remain unchanged.

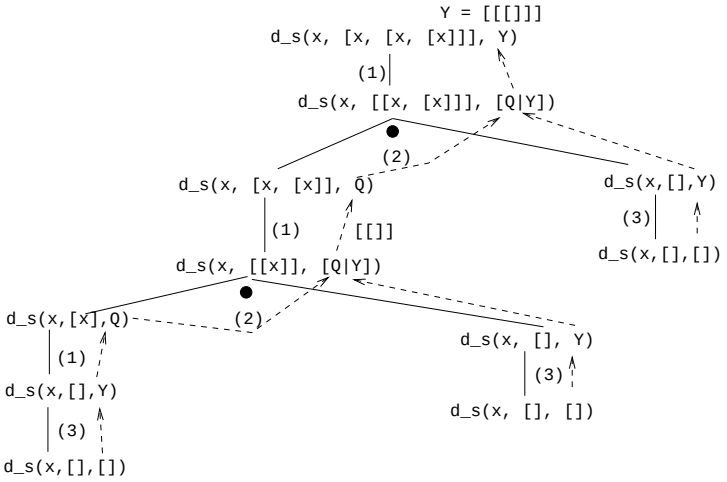
34 Prolog Lists — Delete...

```
delete_one(X,[X|Z],Z).
delete_one(X,[V|Z],[V|Y]) :-
      X \== V,
      delete_one(X,Z,Y).

delete_all(X,[X|Z],Y) :- delete_all(X,Z,Y).
delete_all(X,[V|Z],[V|Y]) :-
      X \== V,
      delete_all(X,Z,Y).
delete_all(X,[],[]).
```

35 Prolog Lists — Delete...

```
(1) delete_struct(X,[X|Z],Y) :-
      delete_struct(X, Z, Y).

(2) delete_struct(X,[V|Z],[Q|Y]):-
      X \== V,
      delete_struct(X, V, Q),
      delete_struct(X, Z, Y).

(3) delete_struct(X, [], []).
(4) delete_struct(X, Y, Y).
```

36 Prolog Lists — Delete...



37

Application: Sorting

38 Sorting – Naive Sort

```
permutation(X,[Z|V]) :-
   delete_one(Z,X,Y),
   permutation(Y,V).
permutation([],[]).

ordered([X]).
ordered([X,Y|Z]) :-
   X =< Y,
   ordered([Y|Z]).

naive_sort(X, Y) :-
   permutation(X, Y),
   ordered(Y).
```

39 Sorting – Naive Sort...

- This is an application of a Prolog cliche known as generate-and-test.

`naive_sort`

1. The `permutation` part of `naive_sort` generates one possible permutation of the input
2. The `ordered` predicate checks to see if this permutation is actually sorted.
3. If the list still isn't sorted, Prolog backtracks to the `permutation` goal to generate an new permutation, which is then checked by `ordered`, and so on.

40 Sorting – Naive Sort...

`permutation`

1. If the list is not empty we:
   (a) Delete some element `Z` from the list
   (b) Permute the remaining elements
   (c) Add `Z` to the beginning of the list

   When we backtrack (ask `permutation` to generate a new permutation of the input list), `delete_one` will delete a different element from the list, and we will get a new permutation.
2. The permutation of an empty list is the empty list.

- Notice that, for efficiency reasons, the boundary case is put *after* the general case.

41 Sorting – Naive Sort...

`delete_one` Removes the first occurrence of `X` (its first argument) from `V` (its second argument).

- Notice that when `delete_one` is called, its first argument (the element to be deleted), is an uninstantiated variable. So, rather than deleting a specific element, it will produce the elements from the input list (+ the remaining list of elements), one by one:

```
?- delete_one(X,[1,2,3,4],Y).
X = 1, Y = [2,3,4] ;
X = 2, Y = [1,3,4] ;
X = 3, Y = [1,2,4] ;
X = 4, Y = [1,2,3] ;
no.
```

42 Sorting – Naive Sort...

The proof tree in the next slide illustrates `permutation([1,2,3],V)`. The dashed boxes give variable values for each backtracking instance:

First instance: `delete_one` will select `X=1` and `Y=[2,3]`. `Y` will then be permuted into `Y'=[2,3]` and then (after having backtracked one step) `Y'=[3,2]`. In other words, we generate `[1,2,3], [1,3,2]`.

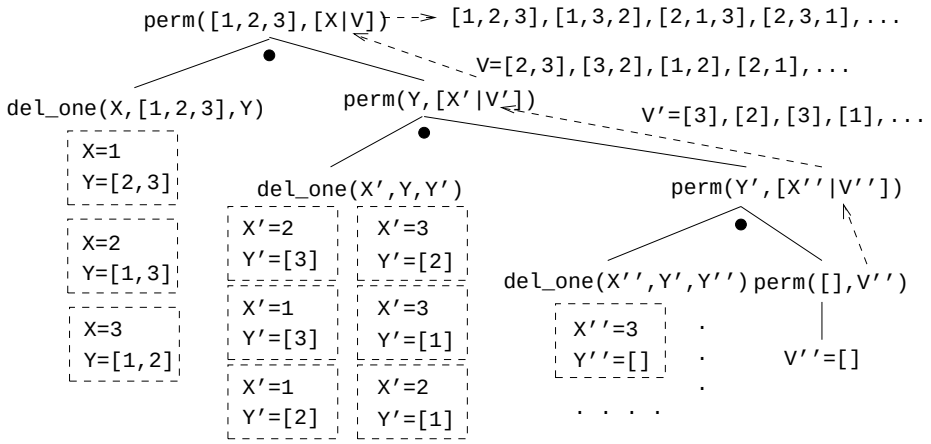
Second instance: We backtrack all the way back up the tree and select `X=2` and `Y=[1,3]`. `Y` will then be permuted into `Y'=[1,3]` and then `Y'=[3,2]`. In other words, we generate `[2,1,3], [2,3,1]`.

43 Sorting – Naive Sort...

Third instance: Again, we backtrack all the way back up the tree and select `X=3` and `Y=[1,2]`. We generate `[3,1,2], [3,2,1]`.

```
?- permutation([1,2,3],V).
V = [1,2,3] ;
V = [1,3,2] ;
V = [2,1,3] ;
V = [2,3,1] ;
V = [3,1,2] ;
V = [3,2,1] ;
no.
```

44 Permutations



45 Sorting Strings

- Prolog strings are lists of ASCII codes.

- `"Maggie" = [77,97,103,103,105,101]`

```
aless(X,Y) :-
   name(X,Xl), name(Y,Yl),
   alessx(Xl,Yl).

alessx([],[_|_]).
alessx([X|_],[Y|_]) :- X < Y.
alessx([A|X],[A|Y]) :- alessx(X,Y).
```

46

Application: Mutant Animals

47 Mutant Animals

- From *Prolog by Example*, Coelho & Cotta.
- We're given a set of words (French animals, in our case).
- Find pairs of words where the ending of the first one is the same as the beginning of the second.
- Combine the words, so as to form new "mutations".

48 Mutant Animals...

1. Find two words, `Y` and `Z`.
2. Split the words into lists of characters. `name(atom, list)` does this.
3. Split `Y` into two sublists, `Y1` and `Y2`.
4. See if `Z` can be split into two sublists, such that the prefix is the same as the suffix of `Y` (`Y2`).
5. If all went well, combine the prefix of `Y` (`Y1`) with the suffix of `Z` (`Z2`), to create the mutant list `X`.
6. Use `name` to combine the string of characters into a new atom.

49 Mutant Animals...

```
mutate(M) :-
   animal(Y), animal(Z), Y \== Z,
   name(Y,Ny), name(Z,Nz),
   append(Y1,Y2,Ny), Y1 \==[],
   append(Y2, Z2, Nz), Y2 \== [],
   append(Y1,Nz,X), name(M,X).

animal(alligator).   /* crocodile*/
animal(tortue).      /* turtle   */
animal(caribou).     /* caribou  */
animal(ours).        /* bear     */
animal(cheval).      /* horse    */
```

```
animal(vache).       /* cow      */
animal(lapin).       /* rabbit   */
```

50 Mutant Animals...

```
?- mutate(X).
   X = alligatortue ;   /* alligator+ tortue */
   X = caribours ;      /* caribou  + ours */
   X = chevalligator ;  /* cheval   + alligator*/
   X = chevalapin ;     /* cheval   + lapin */
   X = vacheval         /* vache    + cheval */
```

51

Summary

52 Prolog So Far...

- Lists are nested *structures*
- Each list node is an object
  - with functor `.` (dot).
  - whose first argument is the head of the list
  - whose second argument is the tail of the list
- Lists can be split into head and tail using `[H|T]`.
- Prolog strings are lists of ASCII codes.
- `name(X,L)` splits the atom X into the string L (or vice versa).