**Prolog Lists**

**Scheme:**

\[\text{cons}(a, \text{cons}(b, \text{cons}(c, \text{null})))\]

**Prolog:**

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

\[L = [a, b, c]\]
Prolog Lists...

?- L = .(a, .(.(1, .(2, []))), .(b, .(c, []))))
L = [a, [1, 2], b, c]
## Matching Lists – [Head | Tail]

<table>
<thead>
<tr>
<th>$A$</th>
<th>$F$</th>
<th>$A \equiv F$</th>
<th>variable subst.</th>
</tr>
</thead>
<tbody>
<tr>
<td>[]</td>
<td>[]</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>[]</td>
<td>a</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>[a]</td>
<td>[]</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>[[]]</td>
<td>[]</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>[a</td>
<td>[b, c]]</td>
<td>L</td>
<td>yes</td>
</tr>
<tr>
<td>[a]</td>
<td>[H</td>
<td>T]</td>
<td>yes</td>
</tr>
</tbody>
</table>
## Matching Lists – [Head | Tail]...

<table>
<thead>
<tr>
<th>$A$</th>
<th>$F$</th>
<th>$A \equiv F$</th>
<th>variable subst.</th>
</tr>
</thead>
<tbody>
<tr>
<td>[a, b, c]</td>
<td>[H</td>
<td>T]</td>
<td>yes</td>
</tr>
<tr>
<td>[a, [1, 2]]</td>
<td>[H</td>
<td>T]</td>
<td>yes</td>
</tr>
<tr>
<td>[[1, 2], a]</td>
<td>[H</td>
<td>T]</td>
<td>yes</td>
</tr>
<tr>
<td>[a, b, c]</td>
<td>[X, Y, c]</td>
<td>yes</td>
<td>X=a, Y=c</td>
</tr>
<tr>
<td>[a, Y, c]</td>
<td>[X, b, Z]</td>
<td>yes</td>
<td>X=a, Y=b, Z=c</td>
</tr>
<tr>
<td>[a, b]</td>
<td>[X, c]</td>
<td>no</td>
<td></td>
</tr>
</tbody>
</table>
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).
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
Prolog Lists — Member...

member1(x, [a, b, x, d])
(1) (2)
member1(x, [a|_]) member1(x, [_|[b,x,d]])
|  
\x_\nfail

member(x, [b,x,d])
(1) (2)
member1(x, [b|_]) member1(x, [_|[x,d]])
|  
\x_\nfail

\x=x

\textbf{succeed}
Prolog Lists — Append

append(L1, L2, L3).

(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 $L_3$. 
Prolog Lists — Append...

app([a, b], [1, 2], L)  \[L=[a, b, 1, 2]\]

(1) (2)

app([], L, L)

app([a|b], [1, 2], [a|L3])

fail

app([b], [1, 2], L3')

(1) (2)

app([], L, L)

app([b|[]], [1, 2], [b|L3''])

fail

app([], [1, 2], [1, 2])

succeed

[10]
Prolog Lists — Append...

\[ L = [a, b, 1, 2] \]

\[ \text{app}([a, b], [1, 2], L) \]

\[ \text{app}([a | [b]], [1, 2], [a | L3]) \]

\[ \text{app}([b | []], [1, 2], [b | L3']) \]

\[ \text{app}([], [1, 2], [1, 2]) \]

?- L = [a | L3], L3 = [b | L3'], L3' = [1, 2].
L = [a, b, 1, 2], L3 = [b, 1, 2], L3' = [1, 2]
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]`?
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`?
Prolog Lists — Using Append...

\[
\begin{align*}
\text{app}([a, b], L, [a, b, 1, 2]) \\
\text{app}([], L, L) \\
\text{app}([a | [b]], L2, [a | [b, 1, 2]]) \\
\text{app}([b], L2', [b, 1, 2]) \\
\text{app}([b | []], L2'', [b | [1, 2]]) \\
\text{app}([a, b], L, [a, b, 1, 2]) \\
\text{app}([], L, L) \\
\text{app}([b | []], L2', [b | [1, 2]]) \\
\text{app}([b | []], L2'', [b | [1, 2]]) \\
\text{app}([], [1, 2], [1, 2]) \\
\text{fail} \\
\text{fail} \\
\text{fail} \\
\text{succeed}
\end{align*}
\]
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
Prolog Lists — Using Append...

```prolog
appl([], [a, b, 1, 2])

app([], [b, 1, 2])
succeed

app([], [1, 2])
```

```
app([L1], L2, [a, b, 1, 2])
appl([a|L1], L2, [a|b, 1, 2])
appl(L1, L2', [b, 1, 2])
appl([b|L1'], L2'', [b|1, 2])
appl(L1', L2''', [1, 2])
succeed

app([], [1, 2], [1, 2])
succeed
```
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?
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]).
**Prolog Lists — Reverse**

- `reverse1` is known as *naive reverse*.
- `reverse1` is *quadratic* in the number of elements in the list.


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.
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.
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).

Prolog Lists — Reverse...
Reverse – Naive Reverse

rev1([a, b, c, d], [d, c, b, a])

rev1([b, c, d], [d, c, b]) app([d, c, b], [a], [d, c, b, a])

rev1([c, d], [d, c]) app([d, c], [b], [d, c, b]) app([c, b], [a], [c, b, a])

rev1([d], [d]) app([d], [c], [d, c]) app([c], [b], [c, b]) app([b], [a], [b, a])

app([], [c], [c]) app([], [b], [b]) app([], [a], [a])

rev1([], []) app([], [d], [d])
Reverse – Smart Reverse

\[
\text{reverse2}([a, b, c, d], D) \quad D = [d, c, b, a]
\]

\[
\text{reverse2}([a, b, c, d], [], D)
\]

\[
\text{reverse2}([b, c, d], [a], D)
\]

\[
\text{reverse2}([c, d], [b, a], D)
\]

\[
\text{reverse2}([d], [c, b, a], D)
\]

\[
\text{reverse2}([], [d, c, b, a], D)
\]
Prolog Lists — Delete...

```prolog
delete(X, L1, L2).
```

*delete_one*  ●  Remove the first occurrence.

*delete_all*  ●  Remove all occurrences.

*delete_struct*  ●  Remove all occurrences from all levels of a list of lists.
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)]
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.
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 \textit{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.
Why do we test for the recursive boundary case (\texttt{delete\_all}(X, \texttt{[]}, \texttt{[]})) 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 \texttt{[]}, the first two rules of \texttt{delete\_all} won’t trigger. Why? Because, \texttt{[]} does not match \texttt{[\texttt{H} \mid \texttt{T}]}, that’s why!
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.
delete_one(X, [X|Z], Z).
define delete_one(X, [V|Z], [V|Y]) :-
    X \== V,
    delete_one(X, Z, Y).
define delete_all(X, [X|Z], Y) :- delete_all(X, Z, Y).
define delete_all(X, [V|Z], [V|Y]) :-
    X \== V,
    delete_all(X, Z, Y).
define delete_all(X, [], []).
(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).
Prolog Lists — Delete...

\[
\begin{array}{c}
Y = [[[[]]]] \\
d_s(x, [x, [x, [x]]], Y) \\
\quad (1) \\
d_s(x, [[x, [x]]], [Q | Y]) \\
\quad (2) \\
d_s(x, [x, [x]], \bar{Q}) \\
\quad (1) \\
d_s(x, [[x]], [Q | Y]) \\
\end{array}
\]
Sorting – Naive Sort

\[ \text{permutation}(X, [Z|V]) :\]
\[ \text{delete\_one}(Z, X, Y), \]
\[ \text{permutation}(Y, V). \]
\[ \text{permutation}([], []). \]

\[ \text{ordered}([X]). \]
\[ \text{ordered}([X, Y|Z]) :\]
\[ X =\leq Y, \]
\[ \text{ordered}([Y|Z]). \]

\[ \text{naive\_sort}(X, Y) :\]
\[ \text{permutation}(X, Y), \]
\[ \text{ordered}(Y). \]
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.
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.
delete_one  Removes the first occurrence of \( x \) (its first argument) from \( V \) (its second argument).

Notice that when \( \text{delete} \_\text{one} \) 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:

\[
?- \text{delete} \_\text{one}(X, [1,2,3,4], Y).
\]

\[
X = 1, \ Y = [2,3,4] \ ;
\]

\[
X = 2, \ Y = [1,3,4] \ ;
\]

\[
X = 3, \ Y = [1,2,4] \ ;
\]

\[
X = 4, \ Y = [1,2,3] \ ;
\]

no.
The proof tree in the next slide illustrates $\text{permutation}([1,2,3], V)$. 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 backtraced 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 $[2, 1, 3], [2, 3, 1]$. 
Third instance: Again, we backtrack all the way back up the tree and select \( X = 3 \) and \( Y = [1, 2] \). We generate \([3, 1, 2], [3, 2, 1]\).

\[- \text{permutation}([1, 2, 3], V).\]

\[
\begin{align*}
V &= [1, 2, 3] ; \\
V &= [1, 3, 2] ; \\
V &= [2, 1, 3] ; \\
V &= [2, 3, 1] ; \\
V &= [3, 1, 2] ; \\
V &= [3, 2, 1] ; \\
\text{no}.
\end{align*}
\]
Permutations

perm([1,2,3],[X|V]) \rightarrow [1,2,3],[1,3,2],[2,1,3],[2,3,1],...

V=[2,3],[3,2],[1,2],[2,1],...

perm(Y,[X'|V'])

V'=[3],[2],[3],[1],...

del_one(X,[1,2,3],Y)

X=1
Y=[2,3]

X=2
Y=[1,3]

X=3
Y=[1,2]

del_one(X',Y,Y')

X'=2
Y'=[3]

X'=3
Y'=[2]

X'=1
Y'=[3]

X'=3
Y'=[1]

Y'= [2]

perm(Y',[X''|V''])

del_one(X'',Y',Y'') perm([],V'')

V''=[3,2,3,1,2,1,3,2,1,3,...]

X''=3
Y''=[]

V''=[]

...
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).
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”.
1. Find two words, $Y$ and $Z$.

2. Split the words into lists of characters. $\text{name(atom, list)}$ does this.

3. Split $Y$ into two sublists, $Y_1$ and $Y_2$.

4. See if $Z$ can be split into two sublists, such that the prefix is the same as the suffix of $Y$ ($Y_2$).

5. If all went well, combine the prefix of $Y$ ($Y_1$) with the suffix of $Z$ ($Z_2$), to create the mutant list $X$.

6. Use $\text{name}$ to combine the string of characters into a new atom.
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(alligater).  /* crocodile*/
animal(tortue).      /* turtle   */
animal(caribou).     /* caribou  */
animal(ours).        /* bear     */
animal(cheval).      /* horse    */
animal(vache).       /* cow      */
animal(lapin).       /* rabbit   */
Mutant Animals...

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