**Introduction**

Prolog Lists

---

<table>
<thead>
<tr>
<th>Haskell:</th>
<th>Prolog:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&gt; 1 : 2 : 3 : []</code></td>
<td><code>?- L = .(a, .(1, .(2, []), .(b, .(c, []))))</code></td>
</tr>
<tr>
<td><code>[1,2,3]</code></td>
<td><code>L = [a, [1, 2], b, c]</code></td>
</tr>
</tbody>
</table>

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

Unlike Haskell, Prolog lists can contain elements of arbitrary type.
Matching Lists — [Head \mid \text{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 \mid T]</td>
<td>yes</td>
<td>(H=a, T=[])</td>
</tr>
</tbody>
</table>

Matching Lists — [Head \mid \text{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 \mid T]</td>
<td>yes</td>
<td>(H=a, T=[b, c])</td>
</tr>
<tr>
<td>[a, [1, 2]]</td>
<td>[H \mid T]</td>
<td>yes</td>
<td>(H=a, T=[[1, 2]])</td>
</tr>
<tr>
<td>[[1, 2], a]</td>
<td>[H \mid T]</td>
<td>yes</td>
<td>(H=[1,2], T=[a])</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

Member

1. \text{member1}(X, [Y\mid\_]) \text{:-} X = Y.
2. \text{member1}(X, [_\mid Y]) \text{:-} \text{member1}(X, Y).

1. \text{member2}(X, [X\mid\_]).
2. \text{member2}(X, [_\mid Y]) \text{:-} \text{member2}(X, Y).

1. \text{member3}(X, [Y\mid Z]) \text{:-} X = Y; \text{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 — Append

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 L₂ onto L₁ to make L₃
   1. Let the first element of L₁ be the first element of L₃.
   2. Append L₂ onto the rest of L₁ to make the rest of L₃.
Prolog Lists — Append...

app([a, b], [1, 2], L)  
1. What’s the result of appending [1, 2] onto [a, b]?
2. Is [a, b, 1, 2] the result of appending [1, 2] onto [a, b]?
3. What do we need to append onto [a, b] to make [a, b, 1, 2]?
4. What’s the result of removing the prefix [a, b] from [a, b, 1, 2]?

append(L, [1, 2], [a, b, 1, 2])
5. What do we need to append [1, 2] onto to make [a, b, 1, 2]?
6. What’s the result of removing the suffix [1, 2] from [a, b, 1, 2]?
7. How can the list [a, b, 1, 2] be split into two lists L1 & L2?

Prolog Lists — Using Append

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

append(L, [1, 2], [a, b, 1, 2])
5. What do we need to append [1, 2] onto to make [a, b, 1, 2]?
6. What’s the result of removing the suffix [1, 2] from [a, b, 1, 2]?
7. How can the list [a, b, 1, 2] be split into two lists L1 & L2?
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
Reversing a List

1. Reverse the tail of the list.
2. Append the head of the list to the reversed tail.

reverse1 works like this:

- reverse1 is known as \textit{naive reverse}.
- reverse1 is \textit{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.

reverse2 is \textit{linear} in the number of elements in the list.

reverse2 works like this:

1. Use an accumulator pair \texttt{In} and \texttt{Out}.
2. \texttt{In} is initialized to the empty list.
3. At each step we take one element (\texttt{X}) from the original list (\texttt{Z}) and add it to the beginning of the \texttt{In} list.
4. When the original list (\texttt{Z}) is empty we instantiate the \texttt{Out} list to the result (the \texttt{In} list), and return this result up through the levels of recursion.

\begin{verbatim}
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]).
\end{verbatim}
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).
delete from this list to yield this one list

```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
?- 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)]
```

---

**delete_one**
- If X is the first element in the list then return the tail of the list.
- Otherwise, look in the tail of the list for the first occurrence of X.

**delete_all**
- If the head of the list is X then remove it, and remove X from the tail of the list.
- 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.
- 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 (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!

The first rule is the same as the first rule in delete_all.

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.

The third rule is the catch-all for lists.

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).
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,[],[]).

(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...

Application: Sorting

Sorting – Naive Sort

```prolog
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).
```

- This is an application of a Prolog cliche known as generate-and-test.
- The permutation part of naive_sort generates one possible permutation of the input.
- The ordered predicate checks to see if this permutation is actually sorted.
- If the list still isn’t sorted, Prolog backtracks to the permutation goal to generate a new permutation, which is then checked by ordered, and so on.
**permutation**

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

When we backtrack (ask *permutation* to generate a new permutation of the input list),\n*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 \textit{after} the general case.

---

**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:

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

\[X = 1, \ Y = [2,3,4] ; X = 2, \ Y = [1,3,4] ; X = 3, \ Y = [1,2,4] ; X = 4, \ Y = [1,2,3] ; \text{no.}\]

---

**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 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 \([2,1,3], [2,3,1]\).

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

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

\[V = [1,2,3] ; V = [1,3,2] ; V = [2,1,3] ; V = [2,3,1] ; V = [3,1,2] ; V = [3,2,1] ; \text{no.}\]
Permutations

**Permutations**

- \( V' = [3], [2], [3], [1], ... \)
- \( X'' = 3 \)
- \( X'' = 3 \)
- \( Y'' = [] \)
- \( X = 1 \)
- \( Y = [2, 3] \)
- \( X = 2 \)
- \( Y = [1, 3] \)
- \( X = 3 \)
- \( Y = [1, 2] \)
- \( \text{del_one}(X'', Y'') \)
- \( \text{perm}(Y, [X' | V'']) \)
- \( V'' = [2, 3], [3, 2], [1, 2], [2, 1], ... \)

**Sorting Strings**

- Prolog strings are lists of ASCII codes.
- "Maggie" = [77, 97, 103, 103, 105, 101]
- \( \text{aless}(X, Y) :- \)
  - \( \text{name}(X, X1), \text{name}(Y, Y1), \)
  - \( \text{alessx}(X1, Y1). \)
- \( \text{alessx}([], []). \)
- \( \text{alessx}([X|\_], [Y|\_]) :- X < Y. \)
- \( \text{alessx}([A|X], [A|Y]) :- \text{alessx}(X, Y). \)

**Application: 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”.

**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”.

Sorting Strings

- Prolog strings are lists of ASCII codes.
- "Maggie" = [77, 97, 103, 103, 105, 101]
- \( \text{aless}(X, Y) :- \)
  - \( \text{name}(X, X1), \text{name}(Y, Y1), \)
  - \( \text{alessx}(X1, Y1). \)
- \( \text{alessx}([], []). \)
- \( \text{alessx}([X|\_], [Y|\_]) :- X < Y. \)
- \( \text{alessx}([A|X], [A|Y]) :- \text{alessx}(X, Y). \)
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.

```
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 */
```

?- mutate(X).
   X = alligator+ tortue ; /* alligator+tortue */
   X = caribou + ours ;     /* caribou + ours */
   X = cheval+ alligator ;  /* cheval + alligator */
   X = cheval+ lapin ;      /* cheval + lapin */
   X = vache + cheval ;     /* 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).