Lists

York University
Department of Computer Science and Engineering

Overview

- Definition and representation of Lists in Prolog
  - Dot functor
- Examples of recursive definition of predicates
  - islist,
  - member, delete
  - append, multiple,
  - prefix, suffix, sublist

[ref.: Clocksin- Chap.3 and Nilsson- Chap. 7]

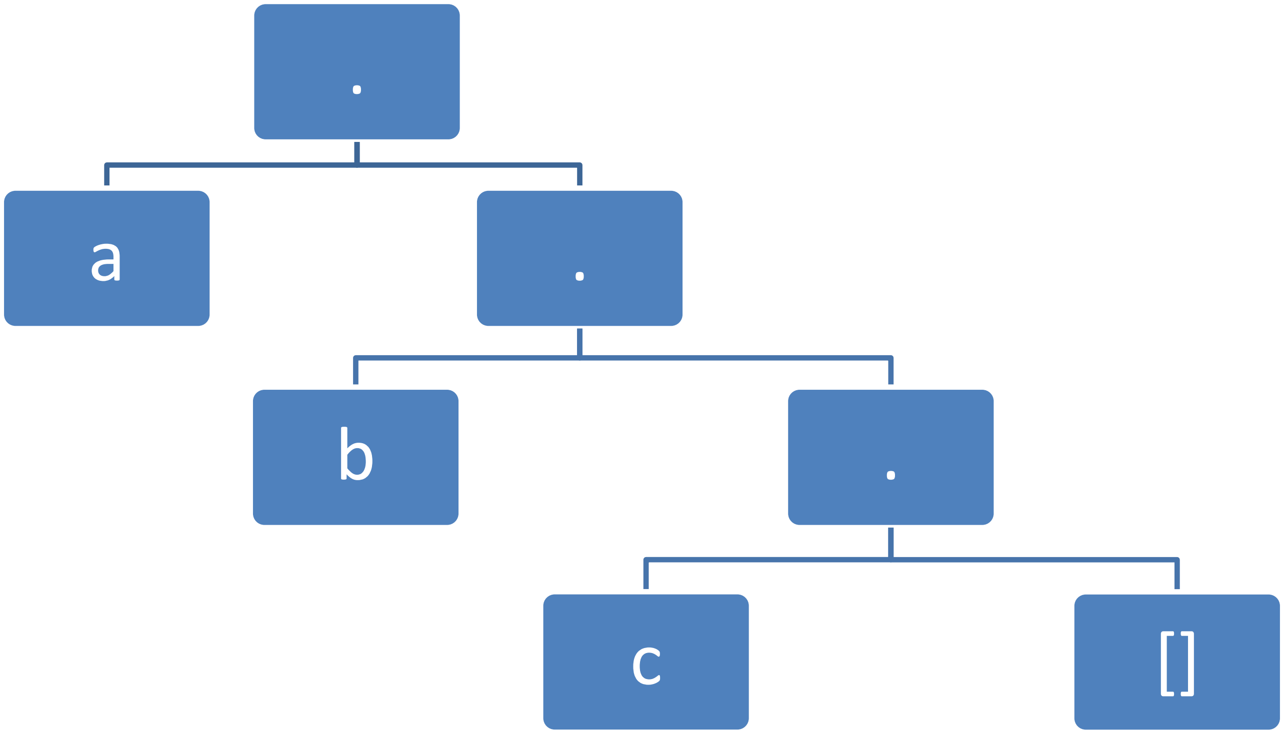
[also Prof. Gunnar Gotshalks' slides]

Lists

- A <u>list:</u>
  - is an <u>ordered</u> sequence of elements that can have any length.
  - List notation in Prolog: [a, b, c, d, ...]
  - Either an empty list [] or it has a <u>head</u> X and a <u>tail</u> L represented as [X|L] where X is a list item and L is a list.
- The <u>dot:</u>
  - is a functor for representing lists with two arguments, the head and the tail of a list
  - A list of one element [a] is [a| []] implemented in Prolog as .(a, [])
  - [a, b] is .(a, .(b, []))
- Note [a, b, c] is not the same as [a, [b,c]]

Lists (cont.)

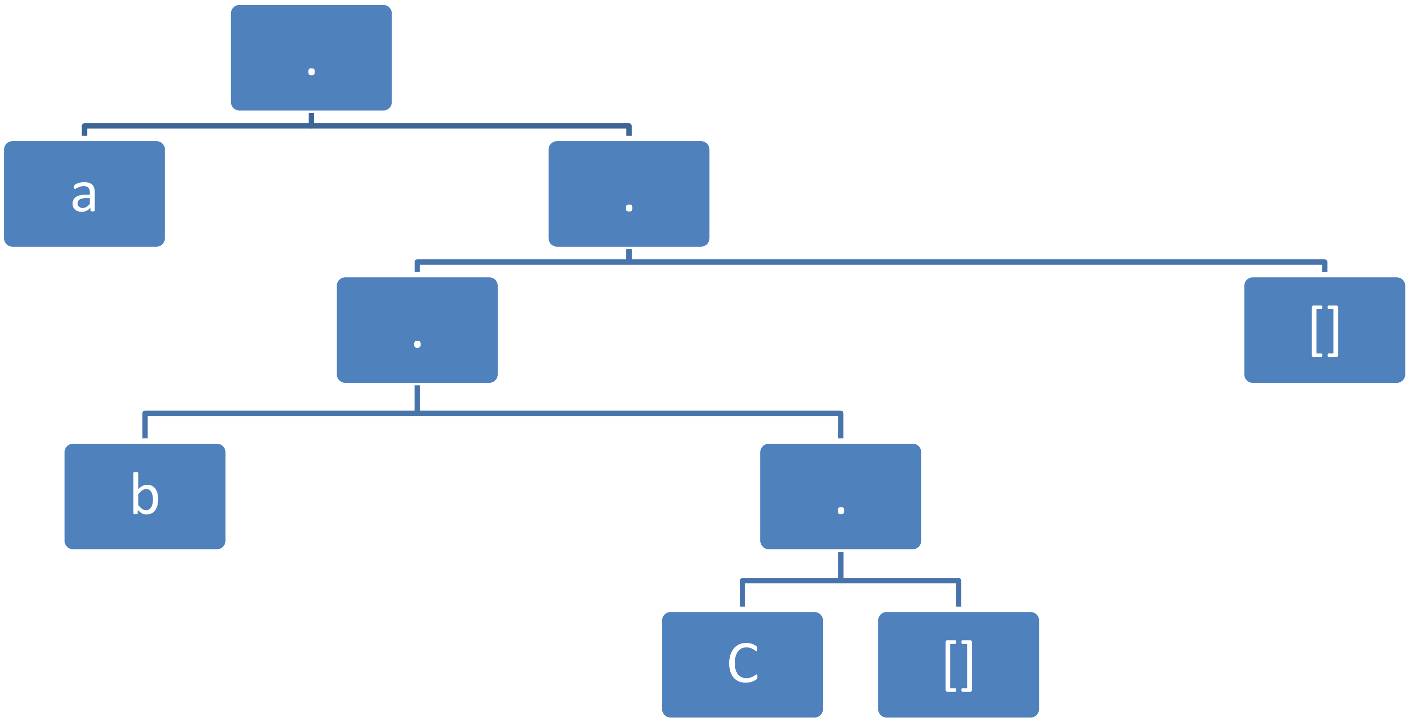
[a, b, c] is .(a, .(b, .(c, [])))



Lists (cont.)

[a, [b, c]] is .(a, L)

where L is [[b,c]] having [b,c] as its head and [] as tail



Examples

- Write the Prolog definition for being a list.

```
islist([]).
islist([Head|Tail]) :- islist(Tail).
```

- Write the Prolog definition for being a member of a list.

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

Examples (cont.)

```
:- member(3, [2, 3, 4, 5]).
true
```

```
:- member(3, [2, [3, 4], 5]).
false
```

Our definition does not consider members of members (nested lists)

```
:- member(X, [1, 2]).
X = 1 ;
X = 2 ;
false
```

Unlike other programming languages, inputs can be unknowns

```
:- member(2, L).
L = [2|_] ;
L=[_, 2 | _];
...
```

Note the <u>recursive </u>definition of member

Recursive Definition

- Example:

  member(X, [X|L]). : boundary condition

  member(X, [Y|L]) :- member(X,L). : recursive case

- Note in a recursive definition, the problem must get smaller in each recursion to guarantee convergence

  member(X,L) is <u>a smaller problem</u> than member(X, [Y|L])

- Prolog does not need to remember Y, so we can use the anonymous variable _ instead:

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

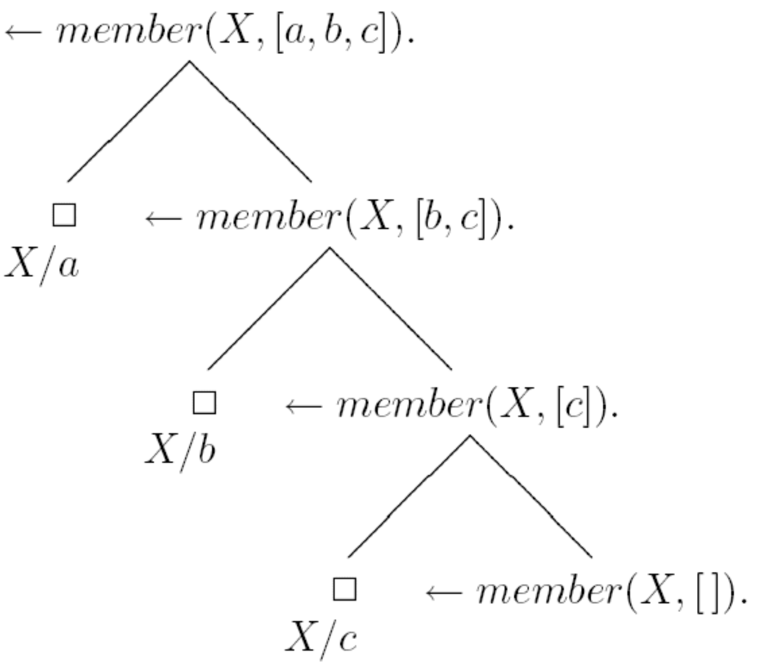
Recursive Search

- Example:

  member(X, [X|L]).

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

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



Delete

- delete(X, L1, L2) is true if L2 is the result of deleting X from L1 (just once).
  - For example: delete(5, [1, 5, 4, 2], [1, 4, 2]).

delete(X, [X|L], L).
delete(X, [Y|L], [Y|L1]) :- delete(X, L, L1).

Append

- Join two lists:
  Example: append([1,2], [3,4], [1,2,3,4])

```
append([], L, L).                                   : boundary condition
append([X|L1], L2, [X|L3]) :- append(L1, L2, L3).   : recursive case
```

a smaller problem

- Possible Queries:
  [Nilsson]
  :- append([a, b], [c, d], [a, b, c, d]).

  true

  :- append([a, b], [c, d], X).

  X=[a, b, c, d]

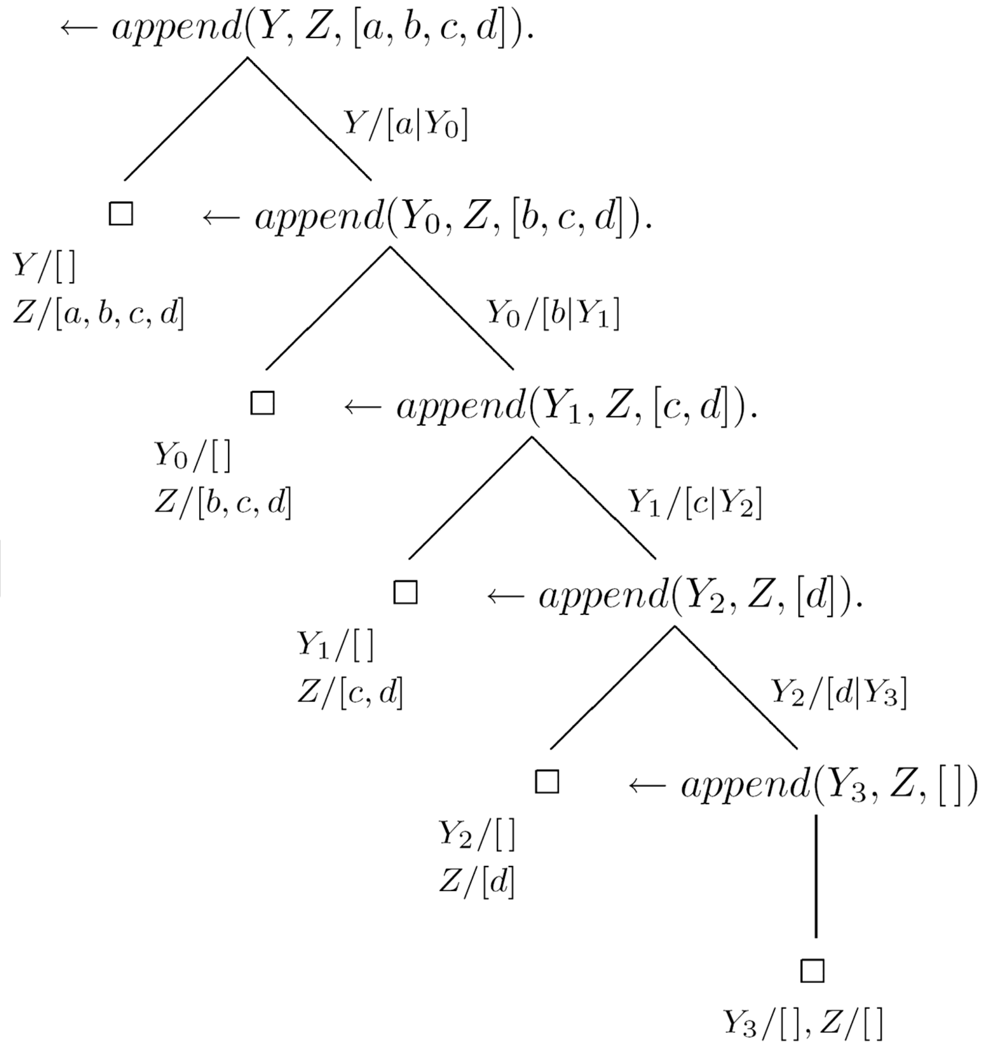
  or even
  :- append(Y, Z, [a, b, c, d]).

Search tree for append query

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

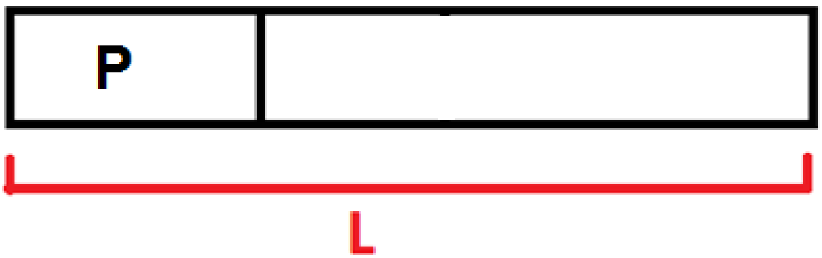
```
:- append(Y, Z, [a, b, c, d]).
```

$$
\begin{array}{ll}
Y = [\,] & Z = [a, b, c, d] \\
Y = [a] & Z = [b, c, d] \\
Y = [a, b] & Z = [c, d] \\
Y = [a, b, c] & Z = [d] \\
Y = [a, b, c, d] & Z = [\,]
\end{array}
$$



Prefix with append

- The list P is a prefix of L, if L can be obtained by appending P to another list.



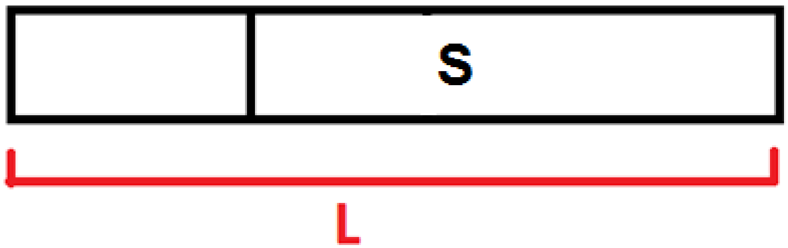
- Write prefix(P,L) which is true if P is a prefix of L.

prefix(P, L):- append(P, _, L).

  – Is [] a prefix of L?

Suffix with append

- The List S is a suffix (or postfix) of L if L can be obtained by appending some other list with S.



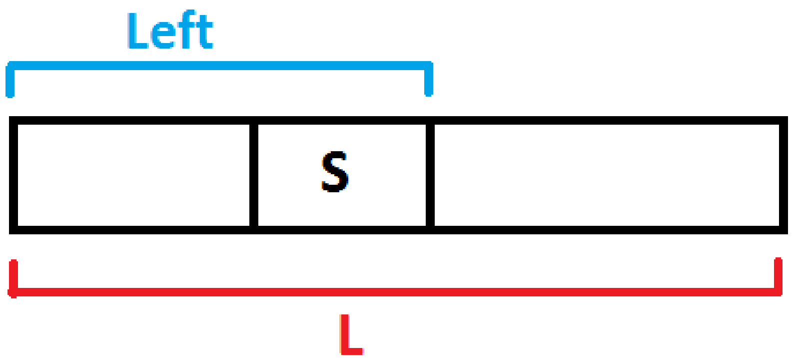
- Write suffix(S,L) which is true if S is a suffix of L

suffix(S,L):- append(_, S, L).

- <u>Exercise:</u> Try writing prefix and suffix without using append.

More Examples with append

- sublist(S,L) is true if S is a sublist of L
  - in other words, S is the suffix of a prefix

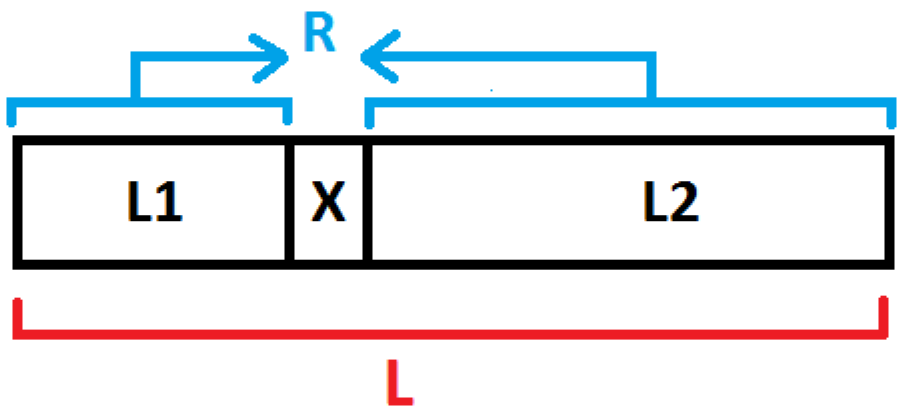


  - Using append(..):

sublist(S,L):- append(_, S, Left), append(Left, _, L).

More Examples with append

- Re-writing delete(X,L1,L2) with append(..):



delete(X, L, R):- append(L1,[X|L2],L), append(L1, L2, R).

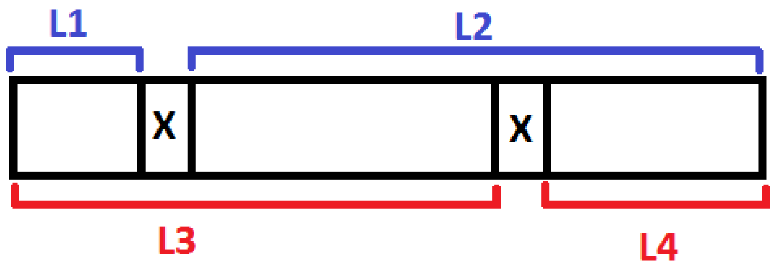
Example: multiple occurrences in a list

- multiple(L) is true if L is a list with multiple occurrences of some element [Nilsson]:

  multiple([Head|Tail]):- member(Head, Tail).
  multiple([Head|Tail]):- multiple(Tail).

  – Writing multiple(..) using append(..)

  multiple(L) :- append(L1, [X|L2], L), append(L3, [X|L4], L).



What is missing in definition of multiple(..)? How can it be corrected?

Append is expensive!

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

- The complexity of appending two lists, L1 and L2, is O(n) where n is the length of the first list.
- Consider reverse(L, R) defined as:
  reverse([], []).
  reverse([X|L], R) :- reverse(L, L1), append (L1, [X], R).
- Complexity of reverse(..) is $O(n^2)$ where n is the length of L.