CSCI-344
Programming Language Concepts (Section 3)

Lecture 22

Prolog: Lists

Instructor: Hossein Hojjat

October 31, 2016

Where we are

Done:

- Prolog
- Logical Programming

This session:

- Lists in Prolog

List

- A list is a finite sequence of elements
- Elements are put within square brackets, separated by commas
- Empty list: `[]`
- Any Prolog term can be element of a list

Examples:

- `[2,3,4]`
- `[Bob,[teaches(Prof,csci344),[]],Z]`
- `[(a(X) :- b(X),c(X))]`
- `[3,sophia,studies(mia,csci344),[]]`

List

List is

- nil (`[]`) or
- a cons cell (`[Head|Tail]`) in which `Tail` is a proper list

Scheme list operations in Prolog:

```
-> car([X|L],X). /* head (car) of [X|L] is X */
-> cdr([X|L],L). /* tail (cdr) of [X|L] is L */
-> cons(X,R,[X|R]). /* construct a list
                       by adding X to the front of R */
```

List

- You don't need `car` and `cdr` functions to split a list
- Prolog can unify a list with the pattern of form `[H|T]`

```
-> L=[1,2,3,4,5].
-> [query].
?- L=[H|T].
H = 1
T = [2, 3, 4, 5].
```

- Extract the third element of the list:

```
?- L=[_,_,X|_].
X = 3.
?- L=[_|[_|[X|_]]].
X = 3.
```

- `[a]` is a shorthand for `[a|[]]`

member

Predicate `member` succeeds when first argument is a member of the list in the second argument

1. `X` is a member of a list with first element `X`
2. `X` is a member of a list with tail `Y`, if `X` is a member of `Y`

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

```
?- member(3,[1,2,3,4]).
yes
?- member(5,[1,2,3,4]).
no
?- member(X,[1,2,3]).
X = 1;
X = 2;
X = 3.
```

Overlapping Lists

- Overlapping lists are those which have some elements in common

```
overlap(X,Y) :- member(M,X), member(M,Y).
```

```
?- overlap([1,2,3,4],[2,4,6]).
yes
```

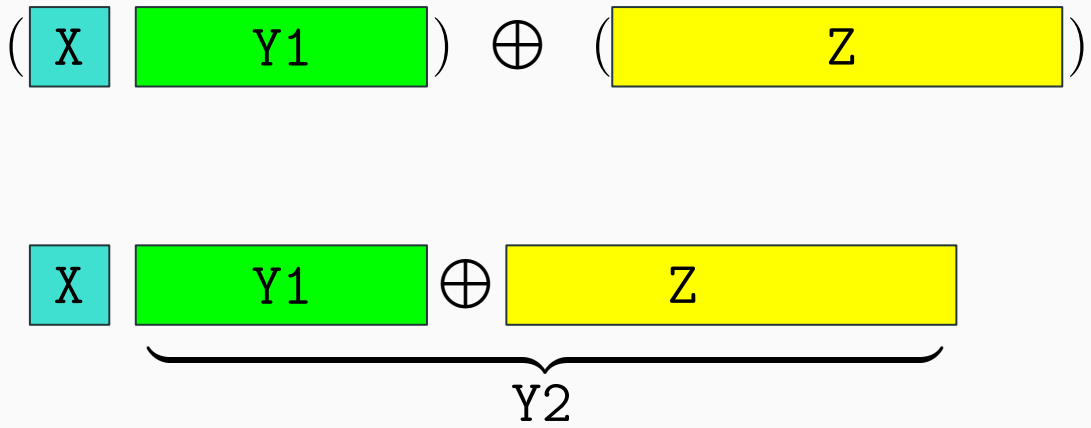
- There is an infinite number of solutions for `X` :

```
?- overlap([1],X).
X = [1|__7135];
X = [__8138, 1|__7143];
X = [__8138, __8146, 1|__7151];
X = [__8138, __8146, __8154, 1|__7159];
X = [__8138, __8146, __8154, __8162, 1|__7167];
...
```

append

- Predicate `append` succeeds when the third argument is a list formed by appending the first two lists

```
append([], X, X).
append([X|Y1],Z,[X|Y2]) :- append(Y1, Z, Y2).
```



append

- Predicate `append` succeeds when the third argument is a list formed by appending the first two lists

```
append([], X, X).
append([X|Y1],Z,[X|Y2]) :- append(Y1, Z, Y2).
```

```
append([1,2],[3],L)
```

append

- Predicate `append` succeeds when the third argument is a list formed by appending the first two lists

```
append([], X, X).
append([X|Y1],Z,[X|Y2]) :- append(Y1, Z, Y2).
```

`append([1,2],[3],L)`

↓

`append([1|[2]],[3],[1|L1]):-append([2],[3],L1)` `L=[1|L1]`

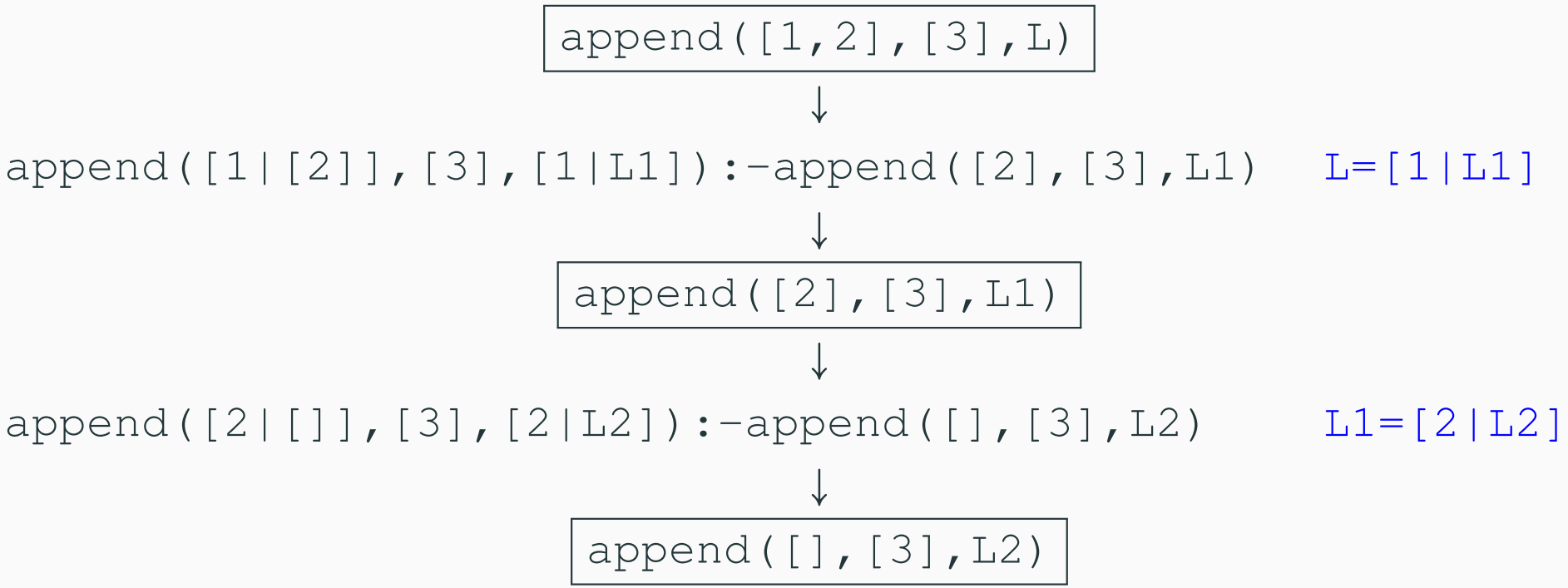
↓

`append([2],[3],L1)`

append

- Predicate `append` succeeds when the third argument is a list formed by appending the first two lists

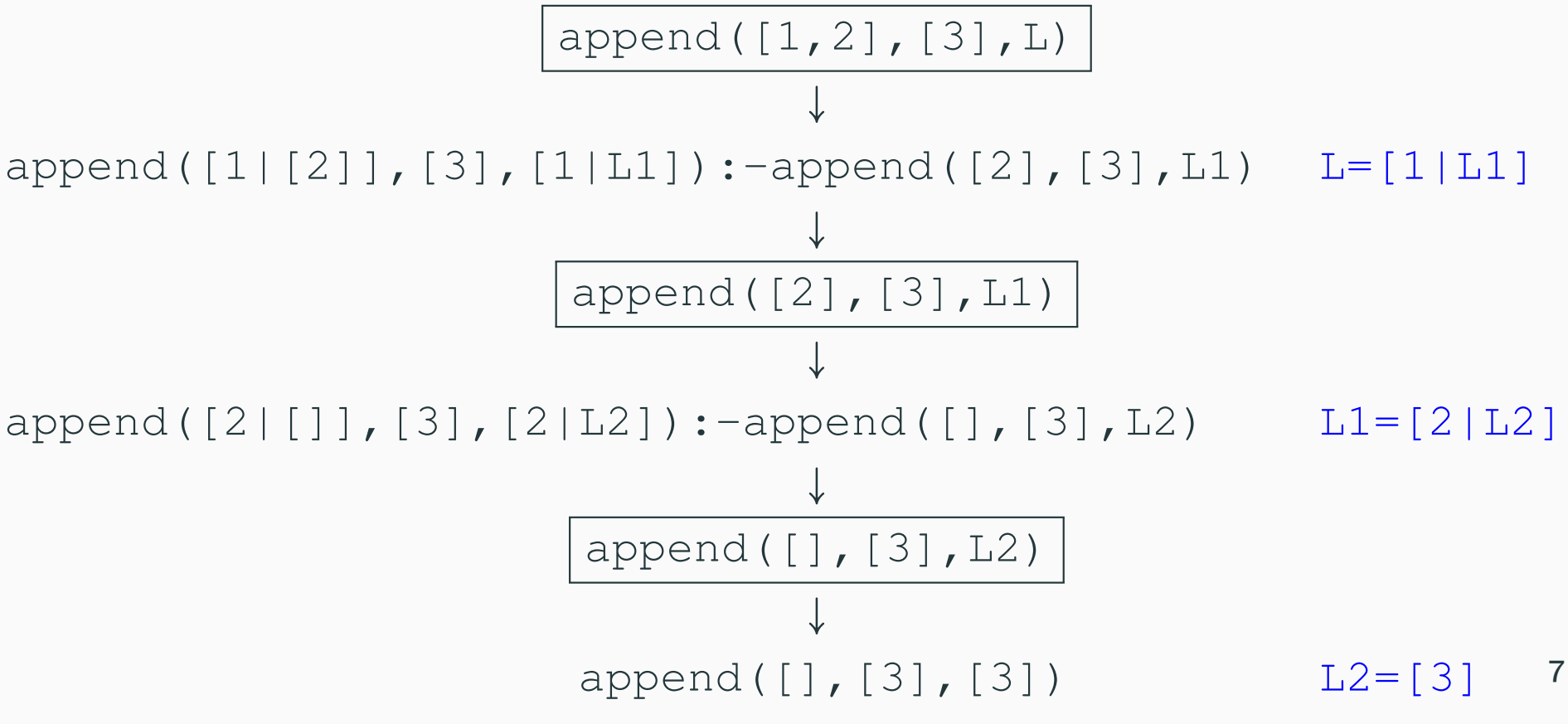
```
append([], X, X).
append([X|Y1],Z,[X|Y2]) :- append(Y1, Z, Y2).
```



append

- Predicate `append` succeeds when the third argument is a list formed by appending the first two lists

```
append([], X, X).
append([X|Y1],Z,[X|Y2]) :- append(Y1, Z, Y2).
```



append

- Predicate `append` succeeds when the third argument is a list formed by appending the first two lists

```
append([], X, X).
append([X|Y1],Z,[X|Y2]) :- append(Y1, Z, Y2).
```

- We can define `member` using `append`:

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

append

- Predicate `append` succeeds when the third argument is a list formed by appending the first two lists

```
append([], X, X).
append([X|Y1],Z,[X|Y2]) :- append(Y1, Z, Y2).
```

Question: Is it possible to use the following clause as base case?

```
append(X, [], X).
```

reverse

- Predicate `reverse`:
  - first argument holds the part of the list that remains to be reversed,
  - second argument is an accumulating parameter: initially `[]`, accumulates answer until the computation is finished

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

```
reverse([1,2,3],[],X)
          ↓
reverse([2,3],[1],X)
          ↓
reverse([3],[2,1],X)
          ↓
reverse([],[3,2,1],X)
          ↓
    yes, X=[3,2,1]
```

reverse

- Predicate `reverse`:
  - first argument holds the part of the list that remains to be reversed,
  - second argument is an accumulating parameter: initially `[]`, accumulates answer until the computation is finished

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

- How can we write a two-parameter version of `reverse` that does not use the accumulating parameter?

reverse

- Predicate `reverse`:
  - first argument holds the part of the list that remains to be reversed,
  - second argument is an accumulating parameter: initially `[]`, accumulates answer until the computation is finished

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

- How can we write a two-parameter version of `reverse` that does not use the accumulating parameter?

```
rev(L,R) :- reverse(L,[],R).
```

reverse

- Predicate `reverse`:
  - first argument holds the part of the list that remains to be reversed,
  - second argument is an accumulating parameter: initially `[]`, accumulates answer until the computation is finished

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

- How can we write a two-parameter version of `reverse` that does not use the accumulating parameter?

```
rev(L,R) :- reverse(L,[],R).
```

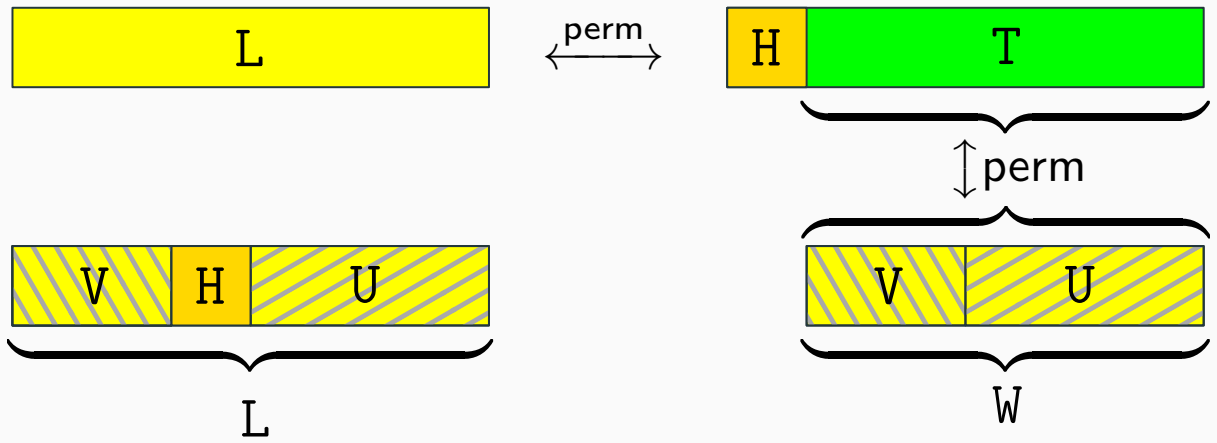
- or,

```
rev([],[]).
rev([X|Xs],R) :- rev(Xs,T),
append(T,[X],R).
```

Permutation

- `perm` succeeds if arguments are rearrangement of each other
- In most languages generating permutations is non-trivial
- Permutations can be defined succintly in Prolog

```
perm([],[]).
perm(L,[H|T]) :- append(V,[H|U],L),
                 append(V,U,W), perm(W,T).
```



Permutation

- `perm` succeeds if arguments are rearrangement of each other
- In most languages generating permutations is non-trivial
- Permutations can be defined succintly in Prolog

```
perm([],[]).
perm(L,[H|T]) :- append(V,[H|U],L),
                 append(V,U,W), perm(W,T).
```

```
?- perm([1,2,3],X).
X = [1, 2, 3];
X = [1, 3, 2];
X = [2, 1, 3];
X = [2, 3, 1];
X = [3, 1, 2];
X = [3, 2, 1].
```

Sorted Lists

- Predicate `sorted` succeeds if argument is sorted
- Empty list and list with a single element are sorted
- Longer lists are sorted if every two elements are ordered

```
sorted([]).
sorted([_]).
sorted([X,Y|T]) :- X =< Y, sorted([Y|T]).
```

Naïve Sort

Permutation Sort:

- Repeatedly generate a permutation of the numbers
- Until it happens that the result is sorted

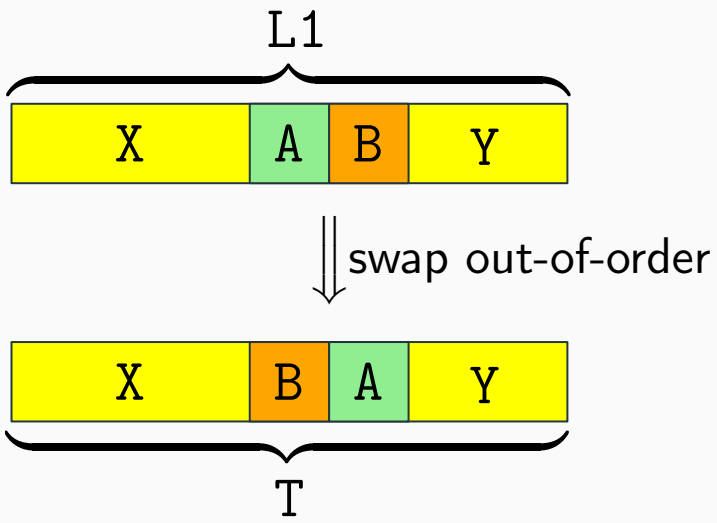
```
naiveSort(L1,L2) :-  perm(L1,L2), sorted(L2).
```

- This technique is very inefficient
- Undirected: does not aim toward a correct ordering

Bubble Sort

- Search for pair of adjacent values that are out-of-order
- If there is such a pair, swap those values

```
bubbleSort(L,L) :- sorted(L).
bubbleSort(L1,L2) :-
 append(X,[A,B|Y],L1), A > B,
 append(X,[B,A|Y],T),
 bubbleSort(T,L2).
```



Sudoku Game

1		3	
	3		
		4	
	4		2

Goal:

- Fill all empty squares such that:
- Numbers 1 to 4 appear exactly once in each row, column and 2x2 box

Sudoku Game

X11	X12	X13	X14
X21	X22	X23	X24
X31	X32	X33	X34
X41	X42	X43	X44

- Each row must be a permutation of [1,2,3,4]
- Each column must be a permutation of [1,2,3,4]
- Each 2x2 box must be a permutation of [1,2,3,4]

Sudoku Game

```
sudoku([[X11,X12,X13,X14],[X21,X22,X23,X24],
        [X31,X32,X33,X34],[X41,X42,X43,X44]]) :-
/*rows*/
perm([X11,X12,X13,X14],[1,2,3,4]),
perm([X21,X22,X23,X24],[1,2,3,4]),
perm([X31,X32,X33,X34],[1,2,3,4]),
perm([X41,X42,X43,X44],[1,2,3,4]),
/*cols*/
perm([X11,X21,X31,X41],[1,2,3,4]),
perm([X12,X22,X32,X42],[1,2,3,4]),
perm([X13,X23,X33,X43],[1,2,3,4]),
perm([X14,X24,X34,X44],[1,2,3,4]),
/*boxes*/
perm([X11,X12,X21,X22],[1,2,3,4]),
perm([X13,X14,X23,X24],[1,2,3,4]),
perm([X31,X32,X41,X42],[1,2,3,4]),
perm([X33,X34,X43,X44],[1,2,3,4]).
```

Sudoku Game

1	2	3	4
4	3	2	1
2	1	4	3
3	4	1	2

```
?- sudoku([[1,X12,3,X14],[X21,3,X23,X24],
           [X31,X32,4,X34],[X41,4,X43,2]]).
```

```
X12 = 2    X14 = 4    X21 = 4    X23 = 2   X24 = 1
X31 = 2    X32 = 1    X34 = 3    X41 = 3   X43 = 1
```

Summary

This Lecture

- List Manipulation in Prolog

Next Lecture

- Cuts