U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

DEPARTMENT OF INFORMATICS ENGINEERING

U.PORTO
FC FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

DEPARTMENT OF COMPUTER SCIENCE

Functional and Logic Programming

Bachelor in Informatics and Computing Engineering
2024/2025 – 1st Semester

Arithmetic, Recursion and Lists

Daniel Castro Silva
dcs@fe.up.pt

Agenda

- Arithmetic
- Recursion
  - Recursion
    - Recursion
      - Recursion
        - Recursion
          - Recursion
- Lists

Arithmetic

- Arithmetic expressions are not evaluated immediately
  - Example: A = 4+2 unifies A with the term +(4, 2), not the value 6
- The *is* predicate can be used to evaluate an arithmetic expression

```
| ?- A = 4+2.
A = 4+2 ?
yes
| ?- B is 4+2.
B = 6 ?
yes
| ?- 6 is 4+2.
yes
| ?- 4+2 is 4+2.
no
```

  - The right-side of *is* needs to be instantiated

```
| ?- C is 4+B.
! Instantiation error in argument 2 of (is)/2
! goal:  _419 is 4+_427
```

See section 4.7 of the SICStus Manual for more information on Arithmetic

Arithmetic

- Arithmetic expressions can be compared for (in)equality
  - Expr1 =:= Expr2 evaluates both expressions and if they are equal
  - Expr1 =\= Expr2 evaluates both expressions and if they are different
  - Comparison

    E1 < E2 E1 > E2 E1 =< E2 E1 >= E2

- Prolog can also compare and order terms

  T1 @< T2 T1 @> T2 T1 @=< T2 T1 @>= T2
  - Term1 == Term2 verifies whether the two terms are literally identical
  - Term1 \== Term2 checks if the two terms are not literally identical

Arithmetic

- There are several functions available
  - X+Y, X-Y, X*Y, X/Y (float quotient)
  - X//Y is the integer quotient, truncated towards 0
  - X div Y is the integer quotient (rounded down)
  - X rem Y is integer remainder: X-Y*(X//Y)
  - X mod Y is integer remainder: X-Y*(X div Y)
  - Many other functions
    - round(X), truncate(X), floor(X), ceiling(X)
    - abs(X), sign(X), min(X, Y), max(X, Y)
    - sqrt(X), log(X), exp(X), X ** Y, X ^ Y
    - sin(X), cos(X), tan(X), ...

```
| ?- A is 5 // 2.
A = 2 ?
yes
| ?- A is -5 // 2.
A = -2 ?
yes
| ?- A is 5 div 2.
A = 2 ?
yes
| ?- A is -5 div 2.
A = -3 ?
yes
| ?- A is 5 rem 2.
A = 1 ?
yes
| ?- A is -5 rem 2.
A = -1 ?
yes
| ?- A is 5 mod 2.
A = 1 ?
yes
| ?- A is -5 mod 2.
A = 1 ?
yes
```

Natural Numbers

- Arithmetic in Prolog deviates from pure Logic Programming
  - It is, however, necessary for efficiency

- A more '*logical*' representation of (natural) numbers
  - 0 is natural
  - The successor of X - *s(X)* - is natural if X is natural
    - 0, s(0), s(s(0)), s(s(s(0))), ...

```
natural_number(0).
natural_number(s(X)):- natural_number(X).
```

Adding Natural Numbers

- Addition can then be seen as a ternary relation

```
% plus(X, Y, Z): X + Y = Z

plus(0, X, X):-
      natural_number(X).

plus(s(X), Y, s(Z)):-
      plus(X,Y,Z).
```

```
| ?- plus( s(s(0)), s(0), Z).
Z = s(s(s(0))) ?
yes
| ?- plus( s(s(0)), Y, s(s(s(0)))).
Y = s(0) ?
yes
| ?- plus( X, s(0), s(s(s(0)))).
X = s(s(0)) ?
yes
| ?- plus( X, Y, s(s(0))).
X = 0,
Y = s(s(0)) ? ;
X = s(0),
Y = s(0) ? ;
X = s(s(0)),
Y = 0 ? ;
no
```

Agenda

- Arithmetic
- Recursion
  - Recursion
    - Recursion
      - Recursion
        - Recursion
          - Recursion
- Lists

Recursion

- Some relations are recursive

```
ancestor(X, Y):-                % X is an ancestor of Y
      parent(X, Y).             % if X is a parent of Y

ancestor(X, Y):-                % X is an ancestor of Y
      parent(X, Z),             % if X is a parent of Z
      ancestor(Z, Y).           % and Z is an ancestor of Y
```

- Recursion is based on the inductive proof
  - One or more base clauses
  - One or more recursion clauses

The order of clauses and goals may influence performance, or even cause infinite computations

Recursion

- Example: sum all numbers between 1 and N

```
sumN(0, 0).                          % Base clause

sumN(N, Sum):- N > 0,                % Guard - make sure we don't
                                     %    have infinite recursion
               N1 is N-1,
               sumN(N1, Sum1),       % Recursive call
               Sum is Sum1 + N.
```

Recursion

- Example: sum all numbers between 1 and N

```
sumN(0, 0).

sumN(N, Sum):- N > 0,
                  N1 is N-1,
                  sumN(N1, Sum1),
                  Sum is Sum1 + N.
```

```
| ?- sumN(2, Sum).
        1      1 Call: sumN(2,_925) ?
        2      2 Call: 2>0 ?
        2      2 Exit: 2>0 ?
        3      2 Call: _1935 is 2-1 ?
        3      2 Exit: 1 is 2-1 ?
        4      2 Call: sumN(1,_1955) ?
        5      3 Call: 1>0 ?
        5      3 Exit: 1>0 ?
        6      3 Call: _6589 is 1-1 ?
        6      3 Exit: 0 is 1-1 ?
        7      3 Call: sumN(0,_6609) ?
?       7      3 Exit: sumN(0,0) ?
        8      3 Call: _1955 is 0+1 ?
        8      3 Exit: 1 is 0+1 ?
?       4      2 Exit: sumN(1,1) ?
        9      2 Call: _925 is 1+2 ?
        9      2 Exit: 3 is 1+2 ?
?       1      1 Exit: sumN(2,3) ?
Sum = 3 ?
```

Tail Recursion

- Tail Recursion can increase efficiency
  - Add a new argument to the predicate: the accumulator
  - Make the recursive call the last call

```
sumN(N, Sum):- sumN(N, Sum, 0).          % Encapsulate
sumN(0, Sum, Sum).                       % Base case – the result is
                                         %     in the accumulator

sumN(N, Sum, Acc):- N > 0,
                    N1 is N-1,
                    Acc1 is Acc + N,
                    sumN(N1, Sum, Acc1).       % Recursive call is now
                                               %   the last sub-goal
```

To increase efficiency, we actually need to add a *cut* in the base clause - we'll see this operator next week

Tail Recursion

```
| ?- trace, sumN(2, S), notrace.
% The debugger will first creep -- showing everything
        1      1 Call: sumN(2,_941) ?
        2      2 Call: 2>0 ?
        2      2 Exit: 2>0 ?
        3      2 Call: _2067 is 2-1 ?
        3      2 Exit: 1 is 2-1 ?
        4      2 Call: sumN(1,_2087) ?
        5      3 Call: 1>0 ?
        5      3 Exit: 1>0 ?
        6      3 Call: _6721 is 1-1 ?
        6      3 Exit: 0 is 1-1 ?
        7      3 Call: sumN(0,_6741) ?
?       7      3 Exit: sumN(0,0) ?
        8      3 Call: _2087 is 0+1 ?
        8      3 Exit: 1 is 0+1 ?
?       4      2 Exit: sumN(1,1) ?
        9      2 Call: _941 is 1+2 ?
        9      2 Exit: 3 is 1+2 ?
?       1      1 Exit: sumN(2,3) ?
       10      1 Call: notrace ?
% The debugger is switched off
S = 3 ?
yes
```

```
| ?- trace, sumN(2, S, 0), notrace.
% The debugger will first creep -- showing
        1      1 Call: sumN(2,_941,0) ?
        2      2 Call: 2>0 ?
        2      2 Exit: 2>0 ?
        3      2 Call: _2111 is 2-1 ?
        3      2 Exit: 1 is 2-1 ?
        4      2 Call: _2129 is 0+2 ?
        4      2 Exit: 2 is 0+2 ?
        5      2 Call: sumN(1,_941,2) ?
        6      3 Call: 1>0 ?
        6      3 Exit: 1>0 ?
        7      3 Call: _8679 is 1-1 ?
        7      3 Exit: 0 is 1-1 ?
        8      3 Call: _8697 is 2+1 ?
        8      3 Exit: 3 is 2+1 ?
        9      3 Call: sumN(0,_941,3) ?
        9      3 Exit: sumN(0,3,3) ?
        5      2 Exit: sumN(1,3,2) ?
        1      1 Exit: sumN(2,3,0) ?
       10      1 Call: notrace ?
% The debugger is switched off
S = 3 ?
yes
```

Agenda

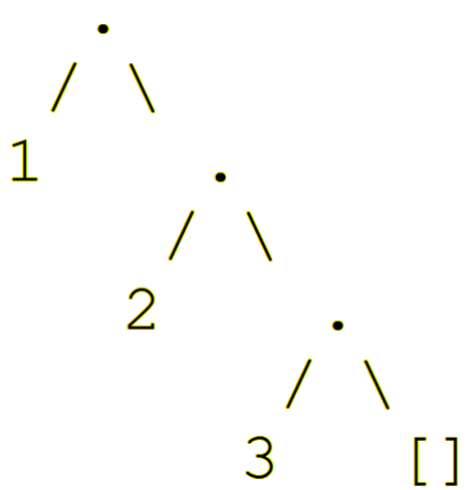
- Arithmetic
- Recursion
  - Recursion
    - Recursion
      - Recursion
        - Recursion
          - Recursion
- Lists

Lists

- Lists are the quintessential data structure in Prolog
- Empty list represented as [ ]
- Elements separated by commas within square brackets
  - [a, b, c]
  - [4, 8, 15, 16, 23, 42]
- Lists elements can be anything, including other lists
  - [1, [a, b, v], g, [2, [D, y], 3], 4]

Lists

- The internal representation uses the . functor and two arguments – the head and tail of the list
  - Ex.: [1, 2, 3] = .(1, .(2, .(3, [])))




```
| ?- A = .(1, .(2, .(3, []) ) ).
A = [1,2,3] ?
yes
```

- Strings are a representation of lists of character ASCII codes

```
| ?- A = "Hello".
A = [72,101,108,108,111] ?
yes
```

Lists

- Easily separate the head of the list from the rest of the list
  - The head of the list can separate more than one element

```
[ H | T ] % where T is a list with the remaining elements of the list
[ 4 ] = [ 4 | [ ] ]          % tail of list with one element is empty list
[4, 8, 15, 16, 23, 42] = [4 | [8, 15, 16, 23, 42] ]
[4, 8, 15, 16, 23, 42] = [ 4, 8 | [ 15, 16, 23, 42] ]
```

- Definition of what is a list
  - An empty list
  - A list construct where tail is a list

```
is_list( [ ] ).
is_list( [H|T] ):- is_list(T).
```

List Length

- There are several useful built-in predicates to work with lists
  - ***length(?List, ?Size)***
    - Size of a list (very flexible)

Can also be easily implemented recursively

```
length( [ ], 0 ).
length( [_|T], L ):-
        length(T, L1),
        L is L1+1.
```

Actually, there's a small caveat with this solution. Can you find it? (homework)

```
| ?- length([1,2,3], 3).
yes
| ?- length([1,2,3], L).
L = 3 ?
yes
| ?- length(L, 3).
L = [_A,_B,_C] ?
yes
| ?- length(L, S).
L = [],
S = 0 ? ;
L = [_A],
S = 1 ? ;
L = [_A,_B],
S = 2 ? ;
L = [_A,_B,_C],
S = 3 ?
yes
```

List Membership

- ***member(?Elem, ?List)***
  - List member (very flexible)
- ***memberchk(?Elem, ?List)***
  - Member verification (determinate)

Can also be easily implemented recursively

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

memberchk( X, [X|_] ).
memberchk( X, [Y|T] ):-
        X \= Y,
        memberchk(X, T).
```

```
| ?- member(2, [1,2,3]).
true ?
yes
| ?- member(2, L).
L = [2|_A] ? ;
L = [_A,2|_B] ?
yes
| ?- member(M, [1,2]).
M = 1 ? ;
M = 2 ? ;
no
| ?- member(M, L).
L = [M|_A] ? ;
L = [_A,M|_B] ?
yes
```

Appending Lists

- ***append(?L1, ?L2, ?L3)***
  - Appends two lists into a third (very flexible)

Can also be easily implemented recursively

```
append( [ ], L2, L2 ).
append( [H|T], L2, [H|T3] ):-
        append(T, L2, T3).
```

```
| ?- append([1,2], [3,4], [1,2,3,4]).
yes
| ?- append([1,2], [3,4], L).
L = [1,2,3,4] ?
yes
| ?- append([1,2], L, [1,2,3,4]).
L = [3,4] ?
yes
| ?- append(L, [3,4], [1,2,3,4]).
L = [1,2] ?
yes
| ?- append(L1, L2, [1,2,3]).
L1 = [],
L2 = [1,2,3] ? ;
L1 = [1],
L2 = [2,3] ? ;
L1 = [1,2],
L2 = [3] ? ;
L1 = [1,2,3],
L2 = [] ? ;
no
```

Sorting Lists

- ***sort(+List, -SortedList)***
  - Sorts a (proper) list
- ***keysort(+PairList, -SortedList)***
  - Sorts a (proper) key-value pair list
  - If a key appears more than once, elements retain original order

```
| ?- sort([4,2,3,1], [1,2,3,4]).
yes
| ?- sort([4,2,3,1], SL).
SL = [1,2,3,4] ?
yes
| ?- keysort([2-1, 1-2, 4-3, 3-4], SL).
SL = [1-2,2-1,3-4,4-3] ?
yes
| ?- keysort([2-1, 1-2, 4-3, 3-4, 1-1], SL).
SL = [1-2,1-1,2-1,3-4,4-3] ?
yes
```

Can also be implemented recursively
Homework!

Lists Library

- The Lists library has numerous predicates to work with lists
- Libraries can be imported using the *use_module* directive:

```
:-use_module(library(lists)).
```

See section 10.25 of the SICStus Manual for a complete description of available predicates

Lists Library

- Some useful predicates from the lists library
  - nth0(?Pos, ?List, ?Elem) / nth1(?Pos, ?List, ?Elem)
  - nth0(?Pos, ?List, ?Elem, ?Rest) / nth1(?Pos, ?List, ?Elem, ?Rest)

```
| ?- nth1(3, R, c, [a,b,d]).
R = [a,b,c,d] ?
yes
| ?- nth1(3, [a,b,c,d], X, R),
     nth1(3, Res, e, R).
X = c,
R = [a,b,d],
Res = [a,b,e,d] ?
yes
```

```
| ?- nth1(3, [a,b,c,d], X).
X = c ?
yes
| ?- nth1(3, [a,b,c,d], X, R).
X = c,
R = [a,b,d] ?
yes
```

Can be used to remove, insert or replace (when used twice) list elements

Lists Library

- select(?X, ?XList, ?Y, ?YList)
- delete(+List, +ToDel, -Rest) / delete(+List, +ToDel, + Count, -Rest)
- last(?Init, ?Last, ?List)

```
| ?- select(g, [a,g,a,g,a], r, X).
X = [a,r,a,g,a] ? ;
X = [a,g,a,r,a] ? ;
no
| ?- delete([a,b,b,a], a, X).
X = [b,b] ?
yes
| ?- delete([a,b,b,a], a, 1, X).
X = [b,b,a] ? ;
no
| ?- last(I, L, [1,2,3,4]).
I = [1,2,3],
L = 4 ?
yes
```

Lists Library

- segment(?List, ?Segment)
- sublist(+List, ?Part, ?Before, ?Length, ?After)

```
| ?- segment([a,b,c], S).
S = [a] ? ;
S = [a,b] ? ;
S = [a,b,c] ? ;
S = [b] ? ;
S = [b,c] ? ;
S = [c] ? ;
S = [] ? ;
no
```

```
| ?- sublist([a,b,c], Part, Bef, Len, Aft).
Part = [],
Bef = 0,
Len = 0,
Aft = 3 ? ;
Part = [a],
Bef = 0,
Len = 1,
Aft = 2 ? ;
Part = [a,b],
Bef = 0,
Len = 2,
Aft = 1 ? ;
```

```
| ?- sublist([a,b,c], S, _, _, _).
S = [] ? ;
S = [a] ? ;
S = [a,b] ? ;
S = [a,b,c] ? ;
S = [] ? ;
S = [b] ? ;
S = [b,c] ? ;
S = [] ? ;
S = [c] ? ;
S = [] ? ;
no
```

Lists Library

- append(+ListOfLists, -List)
- reverse(?List, ?Reversed)
- rotate_list(+Amount, ?List, ?Rotated)

```
| ?- append([[1,2,3], [4,5,6], [7,8,9]], L).
L = [1,2,3,4,5,6,7,8,9] ? ;
no
```

```
| ?- reverse([1,2,3], L).
L = [3,2,1] ? ;
no
| ?- reverse(L, [3,2,1]).
L = [1,2,3] ? ;
no
```

```
| ?- rotate_list(1, [a,b,c,d], L).
L = [b,c,d,a] ? ;
no
| ?- rotate_list(1, L, [a,b,c,d]).
L = [d,a,b,c] ? ;
no
```

Lists Library

- transpose(?Matrix, ?Transposed)
- remove_dups(+List, ?PrunedList)
- permutation(?List, ?Permutation)

```
| ?- transpose([[1,2,3],[4,5,6],[7,8,9]],T).
T = [[1,4,7],[2,5,8],[3,6,9]] ? ;
no
| ?- remove_dups([a,b,b,a], L).
L = [a,b] ? ;
no
```

```
| ?- permutation([a,b,c], P).
P = [a,b,c] ? ;
P = [b,a,c] ? ;
P = [b,c,a] ? ;
P = [a,c,b] ? ;
P = [c,a,b] ? ;
P = [c,b,a] ? ;
no
```

Lists Library

- sumlist(+ListOfNumbers, ?Sum)
- max_member(?Max, +List) / min_member(?Min, +List)
- max_member(:Comp, ?Max, +List) / min_member(:Comp, ?Min, +L)

```
| ?- sumlist([1,2,3,4,5], S).
S = 15 ? ;
no
| ?- max_member(Max, [4,5,3,2,6,1]).
Max = 6 ? ;
no
```

Lists Library

- maplist(:Pred, +L) / maplist(:Pr, +L1, ?L2) / maplist(:Pr, +L1, ?L2, ?L3)
- map_product(:Pred, +Xs, +Ys, ?List)

```
| ?- maplist(even, [2,3,4,5]).
no
| ?- maplist(even, [2,4]).
yes
| ?- maplist(write, [a,b,b,a]).
abba
yes
| ?- maplist(square, [2,3,4,5], L).
L = [4,9,16,25] ? ;
no
```

```
even(X):-
        X mod 2 =:= 0.
square(X, Y):-
        Y is X*X.
pow(X, Y, Z):-
        Z is X**Y.
```

```
| ?- maplist(pow,[2,3,4],[2,3,4],L).
L = [4.0,27.0,256.0] ? ;
no
| ?- map_product(pow,[2,3,4],[2,3,4],L).
L = [4.0,8.0,16.0,9.0,27.0,81.0,16.0,64.0,256.0] ? ;
no
```

Lists Library

- scanlist(:Pred, +Xs, ?Start, ?Final)
- cumlist(:Pred, +Xs, ?Start, ?List)

```
soma(A, B, C):-
        C is A+B.
soma2(A, B, A+B).
soma3(A, B, B+A).
```

```
| ?- cumlist(soma, [2,3,4,5], 1, F).
F = [3,6,10,15] ? ;
no
| ?- cumlist(soma2, [2,3,4,5], 1, F).
F = [2+1,3+(2+1),4+(3+(2+1)),5+(4+(3+(2+1)))] ?
yes
| ?- cumlist(soma3, [2,3,4,5], 1, F).
F = [1+2,1+2+3,1+2+3+4,1+2+3+4+5] ?
yes
```

```
| ?- scanlist(soma, [2,3,4,5], 1, F).
F = 15 ?
yes
| ?- scanlist(soma2, [2,3,4,5], 1, F).
F = 5+(4+(3+(2+1))) ?
yes
| ?- scanlist(soma3, [2,3,4,5], 1, F).
F = 1+2+3+4+5 ?
yes
```

Lists Library

- some(:Pred, +List) / some(:Pred, +Xs, ?Ys) / some(:Pr, +Xs, ?Ys, ?Zs)
- include(:P, +X, ?L) / include(:P, +X, +Y, ?L) / include(:P, +X, +Y, +Z, ?L)
- exclude(:P, +X, ?L) / exclude(:P, +X,+Y, ?L) / exclude(:P, +X,+Y,+Z, ?L)
- group(:Pred, +List, ?Front, ?Back)

```
| ?- some(even, [3,5,7]).
no
| ?- some(even, [3,4,5]).
true ?
yes
```

```
| ?- include(even, [1,2,3,4,5,6,7,8], L).
L = [2,4,6,8] ?
yes
| ?- exclude(even, [1,2,3,4,5,6,7,8], L).
L = [1,3,5,7] ?
yes
```

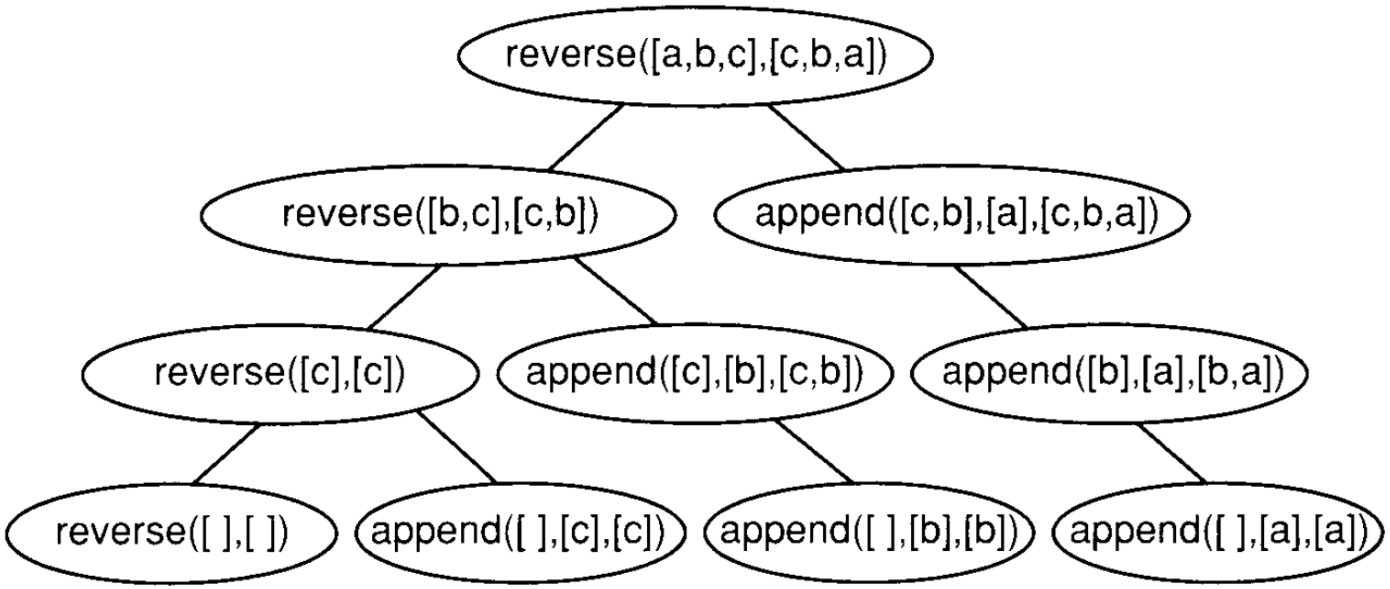
```
| ?- group(even, [2,4,6,1,2,3,4], F, B).
F = [2,4,6],
B = [1,2,3,4] ?
yes
```

Lists

- Several of these predicates can be implemented using append
  - However, sometimes we can find more efficient versions
- Example: list reverse

```
reverse([], []).
reverse([X|Xs], Rev):-
        reverse(Xs, Ys),
        append(Ys, [X], Rev).
```

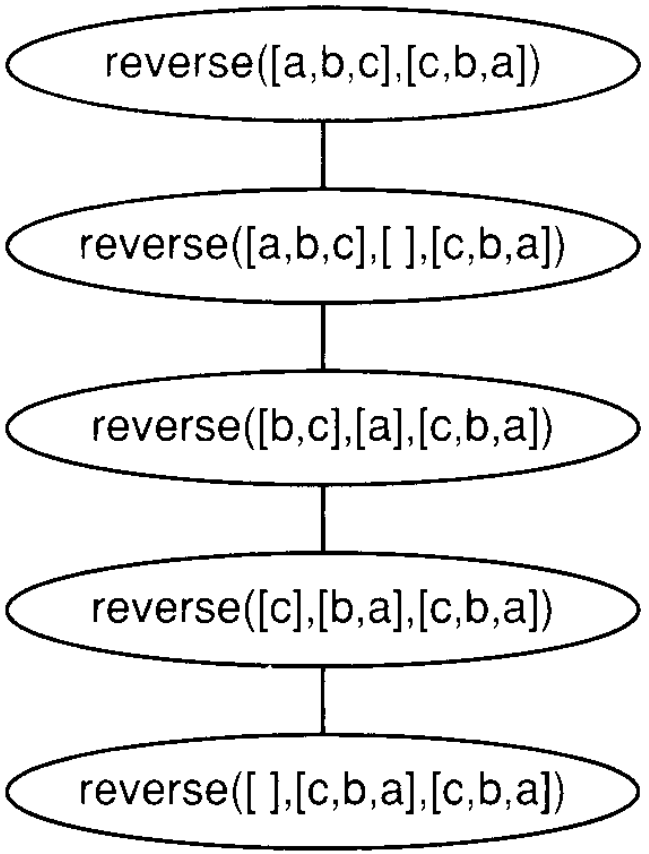
  - Size of proof tree is quadratic to the number of elements in the list



Lists

- We can use an accumulator (tail recursion) to reverse the list

```
reverse(Xs, Rev):- reverse(Xs, [], Rev).
reverse([X|Xs], Acc, Rev):-
        reverse(Xs, [X|Acc], Rev).
reverse([], Rev, Rev).
```



- The accumulator holds the reversed list in the last step of the recursion
- Now the process is linear to the number of elements in the list

Q & A

≤ in different programing languages