

Logic Programming

Georg Moser

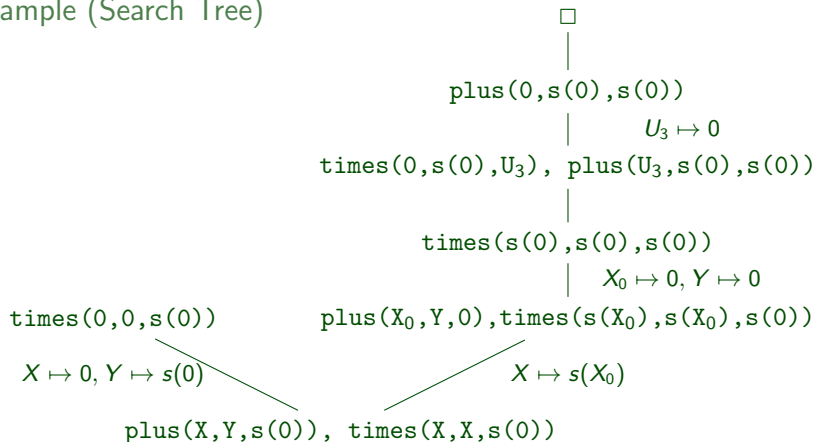
Department of Computer Science @ UIBK

Winter 2016



Summary of Last Lecture

Example (Search Tree)



NB: search trees are a tree representation of SLD-derivations

Definitions

- a **type** is a (possible infinite) set of terms
- types are conveniently defined by unary relations
- a type is **complete** if closed under the instance relation
- with every complete type T one associates an **incomplete** type $!T$ which is a set of terms with instances in T and instances not in T

Definitions

- a list is **complete** if every instances satisfies the above type for lists
- otherwise it is **incomplete**

Example

the lists $[a,b,c]$ and $[a,X,c]$ are complete; the list $[a,b|Xs]$ is not

Outline of the Lecture

Monotone Logic Programs

introduction, basic constructs, logic foundations, unification, semantics, database and recursive programming, termination, complexity

Incomplete Data Structures and Constraints

incomplete data structures, definite clause grammars, constraint logic programming, answer set programming

Full Prolog

semantics (revisited), correctness proofs, meta-logical predicates, cuts non-deterministic programming, efficient programs, complexity

Outline of the Lecture

Monotone Logic Programs

introduction, basic constructs, logic foundations, unification, semantics, **database and recursive programming**, termination, complexity

Incomplete Data Structures and Constraints

incomplete data structures, definite clause grammars, constraint logic programming, answer set programming

Full Prolog

semantics (revisited), correctness proofs, meta-logical predicates, cuts non-deterministic programming, efficient programs, complexity

Proof Trees

Definitions

- a **proof tree** for a program P and a goal G is a tree, whose nodes are goals and whose edges represent reduction of goals
- the root is the query G

Proof Trees

Definitions

- a **proof tree** for a program P and a goal G is a tree, whose nodes are goals and whose edges represent reduction of goals
- the root is the query G
- the edges are labelled with (partial) answer substitutions

Proof Trees

Definitions

- a **proof tree** for a program P and a goal G is a tree, whose nodes are goals and whose edges represent reduction of goals
- the root is the query G
- the edges are labelled with (partial) answer substitutions
- a proof tree for a conjunction of goals G_1, \dots, G_n is the set of proof trees for G_i

Proof Trees

Definitions

- a **proof tree** for a program P and a goal G is a tree, whose nodes are goals and whose edges represent reduction of goals
- the root is the query G
- the edges are labelled with (partial) answer substitutions
- a proof tree for a conjunction of goals G_1, \dots, G_n is the set of proof trees for G_i

Remark

a proof tree is a different representation of **one successful** solution represented by a search tree combining **all** possible selection functions

Example (Proof Tree)

```
plus(0,X,X).                times(0,X,0).  
plus(s(X),Y,s(Z)) :- plus(X,Y,Z).  times(s(X),Y,Z) :- times(X,Y,U),  
                                     plus(U,Y,Z).
```

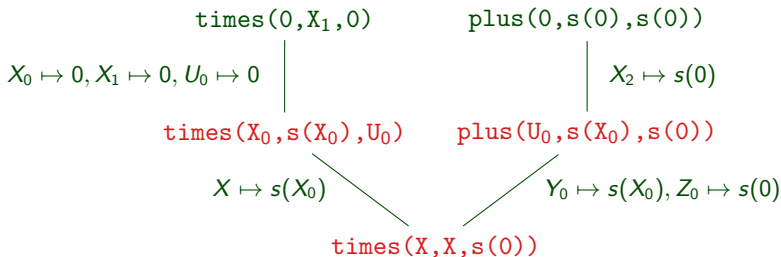
Example (Proof Tree)

```

plus(0,X,X).
plus(s(X),Y,s(Z)) :- plus(X,Y,Z).
    
```

```

times(0,X,0).
times(s(X),Y,Z) :- times(X,Y,U),
                  plus(U,Y,Z).
    
```

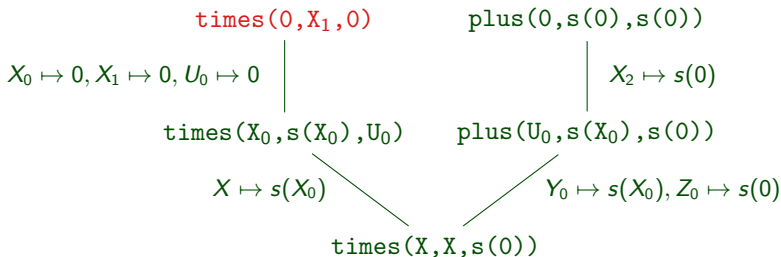


Example (Proof Tree)

```

plus(0,X,X).
plus(s(X),Y,s(Z)) :- plus(X,Y,Z).

times(0,X,0).
times(s(X),Y,Z) :- times(X,Y,U),
                    plus(U,Y,Z).
    
```

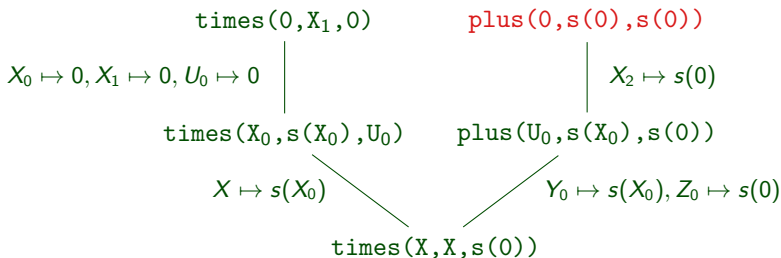


Example (Proof Tree)

```

plus(0,X,X).
plus(s(X),Y,s(Z)) :- plus(X,Y,Z).

times(0,X,0).
times(s(X),Y,Z) :- times(X,Y,U),
                    plus(U,Y,Z).
    
```



Structured Data and Data Abstraction

Example (Unstructured Data)

```
course(discrete_mathematics,tuesday,8,11,sandor,szedmak,  
victor_franz_hess,d).
```

Structured Data and Data Abstraction

Example (Unstructured Data)

```
course(discrete_mathematics,tuesday,8,11,sandor,szedmak,  
       victor_franz_hess,d).
```

Example (Structured Data)

```
course(discrete_mathematics,time(tuesday,8,11),  
       lecturer(sandor,szedmak),location(victor_franz_hess,d)).
```

Structured Data and Data Abstraction

Example (Unstructured Data)

```
course(discrete_mathematics,tuesday,8,11,sandor,szedmak,  
victor_franz_hess,d).
```

Example (Structured Data)

```
course(discrete_mathematics,time(tuesday,8,11),  
lecturer(sandor,szedmak),location(victor_franz_hess,d)).
```

Example

```
lecturer(Lecturer,Course) :-  
    course(Course,Time,Lecturer,Location).  
  
duration(Course,Length) :-  
    course(Course,time(Day,Start,Finish),Lecturer,Location),  
    plus(Start,Length,Finish).
```


Example (cont'd)

```
teaches(Lecturer,Day) : -  
    course(Course,time(Day,Start,Finish),Lecturer,Location).  
  
occupied(Location,Day,Time) : -  
    course(Course,time(Day,Start,Finish),Lecturer,Location),  
    Start ≤ Time, Time ≤ Finish.
```

NB: rules for comparison are as expected

Example (cont'd)

```
teaches(Lecturer,Day) :-  
    course(Course,time(Day,Start,Finish),Lecturer,Location).  
  
occupied(Location,Day,Time) :-  
    course(Course,time(Day,Start,Finish),Lecturer,Location),  
    Start ≤ Time, Time ≤ Finish.
```

NB: rules for comparison are as expected

Why structure Data?

- helps to organise data; databases are usually structured ...
- rules can be written abstractly, hiding irrelevant detail
- modularity becomes possible or is improved

Logic Programs and the Relational Database Model

Observation

the basic operations of relational algebras, namely:

- 1 union
- 2 difference
- 3 cartesian product
- 4 projection
- 5 selection
- 6 intersection

can easily be expressed within logic programming

Logic Programs and the Relational Database Model

Observation

the basic operations of relational algebras, namely:

- 1 union
- 2 difference
- 3 cartesian product
- 4 projection
- 5 selection
- 6 intersection

can easily be expressed within logic programming

Example

```
r_union_s( $X_1, \dots, X_n$ ) :- r( $X_1, \dots, X_n$ ).  
r_union_s( $X_1, \dots, X_n$ ) :- s( $X_1, \dots, X_n$ ).
```

Arithmetic

Example (Type Condition)

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

Arithmetic

Example (Type Condition)

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

Example

```
plus(0,X,X) :- is_number(X).  
plus(s(X),Y,s(Z)) :- plus(X,Y,Z).  
times(0,X,0).  
times(s(X),Y,Z) :- times(X,Y,U), plus(U,Y,Z).
```

Arithmetic

Example (Type Condition)

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

Example

```
plus(0,X,X).  
plus(s(X),Y,s(Z)) :- plus(X,Y,Z).  
times(0,X,0).  
times(s(X),Y,Z) :- times(X,Y,U), plus(U,Y,Z).
```

Arithmetic

Example (Type Condition)

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

Example

```
plus(0,X,X).  
plus(s(X),Y,s(Z)) :- plus(X,Y,Z).  
times(0,X,0).  
times(s(X),Y,Z) :- times(X,Y,U), plus(U,Y,Z).
```

Example

```
factorial(0,s(0)).  
factorial(s(N),F) :- factorial(N,F1), times(s(N),F1,F).
```


Example

$0 \leq X :- \text{is_number}(X).$

$s(X) \leq s(Y) :- X \leq Y.$

$\text{minimum}(N_1, N_2, N_1) :- N_1 \leq N_2.$

$\text{minimum}(N_1, N_2, N_2) :- N_2 \leq N_1.$

Example

$0 \leq X.$

$s(X) \leq s(Y) : - X \leq Y.$

$\text{minimum}(N_1, N_2, N_1) : - N_1 \leq N_2.$

$\text{minimum}(N_1, N_2, N_2) : - N_2 \leq N_1.$

Example

$0 \leq X.$

$s(X) \leq s(Y) : - X \leq Y.$

$\text{minimum}(N_1, N_2, N_1) : - N_1 \leq N_2.$

$\text{minimum}(N_1, N_2, N_2) : - N_2 \leq N_1.$

Example

$\text{mod}(X, Y, Z) : - Z < Y, \text{times}(Y, Q, W), \text{plus}(W, Z, X).$

Example

$0 \leq X.$

$s(X) \leq s(Y) : - X \leq Y.$

$\text{minimum}(N_1, N_2, N_1) : - N_1 \leq N_2.$

$\text{minimum}(N_1, N_2, N_2) : - N_2 \leq N_1.$

Example

$\text{mod}(X, Y, Z) : - Z < Y, \text{times}(Y, Q, W), \text{plus}(W, Z, X).$

$\text{mod}(X, Y, X) : - X < Y.$

$\text{mod}(X, Y, Z) : - \text{plus}(X1, Y, X), \text{mod}(X1, Y, Z).$

Example

$0 \leq X.$

$s(X) \leq s(Y) : - X \leq Y.$

$\text{minimum}(N_1, N_2, N_1) : - N_1 \leq N_2.$

$\text{minimum}(N_1, N_2, N_2) : - N_2 \leq N_1.$

Example

$\text{mod}(X, Y, Z) : - Z < Y, \text{ times}(Y, Q, W), \text{ plus}(W, Z, X).$

$\text{mod}(X, Y, X) : - X < Y.$

$\text{mod}(X, Y, Z) : - \text{plus}(X1, Y, X), \text{ mod}(X1, Y, Z).$

Example

$\text{ackermann}(0, N, s(N)).$

Example

$0 \leq X.$

$s(X) \leq s(Y) : - X \leq Y.$

$\text{minimum}(N_1, N_2, N_1) : - N_1 \leq N_2.$

$\text{minimum}(N_1, N_2, N_2) : - N_2 \leq N_1.$

Example

$\text{mod}(X, Y, Z) : - Z < Y, \text{ times}(Y, Q, W), \text{ plus}(W, Z, X).$

$\text{mod}(X, Y, X) : - X < Y.$

$\text{mod}(X, Y, Z) : - \text{plus}(X1, Y, X), \text{ mod}(X1, Y, Z).$

Example

$\text{ackermann}(0, N, s(N)).$

$\text{ackermann}(s(M), 0, \text{Val}) : - \text{ackermann}(M, s(0), \text{Val}).$

Example

$0 \leq X.$

$s(X) \leq s(Y) : - X \leq Y.$

$\text{minimum}(N_1, N_2, N_1) : - N_1 \leq N_2.$

$\text{minimum}(N_1, N_2, N_2) : - N_2 \leq N_1.$

Example

$\text{mod}(X, Y, Z) : - Z < Y, \text{ times}(Y, Q, W), \text{ plus}(W, Z, X).$

$\text{mod}(X, Y, X) : - X < Y.$

$\text{mod}(X, Y, Z) : - \text{plus}(X_1, Y, X), \text{ mod}(X_1, Y, Z).$

Example

$\text{ackermann}(0, N, s(N)).$

$\text{ackermann}(s(M), 0, \text{Val}) : - \text{ackermann}(M, s(0), \text{Val}).$

$\text{ackermann}(s(M), s(N), \text{Val}) : - \text{ackermann}(s(M), N, \text{Val}_1),$
 $\text{ackermann}(M, \text{Val}_1, \text{Val}).$

Example

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


Example

```
member(X, [X|Xs]).
```

```
member(X, [Y|Xs]) : - member(X, Xs).      : - member(X, [a,b,a]).
```

Example

```
member(X, [X|Xs]).
member(X, [Y|Xs]) : - member(X, Xs).      : - member(X, [a,b,a]).
```

Example

```
append(Xs, Ys, Zs) : -
    Xs = [],
```

```
append(Xs, Ys, Zs) : -
    Xs = [H|Ts],
```

Example

```
member(X, [X|Xs]).
member(X, [Y|Xs]) : - member(X,Xs).      : - member(X, [a,b,a]).
```

Example

```
append(Xs,Ys,Zs) : -
    Xs = [],
```

```
append(Xs,Ys,Zs) : -
    Xs = [H|Ts],
```

Example

```
member(X, [X|Xs]).
member(X, [Y|Xs]) : - member(X, Xs).      : - member(X, [a,b,a]).
```

Example

```
append(Xs, Ys, Zs) : -
    Xs = [],
    Zs = Ys.
append(Xs, Ys, Zs) : -
    Xs = [H|Ts],
```

Example

```
member(X, [X|Xs]).
member(X, [Y|Xs]) : - member(X, Xs).           : - member(X, [a,b,a]).
```

Example

```
append(Xs, Ys, Zs) : -
    Xs = [],
    Zs = Ys.
append(Xs, Ys, Zs) : -
    Xs = [H|Ts],
    append(Ts, Ys, Us),
    Zs = [H|Us].
```

Example

```
member(X, [X|Xs]).
member(X, [_|Xs]) :- member(X, Xs).           :- member(X, [a,b,a]).
```

Example

```
append(Xs, Ys, Zs) :-
    Xs = [],
    Zs = Ys.
append(Xs, Ys, Zs) :-
    Xs = [H|Ts],
    append(Ts, Ys, Us),
    Zs = [H|Us].

append([], Ys, Ys).
append([H|Ts], Ys, [H|Zs]) :-
    append(Ts, Ys, Zs).
```

Example

```
member(X, [X|Xs]).
member(X, [Y|Xs]) :- member(X, Xs).           :- member(X, [a,b,a]).
```

Example

```
append(Xs, Ys, Zs) :-
    Xs = [],
    Zs = Ys.
append(Xs, Ys, Zs) :-
    Xs = [H|Ts],
    append(Ts, Ys, Us),
    Zs = [H|Us].

append([], Ys, Ys).
append([H|Ts], Ys, [H|Zs]) :-
    append(Ts, Ys, Zs).
```

Example

```
prefix([], Xs).
prefix([X|Xs], [X|Ys]) :-
    prefix(Xs, Ys).

suffix(Xs, Xs).
suffix(Xs, [Y|Ys]) :-
    suffix(Xs, Ys).
```

Composition of Programs

five steps to implement relation R

- 1 look up existing definitions of relation R

Composition of Programs

five steps to implement relation R

- 1 look up existing definitions of relation R

- 2 define types of individual data

Composition of Programs

five steps to implement relation R

- 1 look up existing definitions of relation R
- 2 define types of individual data
- 3 think up a suitable name

Composition of Programs

five steps to implement relation R

- 1 look up existing definitions of relation R
- 2 define types of individual data
- 3 think up a suitable name
- 4 write queries (use cases)

Composition of Programs

five steps to implement relation R

- 1 look up existing definitions of relation R
- 2 define types of individual data
- 3 think up a suitable name
- 4 write queries (use cases)
- 5 write the actual program

Composition of Programs

five steps to implement relation R

- 1 look up existing definitions of relation R
 - family relations
 - train tables
- 2 define types of individual data
- 3 think up a suitable name
- 4 write queries (use cases)
- 5 write the actual program

Composition of Programs

five steps to implement relation R

- 1 look up existing definitions of relation R
 - family relations
 - train tables
- 2 define types of individual data
 - `is_number`
 - mainly for documentation
- 3 think up a suitable name
- 4 write queries (use cases)
- 5 write the actual program

Composition of Programs

five steps to implement relation R

- 1 look up existing definitions of relation R
 - family relations
 - train tables
- 2 define types of individual data
 - is_number
 - mainly for documentation
- 3 think up a suitable name
 - convert a verbose description into a name
 - child_of
- 4 write queries (use cases)
- 5 write the actual program

Example (Uses of append)

```
prefix(Xs,Ys) :- append(Xs,As,Ys).  
suffix(Xs,Ys) :- append(As,Xs,Ys).  
member(X,Ys) :- append(As,[X|Xs],Ys).
```


Example (Uses of append)

```
prefix(Xs,Ys) :- append(Xs,As,Ys).  
suffix(Xs,Ys) :- append(As,Xs,Ys).  
member(X,Ys) :- append(As,[X|Xs],Ys).
```

Example

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

Example (Uses of append)

```
prefix(Xs,Ys) :- append(Xs,As,Ys).  
suffix(Xs,Ys) :- append(As,Xs,Ys).  
member(X,Ys) :- append(As,[X|Xs],Ys).
```

Example

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

Example (Uses of append)

```

prefix(Xs,Ys) :- append(Xs,As,Ys).
suffix(Xs,Ys) :- append(As,Xs,Ys).
member(X,Ys) :- append(As,[X|Xs],Ys).

```

Example

```

reverse([],[]).
reverse([X|Xs],Zs) :- reverse(Xs,Ys), append(Ys,[X],Zs).
reverse(Xs,Ys) :- reverse(Xs,[],Ys).
reverse([X|Xs],Acc,Ys) :- reverse(Xs,[X|Acc],Ys).
reverse([],Ys,Ys).

```

Example

```

length([],0).
length([X|Xs],s(N)) :- length(Xs,N).

```

Example (Permutation Sort)

```
select(X, [X|Xs], Xs).  
select(X, [Y|Ys], [Y|Zs]) :- select(X, Ys, Zs).
```

Example (Permutation Sort)

```
permutationsort(Xs,Ys) :- permutation(Xs,Ys), ordered(Ys).
```

```
select(X,[X|Xs],Xs).
```

```
select(X,[Y|Ys],[Y|Zs]) :- select(X,Ys,Zs).
```

Example (Permutation Sort)

```
permutationsort(Xs,Ys) :- permutation(Xs,Ys), ordered(Ys).  
permutation(Xs,[Z|Zs]) :- select(Z,Xs,Ys), permutation(Ys,Zs).  
permutation([],[]).
```

```
select(X,[X|Xs],Xs).  
select(X,[Y|Ys],[Y|Zs]) :- select(X,Ys,Zs).
```

Example (Permutation Sort)

```
permutationsort(Xs,Ys) :- permutation(Xs,Ys), ordered(Ys).  
permutation(Xs,[Z|Zs]) :- select(Z,Xs,Ys), permutation(Ys,Zs).  
permutation([],[]).  
ordered([X]).  
ordered([X,Y|Ys]) :- X ≤ Y, ordered([Y|Ys]).  
select(X,[X|Xs],Xs).  
select(X,[Y|Ys],[Y|Zs]) :- select(X,Ys,Zs).
```

Example (Permutation Sort)

```
permutationsort(Xs,Ys) :- permutation(Xs,Ys), ordered(Ys).  
permutation(Xs,[Z|Zs]) :- select(Z,Xs,Ys), permutation(Ys,Zs).  
permutation([],[]).  
ordered([X]).  
ordered([X,Y|Ys]) :- X ≤ Y, ordered([Y|Ys]).  
select(X,[X|Xs],Xs).  
select(X,[Y|Ys],[Y|Zs]) :- select(X,Ys,Zs).
```

Example (Insertion Sort)

```
insertionsort([X|Xs],Ys) :- insertionsort(Xs,Zs),  
                             insert(X,Zs,Ys).  
insertionsort([],[]).
```


Example (Permutation Sort)

```

permutationsort(Xs,Ys) :- permutation(Xs,Ys), ordered(Ys).
permutation(Xs,[Z|Zs]) :- select(Z,Xs,Ys), permutation(Ys,Zs).
permutation([],[]).
ordered([X]).
ordered([X,Y|Ys]) :- X ≤ Y, ordered([Y|Ys]).
select(X,[X|Xs],Xs).
select(X,[Y|Ys],[Y|Zs]) :- select(X,Ys,Zs).

```

Example (Insertion Sort)

```

insertionsort([X|Xs],Ys) :- insertionsort(Xs,Zs),
                             insert(X,Zs,Ys).
insertionsort([],[]).
insert(X,[],[X]).
insert(X,[Y|Ys],[Y|Zs]) :- X > Y, insert(X,Ys,Zs).
insert(X,[Y|Ys],[X,Y|Ys]) :- X ≤ Y.

```

Example (Quick Sort)

```
quicksort([X|Xs],Ys) :-  
    partition(Xs,X,Littles,Bigs),  
    quicksort(Littles,Ls), quicksort(Bigs,Rs),  
    append(Ls,[X|Rs],Ys).
```

Example (Quick Sort)

```
quicksort([X|Xs],Ys) :-  
    partition(Xs,X,Littles,Bigs),  
    quicksort(Littles,Ls), quicksort(Bigs,Rs),  
    append(Ls,[X|Rs],Ys).  
  
partition([X|Xs],Y,[X|Ls],Bs) :-  
    X <= Y, partition(Xs,Y,Ls,Bs).  
partition([X|Xs],Y,Ls,[X|Bs]) :-  
    X > Y,      partition(Xs,Y,Ls,Bs).  
partition([],Y,[],[]).
```

Example (Quick Sort)

```
quicksort([X|Xs],Ys) :-
    partition(Xs,X,Littles,Bigs),
    quicksort(Littles,Ls), quicksort(Bigs,Rs),
    append(Ls,[X|Rs],Ys).

partition([X|Xs],Y,[X|Ls],Bs) :-
    X <= Y, partition(Xs,Y,Ls,Bs).
partition([X|Xs],Y,Ls,[X|Bs]) :-
    X > Y,      partition(Xs,Y,Ls,Bs).
partition([],Y,[],[]).
```

Example

```
isotree(nil,nil).
isotree(tree(X,Left1,Right1),tree(X,Left2,Right2)) :-
    isotree(Left1,Left2), isotree(Right1,Right2).
isotree(tree(X,Left1,Right1),tree(X,Left2,Right2)) :-
    isotree(Left1,Right2), isotree(Right1,Left2).
```