

# Logic Programming

Cezary Kaliszyk    **Georg Moser**

Institute of Computer Science @ UIBK

Winter 2015



# Summary of Last Lecture

## Example

```
ancestor_of(Ancestor, Descendant) :-  
    child_of(Descendant, Ancestor).  
ancestor_of(Ancestor, Descendant) :-  
    child_of(Person, Ancestor),  
    ancestor_of(Person, Descendant).
```

## Example

```
ancestor_of_2(Ancestor, Descendant) :-  
    child_of(Descendant, Ancestor).  
ancestor_of_2(Ancestor, Descendant) :-  
    ancestor_of_2(Person, Descendant),  
    child_of(Person, Ancestor).
```

# Outline of the Lecture

## Monotone Logic Programs

introduction, basic constructs, unification, database and recursive programming, termination

## Incomplete Data Structures and Constraints

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

## Full Prolog

semantics, correctness proofs, meta-logical predicates, cuts nondeterministic programming, efficient programs, complexity

# Outline of the Lecture

## Monotone Logic Programs

introduction, basic constructs, unification, database and recursive programming, termination

## Incomplete Data Structures and Constraints

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

## Full Prolog

semantics, correctness proofs, meta-logical predicates, cuts nondeterministic programming, efficient programs, complexity

## Example (revisited)

```
ancestor_of(Ancestor, Descendant) :-  
    child_of(Descendant, Ancestor).  
ancestor_of(Ancestor, Descendant) :-  
    child_of(Person, Ancestor),  
    ancestor_of(Person, Descendant).
```

## Example (revisited)

```
ancestor_of(Ancestor, Descendant) :-  
    child_of(Descendant, Ancestor).  
ancestor_of(Ancestor, Descendant) :-  
    child_of(Person, Ancestor),  
    ancestor_of(Person, Descendant).
```

- Ancestor

## Example (revisited)

```
ancestor_of(Ancestor, Descendant) :-  
    child_of(Descendant, Ancestor).  
ancestor_of(Ancestor, Descendant) :-  
    child_of(Person, Ancestor),  
    ancestor_of(Person, Descendant).
```

- Ancestor
- Descendant

## Example (revisited)

```
ancestor_of(Ancestor, Descendant) :-  
    child_of(Descendant, Ancestor).  
ancestor_of(Ancestor, Descendant) :-  
    child_of(Person, Ancestor),  
    ancestor_of(Person, Descendant).
```

- Ancestor
- Descendant
- Person



## Example (revisited)

```
ancestor_of(Ancestor, Descendant) :-  
    child_of(Descendant, Ancestor).  
ancestor_of(Ancestor, Descendant) :-  
    child_of(Person, Ancestor),  
    ancestor_of(Person, Descendant).
```

- Ancestor
- Descendant
- Person

## In English

*Someone is ancestor of a descendant, if the descendant is his (or her) child, or if he (or she) has a child and this person is the ancestor of the descendant.*

binding of logical variables is expressed as references

## Example (revisited)

```

ancestor_of(Ancestor , Descendant) :-
    child_of(Descendant , Ancestor).
ancestor_of(Ancestor , Descendant) :-
    child_of(Person , Ancestor),
    ancestor_of(Person , Descendant).

```

- Ancestor
- Descendant
- Person

## In English

*Someone is ancestor of a descendant, if the descendant is his (or her) child, or if he (or she) has a child and this person is the ancestor of the descendant.*

binding of logical variables is expressed as references

## Example (revisited)

```
ancestor_of(Ancestor, Descendant) :-  
    child_of(Descendant, Ancestor).  
ancestor_of(Ancestor, Descendant) :-  
    child_of(Person, Ancestor),  
    ancestor_of(Person, Descendant).
```

- Ancestor
- Descendant
- Person

## In English

*Someone is ancestor of a **descendant**, if the **descendant** is his (or her) child, or if he (or she) has a child and this person is the ancestor of the **descendant**.*

binding of logical variables is expressed as references

## Example (revisited)

```

ancestor_of(Ancestor , Descendant) :-
    child_of(Descendant , Ancestor).
ancestor_of(Ancestor , Descendant) :-
    child_of(Person , Ancestor),
    ancestor_of(Person , Descendant).

```

- Ancestor
- Descendant
- Person

## In English

*Someone is ancestor of a descendant, if the descendant is his (or her) child, or if he (or she) has a child and this **person** is the **ancestor** of the descendant.*

binding of logical variables is expressed as references

## Example (revisited)

```
ancestor_of(Ancestor, Descendant) :-
    child_of(Descendant, Ancestor).
ancestor_of(Ancestor, Descendant) :-
    child_of(Person, Ancestor),
    ancestor_of(Person, Descendant).
```

- Ancestor
- Descendant
- Person

## In English

*Someone is ancestor of a descendant, if the descendant is his (or her) child, or if he (or she) has a child and this person is the ancestor of the descendant.*

binding of logical variables is expressed as references

# Declarative Reading

## Definition

the declarative reading of a program is its concept as (set of) logical formulas

# Declarative Reading

## Definition

the declarative reading of a program is its concept as (set of) logical formulas

## Analysis

### 1 specialisation

- if we remove clauses of a defined relation, then this relation becomes smaller; the program is **specialised**
- if the specialisation provides wrong answers, the original program certainly will

# Declarative Reading

## Definition

the declarative reading of a program is its concept as (set of) logical formulas

## Analysis

### 1 specialisation

- if we remove clauses of a defined relation, then this relation becomes smaller; the program is **specialised**
- if the specialisation provides wrong answers, the original program certainly will

### 2 generalisation

- if we remove goals from the body of a clause, the relation is extended; the program is **generalised**
- if the generalised program cannot derive correct facts, the original can neither



# Procedure Reading

## Example (multiplication)

### logic program

```
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).
```

### ground queries

```
:- plus(s(s(0)),s(0),s(s(s(0))))
```

# Procedure Reading

## Example (multiplication)

### logic program

```

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).

```

### ground queries

```

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

```

# Procedure Reading

## Example (multiplication)

### logic program

```

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).

```

### ground queries

```

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

```

# Procedure Reading

## Example (multiplication)

### logic program

```

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).

```

### ground queries

```

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

```

# Procedure Reading

## Example (multiplication)

### logic program

```

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).

```

### ground queries

```

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

```

# Procedure Reading

## Example (multiplication)

### logic program

```

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).

```

### ground queries

```

:- plus(s(s(0)),s(0),s(s(s(0))))
:- plus(s(0),s(0),s(s(0)))
:- plus(0,s(0),s(0))           X ↦ s(0)

```

# Procedure Reading

## Example (multiplication)

### logic program

```
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).
```

### ground queries

```
:- plus(s(s(0)),s(0),s(s(s(0))))  
:- plus(s(0),s(0),s(s(0)))  
:- plus(0,s(0),s(0))
```

### solved

## ... is Too Complicated

Example (renaming is needed)

logic program

```
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).
```

query

```
:- plus(s(s(0)),s(0),X)
```



## ... is Too Complicated

Example (renaming is needed)

logic program

```
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).
```

query

```
:- plus(s(s(0)),s(0),X)
```

## ... is Too Complicated

Example (renaming is needed)

logic program

```
plus(0,X,X).
plus(s(X1),Y1,s(Z1)) :- plus(X1,Y1,Z1).
times(0,X,0).
times(s(X),Y,Z) :- times(X,Y,U), plus(U,Y,Z).
```

query

```
:- plus(s(s(0)),s(0),X)      X1 ↦ s(0), Y1 ↦ s(0), X ↦ s(Z1)
```

## ... is Too Complicated

Example (renaming is needed)

logic program

```
plus(0,X,X).
plus(s(X1),Y1,s(Z1)) :- plus(X1,Y1,Z1).
times(0,X,0).
times(s(X),Y,Z) :- times(X,Y,U), plus(U,Y,Z).
```

query

```
:- plus(s(s(0)),s(0),X)           X ↦ s(Z1)
:- plus(s(0),s(0),Z1)
```

## ... is Too Complicated

Example (renaming is needed)

logic program

```
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).
```

query

```
:- plus(s(s(0)),s(0),X)           X ↦ s(Z1)
:- plus(s(0),s(0),Z1)
```

## ... is Too Complicated

Example (renaming is needed)

logic program

```
plus(0,X,X).
plus(s(X2),Y2,s(Z2)) :- plus(X2,Y2,Z2).
times(0,X,0).
times(s(X),Y,Z) :- times(X,Y,U), plus(U,Y,Z).
```

query

```
:- plus(s(s(0)),s(0),X)                X ↦ s(Z1)
:- plus(s(0),s(0),Z1)                X2 ↦ 0, Y2 ↦ s(0), Z1 ↦ s(Z2)
```

## ... is Too Complicated

Example (renaming is needed)

logic program

```
plus(0,X,X).
plus(s(X2),Y2,s(Z2)) :- plus(X2,Y2,Z2).
times(0,X,0).
times(s(X),Y,Z) :- times(X,Y,U), plus(U,Y,Z).
```

query

```
:- plus(s(s(0)),s(0),X)           X ↦ s(Z1)
:- plus(s(0),s(0),Z1)         Z1 ↦ s(Z2)
:- plus(0,s(0),Z2)
```

## ... is Too Complicated

Example (renaming is needed)

logic program

```

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).

```

query

```

:- plus(s(s(0)),s(0),X)           X ↦ s(Z1)
:- plus(s(0),s(0),Z1)          Z1 ↦ s(Z2)
:- plus(0,s(0),Z2)

```

## ... is Too Complicated

Example (renaming is needed)

logic program

```
plus(0,X3,X3).
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).
```

query

```
:- plus(s(s(0)),s(0),X)           X ↦ s(Z1)
:- plus(s(0),s(0),Z1)          Z1 ↦ s(Z2)
:- plus(0,s(0),Z2)           X3 ↦ s(0), Z2 ↦ s(0)
```



# ... is Too Complicated

Example (renaming is needed)

logic program

```
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).
```

query

```
:- plus(s(s(0)),s(0),X)           X ↦ s(Z1)
:- plus(s(0),s(0),Z1)          Z1 ↦ s(Z2)
:- plus(0,s(0),Z2)            Z2 ↦ s(0)
```

solution

# ... is Too Complicated

Example (renaming is needed)

logic program

```
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).
```

query

```
:- plus(s(s(0)),s(0),X)           X ↦ s(Z1)
:- plus(s(0),s(0),Z1)          Z1 ↦ s(Z2)
:- plus(0,s(0),Z2)           Z2 ↦ s(0)
```

solution      $X \mapsto s(s(s(0)))$

## Definition

- a **type** is a (possible infinite) set of terms
- types are conveniently defined by unary relations

## Definition

- a **type** is a (possible infinite) set of terms
- types are conveniently defined by unary relations

## Example

```
male(X).    female(X).
```

## Definition

- a **type** is a (possible infinite) set of terms
- types are conveniently defined by unary relations

## Example

```
male(X).    female(X).
```

## Definition

- to define complex types, **recursive** logic programs may be necessary
- the latter types are called **recursive types**
- recursive types, defined by unary recursive programs, are called **simple recursive types**
- a program defining a type is a **type definition**; a call to a predicate defining a type is a **type condition**

# Simple Recursive Types

## Example

```
is_tree(nil).  
is_tree(tree(Element,Left,Right)) :-  
    is_tree(Left),  
    is_tree(Right).
```

# Simple Recursive Types

## Example

```
is_tree(nil).  
is_tree(tree(Element,Left,Right)) :-  
    is_tree(Left),  
    is_tree(Right).
```

## Definition

- a type is **complete** if closed under the instance relation
- with every complete type  $T$  one associates an **incomplete** type  $IT$  which is a set of terms with instances in  $T$  and instances not in  $T$

# Simple Recursive Types

## Example

```
is_tree(nil).
is_tree(tree(Element,Left,Right)) :-
    is_tree(Left),
    is_tree(Right).
```

## Definition

- a type is **complete** if closed under the instance relation
- with every complete type  $T$  one associates an **incomplete** type  $IT$  which is a set of terms with instances in  $T$  and instances not in  $T$

## Example

- the type  $\{0, s(0), s(s(0)), \dots\}$  is complete
- the type  $\{X, 0, s(0), s(s(0)), \dots\}$  is incomplete



# Arithmetic

## Example

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

# Arithmetic

## Example

```
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

```
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

```
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

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

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

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

# Lists

## Notation

- `[]` empty list

# Lists

## Notation

- $[]$  empty list
- $[H|T]$  list with head  $H$  and tail  $T$

# Lists

## Notation

- $[]$  empty list
- $[H|T]$  list with head  $H$  and tail  $T$
- $[A]$   $[A|[]]$  list with one element

# Lists

## Notation

- $[]$  empty list
- $[H|T]$  list with head  $H$  and tail  $T$
- $[A]$   $[A|[]]$  list with one element
- $[A,B]$   $[A|[B|[]]]$  list with two elements

# Lists

## Notation

- $[]$  empty list
- $[H|T]$  list with head  $H$  and tail  $T$
- $[A]$   $[A|[]]$  list with one element
- $[A,B]$   $[A|[B|[]]]$  list with two elements
- $[A,B|T]$   $[A|[B|T]]$  list with at least two elements



# Lists

## Notation

- `[]` empty list
- `[H|T]` list with head  $H$  and tail  $T$
- `[A]` `[A|[]]` list with one element
- `[A,B]` `[A|[B|[]]]` list with two elements
- `[A,B|T]` `[A|[B|T]]` list with at least two elements

## Example

```
is_list([]). is_list([X|Xs]) :- is_list(Xs).
```

# Lists

## Notation

- `[]` empty list
- `[H|T]` list with head  $H$  and tail  $T$
- `[A]` `[A|[]]` list with one element
- `[A,B]` `[A|[B|[]]]` list with two elements
- `[A,B|T]` `[A|[B|T]]` list with at least two elements

## Example

```
is_list([]). is_list([X|Xs]) :- is_list(Xs).
```

## Notation

formal object	cons pair syntax	element syntax
<code>.(a, [])</code>	<code>[a []]</code>	<code>[a]</code>
<code>.(a, .(b, []))</code>	<code>[a [b []]]</code>	<code>[a,b]</code>

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

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

```

## Example

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

## Example

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

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

## Example

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

## Example (non termination)

```
:-& infinite.
```

```
infinite :- infinite.
```

## Example (non termination)

```
:-& infinite.
```

```
infinite :- infinite.
```

## Example (again, but different)

```
:-& winfinite.
```

```
winfinite :- winfinite.  
winfinite.
```

## Example (non termination)

```
:-& infinite.
```

```
infinite :- infinite.
```

## Example (again, but different)

```
:-& winfinite.
```

```
winfinite :- winfinite.  
winfinite.
```

## Example (non termination yet again)

```
:- hinfinites.  
:-& hinfinites, false.
```

```
hinfinites.  
hinfinites :- hinfinites.
```

# Termination Analysis

## Fact

- *for termination analysis only recursive calls (cycles in call tree) are essential*
- *let's remove non-recursive rules*

# Termination Analysis

## Fact

- *for termination analysis only recursive calls (cycles in call tree) are essential*
- *let's remove non-recursive rules*

## Example (specialised)

```

ancestor_of_2(Ancestor, Descendant) :- false,
    child_of(Descendant, Ancestor),
ancestor_of_2(Ancestor, Descendant) :-
    ancestor_of_2(Person, Descendant),
    child_of(Person, Ancestor).

```

## equivalently

```

ancestor_of_2(Ancestor, Descendant) :-
    ancestor_of_2(Person, Descendant),
    child_of(Person, Ancestor).

```

## Example

```
ancestor_of_2(Ancestor, Descendant) :-  
    ancestor_of_2(Person, Descendant),  
    child_of(Person, Ancestor).
```



## Example

```
ancestor_of_2(Ancestor, Descendant) :-  
    ancestor_of_2(Person, Descendant),  
    child_of(Person, Ancestor).
```

- **Ancestor** doesn't occur in first goal (= recursive call)
- no influence on termination behaviour

## Example

```
ancestor_of_2(Ancestor, Descendant) :-  
    ancestor_of_2(Person, Descendant),  
    child_of(Person, Ancestor).
```

- **Ancestor** doesn't occur in first goal (= recursive call)
- no influence on termination behaviour
- **Descendant** remains unchanged

## Example

```
ancestor_of_2(Ancestor, Descendant) :-  
    ancestor_of_2(Person, Descendant),  
    child_of(Person, Ancestor).
```

- **Ancestor** doesn't occur in first goal (= recursive call)
- no influence on termination behaviour
- **Descendant** remains unchanged
- last goal has no effect

## Example

```
ancestor_of_2(Ancestor, Descendant) :-
    ancestor_of_2(Person, Descendant),
    child_of(Person, Ancestor).
```

- **Ancestor** doesn't occur in first goal (= recursive call)
- no influence on termination behaviour
- **Descendant** remains unchanged
- last goal has no effect

## Example (specialised and generalised)

```
:-& ancestor_of_2(Ancestor, Descendant).
```

```
ancestor_of_2(Ancestor, Descendant) :-
    ancestor_of_2(Person, Descendant), false
    child_of(Person, Ancestor).
```

## Strong vs Weak Normalisation

Fact

*suppose the solution set is infinite, then the query*

*`:- Goal, false.`*

*cannot terminate*

## Strong vs Weak Normalisation

### Fact

*suppose the solution set is infinite, then the query*

*`:- Goal, false.`*

*cannot terminate*

### Example (specialised and generalised)

`:- child_of(X,X).`

`:- ancestor_of(Ancestor, Descendant), false.`

```
ancestor_of(Ancestor, Descendant) :-
    child_of(Person, Ancestor),
    ancestor_of(Person, Descendant).
```

# Termination queries

## Definition

(non) termination queries:

- `:- Goal, false .`
- `:-& Goal, false .`
- `:-$ Goal, false .`

goal terminates

goal does not terminate

goal terminates, but is expensive

# Termination queries

## Definition

(non) termination queries:

- `:- Goal, false .`
- `:-& Goal, false .`
- `:-$ Goal, false .`

goal terminates

goal does not terminate

goal terminates, but is expensive

## Example (useful termination queries)

```
:- ancestor_of_2(Ancestor, Descendant).
:-& ancestor_of_2(Ancestor, Descendant), false.
```