Computational Logic

Logic Programming

Cezary Kaliszyk **Georg Moser**

Institute of Computer Science @ UIBK

Winter 2015

Summary of Last Lecture

Fact

some care is necessary in pruning the search tree, as this may change the meaning of a program

Summary of Last Lecture

Fact

some care is necessary in pruning the search tree, as this may change the meaning of a program

Example

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

Summary of Last Lecture

Fact

some care is necessary in pruning the search tree, as this may change the meaning of a program

Example

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

Example

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

Summary of Last Lecture

Fact

some care is necessary in pruning the search tree, as this may change the meaning of a program

Example

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

Example

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

Observation

`select(a,[a,b,a,c],[a,b,c])` is in the meaning of the 1st program; `select_first(a,[a,b,a,c],[a,b,c])` is not in the meaning of the 2nd

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, complexity, efficient programs

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, complexity, efficient programs

Meta-logical Predicates

Definition

- meta-logical predicates are extensions of the first-order theory of logic programming

Meta-logical Predicates

Definition

- meta-logical predicates are extensions of the first-order theory of logic programming
- meta-logical predicates can
  1. query the state of the proof

Meta-logical Predicates

Definition

- meta-logical predicates are extensions of the first-order theory of logic programming
- meta-logical predicates can
  1. query the state of the proof
  2. treat variables as objects

Meta-logical Predicates

Definition

- meta-logical predicates are extensions of the first-order theory of logic programming
- meta-logical predicates can
  1. query the state of the proof
  2. treat variables as objects
  3. allow conversion of data structures to goals

Meta-logical Predicates

Definition

- meta-logical predicates are extensions of the first-order theory of logic programming
- meta-logical predicates can
  1. query the state of the proof
  2. treat variables as objects
  3. allow conversion of data structures to goals

Remark

meta-logical type predicates allow us to overcome two difficulties:

Meta-logical Predicates

Definition

- meta-logical predicates are extensions of the first-order theory of logic programming
- meta-logical predicates can
  1. query the state of the proof
  2. treat variables as objects
  3. allow conversion of data structures to goals

Remark

meta-logical type predicates allow us to overcome two difficulties:

1. variables in system predicates do not behave as intended

Meta-logical Predicates

Definition

- meta-logical predicates are extensions of the first-order theory of logic programming
- meta-logical predicates can
  1. query the state of the proof
  2. treat variables as objects
  3. allow conversion of data structures to goals

Remark

meta-logical type predicates allow us to overcome two difficulties:

1. variables in system predicates do not behave as intended
2. (logical) variables can be accidentally instantiated

Meta-logical Type Predicates

Definition

- `var`(*Term*) is true if *Term* is at present an uninstantiated variable
- `nonvar`(*Term*) is true if *Term* is at present not a variable
- `ground`(*Term*) is true if *Term* does not contain variables
- `compound`(*Term*) is true if *Term* is compound

Meta-logical Type Predicates

Definition

- `var`(*Term*) is true if *Term* is at present an uninstantiated variable
- `nonvar`(*Term*) is true if *Term* is at present not a variable
- `ground`(*Term*) is true if *Term* does not contain variables
- `compound`(*Term*) is true if *Term* is compound

Example

```
plus(X,Y,Z) :-
    nonvar(X), nonvar(Y), Z is X + Y.
plus(X,Y,Z) :-
    nonvar(X), nonvar(Z), Y is Z - X.
plus(X,Y,Z) :-
    nonvar(Y), nonvar(Z), X is Z - Y.
```

Example

```
unify(X,Y) :- var(X), var(Y), X = Y.
```

Example

```
unify(X,Y) :- var(X), var(Y), X = Y.
unify(X,Y) :- var(X), nonvar(Y), X = Y.
```

Example

```
unify(X,Y) :- var(X), var(Y), X = Y.
unify(X,Y) :- var(X), nonvar(Y), X = Y.
unify(X,Y) :- nonvar(X), var(Y), Y = X.
```

Example

```
unify(X,Y) :- var(X), var(Y), X = Y.
unify(X,Y) :- var(X), nonvar(Y), X = Y.
unify(X,Y) :- nonvar(X), var(Y), Y = X.
unify(X,Y) :-
    nonvar(X), nonvar(Y), constant(X), constant(Y),
    X = Y.
```

Example

```
unify(X,Y) :- var(X), var(Y), X = Y.
unify(X,Y) :- var(X), nonvar(Y), X = Y.
unify(X,Y) :- nonvar(X), var(Y), Y = X.
unify(X,Y) :-
    nonvar(X), nonvar(Y), constant(X), constant(Y),
    X = Y.
unify(X,Y) :-
    nonvar(X), nonvar(Y), compound(X), compound(Y),
    term_unify(X,Y).
```

Example

```
unify(X,Y) :- var(X), var(Y), X = Y.
unify(X,Y) :- var(X), nonvar(Y), X = Y.
unify(X,Y) :- nonvar(X), var(Y), Y = X.
unify(X,Y) :-
    nonvar(X), nonvar(Y), constant(X), constant(Y),
    X = Y.
unify(X,Y) :-
    nonvar(X), nonvar(Y), compound(X), compound(Y),
    term_unify(X,Y).

term_unify(X,Y) :-
    functor(X,F,N), functor(Y,F,N), unify_args(N,X,Y).
```

Example

```
unify(X,Y) :- var(X), var(Y), X = Y.
unify(X,Y) :- var(X), nonvar(Y), X = Y.
unify(X,Y) :- nonvar(X), var(Y), Y = X.
unify(X,Y) :-
    nonvar(X), nonvar(Y), constant(X), constant(Y),
    X = Y.
unify(X,Y) :-
    nonvar(X), nonvar(Y), compound(X), compound(Y),
    term_unify(X,Y).

term_unify(X,Y) :-
    functor(X,F,N), functor(Y,F,N), unify_args(N,X,Y).

unify_args(N,X,Y) :-
    N > 0, unify_arg(N,X,Y), N1 is N - 1, unify_args(N1,X,Y).
unify_args(0,X,Y).
```

Example

```
unify(X,Y) :- var(X), var(Y), X = Y.
unify(X,Y) :- var(X), nonvar(Y), X = Y.
unify(X,Y) :- nonvar(X), var(Y), Y = X.
unify(X,Y) :-
    nonvar(X), nonvar(Y), constant(X), constant(Y),
    X = Y.
unify(X,Y) :-
    nonvar(X), nonvar(Y), compound(X), compound(Y),
    term_unify(X,Y).

term_unify(X,Y) :-
    functor(X,F,N), functor(Y,F,N), unify_args(N,X,Y).

unify_args(N,X,Y) :-
    N > 0, unify_arg(N,X,Y), N1 is N - 1, unify_args(N1,X,Y).
unify_args(0,X,Y).

unify_arg(N,X,Y) :-
    arg(N,X,ArgX), arg(N,Y,ArgY), unify(ArgX,ArgY).
```

Comparing nonground terms

Definition

- X == Y is true if X and Y are identical constants, variables, or compound terms
- X \== Y is true if X and Y are not identical

Comparing nonground terms

Definition

- X == Y is true if X and Y are identical constants, variables, or compound terms
- X \== Y is true if X and Y are not identical

Example

```
:- X == 5
false
```

Unification with Occurs Check

Example

```
not_occurs_in(X,Y) :-
    var(Y), X \== Y.
not_occurs_in(X,Y) :-
    nonvar(Y), constant(Y).
not_occurs_in(X,Y) :-
    nonvar(Y), compound(Y),
    functor(Y,F,N), not_occurs_in(N,X,Y).
not_occurs_in(N,X,Y) :-
    N > 0, arg(N,Y,Arg), not_occurs_in(X,Arg), N1 is N - 1,
    not_occurs_in(N1,X,Y).
not_occurs_in(0,X,Y).
```

Unification with Occurs Check

Example

```
not_occurs_in(X,Y) :-
    var(Y), X \== Y.
not_occurs_in(X,Y) :-
    nonvar(Y), constant(Y).
not_occurs_in(X,Y) :-
    nonvar(Y), compound(Y),
    functor(Y,F,N), not_occurs_in(N,X,Y).

not_occurs_in(N,X,Y) :-
    N > 0, arg(N,Y,Arg), not_occurs_in(X,Arg), N1 is N - 1,
    not_occurs_in(N1,X,Y).
not_occurs_in(0,X,Y).

unify(X,Y) :- var(X), nonvar(Y), not_occurs_in(X,Y), X = Y.
unify(X,Y) :- nonvar(X), var(Y), not_occurs_in(Y,X), Y = X.
```

Meta-Variable Facility

Definition

the meta-variable facility allows a variable to appear as a goal or in the body

Meta-Variable Facility

Definition

the meta-variable facility allows a variable to appear as a goal or in the body

Example

```
X; Y :- X.
X; Y :- Y.
```

Meta-Variable Facility

Definition

the meta-variable facility allows a variable to appear as a goal or in the body

Example

```
X; Y :- X.
X; Y :- Y.
```

Other Control Predicates

- *fail*/0 *false*/0

```
:- fail.          :- false.
false             false
```

Meta-Variable Facility

Definition

the meta-variable facility allows a variable to appear as a goal or in the body

Example

```
X; Y :- X.
X; Y :- Y.
```

Other Control Predicates

- *fail*/0 *false*/0

```
:- fail.          :- false.
false             false
```

- *true*/0

```
:- true.
true
```

Example (Removal of Duplicates)

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

Example (Removal of Duplicates)

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

:- no_doubles([a,b,a,c,b],X).
X ↦ [a,c,b]
```

Example (Removal of Duplicates)

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

:- no_doubles([a,b,a,c,b],X).
X ↦ [a,c,b] ;
X ↦ [b,a,c,b]
```

Example (Removal of Duplicates)

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

:- no_doubles([a,b,a,c,b],X).
X ↦ [a,c,b] ;
X ↦ [b,a,c,b] ;
X ↦ [a,a,c,b]
```

Example (Removal of Duplicates)

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

:- no_doubles([a,b,a,c,b],X).
X ↦ [a,c,b] ;
X ↦ [b,a,c,b] ;
X ↦ [a,a,c,b] ;
X ↦ [a,b,a,c,b]
```

Example (Removal of Duplicates)

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

:- no_doubles([a,b,a,c,b],X).
X ↦ [a,c,b] ;
X ↦ [b,a,c,b] ;
X ↦ [a,a,c,b] ;
X ↦ [a,b,a,c,b] ;
false
```

Example (Removal of Duplicates)

```
no_doubles([],[]).
no_doubles([X|Xs],Ys) :-
    member(X,Xs),
    no_doubles(Xs,Ys).
no_doubles([X|Xs],[X|Ys]) :-
    \+ member(X,Xs),
    no_doubles(Xs,Ys).
```

Example (Removal of Duplicates)

```
no_doubles([],[]).
no_doubles([X|Xs],Ys) :-
    member(X,Xs),
    no_doubles(Xs,Ys).
no_doubles([X|Xs],[X|Ys]) :-
    \+ member(X,Xs),
    no_doubles(Xs,Ys).
```

```
:- no_doubles([a,b,a,c,b],X).
X ↦ [a,c,b]
```

Example (Removal of Duplicates)

```
no_doubles([],[]).
no_doubles([X|Xs],Ys) :-
    member(X,Xs),
    no_doubles(Xs,Ys).
no_doubles([X|Xs],[X|Ys]) :-
    \+ member(X,Xs),
    no_doubles(Xs,Ys).

:- no_doubles([a,b,a,c,b],X).
X ↦ [a,c,b] ;
false
```

Example (Removal of Duplicates)

```
no_doubles([],[]).
no_doubles([X|Xs],Ys) :-
    member(X,Xs),
    no_doubles(Xs,Ys).
no_doubles([X|Xs],[X|Ys]) :-
    \+ member(X,Xs),                    negation as failure
    no_doubles(Xs,Ys).

:- no_doubles([a,b,a,c,b],X).
X ↦ [a,c,b] ;
false
```

Example (Removal of Duplicates)

```
no_doubles([],[]).
no_doubles([X|Xs],Ys) :-
    member(X,Xs), !,                cut
    no_doubles(Xs,Ys).
no_doubles([X|Xs],[X|Ys]) :-

    no_doubles(Xs,Ys).
:- no_doubles([a,b,a,c,b],X).
X ↦ [a,c,b] ;
false
```

Example (Removal of Duplicates)

```
no_doubles([],[]).
no_doubles([X|Xs],Ys) :-
    member(X,Xs), !,                cut
    no_doubles(Xs,Ys).
no_doubles([X|Xs],[X|Ys]) :-

    no_doubles(Xs,Ys).
```

Effect of Cut

- ! succeeds

Example (Removal of Duplicates)

```
no_doubles([],[]).
no_doubles([X|Xs],Ys) :-
    member(X,Xs), !,                cut
    no_doubles(Xs,Ys).
no_doubles([X|Xs],[X|Ys]) :-

    no_doubles(Xs,Ys).
```

Effect of Cut

- ! succeeds
- ! fixes all choices between (and including) moment of matching rule's head with parent goal and cut

Example (Removal of Duplicates)

```
no_doubles([],[]).
no_doubles([X|Xs],Ys) :-
    member(X,Xs), !,                cut
    no_doubles(Xs,Ys).
no_doubles([X|Xs],[X|Ys]) :-

    no_doubles(Xs,Ys).
```

Effect of Cut

- ! succeeds
- ! fixes all choices between (and including) moment of matching rule's head with parent goal and cut

  if backtracking reaches !, the cut fails and the search continues from the last choice made before the clause containing ! was chosen

Example (Removal of Duplicates)

```
no_doubles([],[]).
no_doubles([X|Xs],Ys) :-
    member(X,Xs), !,                cut
    no_doubles(Xs,Ys).
no_doubles([X|Xs],[X|Ys]) :-

    no_doubles(Xs,Ys).
```

Effect of Cut

$$\mathtt{p}(t_{11},\ldots,t_{1n}) \;\mathtt{:-}\; A_1,\ldots,A_k\,.$$

$$\vdots$$

$$\mathtt{p}(t_{i1},\ldots,t_{in}) \;\mathtt{:-}\; B_1,\ldots,B_i,\;\mathtt{!},\;C_1,\ldots,C_j\,.$$

$$\vdots$$

$$\mathtt{p}(t_{m1},\ldots,t_{mn}) \;\mathtt{:-}\; D_1,\ldots,D_l\,.$$

Example (Removal of Duplicates)

```
no_doubles([],[]).
no_doubles([X|Xs],Ys) :-
    member(X,Xs), !,                cut
    no_doubles(Xs,Ys).
no_doubles([X|Xs],[X|Ys]) :-

    no_doubles(Xs,Ys).
```

Effect of Cut

$$\mathrm{p}(t_{11},\ldots,t_{1n}) \;:-\; A_1,\ldots,A_k\,.$$

$$\vdots$$

$$\mathrm{p}(t_{i1},\ldots,t_{in}) \;:-\; B_1,\ldots,B_i,\;!,\;C_1,\ldots,C_j\,.$$

$$\vdots$$

$$\mathrm{p}(t_{m1},\ldots,t_{mn}) \;:-\; D_1,\ldots,D_l\,.$$

Example (Removal of Duplicates)

```
no_doubles([],[]).
no_doubles([X|Xs],Ys) :-
    member(X,Xs), !,                cut
    no_doubles(Xs,Ys).
no_doubles([X|Xs],[X|Ys]) :-

    no_doubles(Xs,Ys).
```

Effect of Cut

$$\mathrm{p}(t_{11},\ldots,t_{1n}) \;:-\; A_1,\ldots,A_k.$$
$$\vdots$$
$$\mathrm{p}(t_{i1},\ldots,t_{in}) \;:-\; B_1,\ldots,B_i,\; !,\; C_1,\ldots,C_j.$$
$$\vdots \qquad\qquad \texttt{blocked}$$
$$\mathrm{p}(t_{m1},\ldots,t_{mn}) \;:-\; D_1,\ldots,D_l.$$

Examples of (Green) Cuts

Example (Without Cuts)

```
merge([X|Xs], [Y|Ys],[X|Zs]) :-
    X < Y,  merge(Xs,[Y|Ys],Zs).
merge([X|Xs],[Y|Ys],[X,Y|Zs]) :-
    X = Y,  merge(Xs,Ys,Zs).
merge([X|Xs],[Y|Ys],[Y|Zs]) :-
    X > Y,  merge([X|Xs],Ys,Zs).
merge(Xs,[],Xs) .
merge([],Ys,Ys) .
```

Examples of (Green) Cuts

Example (With Cuts)

```
merge([X|Xs], [Y|Ys],[X|Zs]) :-
    X < Y, !, merge(Xs,[Y|Ys],Zs).
merge([X|Xs],[Y|Ys],[X,Y|Zs]) :-
    X = Y, !, merge(Xs,Ys,Zs).
merge([X|Xs],[Y|Ys],[Y|Zs]) :-
    X > Y, !, merge([X|Xs],Ys,Zs).
merge(Xs,[],Xs) :- !.
merge([],Ys,Ys) :- !.
```

Examples of (Green) Cuts

Example (With Cuts)

```
merge([X|Xs], [Y|Ys],[X|Zs]) :-
    X < Y, !, merge(Xs,[Y|Ys],Zs).
merge([X|Xs],[Y|Ys],[X,Y|Zs]) :-
    X = Y, !, merge(Xs,Ys,Zs).
merge([X|Xs],[Y|Ys],[Y|Zs]) :-
    X > Y, !, merge([X|Xs],Ys,Zs).
merge(Xs,[],Xs) :- !.
merge([],Ys,Ys) :- !.
```

Example

```
minimum(X,Y,X) :- X =< Y, !.
minimum(X,Y,Y) :- X > Y, !.
```

Fact

(Green) cuts can greatly increase the efficiency by removing redundant computations

Fact

(Green) cuts can greatly increase the efficiency by removing redundant computations

Example

```
ordered([X]).
ordered([X,Y|Xs]) :- X ⩽ Y, ordered([Y|Xs]).

bubblesort(Xs,Ys) :-
    append(As,[X,Y|Bs],Xs),
    X > Y,
    append(As,[Y,X|Bs],Xs1),
    bubblesort(Xs1,Ys).
```

Fact

(Green) cuts can greatly increase the efficiency by removing redundant computations

Example

```
ordered([X]).
ordered([X,Y|Xs]) :- X ⩽ Y, ordered([Y|Xs]).

bubblesort(Xs,Ys) :-
    append(As,[X,Y|Bs],Xs),
    X > Y,
    append(As,[Y,X|Bs],Xs1),
    bubblesort(Xs1,Ys).
 bubblesort(Xs,Xs) :-
    ordered(Xs).
```

Fact

(Green) cuts can greatly increase the efficiency by removing redundant computations

Example

```
ordered([X]).
ordered([X,Y|Xs]) :- X ⩽ Y, ordered([Y|Xs]).

bubblesort(Xs,Ys) :-
    append(As,[X,Y|Bs],Xs),
    X > Y,
    append(As,[Y,X|Bs],Xs1),
    bubblesort(Xs1,Ys).
 bubblesort(Xs,Xs) :-
    ordered(Xs).

 :- bubblesort([3,2,1],Xs)
Xs ↦ [1,2,3]
```

Fact

(Green) cuts can greatly increase the efficiency by removing redundant computations

Example

```
ordered([X]).
ordered([X,Y|Xs]) :- X ⩽ Y, ordered([Y|Xs]).

bubblesort(Xs,Ys) :-
    append(As,[X,Y|Bs],Xs),
    X > Y, !,
    append(As,[Y,X|Bs],Xs1),
    bubblesort(Xs1,Ys).
 bubblesort(Xs,Xs) :-
    ordered(Xs), !.

 :- bubblesort([3,2,1],Xs)
Xs ↦ [1,2,3]
```

Definition (Negation as Failure)

- negation \+ is implemented using cut

Definition (Negation as Failure)

- negation \+ is implemented using cut
- the principle of negation is limited and known as negation as failure

Definition (Negation as Failure)

- negation \+ is implemented using cut
- the principle of negation is limited and known as negation as failure

Example

```
not X :- X, !, fail.
not X.
```

Definition (Negation as Failure)

- negation \+ is implemented using cut
- the principle of negation is limited and known as negation as failure

Example

```
not X :- X, !, fail.
not X.
```

Observation

if G does not terminate, `not`(G) may or may not terminate

Definition (Negation as Failure)

- negation \+ is implemented using cut
- the principle of negation is limited and known as negation as failure

Example

```
not X :- X, !, fail.
not X.
```

Observation

if G does not terminate, `not`(G) may or may not terminate

Example

```
married(abraham,sarah).
married(X,Y) :- married(Y,X)
:- not married(abraham,sarah).
```

Green vs Red Cuts

Definition

- a cut is green if the addition of the cut doesn't cange the meaning of the program; removing it makes the program potentially inefficient, but not wrong

Green vs Red Cuts

Definition

- a cut is green if the addition of the cut doesn't cange the meaning of the program; removing it makes the program potentially inefficient, but not wrong
- a cut is red if its presence changes the meaning of the program; removing it, changes the meaning and thus may make the program wrong

Green vs Red Cuts

Definition

- a cut is green if the addition of the cut doesn't cange the meaning of the program; removing it makes the program potentially inefficient, but not wrong
- a cut is red if its presence changes the meaning of the program; removing it, changes the meaning and thus may make the program wrong

Example (. . .)

```
delete([X|Ys],X,Zs) :- !, delete(Ys,X,Zs).
delete([Y|Ys],X,[Y|Zs]) :- Y ≠ X, !, delete(Ys,X,Zs).
delete([],X,[]).
```

Green vs Red Cuts

Definition

- a cut is green if the addition of the cut doesn't cange the meaning of the program; removing it makes the program potentially inefficient, but not wrong
- a cut is red if its presence changes the meaning of the program; removing it, changes the meaning and thus may make the program wrong

Example (Green Cut)

```
delete([X|Ys],X,Zs) :- !, delete(Ys,X,Zs).
delete([Y|Ys],X,[Y|Zs]) :- Y ≠ X, !, delete(Ys,X,Zs).
delete([],X,[]).
```

Example (...)

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

:- \+ delete([a,b],b,[a,b]).
```

Example (Red Cut)

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

:- \+ delete([a,b],b,[a,b]).
```

Example (Red Cut)

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

:- \+ delete([a,b],b,[a,b]).
```

Example (...)

```
member(X,[X|Xs]) :- !.
member(X,[Y|Ys]) :- member(X,Ys).
```

Example (Red Cut)

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

:- \+ delete([a,b],b,[a,b]).
```

Example (Red Cut)

```
member(X,[X|Xs]) :- !.
member(X,[Y|Ys]) :- member(X,Ys).
```

Example (Red Cut)

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

```
:- \+ delete([a,b],b,[a,b]).
```

Example (Red Cut)

```
member(X,[X|Xs]) :- !.
member(X,[Y|Ys]) :- member(X,Ys).
```

Example (...)

```
minimum(X,Y,X) :- X ⩽ Y, !.
minimum(X,Y,Y).
```

```
:- minimum(2,5,X)
```

$X = 2$

Example (Red Cut)

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

```
:- \+ delete([a,b],b,[a,b]).
```

Example (Red Cut)

```
member(X,[X|Xs]) :- !.
member(X,[Y|Ys]) :- member(X,Ys).
```

Example (...)

```
minimum(X,Y,X) :- X ⩽ Y, .
minimum(X,Y,Y).
```

```
:- minimum(2,5,X)
```

$X = 2$
$X = 5$

Example (Red Cut)

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

:- \+ delete([a,b],b,[a,b]).
```

Example (Red Cut)

```
member(X,[X|Xs]) :- !.
member(X,[Y|Ys]) :- member(X,Ys).
```

Example (...)

```
minimum(X,Y,X) :- X ⩽ Y, !.
minimum(X,Y,Y).

:- minimum(2,5,5)

true
```

Example (Red Cut)

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

```
:- \+ delete([a,b],b,[a,b]).
```

Example (Red Cut)

```
member(X,[X|Xs]) :- !.
member(X,[Y|Ys]) :- member(X,Ys).
```

Example (Bad Cut)

```
minimum(X,Y,X) :- X ⩽ Y, !.
minimum(X,Y,Y).
```

```
:- minimum(2,5,5)
```

```
true
```

Example (Truth Tables for Propositional Formulas)

```
and(A,B) :- A, B.
or(A,B) :- A; B.
implies(A,B) :- or(not(A),B).
```

Example (Truth Tables for Propositional Formulas)

```
and(A,B) :- A, B.
or(A,B) :- A; B.
implies(A,B) :- or(not(A),B).

bind(true).
bind(false).

table(A,B,E) :- bind(A), bind(B), row(A,B,E), fail.
```

Example (Truth Tables for Propositional Formulas)

```
and(A,B) :- A, B.
or(A,B) :- A; B.
implies(A,B) :- or(not(A),B).

bind(true).
bind(false).

table(A,B,E) :- bind(A), bind(B), row(A,B,E), fail.
table(_,_,_) :- nl.
```

Example (Truth Tables for Propositional Formulas)

```
and(A,B) :- A, B.
or(A,B) :- A; B.
implies(A,B) :- or(not(A),B).

bind(true).
bind(false).

table(A,B,E) :- bind(A), bind(B), row(A,B,E), fail.
table(_,_,_) :- nl.

row(A,B,_) :- wr(A), write(' '), wr(B), write(' '), fail.
row(_,_,E) :- E, !, wr(true), nl.
row(_,_,_) :- wr(false), nl.

wr(true) :- write('T').
wr(false) :- write('F').
```

Example (Truth Tables for Propositional Formulas)

```
and(A,B) :- A, B.
or(A,B) :- A; B.
implies(A,B) :- or(not(A),B).

bind(true).
bind(false).

table(A,B,E) :- bind(A), bind(B), row(A,B,E), fail.
table(_,_,_) :- nl.

row(A,B,_) :- wr(A), write(' '), wr(B), write(' '), fail.
row(_,_,E) :- E, !, wr(true), nl.
row(_,_,_) :- wr(false), nl.

wr(true) :- write('T').
wr(false) :- write('F').

:- table(A,B,or(A,implies(B,or(B,and(A,B))))).
```

Example (Truth Tables for Propositional Formulas)

```
and(A,B) :- A, B.
or(A,B) :- A; B.
implies(A,B) :- or(not(A),B).

bind(true).
bind(false).

table(A,B,E) :- bind(A), bind(B), row(A,B,E), fail.
table(_,_,_) :- nl.

row(A,B,_) :- wr(A), write(' '), wr(B), write(' '), fail.
row(_,_,E) :- E, !, wr(true), nl.
row(_,_,_) :- wr(false), nl.

wr(true) :- write('T').
wr(false) :- write('F').

:- table(A,B,or(A,implies(B,or(B,and(A,B))))).

:- table(A,B,false).
```

Cut-Fail Combinations

Example (Implementing $\neq$)

```
X ≠ X → !, fail.
X ≠ Y.
```

Cut-Fail Combinations

Example (Implementing $\neq$)

```
X ≠ X → !, fail.
X ≠ Y.
```

Example (Implementing `if_then_else`)

```
if_then_else(P,Q,R) :- P, !, Q.
if_then_else(P,Q,R) :- R.
```

Cut-Fail Combinations

Example (Implementing $\neq$)

```
X ≠ X → !, fail.
X ≠ Y.
```

Example (Implementing `if_then_else`)

```
if_then_else(P,Q,R) :- P, !, Q.
if_then_else(P,Q,R) :- R.
```

Example (Implementing `same_vars`)

```
same_var(foo,Y) :- var(Y), !, fail.
same_var(X,Y) :- var(X), var(Y).
```

Example (Facts)

```
father(andreas,boris).
father(andreas,christian).
father(andreas,doris).
father(boris,eva).
father(franz,georg).

female(doris).
female(eva).

mother(doris,franz).
mother(eva,georg).

male(andreas).
male(boris).
male(christian).
male(franz).
male(georg).
```

Example (Facts)

```
father(andreas,boris).
father(andreas,christian).
father(andreas,doris).
father(boris,eva).
father(franz,georg).

female(doris).
female(eva).

mother(doris,franz).
mother(eva,georg).

male(andreas).
male(boris).
male(christian).
male(franz).
male(georg).
```

Example

```
children(X,Cs) :- children(X,[],Cs).

children(X,A,Cs) :-
    father(X,C), \+ member(C,A), !, children(X,[C|A],Cs).
children(X,Cs,Cs).
```

Example (Facts)

```
father(andreas,boris).
father(andreas,christian).
father(andreas,doris).
father(boris,eva).
father(franz,georg).

female(doris).
female(eva).

mother(doris,franz).
mother(eva,georg).

male(andreas).
male(boris).
male(christian).
male(franz).
male(georg).
```

Example

```
children(X,Cs) :- children(X,[],Cs).

children(X,A,Cs) :-
    father(X,C), \+ member(C,A), !, children(X,[C|A],Cs).
children(X,Cs,Cs).
```

Example

```
children(X,Kids) :- setof(C,father(X,C),Kids).
```

Second-Order Programming

Definitions

- the predicate *bagof*(*Template*, *Goal*, *Bag*) unifies *Bag* with the alternatives of *Template* that meet *Goal*

Second-Order Programming

Definitions

- the predicate *bagof*(*Template*, *Goal*, *Bag*) unifies *Bag* with the alternatives of *Template* that meet *Goal*
- if *Goal* has free variables besides the one sharing with *Template* *bagof* will backtrack

Second-Order Programming

Definitions

- the predicate *bagof*(*Template*, *Goal*, *Bag*) unifies *Bag* with the alternatives of *Template* that meet *Goal*
- if *Goal* has free variables besides the one sharing with *Template* *bagof* will backtrack
- fails if *Goal* has no solutions

Second-Order Programming

Definitions

- the predicate *bagof*(*Template*, *Goal*, *Bag*) unifies *Bag* with the alternatives of *Template* that meet *Goal*
- if *Goal* has free variables besides the one sharing with *Template* *bagof* will backtrack
- fails if *Goal* has no solutions
- construct *Var* $\uparrow$ *Goal* tells *bagof* to existentially quantify *Var*

Second-Order Programming

Definitions

- the predicate *bagof*(*Template*, *Goal*, *Bag*) unifies *Bag* with the alternatives of *Template* that meet *Goal*
- if *Goal* has free variables besides the one sharing with *Template* *bagof* will backtrack
- fails if *Goal* has no solutions
- construct *Var* $\uparrow$ *Goal* tells *bagof* to existentially quantify *Var*
- the predicate *setof*(*Template*, *Goal*, *Bag*) is similar to *bagof* but sorts the obtained multi-set (bag) and removed duplicates

Second-Order Programming

Definitions

- the predicate *bagof*(*Template*,*Goal*,*Bag*) unifies *Bag* with the alternatives of *Template* that meet *Goal*
- if *Goal* has free variables besides the one sharing with *Template* *bagof* will backtrack
- fails if *Goal* has no solutions
- construct *Var* ↑ *Goal* tells *bagof* to existentially quantify *Var*
- the predicate *setof*(*Template*,*Goal*,*Bag*) is similar to *bagof* but sorts the obtained multi-set (bag) and removed duplicates

Example

```
kids(Kids) :- setof(Y, X ↑ (father(X,Y)),Kids).
```

Simple Application of Set Predicates

Example

```
no_doubles(Xs,Ys) :- setof(X,member(X,Xs),Ys).
```

Simple Application of Set Predicates

Example

```
no_doubles(Xs,Ys) :- setof(X,member(X,Xs),Ys).
```

Definition

- the predicate *findall*(*Template*, *Goal*, *Bag*) works as *bagof* if all excessive variables are existentially quantified
- SWI-Prolog notation for $\uparrow$: `^`