Logic Programming

Georg Moser

Institute of Computer Science @ UIBK

Summer 2015

Summary of Last Lecture

Two Choices

1. goal in sequence of goals
2. rule in logic program

substitution – avoid choice by always taking mgu

Computation Model of Logic Programs

- the choice of goal is arbitrary
  if there is a successful computation for a specific order, then there is a successful computation for any other order
- the choice of rules is essential
  not every choice will lead to a successful computation; thus the computation model is nondeterministic

Outline of the Lecture

Logic Programs

introduction, basic constructs, database and recursive programming, theory of logic programs

The Prolog Language

programming in pure prolog, arithmetic, structure inspection, meta-logical predicates, cuts, extra-logical predicates, how to program efficiently

Advanced Prolog Programming Techniques

nondeterministic programming, incomplete data structures, definite clause grammars, meta-programming , constraint logic programming

Outline of the Lecture

Logic Programs

introduction, basic constructs, database and recursive programming, theory of logic programs

The Prolog Language

programming in pure prolog, arithmetic, structure inspection, meta-logical predicates, cuts, extra-logical predicates, how to program efficiently

Advanced Prolog Programming Techniques

nondeterministic programming, incomplete data structures, definite clause grammars, meta-programming , constraint logic programming

Example

```
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

```
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).
```

Naming Conventions

Example

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

Naming Conventions

- predicates are often denoted together with their arity: *father*/2

Example

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

Naming Conventions

- predicates are often denoted together with their arity: $\mathit{father}/2$
- for each predicate a relation scheme is defined: $\mathit{father}(\mathit{Father}, \mathit{Child})$

Example

```
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).
```

Naming Conventions

- predicates are often denoted together with their arity: *father*/2
- for each predicate a relation scheme is defined: *father*(*Father*, *Child*)
- relation schemes are denoted in italics

Example

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

Naming Conventions

- predicates are often denoted together with their arity: *father/2*
- for each predicate a relation scheme is defined: *father(Father, Child)*
- relation schemes are denoted in italics
- variables should have mnemonic names; each new word in a variable is started with a capital letter: *NieceOrNephew*

Example

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

Naming Conventions

- predicates are often denoted together with their arity: *father/2*
- for each predicate a relation scheme is defined: *father(Father, Child)*
- relation schemes are denoted in italics
- variables should have mnemonic names; each new word in a variable is started with a capital letter: *NieceOrNephew*
- in predicates words are separated by underscores: *schedule_conflict*

Example

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

Naming Conventions

- predicates are often denoted together with their arity: *father/2*
- for each predicate a relation scheme is defined: *father(Father, Child)*
- relation schemes are denoted in italics
- variables should have mnemonic names; each new word in a variable is started with a capital letter: *NieceOrNephew*
- in predicates words are separated by underscores: *schedule_conflict*
- relation schemes are also used in commenting code

Example

```
daughter(X,Y) ← father(Y,X), female(X).
daughter(X,Y) ← mother(Y,X), female(X).

grandfather(X,Y) ← father(X,Z), father(Z,Y).
grandfather(X,Y) ← father(X,Z), mother(Z,Y).

parent(X,Y) ← father(X,Y).
parent(X,Y) ← mother(X,Y).
```

Example

```
daughter(X,Y) ← father(Y,X), female(X).
daughter(X,Y) ← mother(Y,X), female(X).

grandfather(X,Y) ← father(X,Z), father(Z,Y).
grandfather(X,Y) ← father(X,Z), mother(Z,Y).

parent(X,Y) ← father(X,Y).
parent(X,Y) ← mother(X,Y).
```

Relation Schemes

$$daughter(Daughter, Parent) \qquad parent(Parent, Child)$$
$$grandfather(Grandfather, GrandChild)$$

Example

```
daughter(X,Y) ← father(Y,X), female(X).
daughter(X,Y) ← mother(Y,X), female(X).

grandfather(X,Y) ← father(X,Z), father(Z,Y).
grandfather(X,Y) ← father(X,Z), mother(Z,Y).

parent(X,Y) ← father(X,Y).
parent(X,Y) ← mother(X,Y).
```

Relation Schemes

$$daughter(Daughter, Parent) \qquad parent(Parent, Child)$$
$$grandfather(Grandfather, GrandChild)$$

Example

```
brother(Brother,Sib) ←
    parent(Parent,Brother), parent(Parent,Sib), male(Brother).
```

Example

```
andreas ≠ boris.        andreas ≠ georg.    ...
andreas ≠ christian.    boris ≠ christian.
andreas ≠ franz.        boris ≠ franz.

brother(Brother,Sib) ←
    parent(Parent,Brother), parent(Parent,Sib),
    male(Brother), Brother ≠ Sib.
```

Example

```
andreas ≠ boris.        andreas ≠ georg.    ...
andreas ≠ christian.    boris ≠ christian.
andreas ≠ franz.        boris ≠ franz.

brother(Brother,Sib) ←
    parent(Parent,Brother), parent(Parent,Sib),
    male(Brother), Brother ≠ Sib.
```

Example

```
mother(Woman) ← mother(Woman,Child).
```

Example

```
andreas ≠ boris.        andreas ≠ georg.     ...
andreas ≠ christian.    boris ≠ christian.
andreas ≠ franz.        boris ≠ franz.

brother(Brother,Sib) ←
    parent(Parent,Brother), parent(Parent,Sib),
    male(Brother), Brother ≠ Sib.
```

Example

```
mother(Woman) ← mother(Woman,Child).
```

Observation

overloading with the same predicate name, but different arity, is fine

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.
```

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.
```

Why structure Data?

- helps to organise data
- rules can be written abstractly, hiding irrelevant detail
- modularity is improved

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.
```

Why structure Data?

- helps to organise data
- rules can be written abstractly, hiding irrelevant detail
- modularity is improved

The Art of Prolog says

We believe that the appearance of a program is important, particularly when attempting difficult problems

Recursive Rules

Example

```
grandpartent(Ancestor,Descendant) ←
  parent(Ancestor,Person), parent(Person,Descendant).

greatgrandpartent(Ancestor,Descendant) ←
  parent(Ancestor,Person), grandpartent(Person,Descendant).

greatgreatgrandpartent(Ancestor,Descendant) ←
  parent(Ancestor,Person), greatgrandpartent(Person,Descendant).
```

Recursive Rules

Example

```
grandpartent(Ancestor,Descendant) ←
  parent(Ancestor,Person), parent(Person,Descendant).

greatgrandpartent(Ancestor,Descendant) ←
  parent(Ancestor,Person), grandpartent(Person,Descendant).

greatgreatgrandpartent(Ancestor,Descendant) ←
  parent(Ancestor,Person), greatgrandpartent(Person,Descendant).
 ⋮
```

Recursive Rules

Example

```
grandpartent(Ancestor,Descendant) ←
  parent(Ancestor,Person), parent(Person,Descendant).

greatgrandpartent(Ancestor,Descendant) ←
  parent(Ancestor,Person), grandpartent(Person,Descendant).

greatgreatgrandpartent(Ancestor,Descendant) ←
  parent(Ancestor,Person), greatgrandpartent(Person,Descendant).
 ⋮
```

Example

```
ancestor(Ancestor,Descendant) ←
  parent(Ancestor,Person), ancestor(Person,Descendant).
```

Recursive Rules

Example

```
grandpartent(Ancestor,Descendant) ←
  parent(Ancestor,Person), parent(Person,Descendant).

greatgrandpartent(Ancestor,Descendant) ←
  parent(Ancestor,Person), grandpartent(Person,Descendant).

greatgreatgrandpartent(Ancestor,Descendant) ←
  parent(Ancestor,Person), greatgrandpartent(Person,Descendant).
 ⋮
```

Example

```
ancestor(Ancestor,Descendant) ←
  parent(Ancestor,Person), ancestor(Person,Descendant).

ancestor(Ancestor,Descendent) ← parent(Ancestor,Descendent).
```

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

$\texttt{r\_union\_s}(X_1,\dots,X_n) \leftarrow \texttt{r}(X_1,\dots,X_n).$
$\texttt{r\_union\_s}(X_1,\dots,X_n) \leftarrow \texttt{s}(X_1,\dots,X_n).$

Recursive Programming

Definition

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

Recursive Programming

Definition

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

Example

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

Recursive Programming

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

```
natural_number(0).
natural_number(s(X)) ← natural_number(X).
```

Arithmetic

Example

```
natural_number(0).
natural_number(s(X)) ← natural_number(X).
```

Example

```
plus(0,X,X) ← natural_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

```
natural_number(0).
natural_number(s(X)) ← natural_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

```
natural_number(0).
natural_number(s(X)) ← natural_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

$0 \leqslant X \leftarrow \texttt{natural\_number}(X).$
$s(X) \leqslant s(Y) \leftarrow X \leqslant Y.$

$\texttt{minimum}(N_1, N_2, N_1) \leftarrow N_1 \leqslant N_2.$
$\texttt{minimum}(N_1, N_2, N_2) \leftarrow N_2 \leqslant N_1.$

Example

```
0 ≤ X.
s(X) ≤ s(Y) ← X ≤ Y.
```

$\texttt{minimum}(N_1,N_2,N_1) \leftarrow N_1 \leqslant N_2.$

$\texttt{minimum}(N_1,N_2,N_2) \leftarrow N_2 \leqslant N_1.$

Example

```
0 ⩽ X.
s(X) ⩽ s(Y) ← X ⩽ Y.
```

$\texttt{minimum}(N_1,N_2,N_1) \leftarrow N_1 \leqslant N_2.$
$\texttt{minimum}(N_1,N_2,N_2) \leftarrow N_2 \leqslant N_1.$

Example

```
mod(X,Y,Z) ← Z < Y, times(Y,Q,W), plus(W,Z,X).
```

Example

```
0 ⩽ X.
s(X) ⩽ s(Y) ← X ⩽ Y.
```

$$\texttt{minimum}(N_1, N_2, N_1) \leftarrow N_1 \leqslant N_2.$$
$$\texttt{minimum}(N_1, N_2, N_2) \leftarrow N_2 \leqslant N_1.$$

Example

```
mod(X,Y,Z) ← Z < Y, times(Y,Q,W), plus(W,Z,X).
```

```
mod(X,Y,X) ← X < Y.
mod(X,Y,Z) ← plus(X1,Y,X), mod(X1,Y,Z).
```

Example

$0 \leqslant X.$

$s(X) \leqslant s(Y) \leftarrow X \leqslant Y.$

$minimum(N_1,N_2,N_1) \leftarrow N_1 \leqslant N_2.$

$minimum(N_1,N_2,N_2) \leftarrow N_2 \leqslant N_1.$

Example

```
mod(X,Y,Z) ← Z < Y, times(Y,Q,W), plus(W,Z,X).
```

```
mod(X,Y,X) ← X < Y.
mod(X,Y,Z) ← plus(X1,Y,X), mod(X1,Y,Z).
```

Example

```
ackermann(0,N,s(N)).
```

Example

$0 \leqslant X.$

$s(X) \leqslant s(Y) \leftarrow X \leqslant Y.$

$minimum(N_1,N_2,N_1) \leftarrow N_1 \leqslant N_2.$

$minimum(N_1,N_2,N_2) \leftarrow N_2 \leqslant N_1.$

Example

```
mod(X,Y,Z) ← Z < Y, times(Y,Q,W), plus(W,Z,X).
```

```
mod(X,Y,X) ← X < Y.
mod(X,Y,Z) ← plus(X1,Y,X), mod(X1,Y,Z).
```

Example

```
ackermann(0,N,s(N)).
ackermann(s(M),0,Val) ← ackermann(M,s(0),Val).
```

Example

```
0 ≤ X.
s(X) ≤ s(Y) ← X ≤ Y.
minimum(N1,N2,N1) ← N1 ≤ N2.
minimum(N1,N2,N2) ← N2 ≤ N1.
```

Example

```
mod(X,Y,Z) ← Z < Y, times(Y,Q,W), plus(W,Z,X).
mod(X,Y,X) ← X < Y.
mod(X,Y,Z) ← plus(X1,Y,X), mod(X1,Y,Z).
```

Example

```
ackermann(0,N,s(N)).
ackermann(s(M),0,Val) ← ackermann(M,s(0),Val).
ackermann(s(M),s(N),Val) ← ackermann(s(M),N,Val1),
    ackermann(M,Val1,Val).
```

Lists

Notation

- `[]` empty list

Lists

Notation

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

Lists

Notation

- [] empty list
- $[H \mid T]$ list with head H and tail T
- $[A]$ $[A \mid [\,]]$ list with one element

Lists

Notation

• []	empty list	
• $[H \mid T]$	list with head H and tail T	
• $[A]$	$[A \mid []]$	list with one element
• $[A, B]$	$[A \mid [B \mid []]]$	list with two elements

Lists

Notation

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

Example

```
is_list([]).   is_list([X|Xs]) ← is_list(Xs).
```

Notation (cont'd)

formal object	cons pair syntax	element syntax
`.(a,[])`	`[a|[]]`	`[a]`
`.(a,.(b,[]))`	`[a|[b|[]]]`	`[a,b]`

Example

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

```
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).
```

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).
```

Composing Recursive Programs

Example

delete/3 removes all occurrences of an element from a list

Composing Recursive Programs

Example

delete/3 removes all occurrences of an element from a list

Approach

1. craft the predicate with one (procedural) use in mind
2. afterwards see, if alternative uses make declarative sense

Composing Recursive Programs

Example

delete/3 removes all occurrences of an element from a list

Approach

1. craft the predicate with one (procedural) use in mind
2. afterwards see, if alternative uses make declarative sense

Example

```
delete([X|Xs],Z,?)      ← X =  Z                  .
delete([X|Xs],Z,?)      ← X ≠  Z                  .
```

Composing Recursive Programs

Example

delete/3 removes all occurrences of an element from a list

Approach

1. craft the predicate with one (procedural) use in mind
2. afterwards see, if alternative uses make declarative sense

Example

```
delete([X|Xs],Z,Ys)     ← X = Z , delete(Xs,Z,Ys).
```

Composing Recursive Programs

Example

delete/3 removes all occurrences of an element from a list

Approach

1. craft the predicate with one (procedural) use in mind
2. afterwards see, if alternative uses make declarative sense

Example

```
delete([X|Xs],Z,Ys)      ← X = Z , delete(Xs,Z,Ys).
delete([X|Xs],Z,?)       ← X ≠ Z                 .
```

Composing Recursive Programs

Example

delete/3 removes all occurrences of an element from a list

Approach

1. craft the predicate with one (procedural) use in mind
2. afterwards see, if alternative uses make declarative sense

Example

```
delete([X|Xs],Z,Ys)     ← X = Z , delete(Xs,Z,Ys).
delete([X|Xs],Z,[X|Ys]) ← X ≠ Z , delete(Xs,Z,Ys).
```

Composing Recursive Programs

Example

delete/3 removes all occurrences of an element from a list

Approach

1. craft the predicate with one (procedural) use in mind
2. afterwards see, if alternative uses make declarative sense

Example

```
delete([X|Xs],Z,Ys)     ← X = Z , delete(Xs,Z,Ys).
delete([X|Xs],Z,[X|Ys]) ← X ≠ Z , delete(Xs,Z,Ys).
delete([],X,[]).
```

Composing Recursive Programs

Example

delete/3 removes all occurrences of an element from a list

Approach

1. craft the predicate with one (procedural) use in mind
2. afterwards see, if alternative uses make declarative sense

Example

```
delete([X|Xs],Z,Ys)     ← X =  Z , delete(Xs,Z,Ys).
delete([X|Xs],Z,[X|Ys]) ← X ≠ Z , delete(Xs,Z,Ys).
delete([],X,[]).
```

```
delete([X|Xs],X,Ys) ← delete(Xs,X,Ys).
delete([X|Xs],Z,[X|Ys]) ← X ≠ Z, delete(Xs,Z,Ys).
delete([],X,[]).
```

Example

```
delete([X|Xs],X,Ys) ← delete(Xs,X,Ys).
delete([X|Xs],Z,[X|Ys]) ← X ≠ Z, delete(Xs,Z,Ys).
delete([],X,[]).
```

Example

```
delete([X|Xs],X,Ys) ← delete(Xs,X,Ys).
delete([X|Xs],Z,[X|Ys]) ← delete(Xs,Z,Ys).
delete([],X,[]).
```

Example

```
delete₂([X|Xs],X,Ys) ← delete₂(Xs,X,Ys).
delete₂([X|Xs],Z,[X|Ys]) ← delete₂(Xs,Z,Ys).
delete₂([],X,[]).
```

Example

```
delete₂([X|Xs],X,Ys) ← delete₂(Xs,X,Ys).
delete₂([X|Xs],Z,[X|Ys]) ← delete₂(Xs,Z,Ys).
delete₂([],X,[]).
```

```
← delete₂([a,b,c,b],b,[a,c])
true
```

```
← delete₂([a,b,c,b],b,[a,b,c,d])
true
```

Example

```
delete₂([X|Xs],X,Ys) ← delete₂(Xs,X,Ys).
delete₂([X|Xs],Z,[X|Ys]) ←  delete₂(Xs,Z,Ys).
delete₂([],X,[]).
```

```
← delete₂([a,b,c,b],b,[a,c])
true
← delete₂([a,b,c,b],b,[a,b,c,d])
true
```

Example

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

```
← delete([a],b,[a])
true
← select([a],b,X)
false
```

Example

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

Example

```
permutationsort(Xs,Ys) ← permutation(Xs,Ys), ordered(Ys).
```

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

Example

```
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

```
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

```
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

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

Example

```
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

```
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

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

Example

```
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

```
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 (Recursive Datastructures)

```
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).
```