Logic Programming

Georg Moser

Institute of Computer Science @ UIBK

Summer 2015

Summary of Last Lecture

Execution

- Prolog programs are executed using SLD resolution
  - leftmost and topdown selection
  - depth-first search with backtracking
- unification without occur check

Some Observations

1. goal invocation corresponds to procedure invocation
2. differences show when backtracking occurs
3. like LISP, Prolog is a declaration free, typeless language
4. data manipulation is achieved via unification

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

Arithmetic

Numbers

- integers
- floating point numbers

Arithmetic

Numbers

- integers
- floating point numbers

Definition

Prolog provides an arithmetical interface

Value `is` *Expression*

Arithmetic

Numbers

- integers
- floating point numbers

Definition

Prolog provides an arithmetical interface

Value `is` *Expression*

Example

`X is 3+5`	`8 is 3+5`	`N is N+1`
`X` $\mapsto$ `8`	`false`	*`nonsensical`*

Arithmetic Operations

- `+` `-` `*` `//` (integer division) `/` (float division)

Arithmetic Operations

- `+` `-` `*` `//` (integer division) `/` (float division)
- ...

Arithmetic Operations

- `+` `-` `*` `//` (integer division) `/` (float division)
- ...

Arithmetic Comparison Relations

- `<` `=<` `>` `>=`

Arithmetic Operations

- `+` `-` `*` `//` (integer division) `/` (float division)
- ...

Arithmetic Comparison Relations

- `<` `=<` `>` `>=`

```
?- 3 > 2.
true
```

Arithmetic Operations

- `+` `-` `*` `//` (integer division) `/` (float division)
- ...

Arithmetic Comparison Relations

- `<` `=<` `>` `>=`

```
?- 3 > X.
ERROR: >/2: Arguments are not sufficiently instantiated
```

Arithmetic Operations

- `+` `-` `*` `//` (integer division) `/` (float division)
- ...

Arithmetic Comparison Relations

- `<` `=<` `>` `>=`
- `=:=` (equality)

Arithmetic Operations

- `+` `-` `*` `//` (integer division) `/` (float division)
- ...

Arithmetic Comparison Relations

- `<` `=<` `>` `>=`
- `=:=` (equality)

```
?- 1+2 = 3.
false
```

Arithmetic Operations

- `+` `-` `*` `//` (integer division) `/` (float division)
- ...

Arithmetic Comparison Relations

- `<` `=<` `>` `>=`
- `=:=` (equality)

```
?- 1+2 =:= 3.
true
```

Arithmetic Operations

- `+` `-` `*` `//` (integer division) `/` (float division)
- . . .

Arithmetic Comparison Relations

- `<` `=<` `>` `>=`
- `=:=` (equality)
- `=\=` (disequality)

Arithmetic Operations

- `+` `-` `*` `//` (integer division) `/` (float division)
- ...

Arithmetic Comparison Relations

- `<` `=<` `>` `>=`
- `=:=` (equality)
- `=\=` (disequality)

```
?- 1+2 =\= 3.
false
```

Arithmetic Operations

- `+` `-` `*` `//` (integer division) `/` (float division)
- ...

Arithmetic Comparison Relations

- `<` `=<` `>` `>=`
- `=:=` (equality)
- `=\=` (disequality)

```
?- 1+2 =\= 2+1.
false
```

Arithmetic Operations

- `+` `-` `*` `//` (integer division) `/` (float division)
- ...

Arithmetic Comparison Relations

- `<` `=<` `>` `>=`
- `=:=` (equality)
- `=\=` (disequality)

```
?- 1+2 =\= 2+1.
false
```

Non Standard Predicates

- `between`(*Low*,*High*,*Value*) is true when
  1. *Value* is an integer, and $Low \leqslant Value \leqslant High$
  2. *Value* is a variable, and $Value \in [Low, High]$

Arithmetic Operations

- `+` `-` `*` `//` (integer division) `/` (float division)
- ...

Arithmetic Comparison Relations

- `<` `=<` `>` `>=`
- `=:=` (equality)
- `=\=` (disequality)

```
?- 1+2 =\= 2+1.
false
```

Non Standard Predicates

- `between`(*Low*,*High*,*Value*) is true when
  1. *Value* is an integer, and $Low \leqslant Value \leqslant High$
  2. *Value* is a variable, and $Value \in [Low, High]$
- `succ`(Int1,Int2) ...

Example (Factorials)

```
factorial(0,s(0)).

factorial(s(N),F) ←
  factorial(N,F1),
  times(s(N),F1,F).
```

Example (Factorials)

```
factorial(0,s(0)).

factorial(s(N),F) ←
  factorial(N,F1),
  times(s(N),F1,F).
```

```
factorial(N,F) ←
  N>0, N1 is N-1,
  factorial(N1,F1),
  F is N * F1.

factorial(0,1).
```

Example (Factorials)

```
factorial(0,s(0)).

factorial(s(N),F) ←
  factorial(N,F1),
  times(s(N),F1,F).
```

```
factorial(N,F) ←
  N>0, N1 is N-1,
  factorial(N1,F1),
  F is N * F1.

factorial(0,1).
```

Example (Fibonacci Numbers)

```
fibonacci(0,1).
fibonacci(1,1).
fibonacci(N,X) :-
    N > 1,
    fibonacci(N-1,Y),
    fibonacci(N-2,Z),
    X = Y+Z.

?- fibonacci(3,X).

false
```

Example (Factorials)

```
factorial(0,s(0)).

factorial(s(N),F) ←
  factorial(N,F1),
  times(s(N),F1,F).
```

```
factorial(N,F) ←
  N>0, N1 is N-1,
  factorial(N1,F1),
  F is N * F1.

factorial(0,1).
```

Example (Fibonacci Numbers)

```
fibonacci(0,1).
fibonacci(1,1).
fibonacci(N,X) :-
    N > 1,
    fibonacci(N-1,Y),
    fibonacci(N-2,Z),
    X = Y+Z.

?- fibonacci(3,X).

false
```

Example (Factorials)

```
factorial(0,s(0)).

factorial(s(N),F) ←
  factorial(N,F1),
  times(s(N),F1,F).
```

```
factorial(N,F) ←
  N>0, N1 is N-1,
  factorial(N1,F1),
  F is N * F1.

factorial(0,1).
```

Example (Fibonacci Numbers)

```
fibonacci(0,1).
fibonacci(1,1).
fibonacci(N,X) :-
    N > 1,
    N1 is N-1, fibonacci(N1,Y),
    N2 is N-2, fibonacci(N2,Z),
    X = Y+Z.

?- fibonacci(3,X).
```

Example (Factorials)

```
factorial(0,s(0)).

factorial(s(N),F) ←
  factorial(N,F1),
  times(s(N),F1,F).
```

```
factorial(N,F) ←
  N>0, N1 is N-1,
  factorial(N1,F1),
  F is N * F1.

factorial(0,1).
```

Example (Fibonacci Numbers)

```
fibonacci(0,1).
fibonacci(1,1).
fibonacci(N,X) :-
    N > 1,
    N1 is N-1, fibonacci(N1,Y),
    N2 is N-2, fibonacci(N2,Z),
    X = Y+Z.
```

```
?- fibonacci(3,X).
```

X ↦ 1+1

true

Example (Factorials)

```
factorial(0,s(0)).

factorial(s(N),F) ←
  factorial(N,F1),
  times(s(N),F1,F).
```

```
factorial(N,F) ←
  N>0, N1 is N-1,
  factorial(N1,F1),
  F is N * F1.

factorial(0,1).
```

Example (Fibonacci Numbers)

```
fibonacci(0,1).
fibonacci(1,1).
fibonacci(N,X) :-
    N > 1,
    N1 is N-1, fibonacci(N1,Y),
    N2 is N-2, fibonacci(N2,Z),
    X = Y+Z.

?- fibonacci(3,X).

X ↦ 1+1

true
```

Example (Factorials)

```
factorial(0,s(0)).

factorial(s(N),F) ←
  factorial(N,F1),
  times(s(N),F1,F).
```

```
factorial(N,F) ←
  N>0, N1 is N-1,
  factorial(N1,F1),
  F is N * F1.

factorial(0,1).
```

Example (Fibonacci Numbers)

```
fibonacci(0,1).
fibonacci(1,1).
fibonacci(N,X) :-
    N > 1,
    N1 is N-1, fibonacci(N1,Y),
    N2 is N-2, fibonacci(N2,Z),
    X is Y+Z.

?- fibonacci(3,X).
```

Example (Factorials)

```
factorial(0,s(0)).

factorial(s(N),F) ←
  factorial(N,F1),
  times(s(N),F1,F).
```

```
factorial(N,F) ←
  N>0, N1 is N-1,
  factorial(N1,F1),
  F is N * F1.

factorial(0,1).
```

Example (Fibonacci Numbers)

```
fibonacci(0,1).
fibonacci(1,1).
fibonacci(N,X) :-
    N > 1,
    N1 is N-1, fibonacci(N1,Y),
    N2 is N-2, fibonacci(N2,Z),
    X is Y+Z.
```

```
?- fibonacci(3,X).
```

X ↦ 2

true

Transforming Recursion into Iteration

Definitions

- a Prolog clause is called iterative if
  1. it has one recursive call, and
  2. zero or more calls to system predicates, before the recursive call
- a Prolog procedure is iterative if contains only unit clauses and iterative clauses

Transforming Recursion into Iteration

Definitions

- a Prolog clause is called iterative if
  1. it has one recursive call, and
  2. zero or more calls to system predicates, before the recursive call
- a Prolog procedure is iterative if contains only unit clauses and iterative clauses

Example (Factorial Iterative, Version 1)

```
factorial(N,F) ← factorial(0,N,1,F).

factorial(I,N,T,F) ←
    I < N, I1 is I + 1, T1 is T*I1, factorial(I1,N,T1,F).
factorial(N,N,F,F).
```

Transforming Recursion into Iteration

Definitions

- a Prolog clause is called iterative if
  1. it has one recursive call, and
  2. zero or more calls to system predicates, before the recursive call
- a Prolog procedure is iterative if contains only unit clauses and iterative clauses

Example (Factorial Iterative, Version 1)

```
factorial(N,F) ← factorial(0,N,1,F).

factorial(I,N,T,F) ←
    I < N, I1 is I + 1, T1 is T*I1, factorial(I1,N,T1,F).
factorial(N,N,F,F).
```

Transforming Recursion into Iteration

Definitions

- a Prolog clause is called iterative if
  1. it has one recursive call, and
  2. zero or more calls to system predicates, before the recursive call
- a Prolog procedure is iterative if contains only unit clauses and iterative clauses

Example (Factorial Iterative, Version 1)

```
factorial(N,F) ← factorial(0,N,1,F).

factorial(I,N,T,F) ←
    I < N, I1 is I + 1, T1 is T*I1, factorial(I1,N,T1,F).
factorial(N,N,F,F).
```

Example (Factorial Iterative, Version 2)

```
factorial(N,F) ← factorial(N,1,F).

factorial(N,T,F) ←
    N > 0, T1 is T * N, N1 is N-1, factorial(N1,T1,F).
factorial(0,F,F).
```

Example (Factorial Iterative, Version 2)

```
factorial(N,F) ← factorial(N,1,F).

factorial(N,T,F) ←
    N > 0, T1 is T * N, N1 is N-1, factorial(N1,T1,F).
factorial(0,F,F).
```

Example

```
between(I,J,I) ← I ⩽ J.
between(I,J,K) ← I < J, I1 is I+1, between(I1,J,K).
```

Example (Factorial Iterative, Version 2)

```
factorial(N,F) ← factorial(N,1,F).

factorial(N,T,F) ←
    N > 0, T1 is T * N, N1 is N-1, factorial(N1,T1,F).
factorial(0,F,F).
```

Example

```
between(I,J,I) ← I ⩽ J.
between(I,J,K) ← I < J, I1 is I+1, between(I1,J,K).
```

Example

```
sumlist(Is,Sum) ← sumlist(Is,0,Sum).

sumlist([I|Is],Temp,Sum) ←
    Temp1 is Temp + I,sumlist(Is,Temp1,Sum).
sumlist([],Sum,Sum).
```

Example

```
maximum([X|Xs],M) ← maximum(Xs,X,M).

maximum([X|Xs],Y,M) ←
    X ⩽ Y, maximum(Xs,Y,M).
maximum([X|Xs],Y,M) ←
    X > Y, maximum(Xs,X,M).
maximum([],M,M).
```

Example

```
maximum([X|Xs],M) ← maximum(Xs,X,M).

maximum([X|Xs],Y,M) ←
    X ⩽ Y, maximum(Xs,Y,M).
maximum([X|Xs],Y,M) ←
    X > Y, maximum(Xs,X,M).
maximum([],M,M).
```

Example

```
length([X|Xs],N) ←
    N > 0, N1 is N - 1, length(Xs,N1).
length([],0).
```

Example

```
maximum([X|Xs],M) ← maximum(Xs,X,M).

maximum([X|Xs],Y,M) ←
    X ⩽ Y, maximum(Xs,Y,M).
maximum([X|Xs],Y,M) ←
    X > Y, maximum(Xs,X,M).
maximum([],M,M).
```

Example

```
length([X|Xs],N) ←
    N > 0, N1 is N - 1, length(Xs,N1).
length([],0).

length([X|Xs],N) ←
    length(Xs,N1), N is N1 + 1.
length([],0).
```

Example

```
maximum([X|Xs],M) ← maximum(Xs,X,M).

maximum([X|Xs],Y,M) ←
    X ⩽ Y, maximum(Xs,Y,M).
maximum([X|Xs],Y,M) ←
    X > Y, maximum(Xs,X,M).
maximum([],M,M).
```

Example

```
length([X|Xs],N) ←
    N > 0, N1 is N - 1, length(Xs,N1).
length([],0).

length([X|Xs],N) ←
    length(Xs,N1), N is N1 + 1.
length([],0).
```

Type Predicates

Recall

type predicates are unary relations concerning the type of a term

Type Predicates

Recall

type predicates are unary relations concerning the type of a term

Definition

- `integer`: type check for an integer
- `atom`: type check for an atom
- `compound`: type check for a compound term

Type Predicates

Recall

type predicates are unary relations concerning the type of a term

Definition

- `integer`: type check for an integer
- `atom`: type check for an atom
- `compound`: type check for a compound term

Example

```
constant(X) ← integer(X).
constant(X) ← atom(X).
```

Example

```
flatten([X|Xs],Ys) ←
    flatten(X,Ys1), flatten(Xs,Ys2),
    append(Ys1,Ys2,Ys).
```

Example

```
flatten([X|Xs],Ys) ←
    flatten(X,Ys1), flatten(Xs,Ys2),
    append(Ys1,Ys2,Ys).
flatten(X,[X]) ← constant(X), X ≠ [].
```

Example

```
flatten([X|Xs],Ys) ←
    flatten(X,Ys1), flatten(Xs,Ys2),
    append(Ys1,Ys2,Ys).
flatten(X,[X]) ← constant(X), X ≠ [].
flatten([],[]).
```

Example

```
flatten([X|Xs],Ys) ←
    flatten(X,Ys1), flatten(Xs,Ys2),
    append(Ys1,Ys2,Ys).
flatten(X,[X]) ← constant(X), X ≠ [].
flatten([],[]).
?- flatten([[a],[b,[c,d]],e],[a,b,c,d,e])
true
```

Example

```
flatten([X|Xs],Ys) ←
    flatten(X,Ys1), flatten(Xs,Ys2),
    append(Ys1,Ys2,Ys).
flatten(X,[X]) ← constant(X), X ≠ [].
flatten([],[]).
?- flatten([[a],[b,[c,d]],e],[a,b,c,d,e])
true
```

Example

```
flatten(Xs,Ys) ← flatten(Xs,[],Ys).
```

Example

```
flatten([X|Xs],Ys) ←
    flatten(X,Ys1), flatten(Xs,Ys2),
    append(Ys1,Ys2,Ys).
flatten(X,[X]) ← constant(X), X ≠ [].
flatten([],[]).
?- flatten([[a],[b,[c,d]],e],[a,b,c,d,e])
true
```

Example

```
flatten(Xs,Ys) ← flatten(Xs,[],Ys).
flatten([X|Xs],S,Ys) ←
    list(X), flatten(X,[Xs|S],Ys).
```

Example

```
flatten([X|Xs],Ys) ←
    flatten(X,Ys1), flatten(Xs,Ys2),
    append(Ys1,Ys2,Ys).
flatten(X,[X]) ← constant(X), X ≠ [].
flatten([],[]).
?- flatten([[a],[b,[c,d]],e],[a,b,c,d,e])
true
```

Example

```
flatten(Xs,Ys) ← flatten(Xs,[],Ys).
flatten([X|Xs],S,Ys) ←
    list(X), flatten(X,[Xs|S],Ys).
flatten([X|Xs],S,[X,Ys]) ←
    constant(X), X ≠ [], flatten(Xs,S,Ys).
```

Example

```
flatten([X|Xs],Ys) ←
    flatten(X,Ys1), flatten(Xs,Ys2),
    append(Ys1,Ys2,Ys).
flatten(X,[X]) ← constant(X), X ≠ [].
flatten([],[]).
?- flatten([[a],[b,[c,d]],e],[a,b,c,d,e])
true
```

Example

```
flatten(Xs,Ys) ← flatten(Xs,[],Ys).
flatten([X|Xs],S,Ys) ←
    list(X), flatten(X,[Xs|S],Ys).
flatten([X|Xs],S,[X,Ys]) ←
    constant(X), X ≠ [], flatten(Xs,S,Ys).
flatten([],[X|S],Ys) ← flatten(X,S,Ys).
flatten([],[],[]).
```

Accessing compound terms

Definition

- `functor`(*Term*,*F*,*Arity*) is true, if *Term* is a compound term, whose principal functor is *F* with arith *Arity*

Accessing compound terms

Definition

- `functor`(*Term*,*F*,*Arity*) is true, if *Term* is a compound term, whose principal functor is *F* with arith *Arity*
- `arg`(*N*,*Term*,*Arg*) is true, if *Arg* is the N^{th} argument of *Term*

Accessing compound terms

Definition

- `functor`(*Term*,*F*,*Arity*) is true, if *Term* is a compound term, whose principal functor is *F* with arith *Arity*
- `arg`(*N*,*Term*,*Arg*) is true, if *Arg* is the N^{th} argument of *Term*

Example

$\leftarrow$ `functor(father(haran,lot),F,A)`

`F` $\mapsto$ `father`

`A` $\mapsto$ `2`

Accessing compound terms

Definition

- `functor`(*Term*,*F*,*Arity*) is true, if *Term* is a compound term, whose principal functor is *F* with arith *Arity*
- `arg`(*N*,*Term*,*Arg*) is true, if *Arg* is the N^{th} argument of *Term*

Example

$\leftarrow$ `functor(father(haran,lot),F,A)`

`F` $\mapsto$ `father`

`A` $\mapsto$ `2`

Example

$\leftarrow$ `arg(2,father(haran,lot),X)`

`X` $\mapsto$ `lot`

Example

```
subterm(Term,Term).
subterm(Sub,Term) ←
    compound(Term),
    functor(Term,F,N),
    subterm(N,Sub,Term).

subterm(N,Sub,Term) ←
    N > 1,
    N1 is N - 1,
    subterm(N1,Sub,Term).
subterm(N,Sub,Term) ←
    arg(N,Term,Arg),
    subterm(Sub,Arg).
```

Definition

- *Term* `=..` *List* is true if *List* is a list whose head is the principal functor of *Term*, and whose tail is the list of arguments of *Term*
- the operator `=..` is also called `univ`

Definition

- *Term* `=..` *List* is true if *List* is a list whose head is the principal functor of *Term*, and whose tail is the list of arguments of *Term*
- the operator `=..` is also called `univ`

Example

$\leftarrow$ `father(haran,lot) =.. Xs`

`X` $\mapsto$ `[father,haran,lot]`

Definition

- *Term* `=..` *List* is true if *List* is a list whose head is the principal functor of *Term*, and whose tail is the list of arguments of *Term*
- the operator `=..` is also called `univ`

Example

$\leftarrow$ `father(haran,lot) =.. Xs`

`X` $\mapsto$ `[father,haran,lot]`

Remark

- programs written with `functor` and `arg` can also be written with `univ`

Definition

- *Term* `=..` *List* is true if *List* is a list whose head is the principal functor of *Term*, and whose tail is the list of arguments of *Term*
- the operator `=..` is also called `univ`

Example

$\leftarrow$ `father(haran,lot) =.. Xs`

`X` $\mapsto$ `[father,haran,lot]`

Remark

- programs written with `functor` and `arg` can also be written with `univ`
- programs using `univ` are typically simpler

Definition

- *Term* `=..` *List* is true if *List* is a list whose head is the principal functor of *Term*, and whose tail is the list of arguments of *Term*
- the operator `=..` is also called `univ`

Example

$\leftarrow$ `father(haran,lot) =.. Xs`

`X` $\mapsto$ `[father,haran,lot]`

Remark

- programs written with `functor` and `arg` can also be written with `univ`
- programs using `univ` are typically simpler
- programs using `functor` and `arg` are more efficient

Definition

- *Term* `=..` *List* is true if *List* is a list whose head is the principal functor of *Term*, and whose tail is the list of arguments of *Term*
- the operator `=..` is also called `univ`

Example

```
← father(haran,lot) =.. Xs
X ↦ [father,haran,lot]
```

Remark

- programs written with `functor` and `arg` can also be written with `univ`
- programs using `univ` are typically simpler
- programs using `functor` and `arg` are more efficient
- `univ` can be built from `functor` and `arg`

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

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

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

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

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.
```