CSCC24 – Principles of Programming Languages

Anya Tafliovich[1]

[1] with thanks to S.McIlraigh, G.Penn, P.Ragde

Logic Programming and Prolog

Logic programming languages are neither procedural nor functional.

- Specify **relations** between objects
  - `larger(3,2)`
  - `father(tom,jane)`
- Separate logic from control:
  - Programmer declares **what** facts and relations are true.
  - System determines **how** to use facts to solve problems.
  - System **instantiates** variables in order to make relations true.

Prolog

Suppose we state these **facts**:

```
male(charlie).
female(alice).
male(bob).
female(eve).
```

```
parent(charlie,bob).
parent(eve,bob).
parent(charlie,alice).
parent(eve,alice).
```

We can then make **queries**:

```
?- male(charlie).
true
```

```
?- male(eve).
false
```

```
?- female(Person).
Person = alice;
Person = eve;
false
```

```
?- parent(Person, bob).
Person = charlie;
Person = eve;
false
```

```
?- parent(Person, bob),
   female(Person).
Person = eve;
false
```

Prolog

We can also state **rules**, such as this one:

```
sibling(X, Y) :- parent(P, X),
                 parent(P, Y).
```

Then the queries become more interesting:

```
?- sibling(charlie, eve).
false

?- sibling(bob, Sib).
Sib = bob;
Sib = alice;
Sib = bob;
Sib = alice;
false
```

Logic vs Functional Programming

In FP, we program with **functions**.

- $f(x, y, z) = x * y + z - 2$
- Given a function, we can only ask one kind of question:

  Here are the argument values; tell me what the function's value is.

In LP, we program with **relations**.

- $r(x, y, z) = \{(1, 2, 3), (9, 5, 13), (0, 0, 4)\}$
- Given a predicate, we can ask many kinds of question:

  Here are some of the argument values; tell me what the others have to be in order to make a true statement.

Logic Programming

- A program consists of facts and rules.
- Running a program means asking queries.
- The language tries to find one or more ways to prove that the query is true.
- This may have the side effect of freezing variable values.
- The language determines how to do all of this, *not* the program.
- How does the language do it? Using unification, resolution, and backtracking.

Some Prolog Syntax

Layer 0: **constants** and **variables**:

- Constants are:
  - Atoms: any string consisting of alphanumeric characters and underscore (_) that begins with a lowercase letter.
  - Numbers.
- Variables are strings that begin with '_' or an uppercase letter.

Some Prolog Syntax

```
<const>  ::= <atom> | <number>
<var>    ::= <ucase> <string> | _<string>
<atom>   ::= <lcase> <string>
<letter> ::= <ucase> | <lcase>
<string> ::= epsilon | <letter><string> |
             <number><string> | _<string>
<ucase>  ::= A | ... | Z
<lcase>  ::= a | ... | z
<number> ::= ...
```

Some Prolog Syntax

Layer 1: **terms**.

These are inductively defined:

- Constants and variables are terms.
- Compound terms – applications of any *n*-ary **functor** to any *n* terms – are terms.
- Nothing else is a term.

Note: Prolog distinguishes numbers and variables from other terms in certain contexts, but is otherwise untyped/monotyped.

Some Prolog Syntax

Layer 2: **atomic formulae**.
These consist of an *n*-ary relation (also called **predicate**) applied to *n* terms

Note: formulae look a lot like terms because of Prolog's weak typing. But formulae are either true or false; terms denote entities in the world.

In Prolog, atomic formulae can be used in three ways:

- As **facts**: in which they assert truth,
- As **queries**: in which they enquire about truth,
- As components of more complicated statements (see below).

Some Prolog Syntax

```
<term> ::= <const> | <var> |
           <functor>'(' <term> { , <term> } ')'

<pred>   ::= <pname>'(' <term> { , <term> } ')'
```

Prolog Queries

A **query** is a proposed fact that is to be proven.

- If the query has no variables, returns true/false.
- If the query has variables, returns appropriate values of variables (called a substitution).

```
?- male(charlie).
true

?- male(eve).
false

?- female(Person).
Person = alice;
Person = eve;
false
```

Some Prolog Syntax

Layer 3: **complex formulae**.

These are formed by combining simpler formulae. We only discuss a couple:

- conjunction:
  in Prolog, and looks like `','`

Can be used as queries, but not as facts. Using multiple facts is an implicit conjunction.

```
?- parent(Person, bob), female(Person).
Person = eve;
false
```

Some Prolog Syntax

- implication: in Prolog, these are very special.
  They are written backwards (conclusion first), and the conclusion must be an atomic formula. This backwards implication is written as ':-', and is called a **rule**.

Rules can be used in programs to assert implications, but not in queries.

```
sibling(X, Y) :- parent(P, X), parent(P, Y).
```

Horn Clauses (Rules)

A **Horn Clause** is: $c \leftarrow h_1 \wedge h_2 \wedge h_3 \wedge \ldots \wedge h_n$

- Antecedents: conjunction of zero or more atomic formulae.
- Consequent: an atomic formula.

Meaning of a Horn clause:

- "The consequent is true if the antecedents are all true"
- c is true if h_1, h_2, h_3, ..., and h_n are all true

Horn Clause

- Horn Clause = Clause
- Consequent = Goal = Head
- Antecedents = Subgoals = Tail
- Horn Clause with No Tail = Fact
- Horn Clause with Tail = Rule

In Prolog, a Horn clause

$$c \leftarrow h_1 \wedge \ldots \wedge h_n$$

is written

c :- h_1, . . . , h_n.

Syntax elements: ':-' ',' '.'

Prolog Horn Clause

A Horn clause with no tail:

```
male(charlie).
```

I.e., a **fact**: `charlie` is a male dependent on no other conditions.
A Horn clause with a tail:

```
father(charlie,bob):-
 male(charlie), parent(charlie,bob).
```

I.e., a **rule**: `charlie` is the father of `bob` if `charlie` is male and `charlie` is a parent of `bob`'s.

Some Prolog Syntax

A logic program is a collection of Horn clauses.
Finally, a simplified Prolog grammar:

```
<clause> ::= <pred> . |
             <pred> :- <pred> { , <pred> } .

<pred>   ::= <pname>'(' <term> { , <term> } ')'

<term>   ::= <functor>'(' <term> { , <term> } ')'
             | <const> | <var>

<const>  ::= ...
<var>    ::= ...
```

Meaning of Prolog Rules

A prolog rule must have the form:

```
c :- a1, a2, a3, ..., an.
```

which means in logic:

$$a_1 \wedge a_2 \wedge a_3 \wedge \cdots \wedge a_n \rightarrow c$$

Restrictions

- There can be zero or more antecedents, and they are conjoined.
- There cannot be more than one consequent.

non-Horn clauses

Many non-Horn formulae can be converted into logically equivalent Horn-formulae, using propositional tautologies, e.g.:

$\neg\neg a \Leftrightarrow a$	double negation
$\neg(a \vee b) \Leftrightarrow \neg a \wedge \neg b$	DeMorgan
$a \vee (b \wedge c) \Leftrightarrow (a \vee b) \wedge (a \vee c)$	distributivity
$a \rightarrow b \Leftrightarrow \neg a \vee b$	implication

Bending the Restrictions?

Getting disjoined antecedents

Example: $a_1 \vee a_2 \vee a_3 \rightarrow c$.

Solution?

Syntactic sugar: ;

Getting more than 1 consequent, conjoined

Example: $a_1 \wedge a_2 \wedge a_3 \rightarrow c_1 \wedge c_2$.

Solution?

Getting more than 1 consequent, disjoined

Example: $a_1 \wedge a_2 \wedge a_3 \rightarrow c_1 \vee c_2$.

Solution?

We cannot disjoin consequents

Inference with non-Horn formulae is much more difficult, e.g.:

```
a :- b.
a :- c.
b ; c.

?- a.
```

Variables

Variables may appear in the antecedents and consequent of a Horn clause. Prolog interprets free variables of a rule **universally**.
`c(`X_1`,...,`X_n`) :- f(`X_1`,...,`X_n`,`Y_1`,...,`Y_k`)`
is, in first-order logic:

$$\forall X_1, \ldots, X_n, Y_1, \ldots, Y_m \cdot c(X_1, \ldots, X_n) \leftarrow f(X_1, \ldots, X_n, Y_1, \ldots, Y_m)$$

or, equivalently

$$\forall X_1, \ldots, X_n \cdot c(X_1, \ldots, X_n) \leftarrow \exists Y_1, \ldots, Y_m \cdot f(X_1, \ldots, X_n, Y_1, \ldots, Y_m)$$

Because of this equivalence, we sometimes say that free variables that do not appear in the head are quantified **existentially**.
Note that the Prolog **query** `?-q(`X_1`, ..., `X_n`)` means

$$\exists X_1, \ldots, X_n \cdot q(X_1, \ldots, X_n)$$

Horn Clauses with Variables

```
isaMother(X) :- female(X), parent(X, Y).
```

In first-order logic:

$$\forall X \cdot \mathit{isaMother}(X) \leftarrow \exists Y \cdot \mathit{parent}(X, Y) \land \mathit{female}(X).$$

Execution of Prolog Programs

- **Unification**: variable bindings.

- **Backward Chaining/**
  Top-Down Reasoning/
  Goal-Directed Reasoning:
  Reduces a goal to one or more subgoals.

- **Backtracking**:
  Systematically searches for all possible solutions that can be obtained via unification and backchaining.

Unification

Two atomic formulae unify if and only if they can be made syntactically identical by replacing their variables by other terms. For example,

- `parent(bob,Y)` unifies with `parent(bob,sue)` by replacing `Y` by `sue`.

  ```
  ?- parent(bob,Y)=parent(bob,sue).
  Y = sue ;
  false
  ```

- `parent(bob,Y)` unifies with `parent(X,sue)` by replacing `Y` by `sue` and `X` by `bob`.

  ```
  ?- parent(bob,Y)=parent(X,sue).
  Y = sue
  X = bob ;
  false
  ```

Unification

- A **substitution** is a function that maps variables to Prolog terms.
  e.g., { `X/sue, Y/bob` }

- An **instantiation** is an application of a substitution to all of the free variables in a formula or term.

  If `S` is a substitution and `T` is a formula or term, then `ST` denotes the instantiation of `T` by `S`.

```
T = parent(X,Y)
S = {  X/sue, Y/bob }
ST = parent(sue,bob)

T = likes(X,X)
S = {  X/bob }
ST = likes(bob,bob)
```

Unification

- `C` is a **common instance** of formulae/terms `A` and `B`, if there exist substitutions `S1` and `S2` such that `C = S1(A) = S2(B)`.

```
A = parent(bob,X), B = parent(Y,sue)}
S1 =  {  X/sue }, S2 = { Y/bob }
S1(A) = S2(B) = parent(bob,sue) = C
```

- `A` and `B` are **unifiable** if they have a common instance `C`. A substitution that produces a common instance is called a **unifier** of `A` and `B`.

```
parent(bob,X)  and  parent(Y,sue) are unifiable:
{  X/sue, Y/bob } is the unifier
```

Unification

Examples:

`p(a,a)` unifies with `p(a,a)`.

`p(a,b)` does not unify with `p(a,a)`.

`p(X,X)` unifies with `p(b,b)` and with `p(c,c)`, but not with `p(b,c)`.

`p(X,b)` unifies with `p(Y,Y)` with unifier `X/b, Y/b` to become `p(b,b)`.

`p(X,Z,Z)` unifies with `p(Y,Y,b)` with unifier `X/b, Y/b, Z/b` to become `p(b,b,b)`.

`p(X,b,X)` does not unify with `p(Y,Y,c)`.

Unification

Examples:

- `p(f(X),X)` unifies with `p(Y,b)`
  with unifier $\{\mathtt{X}\backslash\mathtt{b},\ \mathtt{Y}\backslash\mathtt{f}(\mathtt{b})\}$
  to become `p(f(b),b)`.
- `p(b,f(X,Y),c)` unifies with `p(U,f(U,V),V)`
  with unifier $\{\mathtt{X}\backslash\mathtt{b},\ \mathtt{Y}\backslash\mathtt{c},\ \mathtt{U}\backslash\mathtt{b},\ \mathtt{V}\backslash\mathtt{c}\}$
  to become `p(b,f(b,c),c)`.

Unification

`p(b,f(X,X),c)` does *not* unify with `p(U,f(U,V),V)`.
Reason:

- To make the first arguments equal,
  we *must* replace `U` by `b`.
- To make the third arguments equal,
  we *must* replace `V` by `c`.
- These substitutions convert
  `p(U,f(U,V),V)` into `p(b,f(b,c),c)`.
- However, *no* substitution for `X` will convert
  `p(b,f(X,X),c)` into `p(b,f(b,c),c)`.

Unification

`p(f(X),X)` should *not* unify with `p(Y,Y)`.
Reason:

- Any unification would require that
  `f(X)` = `Y`   and   `Y` = `X`
- But no substitution can make
  `f(X)` = `X`

However, Prolog claims they unify:

```
?- p(f(X),X)=p(Y,Y).

X = f(**)
Y = f(**) ;

false
```

Unification

If a rule has a variable that appears only once, that variable is called a "singleton variable".

Its value doesn't matter — it doesn't have to match anything elsewhere in the rule.

```
isaMother(X) :- female(X), parent(X, Y).
```

Such a variable consumes resources at run time.

We can replace it with _, the **anonymous variable**. It matches anything. If we don't, Prolog will warn us.

Note that `p(_,_)` unifies with `p(b,c)`. Every instance of the anonymous variable refers to a different, unique variable.

Most General Unifier (MGU)

The atomic formulas `p(X,f(Y))` and `p(g(U),V)` have infinitely many unifiers.

- $\{\texttt{X}\backslash\texttt{g(a)},\ \texttt{Y}\backslash\texttt{b},\ \texttt{U}\backslash\texttt{a},\ \texttt{V}\backslash\texttt{f(b)}\}$
  unifies them to give `p(g(a),f(b))`.
- $\{\texttt{X}\backslash\texttt{g(c)},\ \texttt{Y}\backslash\texttt{d},\ \texttt{U}\backslash\texttt{c},\ \texttt{V}\backslash\texttt{f(d)}\}$
  unifies them to give `p(g(c),f(d))`.

However, these unifiers are more specific than necessary.

The **most general unifier** (mgu) is

$\{\texttt{X}\backslash\texttt{g(U)},\ \texttt{V}\backslash\texttt{f(Y)}\}$

It unifies the two atomic formulas to give `p(g(U),f(Y))`.

Every other unifier results in an atomic formula of this form.

The mgu uses variables to fill in as few details as possible.

Most General Unifier

Example:

$$f(W,\ g(Z),\ Z)$$

$$f(X,\ Y,\ h(X))$$

To unify these two formulae, we need

$$\begin{array}{rcl} Y & = & g(Z) \\ Z & = & h(X) \\ X & = & W \end{array}$$

Working backwards from W, we get

$$\begin{array}{rcl} Y & = & g(Z) = g(h(W)) \\ Z & = & h(X) = h(W) \\ X & = & W \end{array}$$

So, the mgu is

$$\{X \backslash W,\ Y \backslash g(h(W)),\ Z \backslash h(W)\}$$

Most General Unifier

The substitution that results in the most general instance is called the **most general unifier** (mgu).

It is **unique**, up to consistent renaming of variables.

This is the one that Prolog computes.

A substitution, σ, is a most general unifier of a set of expressions E if it unifies E, and for any unifier, ω, of E, there is a unifier, λ, such that $\omega = \sigma \circ \lambda$.

Most General Unifier

A substitution, σ, is a most general unifier of a set of expressions E if it unifies E, and for any unifier, ω, of E, there is a unifier, λ, such that $\omega = \sigma \circ \lambda$.

e.g., given `p(X,f(Y))` and `p(g(U),V)`,
an MGU $\sigma = \{$ `X/g(U), V/f(Y)` $\}$,
and a unifier ω=$\{$`X/g(a), Y/b, U/a, V/f(b)`$\}$,

we have ω`(p(X,f(Y))) = p(g(a),f(b))`,
σ`(p(X,f(Y))) = p(g(U),f(Y))`,
so exists λ= $\{$ `U/a, Y/b` $\}$,
so that ω`(p(X,f(Y)))` $= \lambda(\sigma$`(p(X,f(Y)))`$)$

also ω`(p(g(U),V)) = p(g(a),f(b))`,
σ`(p(g(U),V)) = p(g(U),f(Y))`,
so for λ= $\{$ `U/a, Y/b` $\}$,
we have ω`(p(g(U),V))` $= \lambda(\sigma$`(p(g(U),V))`$)$

Most General Unifier

Examples:

t_1	t_2	MGU
f(X,a)	f(a,Y)	
f(h(X,a),b)	f(h(g(a,b),Y),b)	
g(a,W,h(X))	g(Y,f(Y,Z),Z)	
f(X,g(X),Z)	f(Z,Y,h(Y))	
f(X,h(b,X))	f(g(P,a),h(b,g(Q,Q)))	

Execution of Prolog Programs

- **Unification**: variable bindings.

- **Backward Chaining/**
  Top-Down Reasoning/
  Goal-Directed Reasoning:
  Reduces a goal to one or more subgoals.

- **Backtracking**:
  Systematically searches for all possible solutions that can be obtained via unification and backchaining.

Prolog Search Trees

Encapsulate unification, backward chaining, and backtracking.

- Internal nodes are ordered list of subgoals.
- Leaves are success nodes or failures, where computation can proceed no further.
- Edges are labeled with variable bindings that occur by unification.

Describe all possible computation paths.

- There can be many success nodes.
- There can be infinite branches.

Prolog Search Trees

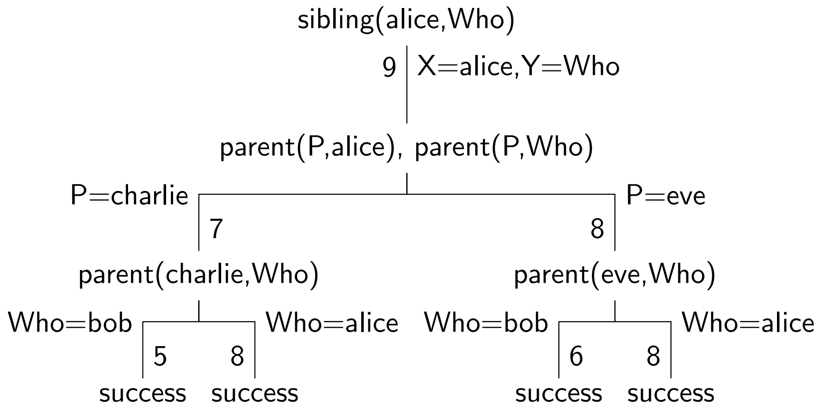
```
1) male(charlie).
2) male(bob).
3) female(alice).
4) female(eve).
5) parent(charlie,bob).
6) parent(eve,bob).
7) parent(charlie,alice).
8) parent(eve,alice).

9) sibling(X, Y) :- parent(P, X), parent(P, Y).
```

```
?- sibling(alice, Who).
Who = bob ;
Who = alice ;
Who = bob ;
Who = alice.
```

Prolog Search Trees

- **Trace it by hand**
- Trace it with **gtrace** (or **trace**).



Logic Programming vs. Prolog

Logic Programming: nondeterministic:

- Arbitrarily choose rule to expand first.
- Arbitrarily choose subgoal to to explore first.
- Results don't depend on rule and subgoal ordering.

Prolog: deterministic:

- Expand first rule first.
- Explore first subgoal first.
- Results may depend on rule and subgoal ordering.

Syntax of Lists in Prolog

A Prolog list is:

- **[]** the empty list, or
- **[Head | Tail]** a list with first element `Head` and rest of the elements list `Tail`

Luckily, we abbreviate

$$[\ e_1 \mid [\ e_2 \mid [\ \ldots\ e_n\ ]]]$$

as

$$[e_1, e_2, \ldots\ e_n\ ]$$

Lists in Prolog

List Unification: two lists unify iff they have the same structure and the corresponding elements unify.

```
?- [X, Y, Z] = [john, likes, fish].
```

```
?- [cat] = [X|Y].
```

```
?- [1,2] = [X|Y].
```

Lists in Prolog

```
?- [a,b,c] = [X|Y].
```

```
?- [a,b|Z]=[X|Y].
```

```
?- [X,abc,Y]=[X,abc|Y].
```

```
?- [[the|Y]|Z] = [[X,hare] | [is,here]].
```

Lists in Prolog

```
% member(?X,?List) iff X is an element of List
member(X,[X|_]).
member(X,[_|Rest]):-member(X,Rest).

?- member(a, [a,b,c]).
true
?- member(b, [a,c]).
false.
?- member(X, [a,b,c]).
X = a ;
X = b ;
X = c ;
false.
?- member(a, X).
X = [a|_] ;
X = [_, a|_] ;
X = [_, _, a|_]
```

Lists in Prolog

```
[trace]  ?- member(c,[a,b,c,d]).
   Call: (7) member(c, [a, b, c, d]) ?
   Call: (8) member(c, [b, c, d]) ?
   Call: (9) member(c, [c, d]) ?
   Exit: (9) member(c, [c, d]) ?
   Exit: (8) member(c, [b, c, d]) ?
   Exit: (7) member(c, [a, b, c, d]) ?
true
```

Lists in Prolog

```
[trace]  ?- member(X,[a,b,c,d]).
   Call: (7) member(_G314, [a, b, c, d]) ?
   Exit: (7) member(a, [a, b, c, d]) ?
X = a ;
   Redo: (7) member(_G314, [a, b, c, d]) ?
   Call: (8) member(_G314, [b, c, d]) ?
   Exit: (8) member(b, [b, c, d]) ?
   Exit: (7) member(b, [a, b, c, d]) ?

X = b ;
   Redo: (8) member(_G314, [b, c, d]) ?
   Call: (9) member(_G314, [c, d]) ?
   Exit: (9) member(c, [c, d]) ?
   Exit: (8) member(c, [b, c, d]) ?
   Exit: (7) member(c, [a, b, c, d]) ?
```

Lists in Prolog

```
  X = c ;
   Redo: (9) member(_G314, [c, d]) ?
   Call: (10) member(_G314, [d]) ?
   Exit: (10) member(d, [d]) ?
   Exit: (9) member(d, [c, d]) ?
   Exit: (8) member(d, [b, c, d]) ?
   Exit: (7) member(d, [a, b, c, d]) ?

X = d ;
   Redo: (10) member(_G314, [d]) ?
   Call: (11) member(_G314, []) ?
   Fail: (11) member(_G314, []) ?
false
```

Lists in Prolog

```
% append(?X,?Y,?Z) iff Z is the result of
%  appending list Y to list X
append([],Y,Y).
append([H|T],Y,[H|Z]):-append(T,Y,Z).

?- append([a],[b,c],Y).
Y = [a, b, c].
?- append(X,[b,c],[a,b,c]).
X = [a] ;
false
?- append(X,Y,[a,b,c]).
X = [], Y = [a, b, c] ;
X = [a], Y = [b, c] ;
X = [a, b], Y = [c] ;
X = [a, b, c], Y = [] ;
false
```

Lists in Prolog

```
[trace]  ?- append([a,b,c],[d,e],Z).
   Call: (7) append([a, b, c], [d, e], _G319) ?
   Call: (8) append([b, c], [d, e], _G385) ?
   Call: (9) append([c], [d, e], _G388) ?
   Call: (10) append([], [d, e], _G391) ?
   Exit: (10) append([], [d, e], [d, e]) ?
   Exit: (9) append([c], [d, e], [c, d, e]) ?
   Exit: (8) append([b, c], [d, e], [b, c, d, e]) ?
   Exit: (7) append([a, b, c], [d, e], [a, b, c, d, e]) ?

Z = [a, b, c, d, e].
```

Lists in Prolog

```
% mylength(?L,?N) iff N is the length of list L
mylength([],0).
mylength([_|T],N):-mylength(T,N-1).

?- mylength([a,b,c],3).
false.
```

Huh?

```
[trace]  ?- mylength([a,b,c],3).
   Call: (7) mylength([a, b, c], 3) ?
   Call: (8) mylength([b, c], 3-1) ?
   Call: (9) mylength([c], 3-1-1) ?
   Call: (10) mylength([], 3-1-1-1) ?
   Fail: (10) mylength([], 3-1-1-1) ?
   Fail: (9) mylength([c], 3-1-1) ?
   Fail: (8) mylength([b, c], 3-1) ?
   Fail: (7) mylength([a, b, c], 3) ?
false.
```

Arithmetic in Prolog

What is the result of these queries:

```
?- X = 97-65,  Y = 32-0,  X = Y.
```

```
?- X = 97-65,  Y = 67,  Z = 95-Y,  X = Z.
```

To evaluate arithmetic expressions is Prolog, use:

- *expr*1 `is` *expr*2, where *expr*2 is fully instantiated; if *expr*1 is a variable, it is bound to the result of evaluating *expr*2
- *expr*1 $<$ *expr*2 or *expr*1 $>$ *expr*2, where both *expr*1 and *expr*2 are instantiated

Arithmetic in Prolog

```
?- 5 is 2+3.
true.

?- X is 2+3.
X = 5.

?- 5 < 6.
true.

?- 2+3 < 3+3.
true.
```

Arithmetic in Prolog

```
% mylength(?L,?N) iff N is the length of list L
mylength([],0).
mylength([_|T],N):-N is NT+1, mylength(T,NT).
```

```
?- mylength([a,b,c],3).
```

```
ERROR: is/2:
  Arguments are not sufficiently instantiated
^  Exception: (8) 3 is _G226+1 ?
```

Why?

```
[trace]  ?- mylength([a,b,c],3).
   Call: (7) mylength([a, b, c], 3) ?
^  Call: (8) 3 is _G358+1 ?
ERROR: is/2: Arguments are not sufficiently instantiated
^  Exception: (8) 3 is _G358+1 ?
   Exception: (7) mylength([a, b, c], 3) ?
```

Arithmetic in Prolog

```
% mylength(?L,?N) iff N is the length of list L
mylength([],0).
mylength([_|T],N):-mylength(T,NT), N is NT+1.

?- mylength([a,b,c],3).
true
```

Lists in Prolog

How?

```
[trace]  ?- mylength([a,b,c],3).
   Call: (8) mylength([a, b, c], 3) ?
   Call: (9) mylength([b, c], _L193) ?
   Call: (10) mylength([c], _L212) ?
   Call: (11) mylength([], _L231) ?
   Exit: (11) mylength([], 0) ?
^  Call: (11) _L212 is 0+1 ?
^  Exit: (11) 1 is 0+1 ?
   Exit: (10) mylength([c], 1) ?
^  Call: (10) _L193 is 1+1 ?
^  Exit: (10) 2 is 1+1 ?
   Exit: (9) mylength([b, c], 2) ?
^  Call: (9) 3 is 2+1 ?
^  Exit: (9) 3 is 2+1 ?
   Exit: (8) mylength([a, b, c], 3) ?
true
```

Lists in Prolog

```
% mylength(?L,?N) iff N is the length of list L
mylength([],0).
mylength([_|T],N):-mylength(T,NT), N is NT+1.

?- mylength([a,b,c], L).
L = 3.

?- mylength([X,Y], L).
L = 2.

?- mylength(X, 3).
X = [_G226, _G229, _G232] ; <-- infinite run
```

Lists in Prolog

```
[trace]  ?- mylength(X, 1).
...
X = [_G373] ;
   Redo: (9) mylength(_G374, _L196) ?
   Call: (10) mylength(_G377, _L215) ?
   Exit: (10) mylength([], 0) ?
^  Call: (10) _L196 is 0+1 ?
^  Exit: (10) 1 is 0+1 ?
   Exit: (9) mylength([_G376], 1) ?
^  Call: (9) 1 is 1+1 ?
^  Fail: (9) 1 is 1+1 ?
```

Lists in Prolog

```
  Redo: (10) mylength(_G377, _L215) ?
   Call: (11) mylength(_G380, _L227) ?
   Exit: (11) mylength([], 0) ?
^  Call: (11) _L215 is 0+1 ?
^  Exit: (11) 1 is 0+1 ?
   Exit: (10) mylength([_G379], 1) ?
^  Call: (10) _L196 is 1+1 ?
^  Exit: (10) 2 is 1+1 ?
   Exit: (9) mylength([_G376, _G379], 2) ?
^  Call: (9) 1 is 2+1 ?
^  Fail: (9) 1 is 2+1 ?
   Redo: (11) mylength(_G380, _L227) ?
...
```

We'll learn how to fix this later.

Logic Programming vs. Prolog

What happens if we change the order of rules:

```
% mylength(?L,?N) iff N is the length of list L
mylength([_|T],N):-mylength(T,NT), N is NT+1.
mylength([],0).

?- mylength([a,b,c], 3).
true.

?- mylength([a,b,c], N).
N = 3.

?- mylength(X, 2).
ERROR: Out of local stack
```

Not very “declarative”.

Negation as Failure

No equivalent of logical negation in Prolog:

- Prolog can only assert that something is true.
- Prolog **cannot** assert that something is false.
- Prolog can assert that the given facts and rules do not allow something to be proven true.

Negation as Failure

Assuming that something unprovable is false is called **negation as failure**.
(Based on a **closed world assumption**.)
The goal \+(G) (or `not G`) succeeds whenever the goal G fails.

```
?- member(b, [a,b,c]).
true

?- \+ member(b, [a,b,c]).
false.

?- not(member(b, [a,b,c])).
false.

?- not(member(b,[a,c])).
true.
```

Negation as Failure

Example: Disjoint Sets

```
overlap(S1,S2) :- member(X,S1),member(X,S2).

disjoint(S1,S2) :- \+(overlap(S1,S2)).

?- overlap([a,b,c],[c,d,e]).
true
?- overlap([a,b,c],[d,e,f]).
false
?- disjoint([a,b,c],[c,d,e]).
false
?- disjoint([a,b,c],[d,e,f]).
true
?- disjoint([a,b,c],X).
false   %<---------Not what we wanted
```

Negation as Failure

```
overlap(S1,S2) :- member(X,S1),member(X,S2).
disjoint(S1,S2) :- \+(overlap(S1,S2)).

[trace]  ?- disjoint([a,b,c],X).
   Call: (7) disjoint([a, b, c], _G293) ? creep
   Call: (8) overlap([a, b, c], _G293) ? creep
   Call: (9) lists:member(_L230, [a, b, c]) ? creep
   Exit: (9) lists:member(a, [a, b, c]) ? creep
   Call: (9) lists:member(a, _G293) ? creep
   Exit: (9) lists:member(a, [a|_G352]) ? creep
   Exit: (8) overlap([a, b, c], [a|_G352]) ? creep
   Fail: (7) disjoint([a, b, c], _G293) ? creep
false
```

Negation as Failure

Proper use of Negation as Failure

`not(G)` works properly only in the following cases:

1. When `G` is fully instantiated at the time prolog processes the goal `not(G)`.

   (In this case, `not(G)` is interpreted to mean "goal `G` does not succeed".)

2. When all variables in `G` are unique to `G`, i.e., they don't appear elsewhere in the same clause.

   (In this case, `not(G(X))` is interpreted to mean "There is no value of `X` that will make `G(X)` succeed".)

Negation as Failure

```
woman(jane).
woman(marilyn).
famous(marilyn).
loves(john,X) :- woman(X), famous(X).
hates(john,X) :- \+ loves(john,X).
```

There are infinitely many women that John hates, not just Jane:

```
?- hates(john,jane).
true
?- hates(john,susan).
true
?- hates(john,betty).
true
...
```

Negation as Failure

```
woman(jane).
woman(marilyn).
famous(marilyn).
loves(john,X) :- woman(X), famous(X).
hates(john,X) :- \+ loves(john,X).
```

Plus John hates many things:

```
?- hates(john,pizza).
?- hates(john,john).
```

We say that the rule `hates` is not **safe**. Solution:

```
hates(john,X) :- woman(X), \+ loves(john,X).
```

`woman(X)` is called a **guard** — it protects from making unwanted inferences.

Execution of Prolog Programs

- **Unification**: variable bindings.
- **Backward Chaining/**
  Top-Down Reasoning/
  Goal-Directed Reasoning:
  Reduces a goal to one or more subgoals.
- **Backtracking**:
  Systematically searches for all possible solutions that can be obtained via unification and backchaining.