

# Prolog

## Basic construct: Horn clause

<fact\_list> .

- or -

<consequence> :- <antecedent\_terms> .

consequence is a term

antecedent\_terms is a conjunction of 1 or more terms (separated by commas)

meaning that the given consequence can be inferred from the truth of all the given terms (or to search for the an assignment of values to variables, search for each of the terms first)

terms can be:

variables

structures: functor( term\_list )

constants and literals

formally, these are 0-ary structures

In practice we have 3 forms we are interested in:

### Facts

Normally atomic structures with only constant/literal terms

### Rules

implications that may contain variables or constants

These are entered via "consult(...)"

A Prolog "program" consists of these

### Goals (queries)

atomic or conjunctive structures with or without variables

These are entered interactively

A particular execution is done with one of these

There are a number of special syntactic forms of interest

Infix arithmetic operators + - \* / // mod  
2 + 2 is shorthand for +(2,2) ( or '+'(2,2) )  
- note that means that Prolog does not automatically evaluate expressions

Equality test

== two terms are identical  
- note that 4 == 2 + 2 fails

Unification

= unify the two terms

Comparisons

:= test equality after evaluating terms  
- note that 4 := 2 + 2 succeeds

=\= test for inequality after evaluating

> test greater than after evaluating

< =< >= ditto for other comparisons

Instantiation

X is <expr> instantiate the given variable to the given value (evaluating expression first)

- fails if variable already instantiated

- variables in <expr> must be instantiated

## More about variables

Variables are limited in scope (local to) the clause they appear in (analogous to local variables)

```
grandParent(X,Z) :- parent(X,Y), parent(Y,Z).  
- The Xs here are the same var (ditto Y, Z)
```

```
parent(X,Y) :- mother(X, Y).  
- but not the same as those in next clause
```

To do unification, sometimes need to rename in order to avoid conflicts (but this happens transparently)

There is a special anonymous variable: "\_" which is used to denote "don't care"

```
member(X, [ X | _ ]).  
member(X, [ _ | Tail ]) :- member(X, Tail).
```

Note that every use of \_ is considered a separate variable

Purely declarative semantics is "fun" but, you can also look at Prolog in a more procedural light

## An Operational Look at Prolog

You can consider clauses to be a kind of procedure declaration

The left hand side (head) of a horn clause indicates the "name of the procedure" (via the functor) and the parameters it takes

The right hand side of the clause gives the "body of the procedure" (steps to carry it out)

All the clauses with the same functor (and same number of parameters) in the head are alternates for executing the "procedure" in different situations

You decide which "procedure body" to use based on the form of the parameters

- find one that matches via unification
- searches in order top to bottom

In order to "invoke a procedure" you need to find a head that can be matched with the "calling parameters" you have

- this may require variables to be instantiated to make the match

Example:

```
[1] located_in(boulder, colorado).  
[2] located_in(X, usa) :- located_in(X, colorado).  
[3] located_in(X, north_america) :- located_in(X, usa).
```

```
?- located_in(boulder, north_america).
```

Query (initial "call") matches 3rd clause if we instantiate X = boulder

Each "procedure" may either succeed  
by getting to the end of its "body"  
or may fail  
by being unable to make a call  
(i.e., by being unable to find a matching head).

Along the way a "procedure" will need to  
instantiate variables to produce matches

If a "procedure body" fails, we backtrack to the  
caller, undo the instantiations, and look for another  
matching head

- starting with the next clause in the order they  
were declared

Trace out earlier query...

[1] located\_in(boulder, colorado).  
[2] located\_in(X, usa) :- located\_in(X, colorado).  
[3] located\_in(X, north\_america) :- located\_in(X, usa).

?- located\_in(boulder, north\_america).

Goal ("call"): ?- located\_in(boulder, north\_america).

Matches 3: located\_in(X, north\_america) :-  
located\_in(X, usa).

Via inst.: X = boulder

New goal: ?- located\_in(boulder, usa).

Matches 2: located\_in(X, usa) :- located\_in(X, colorado).

Via inst: X = boulder {-- note this is a different X}

New goal: ?- located\_in(boulder, colorado).

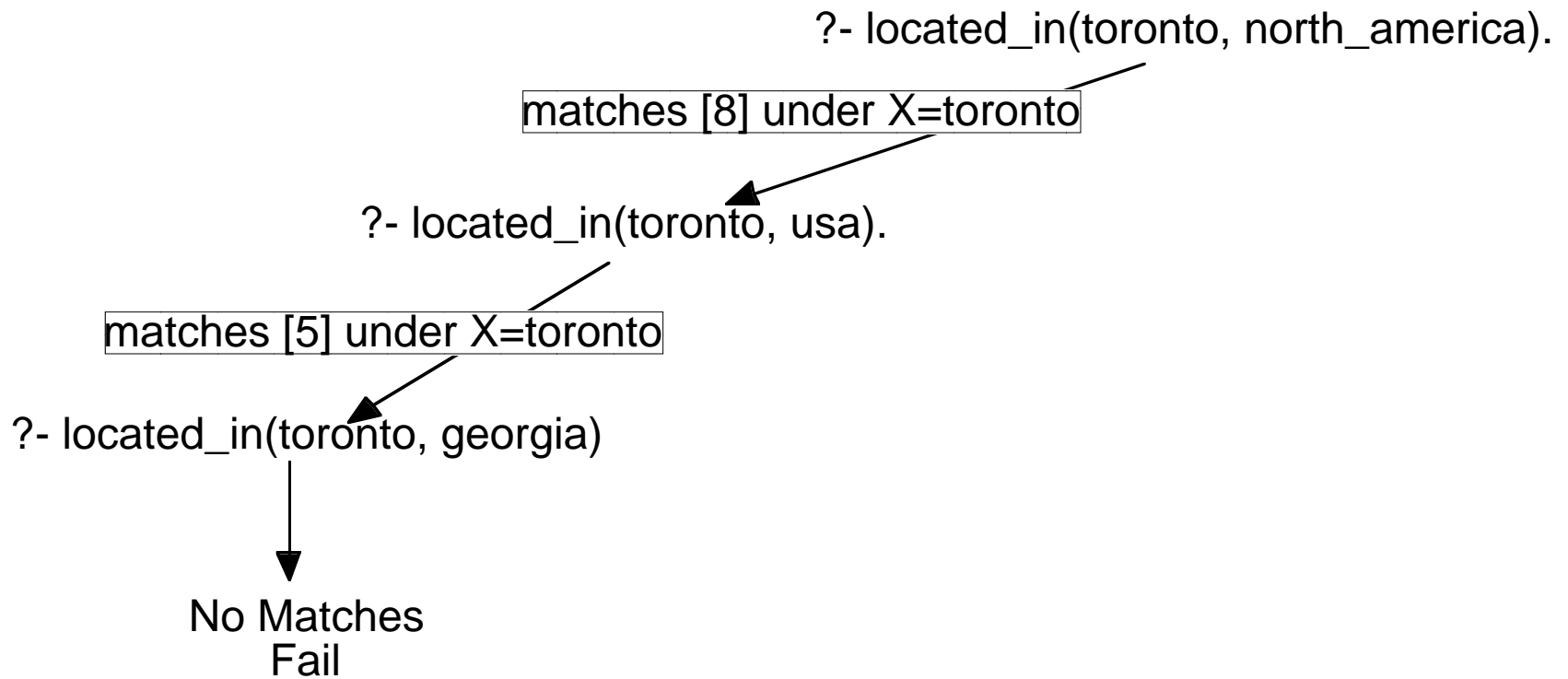
Matches 1: located\_in(boulder, colorado).

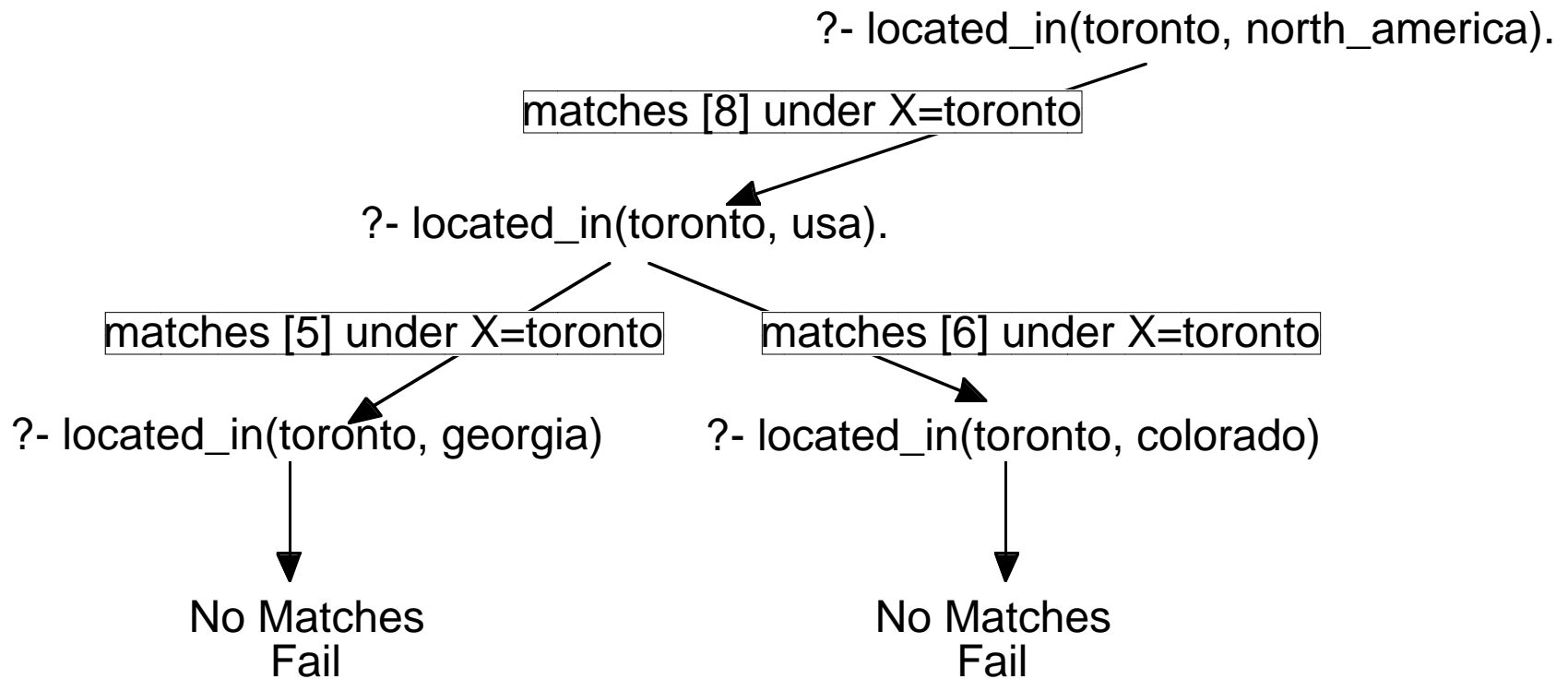
Via inst: <none>

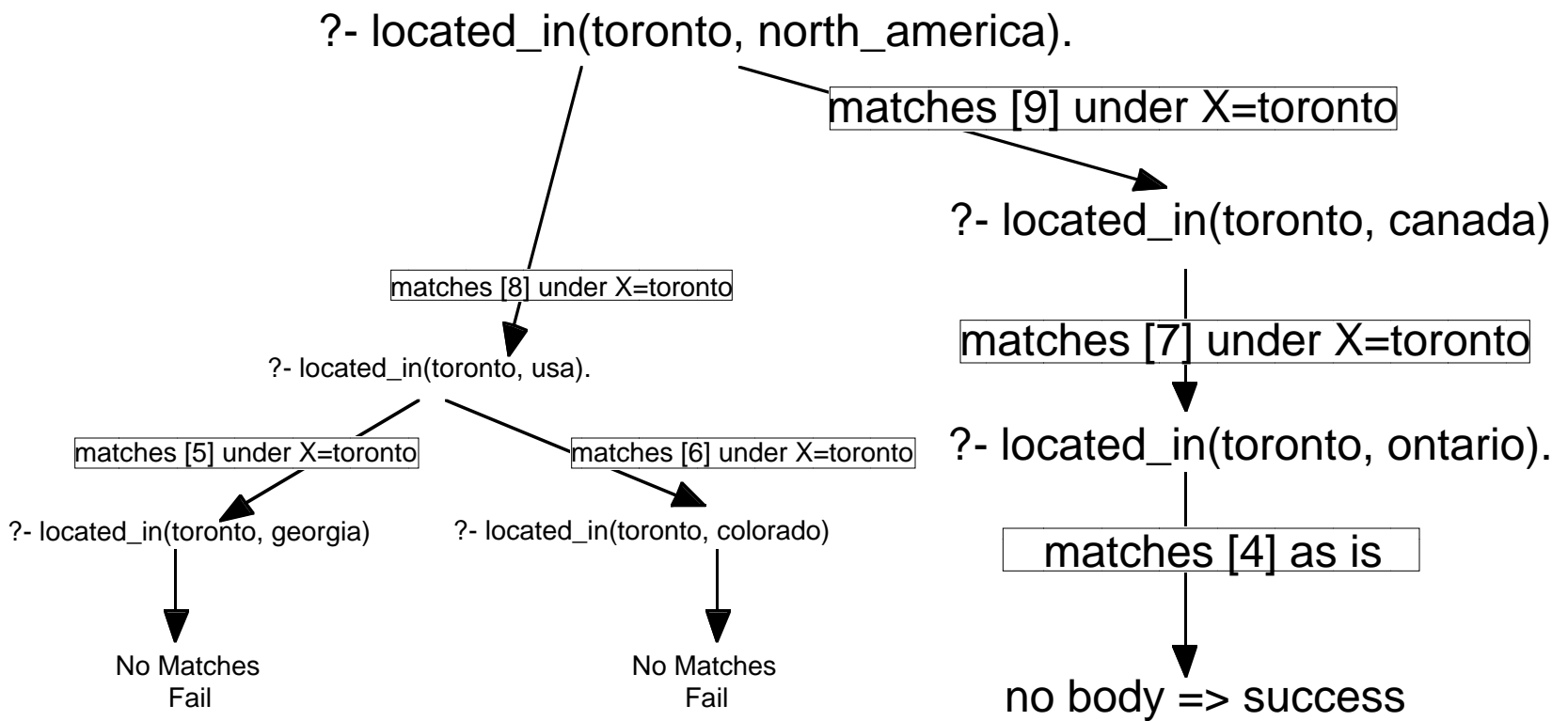
Has empty body, so we reach end of its body and succeed. Each caller also succeeds. Overall result is yes.

```
[1] located_in(atlanta, georgia).
[2] located_in(denver, colorado).
[3] located_in(boulder, colorado).
[4] located_in(toronto, ontario).
[5] located_in(X, usa) :- located_in(X, georgia).
[6] located_in(X, usa) :- located_in(X, colorado).
[7] located_in(X, canada) :- located_in(X, ontario).
[8] located_in(X, north_america) :- located_in(X, usa).
[9] located_in(X, north_america) :- located_in(X, canada).

?- located_in(toronto, north_america).
```







## Lists (again)

Prolog supports Lisp-style nested lists

Notation is:

[ a, b, c, [ d, e, [], f ] ]

which is a list with 4 elements:

a, b, c, and a list with 4 elements:  
d, e, an empty list, and f

Can break apart lists using "|"

[ Head | Tail ]

where Head is the first item as an object  
and Tail is the rest of the list (as a list)

?- [H | T] = [a, b, c].

H = a

T = [b,c]

Can also use this notation to construct lists

?- L = [a | [b, c]].

L = [a, b, c]

That provides List CAR, CDR, and CONS

You can also go beyond that and split multiple items off the head:

?- [A, B | C] = [a, b, c, d, e, f].

A = a

B = b

C = [c, d, e, f]

Also remember that unification works both ways:

?- [A, b | C] = [a, B, c, d, e, f].

A = a

B = b

C = [c, d, e, f]

Example of recursively processing a list  
(expression evaluation)

munch(A, Result) :- integer(A), Result is A.

munch(['+' | [A, B]], R) :- munch(A, V1), munch(B, V2),  
R is V1 + V2.

munch(['\*' | [A, B]], R) :- munch(A, V1), munch(B, V2),  
R is V1 \* V2.