

Let's Write Some List Predicates

1. `member(X, List).`
2. `append(List1, List2, Result).`
3. `swapFirstTwo(List1, List2).`
4. `length(List).`

List Membership

Definition of `member`...

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

Yes

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

No

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

X=a ;

X=b ;

X=c ;

No

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

X=a ;

No

```
?- member(X,Y).
```

X=_G72, Y=[_G72|_G73] ;

X=_G74, Y=[_G72,_G74|_G75] ;

X=_G76, Y=[_G72,_G74,_G76|_G77] ;

...

Lazy evaluation of potentially infinite data structures

Trace of Member

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

Yes

Append - More than "appending"

Definition of append

Build a list:

```
?- append([a],[b],Y).
```

Y=[a,b]

Yes

Break a list up:

```
?- append(X,[b],[a,b]).
```

X=[a]

Yes

```
?- append([a],Y,[a,b]).
```

Y=[b]

Yes

Append (cont.)

```
?- append(X,Y,[a,b].
X=[],Y=[a,b] ;
X=[a],Y=[b] ;
X=[a,b],Y=[] ;
No
```

Generate lists:

```
?- append(X,[b],Z).
X=[],Z=[b] ;
X=[_G98],Z=[_G98,b] ;
X=[_G98,_G102],Z=[_G98,_G102,b] ;
...
```

Trace:

```
[trace] ?- append([a,b,c],[p,q,r],L).
Call: (7) lists:append([a, b, c], [p, q, r], _G303) ? creep
Call: (8) lists:append([b, c], [p, q, r], _G426) ? creep
Call: (9) lists:append([c], [p, q, r], _G429) ? creep
Call: (10) lists:append([], [p, q, r], _G432) ? creep
Exit: (10) lists:append([], [p, q, r], [p, q, r]) ? creep
Exit: (9) lists:append([c], [p, q, r], [c, p, q, r]) ? creep
Exit: (8) lists:append([b, c], [p, q, r], [b, c, p, q, r]) ?
Exit: (7) lists:append
([a, b, c], [p, q, r], [a, b, c, p, q, r]) ? creep
```

L = [a, b, c, p, q, r] ;

No

Try some other traces!

Computing the Length of a List

Definition of length...

```
?- length([a,b,c],L).
```

L = 3

```
?- length([],L).
```

L = 0

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

X = [_G66,_G68,_G70]

```
?- length(X,0).
```

X = []

NOTE: Use built-in length function!!

Trace of Length:

Observe why this doesn't work!

```
xlength([],0).
xlength([_|Y],N) :- xlength(Y,N-1).

[trace] ?- xlength([a,b,c,d],X).
  Call: (7) xlength([a, b, c, d], _G296) ? creep
  Call: (8) xlength([b, c, d], _G296-1) ? creep
  Call: (9) xlength([c, d], _G296-1-1) ? creep
  Call: (10) xlength([d], _G296-1-1-1) ? creep
  Call: (11) xlength([], _G296-1-1-1-1) ? creep
  Fail: (11) xlength([], _G296-1-1-1-1) ? creep
  Fail: (10) xlength([d], _G296-1-1-1) ? creep
  Fail: (9) xlength([c, d], _G296-1-1) ? creep
  Fail: (8) xlength([b, c, d], _G296-1) ? creep
  Fail: (7) xlength([a, b, c, d], _G296) ? creep
```

No

Trace of Length (cont)

But this does work

```
mylength([],0).
mylength([_|Y],N) :- mylength2(Y,M), N is M+1.

[trace] ?- mylength([a,b,c,d],X).
  Call: (7) mylength([a, b, c, d], _G296) ? creep
  Call: (8) mylength([b, c, d], _L206) ? creep
  Call: (9) mylength([c, d], _L225) ? creep
  Call: (10) mylength([d], _L244) ? creep
  Call: (11) mylength([], _L263) ? creep
  Exit: (11) mylength([], 0) ? creep
  ^ Call: (11) _L244 is 0+1 ? creep
  ^ Exit: (11) 1 is 0+1 ? creep
  Exit: (10) mylength([d], 1) ? creep
  ^ Call: (10) _L225 is 1+1 ? creep
  ^ Exit: (10) 2 is 1+1 ? creep
  Exit: (9) mylength([c, d], 2) ? creep
  ^ Call: (9) _L206 is 2+1 ? creep
  ^ Exit: (9) 3 is 2+1 ? creep
  Exit: (8) mylength([b, c, d], 3) ? creep
  ^ Call: (8) _G296 is 3+1 ? creep
  ^ Exit: (8) 4 is 3+1 ? creep
  Exit: (7) mylength([a, b, c, d], 4) ? creep
```

X = 4

Yes

Accessing More Than One Initial Element

Definition of swap_first_two...

```
?- swap_first_two([a,b], [b,a]).
Yes
?- swap_first_two([a,b], [b,c]).
No
?- swap_first_two([a,b,c], [b,a,c]).
Yes
?- swap_first_two([a,b,c], [b,a,d]).
No
?- swap_first_two([a,b,c], X).
X = [b,a,c];
No
?- swap_first_two([a,b|Y], X).
Y = _56, X = [b,a|_56];
No
?- swap_first_two([],X).
No
?- swap_first_two([a],X).
No
?- swap_first_two([a,b],X).
X = [b,a];
No
```

Lists of a Specified Length

Definition of list_of_elem...

```
?- list_elem(X,b,3).
X = [b,b,b];
ERROR: Out of global stack
?- list_of_elem(X,Y,2).
X = [_50,_50]
Y = _50;
ERROR: Out of global stack
```

Lists of a Specified Length

New definition of `list_of_elem...`

```
?- working_list_elem(X,b,3).  
X = [b,b,b];  
No
```

```
?- working_list_elem(X,Y,2).  
X = [_50,_50]  
Y = _50;  
No
```

Beyond Horn Logic

- So far, we have studied what is known as *pure* logic programming, in which all the rules are Horn.
- For some applications, however, we need to go beyond this.
- For instance, we often need
 - Arithmetic
 - Negation
- Fortunately, these can easily be accommodated by simple extensions to the logic-programming framework,

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 get an expression evaluated, use

X is *expression*

where *expression*

- is an arithmetic expression, and
- is fully instantiated.

Examples:

?- X is $10 + 17$.

?- Y is 7 , Z is $3 + 4$, $Y = Z$.

Let's Write Some Predicates with Arithmetic

1. factorial(N , Ans).
2. sumlist($List$, $Total$).

Factorial

```
factorial(0,1).
```

```
factorial(X,Y) :- W is X-1,  
                  factorial(W,Z),  
                  Y is Z*X.
```

What are the preconditions for factorial?

Factorial with an Accumulator:

```
factorial2(0,X,X).
```

```
factorial2(N,A,F) :-  
    N > 0,  
    A1 is N*A,  
    N1 is N -1,  
    factorial2(N1,A1,F).
```

What are the preconditions?

Trace of Factorial

```
[trace] ?- factorial(3,X).  
    Call: (7) factorial(3, _G284) ? creep  
    ^ Call: (8) _L205 is 3-1 ? creep  
    ^ Exit: (8) 2 is 3-1 ? creep  
    Call: (8) factorial(2, _L206) ? creep  
    ^ Call: (9) _L224 is 2-1 ? creep  
    ^ Exit: (9) 1 is 2-1 ? creep  
    Call: (9) factorial(1, _L225) ? creep  
    ^ Call: (10) _L243 is 1-1 ? creep  
    ^ Exit: (10) 0 is 1-1 ? creep  
    Call: (10) factorial(0, _L244) ? creep  
    Exit: (10) factorial(0, 1) ? creep  
    ^ Call: (10) _L225 is 1*1 ? creep  
    ^ Exit: (10) 1 is 1*1 ? creep  
    Exit: (9) factorial(1, 1) ? creep  
    ^ Call: (9) _L206 is 1*2 ? creep  
    ^ Exit: (9) 2 is 1*2 ? creep  
    Exit: (8) factorial(2, 2) ? creep  
    ^ Call: (8) _G284 is 2*3 ? creep  
    ^ Exit: (8) 6 is 2*3 ? creep  
    Exit: (7) factorial(3, 6) ? creep  
X = 6  
Yes
```

Trace of Factorial w/ an Accumulator

```
[trace] ?- factorial2(3,1,Z).
  Call: (8) factorial2(3, 1, _G288) ? creep
  ^ Call: (9) 3>0 ? creep
  ^ Exit: (9) 3>0 ? creep
  ^ Call: (9) _L206 is 3*1 ? creep
  ^ Exit: (9) 3 is 3*1 ? creep
  ^ Call: (9) _L207 is 3-1 ? creep
  ^ Exit: (9) 2 is 3-1 ? creep
  Call: (9) factorial2(2, 3, _G288) ? creep
  ^ Call: (10) 2>0 ? creep
  ^ Exit: (10) 2>0 ? creep
  ^ Call: (10) _L226 is 2*3 ? creep
  ^ Exit: (10) 6 is 2*3 ? creep
  ^ Call: (10) _L227 is 2-1 ? creep
  ^ Exit: (10) 1 is 2-1 ? creep
  Call: (10) factorial2(1, 6, _G288) ? creep
  ^ Call: (11) 1>0 ? creep
  ^ Exit: (11) 1>0 ? creep
  ^ Call: (11) _L246 is 1*6 ? creep
  ^ Exit: (11) 6 is 1*6 ? creep
  ^ Call: (11) _L247 is 1-1 ? creep
  ^ Exit: (11) 0 is 1-1 ? creep
  Call: (11) factorial2(0, 6, _G288) ? creep
  Exit: (11) factorial2(0, 6, 6) ? creep
  Exit: (10) factorial2(1, 6, 6) ? creep
  Exit: (9) factorial2(2, 3, 6) ? creep
  Exit: (8) factorial2(3, 1, 6) ? creep
Z = 6
Yes
```

Sum of List

```
sumlist([],0).
```

```
sumlist([X|Rest],Ans) :- sumlist(Rest,Partial),
                          Ans is Partial+X.
```

Trace:

```
[trace] ?- sumlist([5,10,3],Y).
  Call: (7) sumlist([5, 10, 3], _G293) ? creep
  Call: (8) sumlist([10, 3], _L207) ? creep
  Call: (9) sumlist([3], _L227) ? creep
  Call: (10) sumlist([], _L247) ? creep
  Exit: (10) sumlist([], 0) ? creep
  ^ Call: (10) _L227 is 0+3 ? creep
  ^ Exit: (10) 3 is 0+3 ? creep
  Exit: (9) sumlist([3], 3) ? creep
  ^ Call: (9) _L207 is 3+10 ? creep
  ^ Exit: (9) 13 is 3+10 ? creep
  Exit: (8) sumlist([10, 3], 13) ? creep
  ^ Call: (8) _G293 is 13+5 ? creep
  ^ Exit: (8) 18 is 13+5 ? creep
  Exit: (7) sumlist([5, 10, 3], 18) ? creep
```

Y = 18

Yes

Arithmetic Predicates may not be Invertible

We may not be able to “invert” a predicate that involves arithmetic.

That is, we may not be able to put a variable in a different place.

Tip: Every time you write `is`, you must be sure the expression will be fully instantiated. If necessary, put a precondition on your predicate.

Negation as Failure

No equivalent of logical not 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 $\text{!}(G)$ succeeds whenever the goal G fails.

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

Yes

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

No

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

yes

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]).
```

Yes

```
?- overlap([a,b,c],[d,e,f]).
```

No

```
?- disjoint([a,b,c],[c,d,e]).
```

No

```
?- disjoint([a,b,c],[d,e,f]).
```

Yes

```
?- disjoint([a,b,c],X).
```

No %<-----Not what we wanted

Example: Disjoint Sets (cont.)

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

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

```
?- disjoint([a,b,c],X).
```

```
No    %<-----Not what we wanted
```

```
[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
```

No

Proper use of Negation as Failure

$\backslash+(G)$ works properly only in the following cases:

1. When G is fully instantiated at the time prolog processes the goal $\backslash+(G)$.

(In this case, $\backslash+(G)$ 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, $\backslash+(G(X))$ is interpreted to mean “There is no value of X that will make $G(X)$ succeed”.)

Safety

Consider the following rule:

(*) `hates(tom,X) :- not loves(tom,X).`

This may NOT be what we want, for several reasons:

- The answer is *infinite*, since for any person *p* not mentioned in the database, we cannot infer `loves(tom,p)`, so we must infer `hates(tom,p)`.

Rule (*) is therefore said to be unsafe.

- The rule does not require *x* to be a person.
e.g., since we cannot infer

```
loves(tom,hammer)
loves(tom,verbs)
loves(tom,green)
loves(tom,abc)
```

we must infer that tom hates all these things.

Safety (Cont'd)

To avoid these problems, rules with negation should be guarded:

```
hates(tom,X) :- vegetable(x), green(X),
                not loves(tom,X).
```

i.e., Tom hates every green vegetable that he does not love.

Here, `vegetable` and `green` are called guard literals. They guard against safety problems by binding *x* to specific values in the database.

Data Structure: the “Structure”

Representing a parse tree

Simple grammar:

```
<np> ::= <det> <adjList> <n>
<adjList> ::= { <adj> }
<det> ::= the | a
<n> ::= child | dog | professor
<adj> ::= small | friendly | noisy
```

Parse tree:

Representation as a Prolog structure: