Avoiding Irrelevant Unification

A graph:

```
arc(a,b).
arc(a,c).
arc(b,c).
arc(b,d).
```

Rule for nodes of graph:

```
node(X):- arc(X,Y).
node(X):- arc(Y,X).
```

On reading these in:

```
* Singleton variables, clause 1 of node/1: Y
* Singleton variables, clause 2 of node/1: Y
```


Avoiding Irrelevant Unification

Above has irrelevant unification of Y $\Rightarrow$ wasted effort.

Anonymous variable:
"don't care" symbol _ which means the argument exists but doesn't signify anything.

```
node(X):- arc(X,_).
node(X):- arc(_,X).
```

Removes useless unification.

List Membership

Can write member avoiding unnecessary unification...

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 = _6694,
X = [b,a|Y] ;
| ?- swap_first_two([],X).
no
| ?- swap_first_two([a],X).
no
| ?- swap_first_two([a,b],X).
X = [b,a] ;
no
```

Computing the Length of a List

Definition of `length`...

```
| ?- length([a,b,c],L).
L = 3

| ?- length([],L).
L = 0

| ?- length(X,3).
X = [_6755,_6757,_6759]

| ?- length(X,0).
X = []
```


Lists of a Specified Length

Definition of `list_of_elem`...

```
| ?- list_of_elem(X,b,3).
X = [b,b,b] ;

Prolog interruption (h for help)? a
! Execution aborted

| ?- list_elem(X,Y,2).
X = [Y,Y],
Y = _6691 ;

Prolog interruption (h for help)? a
! Execution aborted
```

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 = [Y,Y],
Y = _6691 ;
no
```


Execution of Prolog Programs

- **Unification**: (variable bindings)
  Specializes general rules to apply to a specific problem.
- **Backward Chaining**:
  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 Execution Example 1

```
male(tom).                  %Male Clause 1.
male(fred).                 %Male Clause 2.

female(jane).               %Female Clause 1.
female(betty).              %Female Clause 2.
female(mary).               %Female Clause 3.

sibling(betty,fred).        %Sibling Clause 1.
sibling(fred,betty).        %Sibling Clause 2.
sibling(mary,fred).         %Sibling Clause 3.
sibling(fred,mary).         %Sibling Clause 4.

parent(tim,jane).           %Parent Clause 1.
parent(mary,jane).          %Parent Clause 2.

uncle(X,Z) :- male(X),      %Uncle Clause.
              sibling(X,Y),
              parent(Y,Z).
```

Search Tree for Example 1

```
| ?- uncle(X,Z).
```


Prolog Execution Trace for Example 1

```
| ?- uncle(X,Z).
   (1) 0 Call: uncle(_6645,_6662) ?
   (1) 1 Head [1]: uncle(_6645,_6662) ?
   (2) 1 Call: male(_6645) ?
   (2) 2 Head [1->2]: male(_6645) ?
   (2) 1 Exit: male(tom) ?
   (3) 1 Call: sibling(tom,_6769) ?
   (3) 1 Fail: sibling(tom,_6769) ?
   (2) 1 Redo: male(tom) ?
   (2) 2 Head [2]: male(_6645) ?
   (2) 1 Done: male(fred) ?
   (4) 1 Call: sibling(fred,_6769) ?
   (4) 2 Head [2->4]: sibling(fred,_6769) ?
   (4) 1 Exit: sibling(fred,betty) ?
```

Prolog Execution Trace (Continued)

```
(5) 1 Call: parent(betty,_6662) ?
(5) 1 Fail: parent(betty,_6662) ?
(4) 1 Redo: sibling(fred,betty) ?
(4) 2 Head [4]: sibling(fred,_6769) ?
(4) 1 Done: sibling(fred,mary) ?
(6) 1 Call: parent(mary,_6662) ?
(6) 2 Head [2]: parent(mary,_6662) ?
(6) 1 Done: parent(mary,jane) ?
(1) 0 Done: uncle(fred,jane) ?

X = fred,
Z = jane ;

no
```


Prolog Execution Example 2

```
flight(ny,chicago).                     %Clause 1.
flight(ny,miami).                       %Clause 2.
flight(miami,austin).                   %Clause 3.

trip(X,Z) :- flight(X,Z).               %Clause 1.
trip(X,Z) :- flight(X,Y),trip(Y,Z).     %Clause 2.
```

Prolog Search Tree for Example 2

```
| ?- trip(ny,austin).
```


Prolog Execution Trace

```
| ?- trip(ny,austin).
  (1) 0 Call: trip(ny,austin) ?
  (1) 1 Head [1->2]: trip(ny,austin) ?
  (2) 1 Call: flight(ny,austin) ?
  (2) 2 Head [1->2]: flight(ny,austin) ?
  (2) 2 Head [2]: flight(ny,austin) ?
  (2) 1 Fail: flight(ny,austin) ?
  (1) 1 Head [2]: trip(ny,austin) ?
  (3) 1 Call: flight(ny,_6715) ?
  (3) 2 Head [1->2]: flight(ny,_6715) ?
  (3) 1 Exit: flight(ny,chicago) ?
  (4) 1 Call: trip(chicago,austin) ?
  (4) 2 Head [1->2]: trip(chicago,austin) ?
  (5) 2 Call: flight(chicago,austin) ?
  (5) 2 Fail: flight(chicago,austin) ?
```

Prolog Execution Trace (Continued)

```
  (4) 2 Head [2]: trip(chicago,austin) ?
  (6) 2 Call: flight(chicago,_6721) ?
  (6) 2 Fail: flight(chicago,_6721) ?
  (4) 1 Fail: trip(chicago,austin) ?
  (3) 1 Redo: flight(ny,chicago) ?
  (3) 2 Head [2]: flight(ny,_6715) ?
  (3) 1 Done: flight(ny,miami) ?
  (7) 1 Call: trip(miami,austin) ?
  (7) 2 Head [1->2]: trip(miami,austin) ?
  (8) 2 Call: flight(miami,austin) ?
  (8) 3 Head [3]: flight(miami,austin) ?
  (8) 2 Done: flight(miami,austin) ?
  (7) 1 Exit: trip(miami,austin) ?
  (1) 0 Exit: trip(ny,austin) ?
yes
```


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 \+(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
```

Proper use of Negation as Failure

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

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

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

Adding Subgoals in Search Trees

```
male(tom).                %Male Clause 1.
male(fred).               %Male Clause 2.

female(jane).             %Female Clause 1.
female(betty).            %Female Clause 2.
female(mary).             %Female Clause 3.

sibling(betty,fred).      %Sibling Clause 1.
sibling(fred,betty).      %Sibling Clause 2.
sibling(mary,fred).       %Sibling Clause 3.
sibling(fred,mary).       %Sibling Clause 4.

parent(tim,jane).         %Parent Clause 1.
parent(mary,jane).        %Parent Clause 2.

uncle(X,Z) :- male(X),         %Uncle Clause.
              sibling(X,Y),
              parent(Y,Z).

niece(X,Y):- female(X), uncle(Y,X). %Neice Clause.
```

Another Example Search Tree

```
| ?- neice(X,Y).
```


Problems with Multiple Solutions

```
isa-mother(X) :- female(X),parent(X,_).
isa-father(X) :- male(X),parent(X,_).

parent(fred,sue).
parent(janet,sue).
parent(fred,tim).
parent(janet,tim).

male(fred).
female(janet).

| ?- isa-mother(X).
X = janet ;
X = janet ;
no
```

Suppressing Multiple Solutions with Cut

The cut symbol **!** is prolog syntax for:

- a goal that always succeeds.
- with the side effect of suppressing backtracking.
  - Prolog will not attempt to find additional solutions to any goals to the left of the cut.

⇒ Cut prunes the search tree that prolog generates.

Suppressing Multiple Solutions with Cut

```
isa-mother(X) :- female(X),parent(X,_),!.
isa-father(X) :- male(X),parent(X,_),!.

parent(fred,sue).
parent(janet,sue).
parent(fred,tim).
parent(janet,tim).

male(fred).
female(janet).

| ?- isa-mother(X).
X = janet ;
no
```

Negation by Failure Revisited

```
not(X):- X, !, fail.
not(\_).
```


Summary

Cuts are:

+ very powerful.

+ can help make programs more efficient.

– difficult to use safely.

– make for difficult to understand programs.

Arithmetic in Prolog

```
| ?- AGE is 1995 - 1956.
AGE = 39 ;

| ?- DATE is 1956 + 39.
DATE = 1995 ;

| ?- 39 is DATE - 1956.
! Instantiation error in argument 2 of is/2
! goal:  39 is _6668-1956

| ?- 1995 is 1956 + AGE.
! Instantiation error in argument 2 of is/2
! goal:  1995 is 1956+_6728
```


Arithmetic in Prolog

At the time Prolog begins processing a goal of the form: `X is <Exp>`, `<Exp>` must be a fully instantiated arithmetic expression, i.e., it cannot have any unbound variables

⇒ Arithmetic programs are not always invertible.

Factorial Function

```
factorial(0,1).
factorial(X,Y) :- W is X-1,
                  factorial(W,Z),
                  Y is Z*X.
```

This calculates X! if X is bound to an integer. Otherwise it aborts in the first "is" clause.

```
| ?- factorial(3,6).
yes
| ?- factorial(5,Z).
Z = 120
| ?- factorial(Y,6).
! Instantiation error in argument 2 of is/2
! goal: _6568 is _6571-1
```


Adding up the Numbers on a List

Definition of `sumlist`...

```
| ?- sumlist([1,2,3],X).
X = 6

| ?- sumlist([],X).
X = 0

| ?- sumlist(X,3).
! Instantiation error in argument 2 of is/2
! goal:  3 is _6786+0
```