More uses of cut:
if-then-else

q and r evaluated twice

\[ p:\neg q, r, s, !, t. \]
\[ p:\neg q, r, u. \]
\[ q. \]
\[ r. \]
\[ u. \]

only evaluated when s is false and both q and r are true

such uses are equivalent to

\[ p:\neg q, r, \text{if}_s\text{then}_t\text{else}_u. \]
\[ \text{if}_s\text{then}_t\text{else}_u:\neg s, !, t. \]
\[ \text{if}_s\text{then}_t\text{else}_u:u. \]
\[ q. \]
\[ r. \]
\[ u. \]
More uses of cut: 
if-then-else built-in

\[ p :\text{-} q, r, \text{if\_then\_else}(S, T, U). \]
\[ \text{if\_then\_else}(S, T, U) :\text{-} S, !, T. \]
\[ \text{if\_then\_else}(S, T, U) :\text{-} U. \]

built-in as \( P \rightarrow Q; R \)

nested if’s:
\( P \rightarrow Q; (R \rightarrow S; T) \)

diagnosis(Patient,Condition) :-
\[ \text{temperature}(Patient, T), \]
\( (T \leq 37 \rightarrow \text{blood\_pressure}(Patient, Condition) \)
\( ; T > 37, T < 38 \rightarrow \text{Condition}=\text{ok} \)
\( ; \text{otherwise} \rightarrow \text{diagnose\_fever}(Patient, Condition) \)

always evaluates to true
More uses of cut: enabling tail recursion optimization

```prolog
play(Board,Player):-
    lost(Board,Player).
play(Board,Player):-
    find_move(Board,Player,Move),
    make_move(Board,Move,NewBoard),
    next_player(Player,Next),!
    play(NewBoard,Next).
:-play(starconfiguration,first).
```

most Prolog’s optimize tail recursion into iterative processes if the literals before the recursive call are deterministic.

would otherwise maintain all previous board configurations and all moves such that they can be undone.

pops choice points from stack before entering next recursion.
Arithmetic in Prolog: is/2

is(Result,Expression) succeeds if Expression can be evaluated as an arithmetic expression and its resulting value unifies with Result.

Peano-encoding of natural numbers is clumsy and inefficient.

Multiplication as repeated addition using recursion.

?-X is 5+7-3.
X = 9

?-X is 5*3+7/2.
X = 18.5

?-9 is 5+7-3.
Yes

?-9 is X+7-3.
Error in arithmetic expression

Defined as an infix operator.

must be instantiated
Arithmetic in Prolog:
is/2 versus =/2

succeeds if its arguments can be unified

\=/2 when its arguments cannot be unified

?- X = 5+7-3
X = 5+7-3

?- 9 = 5+7-3
no

?- X = Y+3
X = _947+3
Y = _947

?- X = f(X)
X = f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(X)

error: term being written is too deep
Prolog practices: accumulators

```prolog
length([],0).
length([H|T],N) :- length(T,N1), N is N1+1.
```

The resolvent collects as many is/2 literals as there are elements in the list before doing any actual calculation. 

```
?-length([a,b,c],N)
length([H|T],N1):-length(T,M1),
  N1 is M1+1
  {H->a, T->[b,c], N1->N}
:-length([b,c],M1),
  N is M1+1
  length([H|T],N2):-length(T,M2),
  N2 is M2+1
  {H->b, T->[c], N2->M1}
:-length([c],M2),
  M1 is M2+1,
  N is M1+1
  length([H|T],N3):-length(T,M3),
  N3 is M3+1
  {H->c, T->[], N3->M2}
:-length([],M3),
  M2 is M3+1,
  M1 is M2+1,
  N is M1+1
  length([],0)
  {M3->0}
:-M2 is 0+1,
  M1 is M2+1,
  N is M1+1
  {M2->1}
:-M1 is 1+1,
  N is M1+1
  {M1->2}
:-N is 2+1
  {N->3}

[]```
Prolog practices:
tail-recursive length/2 with accumulator

```
length(L,N) :- length_acc(L,0,N).
length_acc([],N,N).
length_acc([H|T],N0,N) :-
    N1 is N0+1,
    length_acc(T,N1,N).
```

read `length_acc(L,M,N)` as $N = M + \text{length}(L)$

accumulator represents length so far

```
?-length_acc([a,b,c],0,N)
N11 is N10+1,
length_acc(T,N11,N1)
{H->a, T->[b,c], N10->0, N1->N}
N11 is 0+1,
length_acc([b,c],N11,N)
{N11->1}
N21 is 1+1,
length_acc([c],N21,N)
{N21->2}
N31 is 2+1,
length_acc([],N31,N)
{N31->3}
```
Prolog practices:
tail-recursive reverse/2 with accumulator

naive_reverse([], []).  
naive_reverse([H|T], R) :- 
    naive_reverse(T, R1), 
    append(R1, [H], R).

append([], Y, Y).  
append([H|T], Y, [H|Z]) :- 
    append(T, Y, Z).

reverse(X, Z) :- reverse(X, [], Z).  
reverse([], Z, Z).  
reverse([H|T], Y, Z) :- 
    reverse(T, [H|Y], Z).

reverse(X, [], Z) <- Z = reverse(X)  
moves  
reverse([H|T], Y, Z) <- Z = reverse([H|T]) + Y  
moves  
    <- Z = reverse(T) + [H] + Y  
moves  
    <- Z = reverse(T) + [H|Y]  
moves  
    <- reverse(T, [H|Y], Z)
Prolog practices: difference lists

represent a list by a term L1-L2.

\[
\begin{align*}
[a, b, c, d] & - [d] \\
[a, b, c, 1, 2] & - [1, 2] \\
[a, b, c \leftarrow X] & \leftarrow X
\end{align*}
\]

variable for minus list: can be used as pointer to end of represented list
Prolog practices: appending difference lists in constant time

one unification step rather than as many resolution steps as there are elements in the list appended to

append_d1(XPlus-XMinus,YPlus-YMinus,XPlus-YMinus) :- XMinus=YPlus.

or

append_d1(XPlus-YPlus,YPlus-YMinus,XPlus-YMinus).

?- append_d1([a,b|X]-X,[c,d|Y]-Y,Z).
X = [c,d|Y], Z = [a,b,c,d|Y]-Y
Prolog practices: reversing difference lists

reverse(X,Y,Z) \iff Z=\text{reverse}(X)+Y
\iff \text{reverse}(X)=Z-Y

reverse([H|T],Y,Z) \iff Z=\text{reverse}([H|T])+Y
\iff Z=\text{reverse}(T)+[H|Y]
\iff \text{reverse}(T)=Z-[H|Y]

\text{reverse}(X,Z) :- \text{reverse\_d1}(X,Z-[]).
\text{reverse\_d1}([],Z-Z).
\text{reverse\_d1}([H|T],Z-Y) :- \text{reverse\_d1}(T,Z-[H|Y]).
Second-order predicates: map/3

map(R, [], []).  
map(R, [X|Xs], [Y|Ys]) :- R(X,Y), map(R, Xs, Ys).
?- map(parent, [a,b,c], X)

or, when atoms with variable as predicate symbol are not allowed:

map(R, [], []).  
map(R, [X|Xs], [Y|Ys]) :- Term =.. List succeeds  
            if Term is a constant and List is the list [Term]  
            if Term is a compound term f(A1,...,An)  
            and List is a list with head f and whose tail unifies with [A1,...,An]
Second-order predicates: map/3

map(R, [], []).
map(R, [X|Xs], [Y|Ys]) :- R(X, Y), map(R, Xs, Ys).
?-map(parent, [a, b, c], X)

or, when atoms with variable as predicate symbol are not allowed:

map(R, [], []).
map(R, [X|Xs], [Y|Ys]) :- Goal =.. [R, X, Y],
call(Goal),
map(R, Xs, Ys).

univ operator =.. can be used to construct terms:
?-Term=..[parent, X, peter]
Term=parent(X, peter)

and decompose terms:
?-parent(maria, Y)=..List
List=[parent, maria, Y]

Term=..List succeeds
if Term is a constant and List is the list [Term]
if Term is a compound term f(A1,..,An)
and List is a list with head f and whose tail unifies with [A1,..,An]
Second-order predicates: findall/3

findall(Template,Goal,List) succeeds if List unifies with a list of the terms Template is instantiated to successively on backtracking over Goal. If Goal has no solutions, List has to unify with the empty list.

?-findall(C,parent(john,C),L).
L = [peter,paul,mary]

?-findall(f(C),parent(john,C),L).
L = [f(peter),f(paul),f(mary)]

?-findall(C,parent(P,C),L).
L = [peter,paul,mary,davy,dee,dozy]
Second-order predicates: bagof/3 and setof/3

```prolog
parent(john,peter).
parent(john,paul).
parent(john,mary).
parent(mick,davy).
parent(mick,dee).
parent(mick,dozy).

?-findall(C,parent(P,C),L).
   L = [peter,paul,mary,davy,dee,dozy]

?-bagof(C,parent(P,C),L).
P = john
   L = [peter,paul,mary];
P = mick
   L = [davy,dee,dozy]

?-bagof(C,P^parent(P,C),L).
   L = [peter,paul,mary,davy,dee,dozy]
```

The construct Var^Goal tells bagof/3 not to bind Var in Goal.

setof/3 is same as bagof/3 without duplicate elements in List
findall/3 is same as bagof/3 with all free variables existentially quantified using ^

differ from findall/3 if Goal contains free variables
asserta(Clause)
    adds Clause at the beginning of the Prolog database.
assertz(Clause) and assert(Clause)
    adds Clause at the end of the Prolog database.
retract(Clause)
    removes first clause that unifies with Clause from the Prolog database.

retractall(Term):-
    retract(Term), fail.
    retractall(Term):-
    retract((Term:- Body)), fail.
    retractall(Term).

Backtracking over such literals will not undo the modifications to the database!