The cut (!) is used to affect Prolog's backtracking. It can be used to:
- reduce the search space (save time).
- tell Prolog that a goal is deterministic (has only one solution) (save space).
- construct a (weak form of) negation.
- construct if_then_else and once predicates.

The cut reduces the flexibility of clauses, and destroys their logical structure.

Use cut as a last resort.

Reordering clauses can sometimes achieve the desired effect, without the use of the cut.

If you are convinced that you have to use a cut, try using if_then_else, once, or not instead.

The cut succeeds and commits Prolog to all the choices made since the parent goal was called.

Cut does two things:

**commit**: Don’t consider any later clauses for this goal.

**prune**: Throw away alternative solutions to the left of the cut.
The Cut II

```
p :- a, b, !, c, d.
p :- x, y.
p :- z.
```

**prune:**
Forget alternative solutions

**commit:**
Don’t try these

The Cut III

```
p :- q, !.
p :- r.
q :- s.
q :- t.
s.
```

Prolog Execution — The Boxflow Model I

```
a(X) :- b(X), c(X).
a(X) :- d(X).
```

Try to satisfy the goal

CALL → SUCCEED

FAIL → REDO

Save the current state in case we need to backtrack

No, the goal could not be satisfied

Try to find another solution

Prolog Execution — The Boxflow Model II

```
a(X) :- b(X), c(X).
a(X) :- d(X).
```

CALL → SUCCEED

FAIL → REDO

C S
F R
C S
F R

SUCCEED
Prolog Execution — The Cut

Classifying Cuts

- **grue** No effect on logic, improves efficiency.
- **green** Prune away
  - irrelevant proofs
  - proofs which are bound to fail
- **blue** Prune away
  - proofs a smart Prolog implementation would not try, but a dumb one might.
- **red** Remove unwanted logical solutions.

Green Cuts – Merge I

Produce an ordered list of integers from two ordered lists of integers.

```prolog
merge([X|Xs], [Y|Ys], [X|Zs]) :-
  X < Y, merge(Xs, [Y|Ys], Zs).
merge([X|Xs], [Y|Ys], [X,Y|Zs]) :-
  X = Y, merge(Xs, Ys, Zs).
merge([X|Xs], [Y|Ys], [Y|Zs]) :-
  X > Y, merge([X|Xs], Ys, Zs).
merge(Xs, [], Xs).
merge([], Ys, Ys).

?- merge([1,4], [3,7], L).
L = [1,3,4,7]
```

Green Cuts – Merge II

```
m([2,3,5], [2,3], Xs')
m([2,3,5], [3], Xs')
m([2,3,5], [3], Xs')
m([2,3,5], [3], Xs')
m([2,3,5], [3], Xs')

2<2 m([3,5],[2,3],Xs')=2 m([3,5],[3],Xs')>2 m([2,3,5],[3],Xs')

fail succeed
```
Still, there is no way for Prolog to know that the clauses are mutually exclusive, unless we tell it so. Therefore, Prolog must keep all choice-points (points to which Prolog might backtrack should there be a failure) around, which is a waste of space.

If we insert cuts after each test we will tell Prolog that the procedure is deterministic, i.e. that once one test succeeds, there is no way any other test can succeed. Prolog therefore does not need to keep any choice-points around.

```prolog
merge([X|Xs], [Y|Ys], [X|Zs]) :-
  X < Y, !,
  merge(Xs, [Y|Ys], Zs).
merge([X|Xs], [Y|Ys], [X,Y|Zs]) :-
  X = Y, !,
  merge(Xs, Ys, Zs).
merge([X|Xs], [Y|Ys], [Y|Zs]) :-
  X > Y, !,
  merge([X|Xs], Ys, Zs).
merge(Xs, [], Xs) :- !.
merge([], Ys, Ys) :- !.
```

Green Cuts – Merge IV

```
abs1(X, X) :- X >= 0.
abs1(X, Y) :- Y is -X.
?- abs1(-6, X).
  X = 6 ;
?- abs1(6, X).
  X = 6 ;
  X = -6 ;

abs2(X, X) :- X >= 0, !.
abs2(X, Y) :- Y is -X.
?- abs2(-6, X).
  X = 6 ;
?- abs2(6, X).
  X = 6 ;
```
Red Cuts – Abs II

\[\text{abs3}(X, X) :- X \geq 0.\]
\[\text{abs3}(X, Y) :- X < 0, \quad Y \text{ is } -X.\]

?- \text{abs3}(-6, X).
\(X = 6\); no

?- \text{abs3}(6, X).
\(X = 6\); no

Red Cuts – Intersection I

Find the intersection of two lists \(A \& B\), i.e. all elements of \(A\) which are also in \(B\).

\[
\text{intersect}([H|T], L, [H|U]) :- \\
\text{member}(H, L), \\
\text{intersect}(T, L, U).
\]

\[
\text{intersect}([_|T], L, U) :- \\
\text{intersect}(T, L, U).
\]

\[
\text{intersect}([], [], []). 
\]

Red Cuts – Intersection II

?- \text{intersect}([3,2,1],[1,2], L).
\(L = [2,1]\); no

Red Cuts – Intersection III

succeed

\(m(2,[1,2])\)
fail

\(m(3,[1,2])\)
succeed

\(i([3,2,1],[1,2],L)\)

\(L=\{2,1\}\)

\(i([1],[1,2],L)\)

\(i(\_,\_,\{\})\)

\(L=\{1\}\)

\(i([2,1],[1,2],L)\)

\(i(\_,\_,\{\})\)

\(L=\{\}\)

\(i(\_,\_,\{\})\)

\(i(\_,\_,\{\})\)
Red Cuts – Intersection IV

Red Cuts – Intersection V

Red Cuts – Intersection VI

intersect([H|T], L, [H|U]) :-
    member(H, L),
    intersect(T, L, U).
intersect([_|T], L, U) :-
    intersect(T, L, U).
intersect(_, _, []).

intersect1([H|T], L, [H|U]) :-
    member(H, L), !,
    intersect1(T, L, U).
intersect1([_|T], L, U) :-
    !, intersect1(T, L, U).
intersect1(_, _, []).
Blue Cuts I

First clause indexing will select the right clause in constant time:

\[
\text{clause}(x(5), \ldots) \leftarrow \ldots \\
\text{clause}(y(5), \ldots) \leftarrow \ldots \\
\text{clause}(x(5, f), \ldots) \leftarrow \ldots \\
?\leftarrow \text{clause}(x(C, f), \ldots).
\]

First clause indexing will select the right clause in linear time:

\[
\text{clause}(W, x(5), \ldots) \leftarrow \ldots \\
\text{clause}(W, y(5), \ldots) \leftarrow \ldots \\
\text{clause}(W, x(5, f), \ldots) \leftarrow \ldots \\
?\leftarrow \text{clause}(a, x(C, f), \ldots).
\]

Blue Cuts II

capital(britain, london).
capital(sweden, stockholm).
capital(nz, wellington).
?- capital(sweden, X).
   X = stockholm
?- capital(X, stockholm).
   X = sweden

capital1(britain, london) :- !.
capital1(sweden, stockholm) :- !.
capital1(nz, wellington) :- !.
?- capital1(sweden, X).
   X = stockholm
?- capital1(X, stockholm).
   X = sweden

Red Cuts – Once I

\[
\text{member}(H,[H|\_]). \\
\text{member}(I, [\_|T]) \leftarrow \text{member}(I, T).
\]

\[
- \text{member}(1,[1,1]), \text{write}'x'\,
  \text{fail}. \\
  xx \\
\text{mem1}(H,[H|\_]) \leftarrow !. \\
\text{mem1}(I, [\_|T]) \leftarrow \text{mem1}(I, T). \\
- \text{mem1}(1,[1,1]), \text{write}'x'\,
  \text{fail}. \\
x
\text{once}(G) \leftarrow \text{call}(G), !. \; \text{one_mem}(X, L) \leftarrow \text{once}(\text{mem}(X, L)) \\
- \text{one_mem}(1,[1,1]), \text{write}'x'\,
  \text{fail}. \\
x
\]

Red Cuts – Once II

Red cuts prune away logical solutions. A clause with a red cut has no logical reading.

?- \text{member}(X, [1,2]). \\
   X = 1 ; \\
   X = 2 ; \\
   no \\
?- \text{one_mem}(X, [1,2]). \\
   X = 1 ; \\
   no
### Red Cuts – Abs III

\[
\text{abs2}(X, X) :- X \geq 0, !.
\]
\[
\text{abs2}(X, Y) :- Y \text{ is } -X.
\]

\[
\text{if}_{\text{then}}_{\text{else}}(P, Q, R) :- \text{call}(P), !, Q.
\]
\[
\text{if}_{\text{then}}_{\text{else}}(P, Q, R) :- R.
\]

\[
\text{abs4}(X, Y) :- \text{if}_{\text{then}}_{\text{else}}(X \geq 0, Y = X, Y \text{ is } -X).
\]

?\(- \text{abs4}(-6, X).
\]
\[
\begin{align*}
X &= 6 \\
\text{no}
\end{align*}
\]

?\(- \text{abs4}(6, X).
\]
\[
\begin{align*}
X &= 6 \\
\text{no}
\end{align*}
\]

### IF–THEN–ELSE I

\[
\text{intersect}([H|T], L, [H|U]) :-
\]
\[
\text{member}(H, L), !,
\]
\[
\text{intersect}(T, L, U).
\]

\[
\text{intersect}([-|T], L, U) :-
\]
\[
!, \text{intersect}(T, L, U).
\]
\[
\text{intersect}([-,-,[]]).
\]

IF \( H \in L \) THEN
\[
\text{compute the intersection of } T \text{ and } L,
\]
let \( H \) be in the resulting list.

ELSEIF the list \( \not= [] \) THEN
\[
\text{let the resulting list be the intersection of } T \text{ and } L.
\]

ELSE
\[
\text{let the resulting list be } [].
\]

### IF–THEN–ELSE II

\[
\text{if}_{\text{then}}_{\text{else}}(P, Q, R) :- \text{call}(P), !, Q.
\]
\[
\text{if}_{\text{then}}_{\text{else}}(P, Q, R) :- R.
\]

\[
\text{intersect2}([X|T], L, W) :-
\]
\[
\text{if}_{\text{then}}_{\text{else}}(\text{member}(X, L),
\]
\[
(\text{intersect2}(T, L, U), W = [X|U]),
\]
\[
\text{if}_{\text{then}}_{\text{else}}(T \not= []),
\]
\[
\text{intersect2}(T, L, W),
\]
\[
W = []).
\]

### Open vs. Closed World I

How should we handle negative information?

**Open World Assumption:**

If a clause \( P \) is not currently asserted then \( P \) is neither true nor false.

**Closed World Assumption:**

If a clause \( P \) is not currently asserted then the negation of \( P \) is currently asserted.
Open vs. Closed World II

striker(dahlin).
striker(thern).
striker(andersson).

Open World Assumption:

Dahlin, Thern, and Andersson are strikers, but there may be others we don’t know about.

Closed World Assumption:

x is a striker if and only if x is one of Dahlin, Thern, and Andersson.

Negation in Prolog

Prolog makes the closed world assumption.

Anything that I do not know and cannot deduce is not true.

Prolog’s version of negation is negation as failure.

not(G) means that G is not satisfiable as a Prolog goal.

(1) not(G) :- call(G), !, fail.
(2) not(G).

?- not(member(5, [1,3,5])).
no
?- not(member(5, [1,3,4])).
yes

Prolog Execution – Not

not(P) :- P, !, fail; true.

Negation Example – Disjoint

Do the lists X & Y not have any elements in common?

disjoint(X, Y) :-
not(member(Z, X),
member(Z, Y)).

?- disjoint([1,2],[3,2,4]).
no
?- disjoint([1,2],[3,7,4]).
yes
Prolog Negation Problems I

man(john). man(adam).
woman(sue). woman(eve).
married(adam, eve).

married(X) :- married(X, _).
marrried(X) :- married(_, X).
human(X) :- man(X).
human(X) :- woman(X).

% Who is not married?
?- not married(X).
false

% Who is not dead?
?- not dead(X).
true

Prolog Negation Problems II

If \( G \) terminates then so does not \( G \).
If \( G \) does not terminate then not \( G \) may or may not terminate.
marrried(abraham, sarah).
marrried(X, Y) :- married(Y, X).

?- not married(abraham, sarah).
false
?- not married(sarah, abraham).
non-termination

Open World Assumption I

We can program the open world assumption:
- A query is either true, false, or unknown.
- A false facts \( \bot \) has to be stated explicitly, using false(\( \bot \)).
- If we can’t prove that a statement is true or false, it’s unknown.

% Philip is Charles’ father.
father(philip, charles).

% Charles has no children.
false(father(charles, X)).
Open World Assumption II

prove(P) :- call(P), write('** true'), nl, !.

prove(P) :- false(P), write('** false'), nl, !.

prove(P) :-
    not(P), not(false(P)),
    write('*** unknown'), nl, !.

Open World Assumption III

father(philip, charles).
false(father(charles, X)).

% Is Philip the father of ann?
?- prove(father(philip, ann)).
   ** unknown

% Does Philip have any children?
?- prove(father(philip, X)).
   ** true
   X = charles

% Is Charles the father of Mary?
?- prove(father(charles, mary)).
   ** false