Cut Predicate allows to control the backtracking mechanism of Prolog. With the cut, one can also implement negation.

\[ \text{male}(X) :- \neg \text{female}(X). \]

is not allowed in pure LP, because
\[ \{ \text{male}(X), \text{female}(X) \} \]
is not a Horn clause.

5.4.1. The Cut Predicate

Ex: \[ f(X) = \begin{cases} 0, & \text{if } X < 3 \\ 1, & \text{if } 3 \leq X < 6 \\ 2, & \text{if } 6 \leq X \end{cases} \]

\[ f(X, 0) :- X < 3. \]
\[ f(X, 1) :- 3 =< X, X < 6. \]
\[ f(X, 2) :- 6 =< X. \]

Query: \[ ? - f(1, Y), O < Y. \]

\[ \{ Y/0 \} \quad | \quad \{ Y/1 \} \quad | \quad \{ Y/2 \} \]
\[ 1 < 3, 0 < 0 \quad 3 =< 1, 1 < 6, 0 < 1 \quad 6 =< 1, 0 < 2 \]
Finite Failure, Answer: false

Can we make this program more efficient?

Observation 1: The conditions of the 3 prog. clauses exclude each other. If one of these conditions is true, then we should not consider the other prog. clauses anymore, because their conditions will be false.

This can be exploited by adding the cut predicate to our clauses.

Cut: ! predicate of arity 0

Proof of ! always succeeds, but as a side effect it prunes alternative alternative paths in the SLD tree.

\[
\begin{align*}
  f(X, 0) & : - X < 3, \\
  f(X, 1) & : -3 \leq X, X < 6, \\
  f(X, 2) & : -6 \leq X, \\
  f(Y, 0) & , 0 \leq Y
\end{align*}
\]

Effect: if proof of \( X < 3 \) succeeds, then no alternative proofs for \( X < 3 \) and \( f(X,0) \) are considered, 2 disregard the remaining \( f \)-clauses.
Green Cuts: influence efficiency and termination, but not the results of the program.
If one removes the cuts, one would still get the same solutions.

One can improve efficiency even further:
\[ ?- \neg f(y, y). \]

Answer: \( y = 2 \)

Observation 2: \( x \leq 3 \) and \( 3 \leq x \) are complementary. If \( x \leq 3 \) fails, then we do not have to check \( 3 \leq x \).
anyway, because it must be true.
(similarly for \( X < 6 \) and \( 6 = X \)).

\[
f(X, 0) : \neg X < 3, !.
f(X, 1) : \neg X < 6, !.
f(X, 2).
\]

**Red Cuts:** If one removes the cuts, then one obtains different results.
Then \(?- f(1, Y). Y = 0 ; Y = 1 ; Y = 2\)

\(?- f(\text{gerd}, 2)\)  \(?- f(1, 2)\)

true  true

When adding cuts, one typically has a certain form of query in mind where certain arguments are input (ground terms) and others are output (variables). Here: first arg of \( f \) should be input, second arg of \( f \) should be output.

**Exact Meaning of Cuts**

If a query \(?- A_1, \ldots, A_n\) is resolved with a
prop. clause  \( B := C_1, \ldots, C_i, \neg, C_{i+1}, \ldots, C_n \)

(assuming the argn of \( A_n \) and \( B \)) then one first has to solve the instantiated literals \( C_1, \ldots, C_i \) before reaching \( \neg \).

Effect of the cut: When backtracking, no further alternatives for \( C_1, \ldots, C_i \), \( B = A_n \) are explored.

Ex: To explain the full effect of the cut.

\[ ?- a(X). \]
\[ X=0 ; X=1 ; X=2 ; X=3 ; X=4 ; X=5 \]

If one adds a cut in the second \( B \)-clause, then one obtains:

\[ ?- a(X). \]
\[ X=0 ; X=1 ; X=5 \]

Examples to demonstrate the use of cuts:

\[ \text{gcd}(X, 0, X) := !. \]
\[ \text{gcd}(0, X, X) := !. \]
\[ \text{gcd}(X, Y, Z) := X < Y, !, Y > X, \text{gcd}(X, Y, Z). \]
\[ \text{gcd}(X, Y, Z) := X > Y, \text{gcd}(X, Y, Z). \]

(gcd is only intended for use with nat. numbers)
Another exp.:
remove (X, Xs, Ys) should hold if the list Ys results from the list Xs by removing all occurrences of X.
?- remove (1, [0, 1, 2, 3], Ys).
Ys = [0, 2, 3].

remove (_, [], []). 
remove (X, [X|Xs], Ys) :- !, remove(X, Xs, Ys).
remove (X, [Y|Xs], [Y|Ys]) :- remove(X, Xs, Ys).

The cut ensures that Clause 3 is only reached if the first element of the list is different from X. Otherwise:
?- remove (1, [0, 1, 2, 3], Ys).

This would have the answers
Ys = [0, 2, 3]; Ys = [0, 1, 2, 3]; Ys = [0, 1, 2, 3]; Ys = [0, 1, 2, 3]

5.4.2. Meta-Variablos and Negation

up to now: variables are instantiated by terms
now: variables can also be instantiated by formulas (meta-variables)
Ex. \[ p(a). \]  
\( p \) is a pred. sym. of arity 1  
but its arg. is a formula, not a term 
\( a \) is a predicate  
Symbol of arity 0

Since Prolog is untyped, it does not distinguish between pred. symbols and fun. symbols  
(i.e., atomic formulas and terms).

\[ \text{?- } p(X), X. \]
\[ X = a \]
\[ \text{?- } p(X), X, Y. \]
\text{error}

With meta-variables, we can implement Boolean connectives.  

Disjunction:  
\[ \text{or (X, Y) :- X.} \]
\[ \text{or (X, Y) :- Y.} \]

\( \text{?- X = 4 ; X = 5.} \)  
\( \text{?- or (X = 4, X = 5).} \)
\( \text{versen v} \)  

\( \text{?- X = 4 ; X = 5.} \)  
\( \text{?- or (X = 4, X = 5).} \)
\( \text{versen v} \)
If-Then-Else: if \( A \) then \( B \) else \( C \)

\[
\begin{align*}
\text{if } (A, B, C) & : = A, !, B. \\
\text{if } (A, B, C) & : = C.
\end{align*}
\]

Cut is needed to ensure that if the proof of \( A \) succeeds then we don’t read the 2nd clause.

A similar predicate is pre-defined in Prolog:

\[ A \rightarrow B \rightarrow C \]

With the cut, we can also implement negation.

Up to now, we can only derive statements of the form \( \exists X_1, \ldots, X_n \ A_1 \land \ldots \land A_k \) atomic formulas

Now we also want to include negated atomic formulas.

For negation, Prolog uses 2 assumptions:

- Closed World Assumption: all true statements can be derived from the prog. clauses.
  
  Thus: If a statement can’t be derived,
then it must be false.

(5) If a statement can’t be derived from the program, then this can be detected in finite time.

Then: negation can be implemented as “finite failure.”
To prove \( \neg A \), one tries to prove \( A \). If this proof fails in finite time (i.e., if the SCD tree for \( A \) is finite and does not contain \( \bot \)), then \( \neg A \) holds.

Thus, one can implement “not” as follows:

\[
\text{not} (A) \quad \text{pre-defined pred, which always fails}
\]

\[
\text{not} (A) \quad \text{needed to ensure that one doesn’t read the second clause if } A \text{ could be proved}
\]

"not" is pre-defined, can also be used as a prefix operator \( \downarrow \)

\[
\text{not-equal} (X, Y) \quad \text{not} (X = Y)
\]

\[
\text{not-equal} (1, 2) \quad \text{not-equal} (1, 1)
\]

true

false

\[
\text{not-equal} (1, X) \quad \text{not-equal} (1, X), X = 2
\]
?- X=2, not_equal(X,X).
X=2
not_equal(X,X)
not(1=X)
1=X, !, fail
! fail
fail

?- not_equal(X,X), X=2.
false
not_equal(X,X)
false

1=X true if \exists X, 1=X holds
1=X false if \forall X, 1 \neq X holds
not_equal(X,X) if true

Behavior of negation when assumptions @ or 5 do not hold:

even (0).
even (X) :- X1 is X-2, even (X1).

From these clauses, one can't derive even (1).

?- not (even (1)). ?- even (1)
non-termination non-termination
Proven: Assumption (5) does not hold.

\[
\text{even (0).}
\]
\[
\text{even (X) : - } X \geq 2, \text{ X1 is X-2, even (X1).}
\]

?- not (even (-2)).
true

Problem: Ass. (6)
doesn’t hold.

Programmer forgot to specify that -2 is even. CWA assumes that this was on purpose, i.e., that even (-2) is false.

Alternative correct version:

\[
\text{even (0) : - 1.}
\]
\[
\text{even (X) : - X > 0, \text{ X1 is X-1, not(even (X1)).}
\]
\[
\text{even (X) : - X1 \text{ is X+1, not(even (X1)).}
\]