

Consider the unification of $(X\ X)$ and $(Y\ (a\ Y))$.

One would match the corresponding terms in the clauses to arrive at a conclusion that

- X has be same as Y and
- X has to be same as $(a\ Y)$,

which would imply that X is of the form $(a\ (a\ (a\ (a\ \dots$

Here X does not happen to be finite and so the interpreter must be made aware of cases where it can end up in infinite loops and be able to detect such cases and acknowledge that it is unable to unify the two patterns.

This is one such example, however there can be several such cases, where one accidentally tries to compute a fixed point of a function $f(X) = X$ which may not always be possible to compute.

```
% Facts
% Facts consist of atoms which start
% with lowercase letters.

parent(indira,rajiv).
parent(indira,sanjay).
parent(sanjay,varun).
parent(rajiv,rahul).
parent(rajiv,priyanka).
parent(feroze,sanjay).

% Rules
% variables are atoms inside rules which start
% with uppercase letters or underscore.

grandparent(X,Y):-
    parent(X,Zee),parent(Zee,Y).
```

```
[trace] ?- trace.
true.

[trace] ?- grandparent(X,Y).
Call: (10) grandparent(_14246, _14248) ? creep
Call: (11) parent(_14246, _15572) ? creep
Exit: (11) parent(indira, rajiv) ? creep
Call: (11) parent(rajiv, _14248) ? creep
Exit: (11) parent(rajiv, rahul) ? creep
Exit: (10) grandparent(indira, rahul) ? creep
X = indira,
Y = rahul ;
Redo: (11) parent(rajiv, _14248) ? creep
Exit: (11) parent(rajiv, priyanka) ? creep
Exit: (10) grandparent(indira, priyanka) ? creep
X = indira,
Y = priyanka ;
Redo: (11) parent(_14246, _15572) ? creep
Exit: (11) parent(indira, sanjay) ? creep
Call: (11) parent(sanjay, _14248) ? creep
Exit: (11) parent(sanjay, varun) ? creep
Exit: (10) grandparent(indira, varun) ? creep
X = indira,
Y = varun ;
Redo: (11) parent(_14246, _15572) ? creep
Exit: (11) parent(sanjay, varun) ? creep
Call: (11) parent(varun, _14248) ? creep
Fail: (11) parent(varun, _14248) ? creep
Redo: (11) parent(_14246, _15572) ? creep
Exit: (11) parent(rajiv, rahul) ? creep
Call: (11) parent(rahul, _14248) ? creep
Fail: (11) parent(rahul, _14248) ? creep
Redo: (11) parent(_14246, _15572) ? creep
Exit: (11) parent(rajiv, priyanka) ? creep
Call: (11) parent(priyanka, _14248) ? creep
Fail: (11) parent(priyanka, _14248) ? creep
Redo: (11) parent(_14246, _15572) ? creep
Exit: (11) parent(feroze, sanjay) ? creep
Call: (11) parent(sanjay, _14248) ? creep
Exit: (11) parent(sanjay, varun) ? creep
Exit: (10) grandparent(feroze, varun) ? creep
X = feroze,
Y = varun.
```

Prolog variables are only "variable" until bound (unified) with something else. At that point they cease to be variable and become one with that with which they were unified. Hence the use of the term "unification": to unify is to become one.

For example, `_14246` and `_14248` are two variables which are initially unified with values `indira` and `rajiv` respectively.

The search can be visualised as a tree and would perform a depth first search on this tree to find solutions. After exploring a leaf node, the search backtracks the route it took to reach the leaf node by going one level up on the DFS tree.

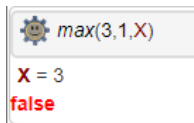
Backtracking at that point, undoes some unification that might have occurred, returning things to the status quo before as it were at that node of the BFS tree.

For example, after producing $X = \text{indira}$ and $Y = \text{rahul}$ as a solution, `_14248` is unbound from `rahul`.

This example shows how backtracking happens to solve the query. For this query, it made perfect sense for the rule to return several solutions.

However in some cases this may not be the desired outcome. For example

```
max(X,Y,X):- X >= Y.  
max(X,Y,Y):- X < Y.
```




In this case, the rule was checked twice and thus it returned two answers. For the first rule, X is 3, whereas the body of the second rule is not satisfied and hence the outcome is false.

In such situations one can employ a mechanism called **cut** !.


```
max(X,Y,X):- X >= Y.  
max(X,Y,Y):- X < Y.
```

```
% notice the ,!. sequence of symbols  
% instead of . at the end of  
% the first rule  
max-with-cut(X,Y,X):- X >= Y,!.  
max-with-cut(X,Y,Y):- X < Y.
```


 max(3,1,X)

X = 3


false

 max-with-cut(3,1,X)

X = 3

 max(1,5,X)

X = 5


 max-with-cut(1,5,X)

X = 5

In this situation, the usage of cut implies that once we find a rule that is satisfied, we need not check for other cases of the same rule to be satisfied and terminate the search immediately thereby stopping the backtracking. The cut does not come into effect, if the rule is not satisfied, as showcased by the last query.

```
teaches(dr_fred, english).  
teaches(dr_fred, history).  
teaches(dr_fred, drama).  
teaches(dr_jimmy, biology).  
teaches(dr_fiona, physics).  
  
studies(alice, english).  
studies(angus, english).  
studies(amelia, drama).  
studies(alex, physics).
```

```
% No cuts  
l1(Course,Student):-  
    teaches(dr_fred, Course),  
    studies(Student, Course).  
  
% l2 Cuts after returning first true case  
% which matches the first condition  
% It may however return different values  
% for the student based on the second condition.  
l2(Course,Student):-  
    teaches(dr_fred, Course),  
    !,  
    studies(Student, Course).  
  
% l3 Cuts after returning first true case  
% which matches both conditions  
l3(Course,Student):-  
    teaches(dr_fred, Course),  
    studies(Student, Course), !.
```

 l1(Course,Student)

Course = english,
Student = alice
Course = english,
Student = angus
Course = drama,
Student = amelia

 l2(Course,Student)

Course = english,
Student = alice
Course = english,
Student = angus

 l3(Course,Student)


Course = english,
Student = alice

ℓ_1 rule has no cut and thus the first query returns all possible solutions. ℓ_2 rule returns only those solutions where course matches the first course offered by dr_fred. It backtracks the choices for the variable Student, but does not backtrack the choice for the variable Course.

Thus if the first and second facts are interchanged, the second $\ell 2$ query would have returned false.


$\ell 3$ rules returns only the first instance where both the conditions are satisfied.


```
sumto(1,1).  
sumto(N,S):- N1 is N-1,  
             sumto(N1,S1),  
             S is S1+N.
```

 `sumto(6,S)`

S = 21

Next 10 100 1,000 Stop

 `sumto(6,S)`

S = 21

Stack limit (0.2Gb) exceeded

Stack sizes: local: 0.2Gb, global: 18.3Mb, trail: 0Kb

Stack depth: 2,403,449, last-call: 0%, Choice points: 12

Possible non-terminating recursion:

[2,403,449] `sumto(-2403415, _1440)`

[2,403,448] `sumto(-2403414, _1466)`

The `sumto(X,N)` rule returns the sum of first X positive integers as N. We know that it has a unique answer, but the prolog implementation causes it to backtrack and find alternative solutions.

Whilst evaluating the goal `sumto(6,S)` the Prolog system will try to find a solution for the goal `sumto(1,S)`.

The first time it does this the first clause is used and the second argument is correctly bound to 1.

Whilst evaluating the goal $\text{sumto}(6,S)$ the Prolog system will try to find a solution for the goal $\text{sumto}(1,S)$.


The first time it does this the first clause is used and the second argument is correctly bound to 1.

On backtracking the first clause is rejected and the system attempts to satisfy the goal using the second clause.

This causes it to subtract one from one and then evaluate the goal $\text{sumto}(0,S)$. Doing this will in turn require it to evaluate $\text{sumto}(-1,S1)$, then $\text{sumto}(-2,S1)$ and so on, until eventually the system runs out of memory.


One could make use of a cut and avoid this issue.

```
sumto(1,1):-!.  
sumto(N,S):- N1 is N-1,  
             sumto(N1,S1),  
             S is S1+N.
```

 `sumto(6,S)`
S = 21


Alternatively, one could avoid the usage of cut and come up with a different solution.

```
sumto(1,1).  
sumto(N,S):- N>1,  
             N1 is N-1,  
             sumto(N1,S1),  
             S is S1+N.
```

 `sumto(6,S)`
S = 21
false

However, in some cases the usage of cut may not be avoidable.

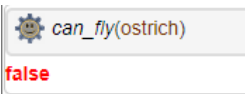
```
1 bird(sparrow).  
2 bird(ostrich).  
3 bird(swan).  
4 bird(owl).  
5  
6 can_fly(ostrich):-fail.  
7 can_fly(X):-bird(X).
```

 `can_fly(ostrich)`
`true`

One would ideally want the prolog interpreter to return false in this situation, however, after the failure of rule at line 6, prolog checks the rule at line 7 and returns true.

To stop this from happening we can use a cut.

```
bird(sparrow).  
bird(ostrich).  
bird(swan).  
bird(owl).  
  
can_fly(ostrich):-!,fail.  
can_fly(X):-bird(X).
```



As before, the `can_fly(ostrich)` goal is matched with the head of the first `can_fly` clause.

Attempting to satisfy the goal in the body of that clause (i.e. `fail`) fails, but the cut prevents the system from backtracking and so the `can_fly(ostrich)` goal fails.

The combination of goals `!,fail` is known as **cut with failure**.

```
:- use_module(library(clpfd)).
```

```
square(X,Y) :- Y is X*X.
```

```
square2(X,Y):- Y #= X*X.
```

*% In this situation the rule is not unable to
% provide the value of X given the value of Y.*







```
poly(X,Y):- Y #= X*X*X+3*X*X+3*X+1.
```

```
⚙ square(5,X).  
X = 25  
⚙ square(X,25).  
Arguments are not sufficiently instantiated  
In:  
[1] 25 is _1706*_1708  
⚙ square2(5,X).  
X = 25  
⚙ square2(X,25).  
X in -5V5  
⚙ poly(5,Y).  
Y = 216  
⚙ poly(X,216).  
3*X#=_3524,  
_3552*X#=_3548,  
3*X#=_3552,  
X^3#=_1240,  
_1256+_1204#=_215,  
_1240+_1216#=_1256
```

The square rule only works in one direction. The square2 rule works in both directions as expected, thanks to the use of the clpfd library. Even with the use of the library, it might not be possible to invert several mathematical functions, like poly.


Arithmetic Expression Equality


$==$ and $\backslash=$


 <code>sqrt(36)+4 == 5*11-45.</code>	 <code>sqrt(36)+4 == 5*11-45</code>
true	false
 <code>sqrt(36) \= 7.</code>	 <code>X is 11, pred1(X) == pred1(10)</code>
true	false
	 <code>X is 10, pred1(X) == pred1(10)</code>
	X = 10
	 <code>10 \== 8+2</code>
	true


Term equality
 $==$ and $\backslash==$

A third version of equality checks whether a unification is possible between two patterns.

 `likes(X,prolog)=likes(john,Y).`
`X = john,`
`Y = prolog`

 `6+X=6+3.`
`X = 3`

 `likes(X,prolog)=likes(Y,java).`
false

 `likes(X,prolog)\=likes(john,Y).`
false

 `likes(X,prolog)\=likes(X,java).`
true

In general, the query evaluator uses the following method to apply a rule when trying to establish a query pattern in a frame that specifies bindings for some of the pattern variables:

- ▶ Unify the query with the conclusion of the rule to form, if successful, an extension of the original frame.
- ▶ Relative to the extended frame, evaluate the query formed by the body of the rule.

This is very to the method for applying a procedure in the eval/apply evaluator for Scheme:

- ▶ Bind the procedure's parameters to its arguments to form a frame that extends the original procedure environment.
- ▶ Relative to the extended environment, evaluate the expression formed by the body of the procedure.

- ▶ Map coloring
- ▶ Timetable
- ▶ Sudoku/Kakuro/Wordle
- ▶ N-Queens
- ▶ Tournament Scheduling
- ▶ Einstein puzzles
- ▶ Logic Puzzles like Knight and Knaves

These puzzles do not require arithmetic skills, however knowledge of graph theory and logic deductions can be quite essential and prolog is quite adept at dealing with such situations.

Premise: You are on an island where every inhabitant is either a knight or a knave.

Knights always tell the truth, and knaves always lie.

To make deductions we shall use Constraint Logic Programming over Boolean variables library by using

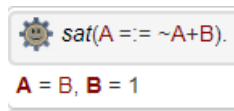
`:- use_module(library(clpb)).`

Using Prolog and its CLP(B) constraints, we can model this situation as follows:

- ▶ use 0 (false) to denote a knave.
- ▶ use 1 (true) to denote a knight.
- ▶ use one Boolean variable for each inhabitant, its truth value representing whether the person is a knave or a knight.

Example 1: You meet 2 inhabitants, A and B.
A says: "Either I am a knave or B is a knight."

We can model this statement as follows and use sat



sat/1 (aka sat(Expr)) predicate returns a True \iff Expr is a satisfiable Boolean expression.

In this situation, the output implies that both A and B are knights.
The usage of 'either' here is in mathematical sense of OR and not in sense of XOR.

Example 2: A says: "I am a knave, but B isn't."

Corresponding query and response



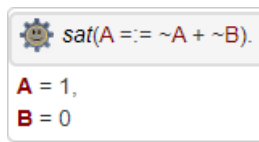
$\text{sat}(A \text{ := } \sim A * B).$

$A = B, B = 0$

Inference:- Both A and B are knaves.

Example 3: A says: "At least one of us is a knave."

Corresponding query and response




Inference:- A is a knight and B is a knave.

Example 4: You meet 3 inhabitants. A says: "All of us are knaves."

B says: "Exactly one of us is a knight."

Corresponding query and response

 `sat(A := (~A * ~B * ~C)), sat(B := card([1],[A,B,C])).`

`A = C, C = 0,`

`B = 1`

Using CLP(B) constraints, we can use the Boolean expression `card(Ls,Exprs)` to express a cardinality constraint.

This expression is true \iff the number of expressions in `Exprs` that evaluate to true is a member of the list `Ls` of integers.

Inference:- A and C are knaves, but B is a knight.

Example 5: A says: "B is a knave." B says: "A and C are of the same kind." What is C?

Corresponding query and response



$\text{sat}(A = \text{:=} \sim B), \text{sat}(B = \text{:=} (A = \text{:=} C)).$

$C = 0,$

$\text{sat}(A \neq B)$

Inference:- C is definitely a knave. A and B can both be either knights or knaves, and they are of different kinds, which is indicated by the residual goal.