Utrecht University

Introduction to Prolog 3

Cut, Negation and Dynamic Predicates

Alvaro H. C. Correia

6th of June 2019

Utrecht University

Comparison Operators

Unify - Do not Unify

- **X = Y**

  X and Y unify (match).

- **X \= Y**

  X and Y do not unify (differ). No instantiation occurs.

Unify - Do not Unify

- **X = Y**

  X and Y unify (match).

- **X \= Y**

  X and Y do not unify (differ). No instantiation occurs.

- **X == Y**

  X and Y are identical. No instantiation occurs.

- **X \== Y**

  X and Y are not identical. No instantiation occurs.

Unify - Do not Unify - Examples

Comparison operators

Identity

```
?- a == a.
```

```
true.
```

Unify - Do not Unify - Examples

Comparison operators

Unification predicate. X is instantiated to a

```
?- a == a.
true.
?- X = a.
X = a.
```

Unify - Do not Unify - Examples

Comparison operators

Negation of the unification predicate.

```
?- a == a.
true.
?- X = a.
X = a.
?- X \= a.
false.
```

Unify - Do not Unify - Examples

Comparison operators

Identity. No instantiation occurs, and hence Prolog returns false.

```
?- a == a.
true.
?- X = a.
X = a.
?- X \= a.
false.
?- X == a.
false.
```

Unify - Do not Unify - Examples

Comparison operators

Negation of the identity predicate.

```
?- a == a.
true.
?- X = a.
X = a.
?- X \= a.
false.
?- X == a.
false.
?- X \== a.
true.
```

Standard Order of Terms

Variables < Numbers < Strings < Atoms < Compound Terms[1]

1. Variables are sorted by address.
2. Numbers are compared by value.
3. Strings are compared alphabetically.
4. Atoms are compared alphabetically.
5. Compound terms are first checked on their arity, then on their functor name (alphabetically).

[1] http://www.swi-prolog.org/pldoc/man?section=compare

Comparison Operators - Numeric

X < Y	numeric value of X is smaller than that of Y.
X =< Y	numeric value of X is smaller than or equal to that of Y.
X > Y	numeric value of X is bigger than that of Y.
X >= Y	numeric value of X is bigger than or equal to that of Y.
X =:= Y	X evaluates to same number as Y (numeric identity).
X =\= Y	X does not evaluate to same number as Y.

Comparison Operators - Numeric

The =:= operator

It denotes the arithmetic equality of two arithmetic expressions.

It is true if and only if its arguments evaluate to the same number.

```
?- 1+2 == 3*1.
false.
?- 1+2 is 3*1.
false.
?- 1+2 =:= 3*1.
true.
```

Comparison Operators - Numeric

The =:= operator

It denotes the arithmetic equality of two arithmetic expressions.

Note that both expressions should be fully instantiated!

```
?- 1+2 == 3*1.
false.
?- 1+2 is 3*1.
false.
?- 1+2 =:= 3*1.
true.
?- 1+X =:= 3*1.
ERROR: Arguments are not sufficiently instantiated
```

Comparison Operators - Alphabetic

X @< Y	X precedes Y alphabetically.
X @=< Y	X is equal to or precedes Y alphabetically.
X @> Y	X succeeds Y alphabetically.
X @>= Y	X is equal to or succeeds Y alphabetically.
X == Y	X and Y are identical.
X \== Y	X and Y are not identical.

Comparison Operators - Examples

Comparison is useful when writing rules

We can only arrange a match between players who are not the same.

```
player(federer).
player(nadal).
player(djokovic).
arrangeMatch(X, Y) :-
    player(X),
    player(Y),
    X \= Y.
```

```
?- arrangeMatch(nadal, X).
X = federer;
X = djokovic.
```

Comparison Operators - Examples

Comparison is useful when writing rules

Notice that we cannot have the comparison as first subgoal. Why?

```
player(federer).
player(nadal).
player(djokovic).
arrangeMatch(X, Y) :-
    X \= Y,
    player(X),
    player(Y).
```

```
?- arrangeMatch(nadal, X).
false.
```

Comparison Operators - Examples

Comparison is useful when writing rules

We sell if we are offered at least what we ask.

We buy if they ask for at most what we are willing to offer.

```
sell(Bid, Ask) :-
    Bid >= Ask.
buy(Bid, Ask) :-
    Bid =< Ask.
```

```
?- sell(10, 8).
true.
```

Comparison Operators - Examples

Comparison is useful when writing rules

Arithmetic comparison only works with ground terms (no free variables).

```
sell(Bid, Ask) :-
    Bid >= Ask.
buy(Bid, Ask) :-
    Bid =< Ask.
```

```
?- sell(10, 8).
true.
?- sell(10, X).
ERROR: Arguments are not
sufficiently instantiated
```

Useful Predicates

fail

A special predicate that is always false.

When Prolog fails to prove a clause, it tries to backtrack.

`fail/0` is then a way to force Prolog to backtrack.

Predicate `p/1` below will always be false.

```
a.
b.
c.
p(_) :- a, b, fail, c.
```

fail - Example

We can use fail to print all possible solutions for a goal

Why do we need `fail` for this?

```
print_all(Goal):-
    Goal,
    writeln(Goal),
    fail.
print_all(_). % base case
```

fail - Example

We can use fail to print all possible solutions for a goal

Because of the `fail` predicate Prolog never succeeds and goes through all solutions before stopping on `print_all(_)`.

```
player(federer).
player(nadal).
player(djokovic).
print_all(Goal):-
    Goal,
    writeln(Goal),
    fail.
print_all(_). % base case
```

```
?- print_all(player(X)).
player(federer)
player(nadal)
player(djokovic)
true.
```

call

call(:Goal, +ExtraArg1, ...)

Append ExtraArg1, ExtraArg2, ... to the argument list of Goal and call the result.

using call/2

```
call(write, "Hello world").
Hello world
true.
```

equivalent expression

```
write("Hello world").
Hello world
true.
```

[2]

[2] http://www.swi-prolog.org/pldoc/doc_for?object=call/2

call

call(:Goal, +ExtraArg1, ...)

Append ExtraArg1, ExtraArg2, ... to the argument list of Goal and call the result.

using call/2

```
call(write, "Hello world").
Hello world
true.
?- call(is, 3, 2+1)
true.
```

equivalent expression

```
write("Hello world").
Hello world
true.
?- is(3, 2+1).
true.
```

[2]

[2] http://www.swi-prolog.org/pldoc/doc_for?object=call/2

call - Example

We can use call to handle multiple predicates with a single rule

Suppose we use a different predicate for each item in a database.

```
pen(1.50).
pencil(0.70).
eraser(0.50).
getPrice(Item, Price) :-
    call(Item, Price).
```

```
?- getPrice(pencil, Price).
Price = 0.70.
?- getPrice(eraser, Price).
Price = 0.50.
```

call - Example

We can use call to handle multiple predicates with a single rule

`getPrice/2` is a meta-predicate because it take goals or other predicates as arguments.

```
pen(1.50).
pencil(0.70).
eraser(0.50).
getPrice(Item, Price) :-
    call(Item, Price).
```

```
?- getPrice(pencil, Price).
Price = 0.70.
?- getPrice(eraser, Price).
Price = 0.50.
```

between

between(+Low, +High, ?Value)

True if Low =< Value =< High.

```
?- between(1, 3, X).
X = 1;
X = 2;
X = 3.
```

[3]

[3] http://www.swi-prolog.org/pldoc/man?predicate=between/3

between

between(+Low, +High, ?Value)

We can only have a variable as third argument.

```
?- between(1, 3, X).
X = 1;
X = 2;
X = 3.
?- between(1, X, 3).
ERROR: Arguments are not sufficiently instantiated
```

3

[3] http://www.swi-prolog.org/pldoc/man?predicate=between/3

between - Example

Let us use between/3 to create a divisor/2 predicate

`divisor(N, D): True if D is a divisor of N.`

between - Example

Let us use between/3 to create a divisor/2 predicate

```
divisor(N, D): True if D is a divisor of N.

divisor(N, D) :-
    between(1, N, D),
    0 is N mod D.
```

between - Example

Let us use between/3 to create a divisor/2 predicate

```
divisor(N, D): True if D is a divisor of N.
```

```
divisor(N, D) :-
    between(1, N, D),
    0 is N mod D.
```

```
?- divisor(10, X).
X = 1 ;
X = 2 ;
X = 5 ;
X = 10.
```

findall

findall(+Template, :Goal, -Bag)

Bag is a list of solutions to Goal organized according to Template.

```
?- findall(X, divisor(10, X), L).
L = [1, 2, 5, 10].
```

4

[4] http://www.swi-prolog.org/pldoc/doc_for?object=findall/3

findall

findall(+Template, :Goal, -Bag)

Notice that the template can be almost anything.

```
?- findall(X, divisor(10, X), L).
L = [1, 2, 5, 10].
?- findall([X], divisor(10, X), L).
L = [[1], [2], [5], [10]].
```

4

[4] http://www.swi-prolog.org/pldoc/doc_for?object=findall/3

findall

findall(+Template, :Goal, -Bag)

Templates can also be functors, but Prolog will not try to match them.

```
?- findall(X, divisor(10, X), L).
L = [1, 2, 5, 10].
?- findall([X], divisor(10, X), L).
L = [[1], [2], [5], [10]].
?- findall(write(X), divisor(10, X), L).
L = [write(1), write(2), write(5), write(10)].
```

4

[4] http://www.swi-prolog.org/pldoc/doc_for?object=findall/3

findall

findall(+Template, :Goal, -Bag)

There are other built-in predicates with similar behaviour.

bagof/3 similar but fails if there is no solution.

setof/3 similar but removes duplicates.

```
?- findall(X, divisor(-1, X), L).
L = [].
?- bagof(X, divisor(-1, X), L).
false.
?- findall(1, divisor(10, X), L).
L = [1, 1, 1, 1].
?- setof(1, divisor(10, X), L).
X = 1, L = [1].
```

forall

forall(:Cond, :Action)

For all alternative bindings of Cond, Action can be proven.

We can then write a new predicate to print all solutions.

`forall/2` actually uses a failure-driven loop in its definition.

In contrast to `findall/3`, `forall/2` does call the predicate in Action.

```
print_all_2(Goal) :-
    forall(Goal, writeln(Goal)).
```

```
?- print_all_2(player(X)).
player(federer)
player(nadal)
player(djokovic)
true.
```

[5]

[5] http://www.swi-prolog.org/pldoc/man?predicate=forall/2

forall

forall(:Cond, :Action)

We can combine `forall` with `call` to create a more general predicate.

```
print_all_3(Pred) :-
    forall(
        call(Pred, X),
        writeln(X)
    ).
```

forall

forall(:Cond, :Action)

Notice that we can no longer print the whole goal with `writeln/1`. Why?

```
print_all_3(Pred) :-
    forall(
        call(Pred, X),
        writeln(X)
    ).
```

```
?- print_all(player).
federer
nadal
djokovic
true.
```

forall

forall(:Cond, :Action)

To get the same result, we can use another predicate, `format/2`.

```
print_all_3(Pred) :-
    forall(
        call(Pred, X),
        format(
            "~w(~w)~n",
            [Pred, X]
        )
    ).
```

```
?- print_all(player).
player(federer)
player(nadal)
player(djokovic)
true.
```

Checking the type of a term

There are few predicates to distinguish Prolog terms.

atom/1	True if the term is an atom.
var/1	True if the term is a free variable.
ground/1	True if the term does not contain free variables.
number/1	True if the term is bound to an integer or float.
string/1	True if the term is bound to a string.

```
?- atom(a).
true.
?- var(X).
true.
?- X=a, var(X).
false.
```

```
?- ground(write(X)).
false.
?- X=a, ground(write(X)).
X = a.
```

Checking the type of term - Example

Creating a predicate gate/1

`gate(X)` is true if `X=0` or `X` is a free variable.

```
?- gate(3).
false.
?- gate(0).
true.
?- gate(X).
true.
?- X=1, gate(X).
false.
```

Checking the type of term - Example

Creating a predicate gate/1

`gate(X)` is true if `X=0` or `X` is a free variable.

```
gate(X) :- var(X).
gate(X) :- X=0.
```

```
?- gate(3).
false.
?- gate(0).
true.
?- gate(X).
true.
?- X=1, gate(X).
false.
```

Negation

Negation \+

Negating a subgoal

We can make the success of a goal dependent on the failure of a subgoal.

In Prolog, we do that using the `\+` operator.

```
head :- subgoal_1,
    subgoal_2,
    \+ subgoal_3,
    ...
    subgoal_n.
```

Negation \+

Negating a subgoal

We can make the success of a goal dependent on the failure of a subgoal.

In Prolog, we do that using the `\+` operator.

```
head :- subgoal_1,
    subgoal_2,
    \+ subgoal_3,
    ...
    subgoal_n.
```

We cannot use negation in the head of a rule!

Negation \+

Negating a subgoal

We can make the success of a goal dependent on the failure of a subgoal.

In Prolog, we do that using the `\+` operator.

```
head :- subgoal_1,
    subgoal_2,
    \+ subgoal_3,
    ...
    subgoal_n.
```

We cannot use negation in the head of a rule!

Obs: In SWI-Prolog we could also write `not(subgoal_3)` but `\+` is preferred.

Negation \+

We can use negation to express defaults

Assume we have match between Ajax and PSV.

```
ajax(hakim).
ajax(david).
ajax(donny).
```

Negation \+

We can use negation to express defaults

If a player does not play for Ajax, he must play for PSV.

```
ajax(hakim).
ajax(david).
ajax(donny).
psv(X) :- \+ajax(X).
```

Negation \+

We can use negation to express defaults

If a player does not play for Ajax, he must play for PSV.

```
ajax(hakim).
ajax(david).
ajax(donny).
psv(X) :- \+ajax(X).
```

```
?- psv(erick).
true.
```

Negation \+

We can use negation to express defaults

If it does not follow from the database, it is false! Closed-world assumption.

The whole world will be on the PSV side.

```
ajax(hakim).
ajax(david).
ajax(donny).
psv(X) :- \+ajax(X).
```

```
?- psv(erick).
true.
?- psv(cat).
true.
```

Negation \+

We can use negation to express defaults

It does not work with variables either. Why?

```
ajax(hakim).
ajax(david).
ajax(donny).
psv(X) :- \+ajax(X).
```

```
?- psv(erick).
true.
?- psv(cat).
true.
?- psv(X).
false.
```

Negation \+

We can use negation to express defaults

`\+ajax(X)` fails because there is at least one ajax player.

```
ajax(hakim).
ajax(david).
ajax(donny).
psv(X) :- \+ajax(X).
```

```
?- psv(erick).
true.
?- psv(cat).
true.
?- psv(X).
false.
```

Negation \+

\+ goal succeeds if and only if goal fails.

This is known as negation as failure and differs from our intuition of negation. In Prolog, negation is defined via its failure to provide a proof.

```
student(john).
student(mike).
married(mike).
unmarried_student(X) :-
    \+ married(X), student(X).
```

Negation \+

\+ goal succeeds if and only if goal fails.

Consider predicate `unmarried_student/1`.

```
student(john).
student(mike).
married(mike).
unmarried_student(X) :-
    \+ married(X), student(X).
```

```
?- unmarried_student(john).
true.
?- unmarried_student(mike).
false.
```

Negation \+

\+ goal succeeds if and only if goal fails.

Notice that `unmarried_student(X)` does not return a solution as expected.

```
student(john).
student(mike).
married(mike).
unmarried_student(X) :-
    \+ married(X), student(X).
```

```
?- unmarried_student(john).
true.
?- unmarried_student(mike).
false.
?- unmarried_student(X).
false.
```

Negation \+

\+ goal succeeds if and only if goal fails.

The position of the negation in the rule matters!

```
student(john).
student(mike).
married(mike).
unmarried_student(X) :-
    student(X), \+ married(X).
```

Negation \+

\+ goal succeeds if and only if goal fails.

The position of the negation in the rule matters!

```
student(john).
student(mike).
married(mike).
unmarried_student(X) :-
    student(X), \+ married(X).
```

```
?- unmarried_student(john).
true.
?- unmarried_student(mike).
false.
?- unmarried_student(X).
john.
```

Cut!

The Cut Operator

!

Cut is a special operator represented by the exclamation sign.

- It always succeeds.
- It discards all previous choicepoints.

We say that cut commits to the current solution. No alternative solutions for already instantiated variables will be considered.

Taking the Minimum

Suppose we want to take the minimum of two numbers

`minimum(+Number1, +Number2, ?Minimum).`

Succeeds if Minimum is the minimum value of Number1 and Number2.

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


Dave Robertson. *Quick Prolog*. Sept. 2005. URL: http://www.inf.ed.ac.uk/teaching/courses/lp/prolognotes.pdf.

Taking the Minimum

Suppose we want to take the minimum of two numbers

`minimum(+Number1, +Number2, ?Minimum).`

Succeeds if Minimum is the minimum value of Number1 and Number2.

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

```
?- minimum(1, 2, X).
X = 1;
```

Dave Robertson. *Quick Prolog*. Sept. 2005. URL: http://www.inf.ed.ac.uk/teaching/courses/lp/prolognotes.pdf.

Taking the Minimum

Suppose we want to take the minimum of two numbers

`minimum(+Number1, +Number2, ?Minimum).`

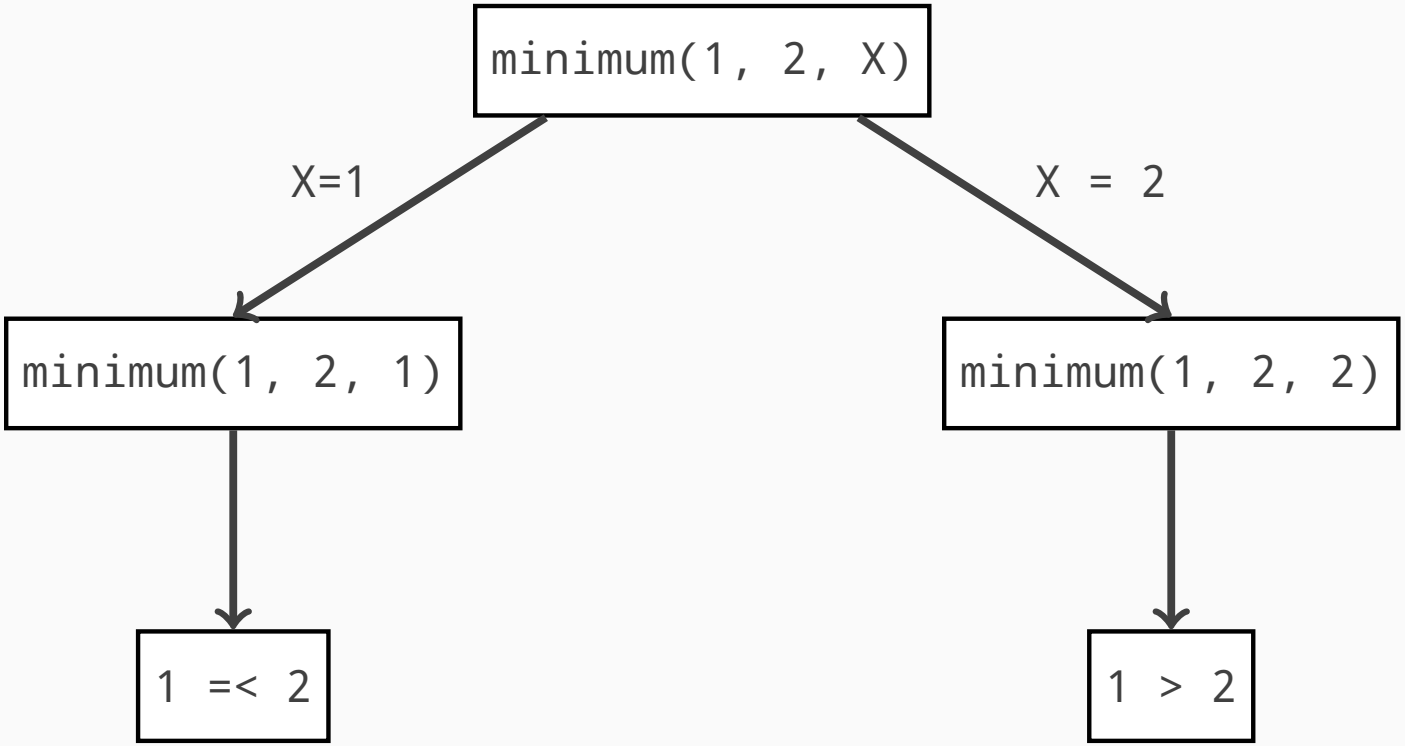
Succeeds if Minimum is the minimum value of Number1 and Number2.

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

```
?- minimum(1, 2, X).
X = 1;
false.
```

Dave Robertson. *Quick Prolog*. Sept. 2005. URL: http://www.inf.ed.ac.uk/teaching/courses/lp/prolognotes.pdf.

Taking the Minimum - Search Tree



Dave Robertson. *Quick Prolog*. Sept. 2005. URL: http://www.inf.ed.ac.uk/teaching/courses/lp/prolognotes.pdf.

Taking the Minimum - Green Cut

We introduce a cut to remove pointless computations

`gc_minimum(+Number1, +Number2, ?Minimum).`

Succeeds if Minimum is the minimum value of Number1 and Number2.

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


Dave Robertson. *Quick Prolog*. Sept. 2005. URL: http://www.inf.ed.ac.uk/teaching/courses/lp/prolognotes.pdf.

Taking the Minimum - Green Cut

We introduce a cut to remove pointless computations

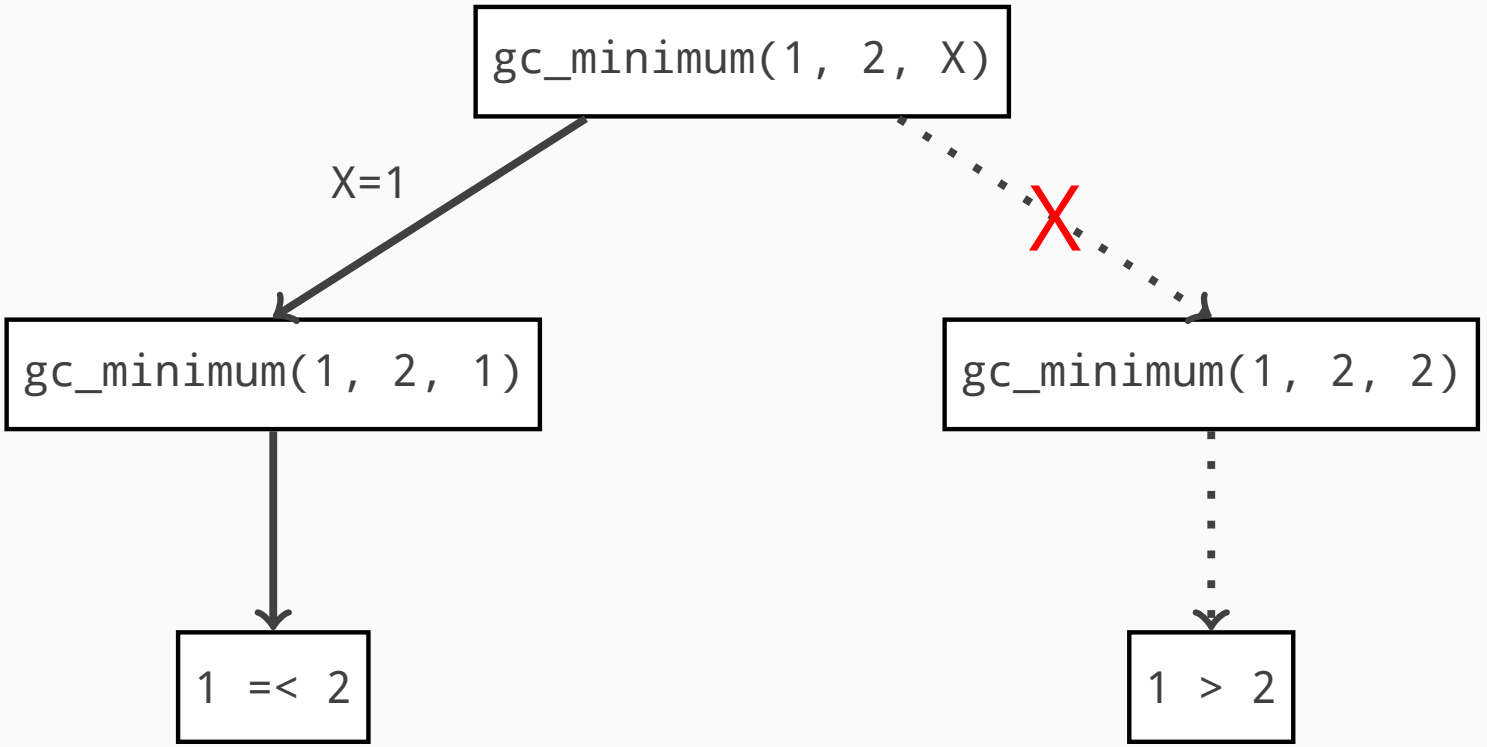
It works exactly as before, but we get a single solution.

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

```
?- gc_minimum(1, 2, X).
X = 1.
```

Dave Robertson. *Quick Prolog*. Sept. 2005. URL: http://www.inf.ed.ac.uk/teaching/courses/lp/prolognotes.pdf.

Green Cut - Search Tree




```
true.
```

Dave Robertson. *Quick Prolog*. Sept. 2005. URL: http://www.inf.ed.ac.uk/teaching/courses/lp/prolognotes.pdf.

Taking the Minimum - Red Cut

We could remove the second condition altogether

`rc_minimum(+Number1, +Number2, ?Minimum).`

Succeeds if Minimum is the minimum value of Number1 and Number2.

```
rc_minimum(X, Y, X):-
    X =< Y, !.
rc_minimum(_, Y, Y).
```


Dave Robertson. *Quick Prolog*. Sept. 2005. URL: http://www.inf.ed.ac.uk/teaching/courses/lp/prolognotes.pdf.

Taking the Minimum - Red Cut

We could remove the second condition altogether

It still gives us the correct minimum.

```
rc_minimum(X, Y, X):-
    X =< Y, !.
rc_minimum(_, Y, Y).
```

```
?- rc_minimum(1, 2, X).
X = 1.
```

Dave Robertson. *Quick Prolog*. Sept. 2005. URL: http://www.inf.ed.ac.uk/teaching/courses/lp/prolognotes.pdf.

Taking the Minimum - Red Cut

We could remove the second condition altogether

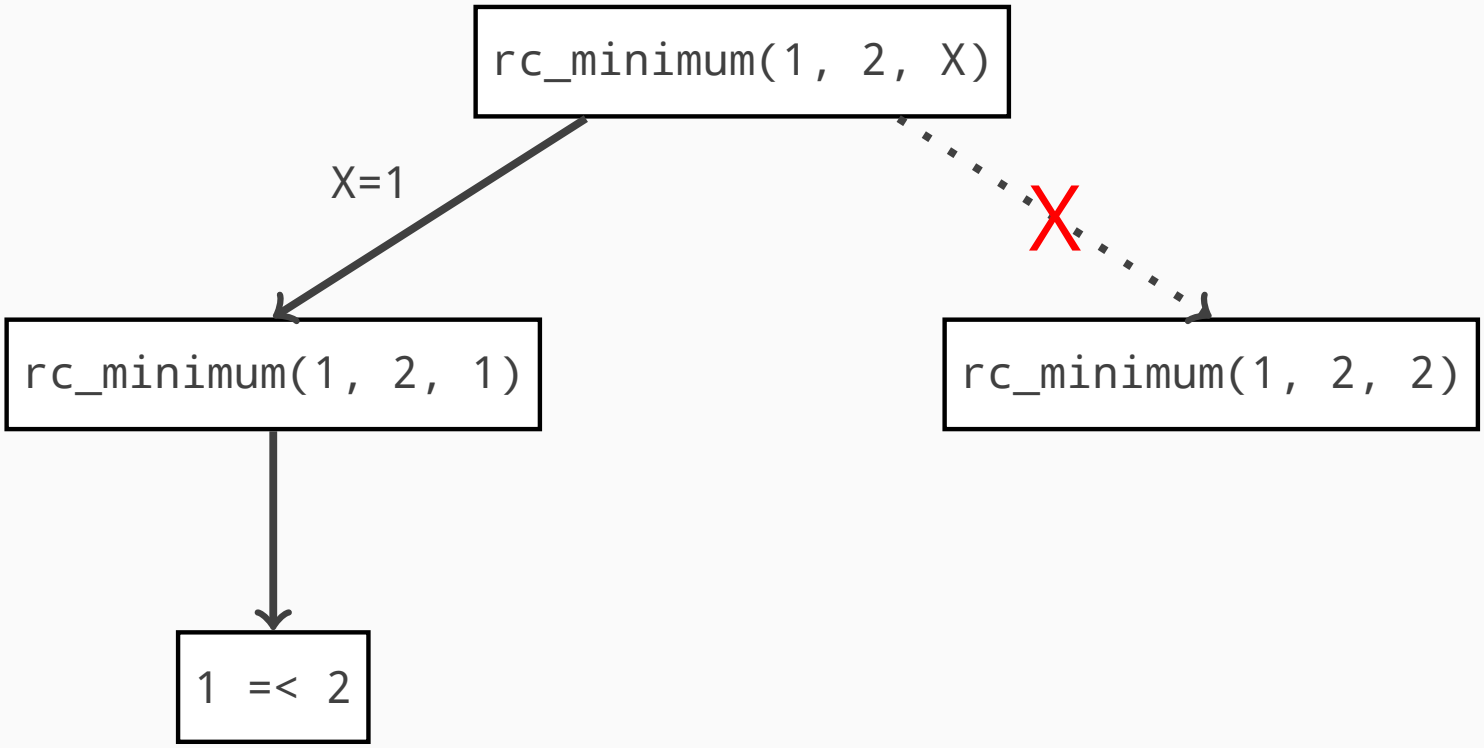
But it gives some unexpected results.

```
rc_minimum(X, Y, X):-
    X =< Y, !.
rc_minimum(_, Y, Y).
```

```
?- rc_minimum(1, 2, X).
X = 1.
?- rc_minimum(1, 2, 2).
true.
```

Dave Robertson. *Quick Prolog*. Sept. 2005. URL: http://www.inf.ed.ac.uk/teaching/courses/lp/prolognotes.pdf.

Red Cut - Search Tree




```
true.
```

Dave Robertson. *Quick Prolog*. Sept. 2005. URL: http://www.inf.ed.ac.uk/teaching/courses/lp/prolognotes.pdf.

Green and Red Cuts

Cuts prune unnecessary backtracking

but they may or may not change the meaning of a procedure.

Green Cut

Does not change the meaning of a procedure.

Red Cut

Introduces unexpected behaviour that differs from the original procedure with no cuts.

Using Cuts - Switch

We can use cuts to switch between cases.

The cut commits to a case: Prolog executes only one at each call to switch.

```
switch(1):-
    writeln("Case 1"), !.
switch(2):-
    writeln("Case 2"), !.
switch(3):-
    writeln("Case 3"), !.
switch(_) :-
    writeln("Invalid option"),
    fail.
```

```
?- switch(1).
Case 1
true.
?- switch(2).
Case 2
true.
```

Using Cuts - Switch

We can use cuts to switch between cases.

We add the final rule so we can print a message before failing.

```
switch(1):-
    writeln("Case 1"), !.
switch(2):-
    writeln("Case 2"), !.
switch(3):-
    writeln("Case 3"), !.
switch(_) :-
    writeln("Invalid option"),
    fail.
```

```
?- switch(1).
Case 1
true.
?- switch(2).
Case 2
true.
?- switch(4).
Invalid option
false.
```

Using Cuts - Switch

We can use cuts to switch between cases.

We can also combine cut and fail to avoid undesirable cases.

```
switch(1):-
    writeln("Case 1"), !.
switch(2):-
    writeln("This is bad"), !,
    fail.
switch(3):-
    writeln("Case 3"), !.
switch(_) :-
    writeln("Invalid option"),
    fail.
```

```
?- switch(1).
Case 1
true.
?- switch(2).
This is bad
false.
?- switch(4).
Invalid option
false.
```

Using Cuts - Defaults

We can use cuts to define defaults

If X does not buy neither a shirt or shoes, X gets nothing.

```
buys(john, shirt).
buys(mike, shoes).
gets(X, bag) :-
    buys(X, shirt), !.
gets(X, box) :-
    buys(X, shoes), !.
gets(_, nothing).
```

Using Cuts - Defaults

We can use cuts to define defaults

If `X` does not buy neither a shirt or shoes, `X` gets nothing.

```
buys(john, shirt).
buys(mike, shoes).
gets(X, bag) :-
    buys(X, shirt), !.
gets(X, box) :-
    buys(X, shoes), !.
gets(_, nothing).
```

```
?- gets(X, box).
X = mike.
?- gets(X, bag).
X = john.
?- gets(bill, X).
X = nothing.
```

Using Cuts - Defaults

We can use cuts to define defaults

However, the last clause leads to wrong conclusions.

```
buys(john, shirt).
buys(mike, shoes).
gets(X, bag) :-
    buys(X, shirt), !.
gets(X, box) :-
    buys(X, shoes), !.
gets(_, nothing).
```

```
?- gets(X, box).
X = mike.
?- gets(X, bag).
X = john.
?- gets(bill, X).
X = nothing.
?- gets(mike, nothing).
true.
```

Using Cuts - Defaults

We can use cuts to define defaults

To solve that, we create a new predicate and use negation.

```
buys(john, shirt).
buys(mike, shoes).
gets(X, Y) :-
    package(X, Y), !.
gets(_, nothing) :-
    \+ package(X, Y).
package(X, box) :-
    buys(X, shoes).
package(X, bag) :-
    buys(X, shirt).
```

Using Cuts - Defaults

We can use cuts to define defaults

To solve that, we create a new predicate and use negation.

```
buys(john, shirt).
buys(mike, shoes).
gets(X, Y) :-
    package(X, Y), !.
gets(_, nothing) :-
    \+ package(X, Y).
package(X, box) :-
    buys(X, shoes).
package(X, bag) :-
    buys(X, shirt).
```

```
?- gets(X, box).
X = mike.
?- gets(X, bag).
X = john.
?- gets(bill, X).
X = nothing.
?- gets(mike, nothing).
false.
```

More About Cuts

The position of the cut matters

The cut ensures a single solution to the choices of bread and filling.

```
bread(white). bread(brown).
filling(melt). filling(bmt).
salad(tomato). salad(lettuce).
picks(john, bread(brown)).
picks(john, bread(white)).
picks(john, filling(melt)).
order(X, Sub) :-
    picks(X, bread(B)),
    picks(X, filling(F)), !,
    Sub = (B, F).
```

The position of the cut matters

Even if `john` tries to change his bread, he only gets his first choice.

```
bread(white). bread(brown).
filling(melt). filling(bmt).
salad(tomato). salad(lettuce).
picks(john, bread(brown)).
picks(john, bread(white)).
picks(john, filling(melt)).
order(X, Sub) :-
    picks(X, bread(B)),
    picks(X, filling(F)), !,
    Sub = (B, F).
```

```
?- order(john, Sub).
Sub = (brown, melt).
```

The position of the cut matters

As a side note, see how the program works for database management.

```
bread(white). bread(brown).
filling(melt). filling(bmt).
salad(tomato). salad(lettuce).
picks(john, bread(brown)).
picks(john, bread(white)).
picks(john, filling(melt)).
order(X, Sub) :-
    picks(X, bread(B)),
    picks(X, filling(F)), !,
    Sub = (B, F).
```

```
?- order(john, Sub).
Sub = (brown, melt).
?- order(X, Sub).
X = john.
Sub = (brown, melt).
```

More About Cuts

The position of the cut matters

Suppose we add the choice of salad **after the cut**.

```
% Menu omitted for brevity
picks(john, bread(brown)).
picks(john, bread(white)).
picks(john, filling(melt)).
picks(john, salad(tomato)).
picks(john, salad(lettuce)).
order(X, Sub) :-
    picks(X, bread(B)),
    picks(X, filling(F)), !,
    picks(X, salad(S)),
    Sub = (B, F, S).
```

The position of the cut matters

Now we get multiple solutions, but only varying the salad choice. Why?

```
% Menu omitted for brevity
picks(john, bread(brown)).
picks(john, bread(white)).
picks(john, filling(melt)).
picks(john, salad(tomato)).
picks(john, salad(lettuce)).
order(X, Sub) :-
    picks(X, bread(B)),
    picks(X, filling(F)), !,
    picks(X, salad(S)),
    Sub = (B, F, S).
```

```
?- order(john, Sub).
Sub = (brown, melt, tomato);
Sub = (brown, melt, lettuce).
```

More About Cuts

The position of the cut matters

All variables instantiated before the cut stay fixed.

```
% Menu omitted for brevity
picks(john, bread(brown)).
picks(john, bread(white)).
picks(john, filling(melt)).
picks(john, salad(tomato)).
picks(john, salad(lettuce)).
order(X, Sub) :-
    picks(X, bread(B)),
    picks(X, filling(F)), !,
    picks(X, salad(S)),
    Sub = (B, F, S).
```

```
?- order(john, Sub).
Sub = (brown, melt, tomato);
Sub = (brown, melt, lettuce).
```

Cuts and Negation as Failure

We can reimplement \+ using cuts

```
neg(Goal) :- call(Goal), !, fail.
neg(_).
```

We call Goal.

- If Goal succeeds, we reach the cut and then fail.
  Because of the cut, we will not explore other possible solutions, including the second predicate `neg(_)`, and Prolog will return false.
- If Goal fails, Prolog will try the second fact, `neg(_)` which always succeeds. It will then return true.

Cuts and Negation as Failure

We can reimplement \+ using cuts

```
neg(Goal) :- call(Goal), !, fail.
neg(_).
```

Examples

```
?- neg(3 is 2+1).
false.
?- neg(3 is 2+2).
true.
```

Dynamic Predicates

Dynamic Predicates

Static

Static clauses cannot be changed during execution.

Facts and rules loaded from a file are static by default.

Dynamic

Dynamic clauses can be changed during execution.

Dynamic predicates must be declared as such in the program.

Two ways of defining dynamic predicates:

```
:- dynamic my_predicate/2.
```

```
dynamic(my_predicate/2).
```

listing

Built-in predicate to check current state of the knowledge base

listing/1

Prints all the clauses concerning a single predicate.

```
?-listing(buys).
:- dynamic buys/2.
buys(john, shirt).
buys(mike, shoes).
```

listing/0

Prints all the clauses in the program.

assertz and asserta

Built-in predicates to add clauses to the knowledge base

Only works with dynamic predicates.

assertz/1

Adds a clause to the end of the program.

```
?-assertz(buys(carl, socks)).
true.
```

assertz and asserta

Built-in predicates to add clauses to the knowledge base

Only works with dynamic predicates.

assertz/1

Adds a clause to the end of the program.

```
?-assertz(buys(carl, socks)).
true.
?-listing(buys).
:- dynamic buys/2.
buys(john, shirt).
buys(mike, shoes).
buys(carl, socks).
```

assertz and asserta

Built-in predicates to add clauses to the knowledge base

Only works with dynamic predicates.

asserta/1

Adds a clause to the beginning of the program.

```
?-asserta(buys(carl, socks)).
true.
```

assertz and asserta

Built-in predicates to add clauses to the knowledge base

Only works with dynamic predicates.

asserta/1

Adds a clause to the beginning of the program.

```
?-asserta(buys(carl, socks)).
true.
?-listing(buys).
:- dynamic buys/2.
buys(carl, socks).
buys(john, shirt).
buys(mike, shoes).
```

assertz and asserta

Built-in predicates to add clauses to the knowledge base

Only works with dynamic predicates.

We can also add rules to the program.

```
?-assertz(buys(carl, X) :- buys(john, X)).
true.
```

assertz and asserta

Built-in predicates to add clauses to the knowledge base

Only works with dynamic predicates.

We can also add rules to the program.

```
?-assertz(buys(carl, X) :- buys(john, X)).
true.
?-listing(buys).
:- dynamic buys/2.
buys(john, shirt).
buys(mike, shoes).
buys(carl, A) :- buys(john, A).
```

retract and retractall

Built-in predicates to remove clauses from the knowledge base

Only works with dynamic predicates.

retract/1

Removes a specific clause from the program.

```
?-retract(buys(john, shirt)).
true.
```

retract and retractall

Built-in predicates to remove clauses from the knowledge base

Only works with dynamic predicates.

retract/1

Removes a specific clause from the program.

```
?-retract(buys(john, shirt)).
true.
?-listing(buys).
:- dynamic buys/2.
buys(mike, shoes).
```

retract and retractall

Built-in predicates to remove clauses from the knowledge base

Only works with dynamic predicates.

retract/1

Returns false if the clause is not in the program.

```
?-retract(buys(bill, ice_cream)).
false.
```

retract and retractall

Built-in predicates to remove clauses from the knowledge base

Only works with dynamic predicates.

retractall/1

Removes all clauses that unify with a given clause.

```
?-retractall(buys(_, _)).
true.
```

retract and retractall

Built-in predicates to remove clauses from the knowledge base

Only works with dynamic predicates.

retractall/1

Removes all clauses that unify with a given clause.

```
?-retractall(buys(_, _)).
true.
?-listing(buys).
:- dynamic buys/2.
```

retract and retractall

Built-in predicates to remove clauses from the knowledge base

Only works with dynamic predicates.

retract(buys(_, _)) vs **retractall(buys(_, _))**

`retract(buys(_, _))` removes only the first matching clause.

```
?-retract(buys(_, _)).
true.
?-listing(buys).
:- dynamic buys/2.
buys(mike, shoes).
```

abolish

Built-in predicates to remove clauses from the knowledge base

Works with both dynamic and static predicates in SWI-Prolog.

abolish/1

Removes all clauses of a given predicate.

```
?-abolish(buys/2).
true.
```

abolish

Built-in predicates to remove clauses from the knowledge base

Works with both dynamic and static predicates in SWI-Prolog.

abolish/1

Removes all clauses of a given predicate.

```
?-abolish(buys/2).
true.
?-listing(buys).
ERROR: procedure ‘buys’ does not exist.
```

abolish

Built-in predicates to remove clauses from the knowledge base

Works with both dynamic and static predicates in SWI-Prolog.

abolish/1

Removes all clauses of a given predicate.

```
?-abolish(buys/2).
true.
```

Use with care. It is preferable to use retractall/1.

Calculating Fibonacci numbers

$$F(0) = 0 \quad F(1) = 1 \quad F(n) = F(n-1) + F(n-2)$$

Calculating Fibonacci numbers

We start with the base cases.

```
fibo(0, 0). % base cases
fibo(1, 1).
```

Calculating Fibonacci numbers

To compute $F(n)$, we just need to compute $F(n-1)$ and $F(n-2)$.

```
fibo(0, 0). % base cases
fibo(1, 1).
fibo(N, F) :-
    N >= 2,
    N1 is N - 1, fibo(N1, F1),
    N2 is N - 2, fibo(N2, F2),
    F is F1 + F2.
```

Calculating Fibonacci numbers

To compute $F(n)$, we just need to compute $F(n-1)$ and $F(n-2)$.

```
fibo(0, 0). % base cases
fibo(1, 1).
fibo(N, F) :-
    N >= 2,
    N1 is N - 1, fibo(N1, F1),
    N2 is N - 2, fibo(N2, F2),
    F is F1 + F2.
```

```
?- fibo(3, F).
F = 2.
?- fibo(10, F).
F = 55.
```

Using Dynamic Predicates

Calculating Fibonacci numbers

How can we use dynamic predicates to make it more efficient?

```
fibo(0, 0). % base cases
fibo(1, 1).
fibo(N, F) :-
    N >= 2,
    N1 is N - 1, fibo(N1, F1),
    N2 is N - 2, fibo(N2, F2),
    F is F1 + F2.
```

Using Dynamic Predicates

Calculating Fibonacci numbers

We can use dynamic predicates to store solutions.

```
:- dynamic fibo/2.
fibo(0, 0). % base cases
fibo(1, 1).
fibo(N, F) :-
    N >= 2,
    N1 is N - 1, fibo(N1, F1),
    N2 is N - 2, fibo(N2, F2),
    F is F1 + F2,
    asserta(fibo(N, F):-!).
```

```
?- fibo(3, F).
F = 2.
?- fibo(10, F).
F = 55.
```

Calculating Fibonacci numbers

Without the cut we would keep adding clauses to the program.

```
:- dynamic fibo/2.
fibo(0, 0). % base cases
fibo(1, 1).
fibo(N, F) :-
    N >= 2,
    N1 is N - 1, fibo(N1, F1),
    N2 is N - 2, fibo(N2, F2),
    F is F1 + F2,
    asserta(fibo(N, F):-!).
```

```
?- fibo(3, F).
F = 2.
?- fibo(10, F).
F = 55.
```

Using Dynamic Predicates

Calculating Fibonacci numbers

As an exercise try to write the same program with a single recursive call.

```
:- dynamic fibo/2.
fibo(0, 0). % base cases
fibo(1, 1).
fibo(N, F) :-
    N >= 2,
    N1 is N - 1, fibo(N1, F1),
    N2 is N - 2, fibo(N2, F2),
    F is F1 + F2,
    asserta(fibo(N, F):-!).
```

```
?- fibo(3, F).
F = 2.
?- fibo(10, F).
F = 55.
```

Using Dynamic Predicates

Ordering Coffee

We want a program to maintain a database of coffee orders.

Using Dynamic Predicates

Ordering Coffee

We create a dynamic predicate `coffee/3` that stores the orders.

```
:- dynamic coffee/3.
```

Using Dynamic Predicates

Ordering Coffee

`makeCoffee/3` places an order for us.

```
:- dynamic coffee/3.

makeCoffee(Type, Sugar, Size) :-
    \+ coffee(Type, Sugar, Size),
    writeln("Preparing your coffee."),
    assertz(coffee(Type, Sugar, Size)).
```

Using Dynamic Predicates

Ordering Coffee

We add another rule in case the order has already been placed.

```
:- dynamic coffee/3.

makeCoffee(Type, Sugar, Size) :-
    \+ coffee(Type, Sugar, Size),
    writeln("Preparing your coffee."),
    assertz(coffee(Type, Sugar, Size)).

makeCoffee(Type, Sugar, Size) :-
    coffee(Type, Sugar, Size),
    writeln("You have already ordered.").
```

Using Dynamic Predicates

Ordering Coffee

We add a cut to the first rule. Why is it a good idea?

```
:- dynamic coffee/3.

makeCoffee(Type, Sugar, Size) :-
    \+ coffee(Type, Sugar, Size),
    writeln("Preparing your coffee."),
    assertz(coffee(Type, Sugar, Size)), !.

makeCoffee(Type, Sugar, Size) :-
    coffee(Type, Sugar, Size),
    writeln("You have already ordered.").
```

Ordering Coffee

We can run the program and check if an order is properly placed.

```
?- makeCoffee(filter, no, large).
Preparing your coffee.
true.
```

Using Dynamic Predicates

Ordering Coffee

We can run the program and check if an order is properly placed.

```
?- makeCoffee(filter, no, large).
Preparing your coffee.
true.
?- listing(coffee).
:- dynamic coffee/3.
coffee(filter, no, large).
true.
```

Using Dynamic Predicates

Ordering Coffee

We can also add a predicate to take a coffee.

```
takeCoffee(Type, Sugar, Size) :-
    coffee(Type, Sugar, Size),
    writeln("Here you go!"),
    retract(coffee(Type, Sugar, Size)), !.

takeCoffee(Type, Sugar, Size) :-
    coffee(Type, Sugar, Size),
    writeln("I am sorry, your coffee is not ready yet."),
    makeCoffee(Type, Sugar, Size).
```

Ordering Coffee

We can now remove a coffee from the database when someone takes it.

```
?- listing(coffee).
:- dynamic coffee/3.
coffee(filter, no, large).
true.
?- takeCoffee(filter, no, large).
Here you go!
true.
?- listing(coffee).
:- dynamic coffee/3.
true.
```