# Prolog Cheat Sheet

## Basics

*Everything is a relation!* —I.e., a table in a database!

Whence programs are unidirectional and can be 'run in reverse': Input arguments and output arguments are the same thing! Only perspective shifts matter.

For example, defining a relation `append(XS, YS, ZS)` *intended* to be true precisely when `ZS` is the catenation of `XS` with `YS`, gives us three other methods besides being a predicate itself! List construction: `append([1, 2], [3, 4], ZS)` ensures `ZS` is the catenation list. List subtraction: `append([1,2], YS, [1, 2, 3, 4])` yields all solutions `YS` to the problem `[1, 2] ++ YS = [1, 2, 3, 4]`. Partitions: `append(XS, YS, [1, 2, 3, 4])` yields all pairs of lists that catenate to `[1,2, 3, 4]`. Four methods for the price of one!

Prolog is PROgramming in LOGic.

- Prolog is declarative: A program is a collection of 'axioms' from which 'theorems' can be proven. For example, consider how sorting is performed:
  - Procedurally: Find the minimum in the remainder of the list, swap it with the head of the list; repeat on the tail of the list.
  - Declaratively: `B` is the sorting of `A` *provided* it is a permutation of `A` and it is ordered.

  Whence, a program is a theory and computation is deduction!

## Unification

**Unification** Can the given terms be made to represent the same structure?

- This is how type inference is made to work in all (?) languages.

**Backtracking** When a choice in unification causes it to fail, go back to the most recent choice point and select the next avialable choice.

Nullary built-in predicate `fail` always fails as a goal and causes backtracking.

Unification:

1. A constant unified only with itself.
2. A variable unifies with anything.
3. Two structures, terms, unify precisely when they have the same head and the same number of arguments, and the corresponding arguments unify recursively.

Unification performs no simplification, whence no arithmetic. This means, for example, we can form pairs by sticking an infix operator between two items; moreover we can form distinct kinds of pairs by using different operators:

```
?- C + "nice" = woah + Z.
C = woah,
Z = "nice".

% '+' and '/' are different, so no way to make these equal.
?- C + "nice" = woah / Z.
false.
```

## Facts & Relations

We declare relations by having them begin with a lowercase letter; variables are distinguished by starting with a capital letter.

```
jasim_is_nice.

% ?- jasim_is_nice. % ⇒ true: We declared it so.

it_is_raining. /* Another fact of our world */

% ?- it_is_raining. % ⇒ true

eats(fred, mangoes).
eats(bob, apples).
eats(fred, oranges).

% ?- eats(bob, apples). % ⇒ true

% Which foods are eaten by fred?
% ?- eats(fred, what). % ⇒ false; 'what' is name!
% ?- eats(fred, What). % ⇒ mangoes oranges
```

Here's a cute one:

```
% All men are mortal.
mortal(X) :- man(X).

% Socrates is a man.
man(socrates).

% Hence, he's expected to be mortal.
% ?- mortal(socrates). % ⇒ true

% What about Plato?
% ?- mortal(plato). % ⇒ false, plato's not a man.

% Let's fix that.
man(plato).

% Who is mortal?
% ?- mortal(X). % ⇒ socrates plato
```

## Hidden Quantifiers

```
head(X) :- body(X,Y).
% Semantics: ∀ X. head(X) ⇐ ∃ Y. body(X,Y).
```

Queries are treated as headless clauses.

```
?- Q(X)
% Semantics: ∃ X. Q(X).
```

## Conjunction

- Conjunction: `p(X), q(X)` means "let `X` be *a* solution to `p`, then use it in query `q`."
- Operational semantics: Let `X` be the first solution declared, found, for `p`, then try `q`; if it fails, then *backtrack* and pick the next declared solution to `p`, if any, and repeat until `q` succeeds.
- For example, `p(X), print(X), fail.` gets a solution to `p`, prints it, then fails thereby necessitating a backtrack to obtain a different solution `X` for `p`, then repeats. In essence, this is prints all solutions to `p` —a so-called "fail driven loop".

For example,

```
yum(pie).
yum(apples).
yum(maths).

% ?- yum(Y), writeln(Y), fail. %⇒ pie apples maths false.
```

## Disjunction

Since a Prolog program is the conjunction of all its clauses:

```
% (head ⇐ body₁) ∧ (head ⇐ body₂)
head :- body₁.
head :- body₂.

≈

% head ⇐ body₁ ∨ body₂
head :- body₁ ; body₂.
```

## Arithmetic with `is`

- Unification only tries to make both sides of an equality true by binding free variables to expressions. It does not do any arithmetic.
- Use `is` to perform arithmetic with `+`, `-`, `*`, `/`, `mod`.

```
% ?- X = 3 + 2.  %% X = 3 + 2

% ?- X is 3 + 2. %% X = 5

% ?- X is 6 / 3. %⇒ X = 2.

% ?- X is  5 / 3. %⇒ X = 1.6666666666666667.

% ?- X is 5 // 3. %⇒ X = 1.

% ?- X is 5 mod 3. %⇒ X = 2.
```

- Comparison operators: `=`, `\=`, `<`, `>`, `=<`, and `>=`.

Atoms, or nullary predicates, are represented as a lists of numbers; ASCII codes.

```
% ?- name(woah_hello, X). %⇒ X = [119,111,97,104,95,104,101,108,108,111]
% ?- name(woah, X).       %⇒ X = [119,111,97,104]
```

Exercise: We can use this to compare two atoms lexicocraphically.
Incidentally, we can obtain the characters in an atom by using the built-in `atom_chars`.

```
% ?- atom_chars(nice, X). %⇒ X = [n, i, c, e].
```

## Declaration Ordering Matters

When forming a recursive relation, ensure the base case, the terminating portion, is declared before any portions that require recursion. Otherwise the program may loop forever.

Unification is performed using depth-first search using the order of the declared relationships. For example, the following works:

```
% Graph
edge(a, b). edge(b ,c). edge(c, d).

% Works
path(X, X).
path(X, Y) :- edge(Z, Y), path(X, Z).
% ?- path(a, d). %⇒ true.

% Fails: To find a path, we have to find a path, before an edge!
% The recursive clause is first and so considerd before the base clause!
path_(X, Y) :- path_(X, Z), edge(Z, Y).
path_(X, X).
% ?- path_(a, d). %⇒ loops forever!
```

## ADT: Pairs, Numbers, Lists, and Trees

- Uniform treatment of all datatypes as predicates!

```
% In Haskell: Person = Me | You | Them
person(me).
person(you).
person(them).

% In Haskell: Pair a b = MkPair a b

pair(_, _).

% ?- pair(1, "nice").
% ?- pair(1, "nice") = pair(A, "nice"). %⇒ A = 1

% In Haskell: Nat = Zero | Succ Nat

nat(zero).
nat(succ(N)) :- nat(N).

% ?- nat(succ(succ(zero))).
```

```prolog
sum(zero, N, N).
sum(succ(M), N, succ(S)) :- sum(M, N, S).

% ?- Two = succ(succ(zero)), Four = succ(succ(succ(succ(zero)))), sum(Two, Two, Four).

% In Haskell: Tree a = Leaf a | Branch (Tree a) (Tree a)

tree(leaf(_)).
tree(branch(L, R)) :- tree(L), tree(R).

% ?- A = leaf(1), B = leaf(2), L = branch(A, B), R = branch(A, A), tree(branch(L, R)).
```

Programming via specification: Lisp lists, for example, are defined by the following equations.

```prolog
% Head: (car (cons X Xs)) = X
% Tail: (cdr (cons X Xs)) = Xs
% Extensionality: (cons (car Xs) (cdr Xs)) = Xs, for non-null Xs.

% We can just write the spec up to produce the datatype!
% We simply transform /functions/ car and cdr into relations;
% leaving the constructor, cons, alone.

% What are lists?
list(nil).
list(cons(_, Xs)) :- list(Xs).

null(nil).

car(cons(X, Xs), X) :- list(Xs).
cdr(cons(_, Xs), Xs) :- list(Xs).

% ?- true.
% - list(Ys), not(null(L)), list(cons(car(Ys, Y), cdr(Ys, L))). % loops.

% ?- [1] = [1|[]].
```

## Built-in Lists

Lists are enclosed in brackets, separated by commas, and constructed out of cons "|".

```prolog
% ?- ["one", two, 3] = [Head|Tail]. % ⇒ Head = "one", Tail = [two, 3].
% ?- ["one", two, 3] = [_,Second|_]. % ⇒ Second = two.
% ?- [[the, Y], Z] = [[X, hare], [is, here]]. % ⇒ X = the, Y = hare, Z = [is, here]]

% Searching: x ∈ l?
elem(Item, [Item|Tail]). % Yes, it's at the front.
elem(Item, [_|Tail]) :- elem(Item, Tail). % Yes, it's in the tail.

% ?- elem(one, [this, "is", one, thing]). % ⇒ true
% ?- elem(onE, [this, "is", one, thing]). % ⇒ false
```

In Haskell, we may write `x:xs`, but trying that here forces us to write `[X|XS]` or `[X|Xs]` and accidentally mismatching the capitalisation of the 's' does not cause a compile-time error but will yield an unexpected logical error –e.g., in the recursive clause use `Taill` instead of `Tail`. As such, prefer the `[Head|Tail]` or `[H|T]` naming.
See here for the list library, which includes:

```prolog
member(element, list)
append(list1, list2, lists12)
prefix(part, whole)
nth0(index, list, element)
last(list, element)
length(list, number)
reverse(list1, list2)
permutation(list1, list2)
sum_list(list, number)
max_list(list, number)
is_set(list_maybe_no_duplicates)
```

Exercise: Implement these functions. Hint: Arithmetic must be performed using `is`.

## The Cut

- Ensure deterministic behaviour: Discard choice points of ancestor frames.
  - Once a goal has been satisfied, don't try anymore. —Efficient: We wont bother going through all possibilities, the first solution found is sufficient for our needs.
  - When a cut, "`!`", is encountered, the system is committed to all choices made since the parent goal was invoked. All other alternatives are discarded.
- `p(X, a), !` only produces one answer to `X`: Do not search for additional solutions once *a* solution has been found to `p`.
  E.g., only one `X` solves the problem and trying to find another leads to infinite search —"green cut"— or unintended candidate results —"red cut".

Example `a`: The first solution to `b` is 1, and when the cut is encountered, no other solutions for `b` are even considered. After a solution for `Y` is found, backtracking occurs to find other solutions for `Y`.

```prolog
a(X,Y) :- b(X), !, c(Y).
b(1). b(2). b(3).
c(1). c(2). c(3).

% ?- a(X, Y). % ⇒ X = 1 ∧ Y = 1, X = 1 ∧ Y = 2, X = 1 ∧ Y = 3
```

Below the first solution found for `e` is 1, this is not a solution for `f`, but backtracking cannot assign other values to `X` since `X`'s value was determined already as 1 and this is the only allowed value due to the cut. But `f(1)` is not true and so `d` has no solutions. In contrast, `d_no_cut` is just the intersection.

```prolog
d(X) :- e(X), !, f(X).
e(1). e(2). e(3). f(2).

% ?- not(d(X)). % ⇒ "no solution" since only e(1) considered.
% ?- d(2). % ⇒ true, since no searching performed and 2 ∈ e ∩ f.

d_no_cut(X) :- e(X), f(X).
% ?- d_no_cut(X). % ⇒ X = 2.
```

The cut not only commits to the instantiations so far, but also commits to the clause of the goal in which it occurs, whence no other clauses are even tried!

```
g(X) :- h(X), !, i(X).
g(X) :- j(X).

h(1). h(4). i(3). j(4).

% ?- g(X). %⇒ fails
% ?- f(
```

There are two clauses to prove `g`, by default we pick the first one. Now we have the subgoal `h`, for which there are two clauses and we select the first by default to obtain `X = 1`. We now encounter the cut which means we have committed to the current value of `X` and the current clause to prove `g`. The final subgoal is `i(1)` which is false. Backtracking does not allow us to select different goals, and it does not allow us to use the second clause to prove `g`. Whence, `g(X)` fails. Likewise we fail for `g(4)`. Note that if we had failed `b` before the cut, say `b` had no solutions, then we fail that clause before encountering the cut and so the second rule is tried.

Common use: When disjoint clauses cannot be enforced by pattern matching.

```
sum_to(0, 0).
sum_to(N, Res) :- M is N - 1, sum_to(M, ResM), Res is ResM + N.

% Example execution
  sum_to(1, X)
 ⇒ M is 0     --only clause 2 applies

Now both clauses apply.

Clause1:  ⇒  ResM = 0, Res = 0.
Clause2:  ⇒  M is -1, sum_to(M, ResM), ···
          ⇒  Clause 2 applies here, with M = -2.
          ⇒  Loop forever.
```

After we commit to the first clause, *cut* out all other alternative clauses:

```
sum_to(0, 0) :- !.
sum_to(N, Res) :- M is N - 1, sum_to(M, ResM), Res is ResM + N.

% ?- sum_to(1, X).
```

It may be clearer to replace cuts with negations so as to enforce disjoint clauses.

```
sum_to_not(0, 0).
sum_to_not(N, Res) :- N \= 0, M is N - 1, sum_to(M, ResM), Res is ResM + N.

% ?- sum_to_not(5, X). %⇒ X = 15.
```

In general, `not(G)` succeeds when *goal* `G` fails.

## Using Modules

The Constraint Logic Programming over Finite Domains library provides a number of useful functions, such as `all_distinct` for checking a list has unique elements.

```
use_module(library(clpfd)).

% ?- all_distinct([1,"two", two]).
```

See here for a terse solution to Sudoku.

## Higher-order

- Prolog is limited to first-order logic: We cannot bind variables to relations.
- Prolog *indirectly* supports higher-order rules.

```
colour(bike, red).
colour(chair, blue).

% Crashes!
% is_red(C, X, Y) :- C(X, Y)

% Works
is_red(C, X, Y) :- call(C, X, Y).

% ?- is_red(colour, bike, X). %⇒ X = red.
```

Translate between an invocation and a list representation by using 'equiv' `=..` as follows:

```
% ?- p(a, b, c) =.. Y. %⇒ Y = [p, a, b, c].
% ?- Y =.. [p, a, b, c]. %⇒ Y = p(a, b, c).
```

## `Print, var, nonvar, arg`

`Print` predicate always succeeds, never binds any variables, and prints out its parameter as a side effect.

Use built-ins `var` and `nonvar` to check if a variable is free or bound.

```
% ?- var(Y).  %⇒ true
% ?- Y = 2, var(Y). %⇒ false
% ?- Y = 2, nonvar(Y). %⇒ true
```

Built-in `arg(N,T,A)` succeeds if `A` is the `N`-th argument of the term `T`.

```
% ?- arg(2, foo(x, y), y). %⇒ true
```

## Meta-Programming

- ⋄ Programs as data.
- ⋄ Manipulating Prolog programs with other Prolog programs.

`clause(X, Y)` succeeds when `X` is the signature of a relation in the knowledge base, and `Y` is the body of one of its clauses. `X` must be provided.

```
test(you, me, us).
test(A, B, C) :- [A, B, C] = [the, second, clause].

% ?- clause(test(Arg1, Arg2, Arg3), Body).
% ⇒ ‘Body’ as well as ‘Argi’ are unified for each clause of ‘test’.
```

Here is a Prolog interpreter in Prolog —an approximation to `call`.

```
% interpret(G) succeeds as a goal exactly when G succeeds as a goal.

% Goals is already true.
interpret(true) :- !.

% A pair of goals.
interpret((G, H)) :- !, interpret(G), interpret(H).

% Simple goals: Find a clause whose head matches the goal and interpret its subgoals.
interpret(Goal) :- clause(Goal,Subgoals), interpret(Subgoals).

% ?- interpret(test(A, B, C)).
```

## Reads

- ☒ Introduction to logic programming with Prolog —12 minute read.
- ☒ Introduction to Prolog —with interactive quizzes
- ☐ Derek Banas’ Prolog Tutorial —1 hour video
- ☒ A Practo-Theoretical Introduction to Logic Programming —a **colourful** read showing Prolog ≅ SQL.
- ☐ Prolog Wikibook —slow-paced and cute
- ☐ James Power’s Prolog Tutorials
- ☒ Introduction to Logic Programming Course —Nice slides
- ☐ Stackoverflow Prolog Questions —nifty FAQ stuff
- ☐ 99 Prolog Problems —with solutions
- ☐ The Power of Prolog –up to date tutorial, uses libraries ;-)
- ☐ Backtracking
- ☐ Escape from Zurg: An Exercise in Logic Programming
- ☐ Efficient Prolog –Practical tips
- ☐ Use of Prolog for developing a new programming language —Erlang!
- ☐ prolog :- tutorial —Example oriented
- ☐ Learn Prolog Now! —thorough, from basics to advanced
- ☐ Real World Programming in SWI-Prolog