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 `plus(X, Y, Sum)` *intended* to be true precisely when `Sum` ≈ `X + Y` gives us two other methods! Subtract: `plus(4, Y, 8)` yields all solutions `Y` to the problem `8 = 4 + Y`. Partitions: `plus(X, Y, 8)` yields all pairs `X, Y` that sum to 8.
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.

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 —Nullary 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.

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

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!
```

Cuts

- Ensure deterministic behaviour: Discard choice points of a ancestor frames.
- `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".

ADT: Pairs, Numbers, Lists, and Trees

- Uniform treatment of all datatypes as predicates!

```
% 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))).

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, Fou

% 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.

```
% 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 "|".

```
% ?- ["one", two, 3] = [Head|Tail].
```

See here for the list library, which includes:

```
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)
```

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.
```

`Print, var, nonvar`

`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
```

Reads

Organised in terms of length:

- 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 notes and more!
- Stackoverflow Prolog Questions —nifty FAQ stuff
- 99 Prolog Problems —with solutions
- Backtracking
- Escape from Zurg: An Exercise in Logic Programming
- 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