Prolog CheatSheet

Administrivia

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.

In Prolog, the task of the programmer is simply to *describe* problems —write down, logically, the situation— rather than telling the computer what to do, then obtains information by asking questions —the logic programming system *figures out how* to get the answer.

- 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!
- `swipl -s myprogram.pl` –Load your program into a REPL, `?-....`
- `make.` –Reload your program.
- `halt.` –Exit the REPL.
- `consult('CheatSheet.pl').` –Load the contents of the given file as the new knowledge base.
- `assert((⋯)).` –Add a new rule to the knowledge base, from within the REPL. Use `retract((⋯))` to remove rules from the knowledge base.
  - `assert` is useful when we want to cache computations.
- `listing.` –Display the contents of the current knowledge base; i.e., what Prolog 'knows'.
- `listing(name).` –List all information in the knowledge base about the `name` predicate.

Syntax

There are three types of terms:

- Constants: Numbers such as -24, and atoms such as `jasim`, `'hello world'`, `'&~%&#@$ &*'`, and `' '` —a space in quotes.
- Variables: Words starting with a capital letter or an underscore.
  - The variable `_` is called the *anonymous variable*.

    It's for when we need a variable, say when pattern matching, but don't care about the value.
- Structures: Terms of the form $\texttt{functor(term}_1\texttt{,...,term}_n\texttt{)}$.

The characters between single quotes are the *name* of an atom and so Prolog admits `symbol = 'symbol'` as true.

- Atoms, or nullary predicates, are represented as a lists of numbers; ASCII codes.
- We can use this to compare two atoms lexicographically.
- We can obtain the characters in an atom by using the built-in `atom_chars`.

```
?- name(woah, X).       % ⇒  X = [119,111,97,104]
?- atom_chars(nice, X). % ⇒  X = [n, i, c, e].
```

Facts & Relations

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

```
/* Some facts of our world */
jasim_is_nice.
it_is_raining.

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

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

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

Relational constraints are formed using `:-`, which acts as the "provided", $\Leftarrow$, operator from logic. Read `P :- Q` as *P is true, provided Q is true.*

```
% 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 ... in the REPL!
?- assert((man(plato))).

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

Mixfix Syntax

It may feel awkward to write `father_of(homer, bart)` and instead prefer `homer father_of bart`. We may declare relations to be prefix, infix, or postfix with patterns `xf`, `xfx`, and `fx` respectively. For left associativity we use pattern `yfx` and use `xfy` for right associativity.

```
:- op(35,xfx,father_of).

father_of(me, you).
homer father_of bart.
homer father_of lisa.
```

- ◇ Precedence, or binding power, is lowest at 1200 and highest at 0.
- ◇ Note: `father_of(X,Y) = X father_of Y` is true.

We may learn about existing operators too;
e.g., `?- current_op(Prec, Fixity, =:=)` ^_^

Trace & Backtracking

We can see what Prolog does at each step of a computation by invoking `trace`; we turn off this feature with `notrace.`

This' an excellent way to learn how Prolog proof search works! (Debugging!)

Suppose we have the following database.

```
q(1). q(2). q(3).
r(2). r(3).
p(X) :- q(X), r(X).
```

With trace, query `p(X)` and press *SPACE* each time to see what Prolog is doing. At one point, the goal `r(1)` will *fail* and that choice $X = 1$ will be redone with the next possibility for `q`, namely $X = 2$.

The line marked `redo` is when Prolog realizes its taken the wrong path, and backtracks to instantiate the variable to 2.

Operationally, query `p(X)` is answered by:

1. Find match for the first goal: `q` at `1`.
2. Then see if matches the second: `r` at `1`.
3. (Redo) If not, find another match for the first: `q` at `2`.
4. See if this matches the second, `r`.
5. Etc.

- ◇ `findall(X, Goal, L)` succeeds if `L` is the list of all those `X`'s for which `Goal` holds.
- ◇ `fail/0` immediately fails when encountered. Remember: Prolog tries to backtrack when its fails; whence `fail` can be viewed as an instruction to force backtracking.

  The opposite of forcing backtracking is to block it, which is done with 'cut' `!` —see below.

What is a Prolog Program Exactly?

A program *denotes* all true facts derivable from its clauses using **modus ponens, unification, term rewriting, and logical or-&-and** for the execution model.

Hidden Quantifiers:

Syntax	Semantics
`head(X) :- body(X,Y).`	$\forall X.\, head(X) \Leftarrow \exists Y.\, body(X,Y)$
`?- Q(X)`	$\exists X.\, Q(X)$

1. " `head(X)` is true provided there's some `Y` such that `body(X,Y)` is true "
   - ◇ `head.` is an abbreviation for `head :- true.`
   - ◇ Indeed, $p \equiv (p \Leftarrow true)$.
2. " Is there an `X` so that `Q(X)` is true? "

"One-Point Rule": Provided `X` is a fresh variable,

`f(···X···) :- X = `$\mathcal{E}xpr$`.` $\quad\approx\quad$ `f(···`$\mathcal{E}xpr$`···).`

Overloading! Predicates of different arities are considered different.

Documentation Convention: `f/N` $\approx$ *relation* `f` *takes* `N`*-many arguments*

Modus Ponens — Computation ≈ Deduction

The logical rule $p \wedge (p \Rightarrow q) \Rightarrow q$ says if we have p, and from that we know we can get a q, then we have a q. From the following program on the left, we get `q(a)` is true.

```
p(a).
q(X) :- p(X).
```

We *rewrite* term `X` with atom `a` to obtain `q(a) :- p(a)` from the second rule, but we know `p(a)`, and so we have *computed* the new fact `q(a)` by using the deduction rule modus ponens.

Conjunction ≈ Constraints — Disjunction ≈ Alternatives

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`, in the user's script, then try `q`; if it fails, then *backtrack* and pick the next declared solution to `p`, if any, and repeat until `q` succeeds —if possible, otherwise fail.

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

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

"Fail driven loop" `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`.

"Let Clauses": Provided `X` is a fresh variable,

$\cdots\mathcal{E}xpr\cdots\mathcal{E}xpr\cdots \quad\approx\quad$ `X = `$\mathcal{E}xpr$`, ···X···X···`

A Prolog program is the conjunction of all its clauses, alternatives ';'.

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

Read ' ⇐ ' as '≥', and '∨' as maximum, then the following is the "characterisation of least upper bounds".

$$(p \Leftarrow q) \wedge (p \Leftarrow r) \equiv p \Leftarrow (q \vee p)$$

"And binds stronger than Or": `a,b;c` ≈ `(a,b);c`.

Unification

A program can be written by having nested patterns, terms, then we use matching to pull out the information we want!

Two terms *match* or *unify*, if they are equal or if they contain variables that can be instantiated in such a way that the resulting terms are equal.

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 available choice.

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

The unification predicate is `=/2`. It can be written with the usual notation `=(L, R)` but can also be written infix `L = R`.

```
% Query: Who is loved by Jay?
?- loves(jay, X) = loves(jay, kathy).
% ⇒ X = kathy
```

Operationally $\mathcal{L}$ `=` $\mathcal{R}$ behaves as follows:

1. If either is an unbound variable, assign it to the other one.
   - ◇ A constant unifies only with itself.
   - ◇ A variable unifies with anything.
2. Otherwise, they are both terms.
   - ◇ Suppose $\mathcal{L} \approx f(e_1, \ldots, e_n)$ and $\mathcal{R} \approx g(d_1, \ldots, d_m)$.
   - ◇ If `f` is different from `g`, or `n` different from `m`, then crash.
   - ◇ Recursively perform e_i `=` d_i.

   Ensure the variable instantiations are compatible in that a variable is associated with at most one value —which is not true in `f(1,2) = f(X,X)`.

   Thus variables are single 'assignment'!

   Exception! Each occurrence of the anonymous variable `_` is independent: Each is bound to something different.
3. If two terms can't be shown to match using the above clauses, then they don't match.

Unification lets us solve equations! It lets us **compute!**

The query `'symbol' = symbol` is true since both are considered to be the same atom. Whereas `'2' = 2` is false since `'2'` is a symbolic atom but `2` is a number.

The *discrepancy predicate* `\=/2` succeeds when its arguments don't unify; e.g., `'5' \= 5` is true.

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.
```

Algebraic Datatypes

Uniform treatment of all datatypes as predicates! Enumerations, pairs, recursives:

Haskell

```
data Person = Me | You | Them


data Pair a b = MkPair a b

data Nat = Zero | Succ Nat


sum Zero     n = n
sum (Succ m) n = Succ (sum m n)
```

Prolog

```
person(me).
person(you).
person(them).

pair(_, _).

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

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

Exercise: Form binary trees.

Arithmetic with `is` —Using Modules

Use `is` to perform arithmetic with `+, -, *, /, **, mod`, and `//` for integer division.

```
% How do we make this equation equal?
?- X = 3 + 2.
% ⇒ X = 3 + 2; this choice of variables make its equal!

% Everything is a term! Terms don't 'compute'!
?- +(3, 2) = 3 + 2. % ⇒ true
?- +(3, 2) = 6 - 1. % ⇒ false

?- X is 3 + 2. % ⇒ X = 5
?- 5 is 6 - 1. % ⇒ true
?- 5 is X. % ⇒ CRASH!
?- 3 + 2 is 6 - 1. % ⇒ CRASH!

?- +(3, 2) =:= 6 - 1. % ⇒ true
?- 1 =:= sin(pi/2).   % ⇒ true
?- X =:= 3 + 2.       % ⇒ CRASH!
?- X = 2, Y = 3, X + Y =:= 5. % ⇒ true
```

- ◇ `is` takes a *variable, or a numeric constant,* and an arithmetical expression as

arguments.

- `L is R` means " unify `L` with the result of simplifying `R` "
- If `R` mentions an unbound variable, crash!

◇ `=:=` has both arguments as *concrete terms*, it evaluates them and compares the results.

$$l \texttt{ =:= } r \quad \approx \quad \texttt{L is } l\texttt{, R is } r\texttt{, L = R.}$$

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.
See here for a terse solution to Sudoku.
In particular, `=:=` is too low level —e.g., it doesn't admit unbound variables— instead one uses `clpfd`'s `#=/2` predicate.
Likewise we could use `in` to check if a number is in a particular range, or instead use `#>/2` and `#</2`.

```
use_module(library(clpfd)).

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

?- X + 2 #= 3. %⇒  X = 1
?- 1 + Y #= 3. %⇒  Y = 2.
?- X #= Y.     %⇒  Also works ;-)

?- 2 in 0..3.       %⇒  true.
?- 3 #> X, 0 #< X.  %⇒  X in 1..2.

% All parititions of number N ^_^
?- N = 5, between(0, N, X),
   between(0, N, Y), X + Y #= N.
```

Lists

Lists are enclosed in brackets, separated by commas, and can be split up at any point by using cons "|". The empty list is `[]`.

```
?- ["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: $x \in l$?

```
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
```

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)
```

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.

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

Declaration Ordering Matters —Recursion

Prolog searches the knowledge base from top to bottom, clauses from left to right, and uses backtracking to recover from bad choices.

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:

```
% Acyclic graph: a ⟶ b ⟶ c ⟶ d
edge(a, b). edge(b ,c). edge(c, d).

% Works
path(X, X).
path(X, Y) :- edge(Z, Y)  % Can we get to Y from some intermediary Z?
            , path(X, Z). % Can we get to the intermediary Z from X?
% ?- 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!
```

The Cut

Automatic backtracking is great, but can be a waste of time exploring possibilities that lead nowhere. The atom *cut*, `!`, offers a way to control how Prolog looks for solutions: It always succeeds with a side-effect of committing to any choices made thus far —including variable instantiations **and** rule, clause, chosen— whence ignoring any other possible branches and no backtracking!

`q :- p1, ..., pn, !, r1, ..., rm` ⇒ Once we reach the cut, we're commited to the choices made when evaluating the $\texttt{p}_i$, but we are free to backtrack among the $\texttt{r}_i$ **and** we may backtrack among the alternatives for choices that were made before reaching goal `q`. Here's an example.

```
i(1). i(2).
j(1). j(2). j(3).
```

```
k(X, Y) :- i(X), !, j(Y).
```

```
l(X,Y) :- k(X,Y).
l(0,0).
```

Query `l(X, Y)` yields solutions 1-1, 1-2, 1-3, and 0-0. Notice that `X = 0, Y = 0` is not truthified by by the first clause of `l` but the choice of clause happened before the `k`-clause containing the cut `!` and so backtracking may pick another `l`-clause. Notice that without the cut, we have the extra solutions 2-1, 2-2, 2-3 which are "cut out" by `!` since `i(1)` is the choice we committed to for `X = 1` and we can backtrack for `Y` only since it comes after the cut.

Suppose $\mathtt{x}_1$ is the first solution found for `p`, then:

$$\begin{array}{lcl} \texttt{p(X), q(Y)} & \approx & \{(x,y) \mid p\,x \wedge q\,y\} \\ \texttt{p(X), !, q(Y)} & \approx & \{(x_1,y) \mid q\,y\} \end{array}$$

Remember, 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(2).

% ?- g(X). % ⇒ fails
```

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 `h` before the cut, as is the case with `g(2)`, then we fail that clause before encountering the cut and so the second rule is tried.

When there are disjoint clauses, i.e., only one succeeds, then if backtracking is forced at some point, trying other cases is a waste of time since only one clause, say the first one, succeeds. An example of this would be the maximum function or the $\sum_{i=0}^{n} i$ function.

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

% ?- trace.
% ?- max_(3, 4, Y).
% ⇒ Wastes time trying both clauses

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

% ?- trace.
% ?- max(3, 4, Y).
% ⇒ Only first clause is tried ^_^
```

```
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).
% ⇒ Loops forever: Both clauses apply!

% The fix is to mark the
% first clause as a "base case".
sum_to(0, 0) :- !.
```

The resulting code gives the *same* results but is more *efficient*. Such cuts are called *green cuts*. Changes to a program that *depend* on a cut rather than the logic are called *red cuts* and are best avoided —e.g., `maxNo(X, Y, Y) :- X =< Y, !. maxNo(X, Y, X).` works by relying on the cut: It works with variables, but `maxNo(2, 3, 2)` matches the second clause unconditionally even though 2 is not the maximum of 2 and 3!

- ◇ Cut at the end ⇒ Don't consider any more clauses of the current predicate.

Lazy Conditional

`A -> B; C` If `A` is true, then prove `B` and ignore `C`; else prove `C` and ignore `B`.

- ◇ The "; C" portion is *optional* and `C` defaults to `fail`.
- ◇ We can also nest conditionals: $\mathtt{A}_1$ `->` $\mathtt{B}_1$`;` $\cdots$ `;` $\mathtt{A}_n$ `->` $\mathtt{B}_n$`; C` —again, `C` is optional.

We may use this form when we have disjoint conditions $\mathtt{A}_i$*!*

However, using multiple clauses is preferable as it clearly separates concerns.

Suppose we want all solutions to `p` except `e`, then we write:

```
all_but_e(X) :- X = e, !, fail.
all_but_e(X) :- p(X).
```

When we pose the query `all_but_e(e)`, the first rule applies, and we reach the cut. This commits us to the choices we have made, and in particular, blocks access to the second rule. But then we hit `fail`. This tries to force backtracking, but the cut blocks it, and so our query fails —as desired.

We can package up this red cut into a reusable form, 'negation as failure':

```
% neg(Goal) succeeds iff Goal fails.
neg(Goal) :- Goal, !, fail.
neg(Goal).

all_but_e(X) :- p(X), neg(X = e).
```

The built-in prefix operator `\+` is negation as failure —you may use `not(···)` but must use the parens and no space before them.

Remember: Order matters with Prolog's conjunction!

Hence, `\+ X = e, p(X)` always fails —see `neg` above— but `p(X), \+ X = e` yields all solutions to `p` except `e`.

Comprehension Exercise: With the left-side database, answer the right-side queries.

```
p(1).
p(2) :- !.
p(3).
```

```
?- p(X).
?- p(X), p(Y).
?- p(X), !, p(Y).
```

Higher-order Support with `call`

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).
```

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 in the form `f(`X_1`, ..., `X_n`)`.

```
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)).
```

Challenge: There are many shortcomings with this interpreter, such as no support for interpreting recursive functions, negation, failures, and disjunctions. Fix it!

The `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
```

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! (or here) —thorough, from basics to advanced
- ☐ Real World Programming in SWI-Prolog
- ☐ Adventures in Prolog —Amzi! inc.

Also, here's a nice set of 552 slides ^_^