The Prolog Dictionary

Contact Modified: 25 July 2015

This version of the Prolog dictionary assumes the syntax of SWI Prolog. Some examples assume a Unix command interpreter, as in Linux and MacOS X/Terminal.

You should use The Prolog Dictionary to clarify or revise concepts that you have already met. The Prolog Dictionary is *not* a suitable way to *begin* to learn about Prolog. That said, this dictionary *is* designed to be used by beginner and intermediate Prolog programmers. Further information on Prolog can be found in the SWI Prolog documentation linked above.

Other related dictionaries:
The AI Dictionary – URL: `http://www.cse.unsw.edu.au/~billw/aidict.html`
The Machine Learning Dictionary – URL: `http://www.cse.unsw.edu.au/~billw/mldict.html`
The NLP Dictionary (Natural Language Processing) – URL: `http://www.cse.unsw.edu.au/~billw/nlpdict.html`

Other places to find out about artificial intelligence include the AAAI (American Association for Artificial Intelligence) AI Reference Shelf

The URL of this Prolog Dictionary is `http://www.cse.unsw.edu.au/~billw/dictionaries/prologdict.html`

Topics Covered

To see the topic index, click on the link in the menu at left.

This dictionary was limited to the Prolog concepts covered in COMP9414 Artificial Intelligence at the University of New South Wales, Sydney. It has subsequently been expanded to cover the Prolog concepts in COMP9814 Extended Artificial Intelligence as well. Either way, it does not currently cover certain concepts *underlying* Prolog (resolution, etc.) It also does not cover all of ISO Prolog – once you understand everything in the Prolog Dictionary, you should certainly be ready to dive into the documentation that comes with your favourite Prolog interpreter.

More details on Prolog can be found in the COMP9414 lecture notes and COMP9814 lecture notes . The basic Prolog concepts – that is, the ones covered/used in COMP9414 – are marked with green dots (•) in the index, and also have the headword of the article about them in **green**.

Index

There are no entries (yet) for the letters J, K, X, Y, and Z.

append

The built-in predicate `append(?List1, ?List2, ?List1_then_List2)` succeeds if `List1` followed by `List2 = List1_then_List2`. Examples:

```
?- append([a, b], [c, d, e], Result).
Result = [a, b, c, d, e].
true.
?- append([a, b], SecondBit, [a, b, c, d, e]).
SecondBit = [c, d, e]
true.
?- append(FirstBit, [c, d, e], [a, b, c, d, e]).
FirstBit = [a, b]
true.
```

A not uncommon beginner's mistake is to use `append([Head], Tail, List)` to build a list instead of something like `List = [Head | Tail]`. This will work, but is like using a bulldozer to dig a hole to plant out a seedling.

Note: there is also an unrelated version of `append` that takes a single parameter. This is described under files, and is referred to as `append/1` – i.e. 1 argument, while the `append` in this article is referred to as `append/3` – 3 arguments.

arg

Not a cry of frustration at yet another error message, but rather, an abbreviation for argument, and also the name of a built-in metalogical predicate, that extracts arguments from a structure. Example:

```
?- arg(2, buys(john, beer), Arg).
Arg = beer
```

It is also possible to put arguments into terms using `arg`, should this prove useful:

```
?- X = likes(mary, Y), arg(2, X, pizza).
X = likes(mary, pizza)
```

```
Y = pizza
```

argument

See arg, term.

arithmetic

Many of the usual arithmetic operators are available in Prolog:

Operator	Meaning	Example
+	addition	`2 is 1 + 1.`
–	subtraction	`1 is 2 – 1.`
–	unary minus	Try the query `X is 1, Y is - X.`
*	multiplication	`4 is 2 * 2.`
/	division	`2 is 6 / 3.`
//	integer division	`1 is 7 // 4.`
mod	integer remainder	`3 is 7 mod 4.`
**	exponentiation	`1.21 is 1.1 ** 2.`

Except in the context of an arithmetic comparison operator, arithmetic expressions need to be explicitly *evaluated* in Prolog, using the `is` built-in predicate.

Mostly one uses these operators in a goal more like this:

```
X is Y + 2
```

where `Y` is a variable already bound to a numeric value, or perhaps a arithmetic comparison goal like this:

```
X > Y * Z
```

Here's another example: a rule to tell if a (whole) number is *odd*:

```
odd(Num) :- 1 is Num mod 2.
```

Another way to do this one is to use `=:=`:

```
odd(Num) :- Num mod 2 =:= 1.
```

If you wanted to be more cautious, you could first check that `Num` is in fact a whole number:

```
odd(Num) :- integer(Num), 1 is Num mod 2.
```

As usual in digital computations, with fractional numbers, it is necessary to be careful with approximation issues. Thus the final example in the table above, `1.21 is 1.1 ** 2.` actually fails when typed into Prolog as a query. This is because 1.1 ** 2, as represented in computers, actually comes out to be something like 1.210000000001. See the section on comparison operators for a solution to this issue.

Note that SWI Prolog does some apparently strange stuff in the domain of arithmetic (dialogue below done with SWI-Prolog Version 5.6.58):

```
?- X is [a].
X = 97.

?- X is [a] + [b].
X = 195.
```

```
?- X is sqrt(23456).
X = 153.154.

?- X is sqrt([23456]).
X = 153.154.

?- X is sqrt(123456789).
X = 11111.1.

?- X is sqrt([123456789]).
X = 13.3791.
```

The ASCII code for *a* is 97, which sort of explains the first two query outcomes. The last pair of queries are particularly strange. It is best to avoid relying on this sort of thing, and stick to standard arithmetical notation. Avoid strange tricks: code should be comprehensible.

See also comparison operators, built-in functions.

arity

The arity of a functor is the number of arguments it takes. For example, the arity of `likes/2`, as in `likes(jane, pizza)`, is 2, as it takes two arguments, `jane` and `pizza`.

Arity	Example	Could Mean ...
0	`linux.`	a bit like `#define LINUX` in* C
1	`happy(fido).`	Fido is happy.
2	`likes(jane, pizza).`	
3	`gave(mary, john, book).`	Mary gave John a book.

* hard to use in practice in Prolog: if the code contains `linux.` then you could test for this, but if it doesn't, then testing `linux` in your code will trigger an "Undefined procedure: linux/0" error. You can work around this by declaring `linux` to be dynamic – i.e. putting the line

```
:- dynamic linux/0.
```

in your code (usually at the start).

Every fact and rule in a Prolog program, and every built-in predicate has an arity. Often this is referred to in descriptions of these facts and rules, by appending `/` and the arity to the name of the rule or fact. For example, the built-in predicate member/2 has arity 2.

A fact or rule can have more than one arity. For example, you might want to have two versions of `make` terms:

```
% make(X, Y, Z): X makes Y for Z
make(fred, chest_of_drawers, paul).
make(ken, folding_bicycle, graham).

% make(X, Y): X makes Y
make(steve, fortune).
make(kevin, mistake).
```

What we have here are `make/3` and `make/2`.

assert, asserta, assertz

assert is a meta-predicate that adds its single argument, which may be a fact or a rule, to the Prolog database. The idea is that you construct or learn a new fact or rule in the course of executing your Prolog program, and you want to add it to the database. `asserta` ensures that the added fact/rule is added before any other facts or rules with the same functor), while `assertz` adds the fact after any other rules or facts with the same functor. When more than one rule/fact with the same functor is present in the database, they are tried in the order that they appear in the database, hence the need for `asserta` and `assertz`. You would use `asserta` in the common case where the new fact is supposed to save the effort involved in checking the fact using rules and other facts. You might use `assertz`, for example, if you were trying to construct a queue data structure (where items are added to the *end* of the queue. Examples:

```
?- assert(rich(mary)).
true.

?- rich(mary).
true.

?- assert((happy(X) :- rich(X), healthy(X))).
X = _G180

?- assert(healthy(mary)).
true.

?- happy(X).
X = mary
```

Facts/rules that are loaded from a file cannot be mixed with facts/rules that are created using `assert`/`asserta`/`assertz` without take the special step of declaring the procedure in question to be dynamic. For example, if you have a rule to say, compute *N*!, with header `factorial(+N, -Result)`, i.e with functor `factorial/2` and so arity 2, *and* you also want to use `asserta` to add pre-computed values of some factorials to the Prolog database (see also memoisation), then you need to declare `factorial` to be dynamic, by including the following with your loaded rules for `factorial`:

```
:- dynamic factorial/2.
```

Programs with `assert`ed facts and rules can be extra hard to debug.

See also `retract, retractall`, `dynamic`.

atom

An atom, in Prolog, means a single data item. It may be of one of four types:

- a string atom, like 'This is a string' or
- a symbol, like `likes`, `john`, and `pizza`, in `likes(john, pizza)`. Atoms of this type must start with a lower case letter. They can include digits (after the initial lower-case letter) and the underscore character (_).
- the empty list `[]`. This is a strange one: other lists are not atoms. If you think of an atom as something that is not divisible into parts (the original meaning of the word *atom*, though subverted by subatomic physics) then `[]` being an atom is less surprsing, since it is certainly not divisible into parts.
- strings of special characters, like `<--->`, `...`, `===>`. When using atoms of this type, some care is needed to avoid using strings of special characters with a predefined meaning, like the neck symbol `:-`, the cut symbol `!`, and various arithmetic and comparison operators.
  The available special characters for constructing this class of atom are: `+`, `-`, `*`, `/`, `<`, `>`, `=`, `:`, `.`, `&`, `_`, and `~`.

Numbers, in Prolog, are not considered to be atoms.

`atom` is also the name of a built-in predicate that tests whether its single argument is an atom.

```
?- atom(pizza).
true.

?- atom(likes(mary, pizza)).
false.

?- atom(<-->).
true.

?- atom(235).
false.

?- X = some_atom, atom(X).
X = some_atom.
```

The final example means that `atom(X)` has succeeded, with `X` bound to `some_atom`.

atom_chars

The built-in Prolog predicate `atom_chars` can convert an atom into the list of its constituent letters, or vice-versa. A fairly broad concept of atom is used: this predicate will glue together (or split up) any reasonable characters you give it. A possible list would be to put together a list of letters read, one character at a time, to make a word – that is, an atom whose name is the word. Examples:

```
?- atom_chars(pizza, List).
List = [p, i, z, z, a]

?- atom_chars(Atom, [b, e, e, r]).
Atom = beer

?- atom_chars(2007, List).
List = ['2', '0', '0', '7']

?- atom_chars(Atom, ['[', '3', ' ', ',', '4', ']']).
Atom = '[3 ,4]'
```

See also `atom_codes`.

atom_codes

The built-in Prolog predicate `atom_codes` can convert an atom into the list of the numeric codes used internally to represent the characters in the atom, or vice-versa. Examples:

```
?- atom_codes(pizza, List).
List = [112, 105, 122, 122, 97]

?- atom_codes(Atom, [98, 101, 101, 114]).
Atom = beer
```

See also `atom_chars`.

backtracking

Backtracking is basically a form of searching. In the context of Prolog, suppose that the Prolog interpreter is trying to satisfy a sequence of goals *goal_1, goal_2*. When the Prolog interpreter finds a set of variable bindings which allow *goal_1* to be satisfied, it commits itself to those bindings, and then seeks to satisfy *goal_2*. Eventually one of two things happens: (a) *goal_2* is satisfied and finished with; or (b) *goal_2* cannot be satisfied. In either case, Prolog backtracks. That is, it "un-commits" itself to the variable bindings it made in satisfying *goal_1* and goes looking for a *different* set of variable bindings that allow *goal_1* to be satisfied. If it finds a second set of such bindings, it commits to them, and proceeds to try to satisfy *goal_2* again, with the new bindings. In case (a), the Prolog interpreter is looking for *extra* solutions, while in case (b) it is still looking for the first solution. So backtracking may serve to find extra solutions to a problem, or to continue the search for a first solution, when a first set of assumptions (i.e. variable bindings) turns out not to lead to a solution.

Example: here is the definition of the built-in Prolog predicate `member`:

```
member(X, [X | Rest]). % X is a member if its the first element
member(X, [Y | Rest]) :-
    member(X, Rest).   % otherwise, check if X is in the Rest
```

You may not think of `member` as a backtracking predicate, but backtracking is built into Prolog, so in suitable circumstances, `member` will backtrack:

```
?- member(X, [a, b, c]).
X = a ;
X = b ;
X = c ;
false.
```

Here `member` backtracks to find every possible solution to the query given to it. Consider also:

```
?- member(X, [a, a, a]).
X = a ;
X = a ;
X = a ;
false.
```

Here `member` backtracks even though it keeps on finding the same answer. What about

```
?- member(a, [a, a, a]).
true ;
true ;
true ;
false.
```

This is because prolog has three ways of proving that `a` is a member of `[a, a, a]`.

The term backtracking also applies to seeking several sets of variable bindings to satisfy a single goal.

In some circumstances, it may be desirable to inhibit backtracking, as when only a single solution is required. The Prolog cut goal allows this.

Tracing the execution of a piece of Prolog code that backtracks can be a good way to figure out what happens during backtracking. If you wanted to experiment with backtracking by tracing `member`, you could achieve this by copying the code for `member`, given above, changing the name from `member` to say `mem` in the three places where it appears, and then tracing your `mem` procedure.

bagof

The built-in predicate `bagof(+Template, +Goal, -Bag)` is used to collect a list `Bag` of all the items `Template` that satisfy some goal `Goal`. Example: assume

```
likes(mary, pizza).
likes(marco, pizza).
likes(Human, pizza) :- italian(Human).
italian(marco).
```

Then

```
?- bagof(Person, likes(Person, pizza), Bag).
Person = _G180
Bag = [mary, marco, marco]
```

Notice that `Bag` contains the item `marco` twice, because there are two ways to prove that `marco` likes pizza – the fact and via the rule. `bagof` fails if `Goal` has no solutions.

See also `setof`, and, for differences between `bagof` and `findall`, see `findall`.

binding

Binding is a word used to describe giving a value to a variable. It normally occurs during application of a Prolog rule, or an attempt to satisfy the goals in a Prolog query.

Suppose that the Prolog database contains just the single fact `likes(mary, pizza).` and that there is a query:

```
?- likes(mary, What).
```

Prolog will search the (tiny) database, find that the query can be satisfies if `What = pizza`, do it will bind `What` to `pizza` and report success:

```
?- likes(mary, What).
What = pizza ;
false.
```

Now suppose that there is a rule and facts in Prolog's database:

```
teaches(Teacher, Student):-
    lectures(Teacher, Subject), studies(Student, Subject).
lectures(codd, databases).
studies(fiona, databases).
studies(fred, databases).
```

and that the user issues the query:

```
?- teaches(codd, Who).
```

The Prolog interpreter first matches the head of the rule with the query, and so binds `Teacher` to `codd`. It then finds a fact that indicates a subject that Codd lectures – namely `lectures(codd, databases)`. At this point, the variable `Subject` is bound to `databases` (`Subject = databases`). In other words, `Subject`, perhaps temporarily, has the value `databases`. Then Prolog tries to satisfy the second goal `studies(Student, Subject)` with `Subject = databases`, i.e. it tries to satisfy `studies(Student, databases)`. When it finds the solution, `studies(fiona, databases)` it will bind `Subject` to `fiona`, and report the solution:

```
?- teaches(codd, Who).
Who = fiona
```

Notice that the binding `Subject = databases` was made in solving the query, but it is not reported, as it is not explicitly part of the query.

Then, if the user types "`;`", Prolog will backtrack and undo the binding `Student = fiona` and look for another value for `Subject` that satisfies `studies(Student, databases)`, and find as `Student = fred`. However, while binding (and unbinding) is involved in this step, it is properly treated under backtracking.

body

the last part of a Prolog rule. It is separated from the head by the neck symbol, written as `:-`. It has the form of a comma-separated list of goals, each of which is a the name part of a functor, possibly followed by a comma-separated list of arguments, in parentheses. E.g. in the rule

```
sister_of(X,Y) :-
    female(Y),
    X \== Y,
    same_parents(X,Y).
```

the three goals `female(Y), X \== Y, same_parents(X,Y)` form the body.

built-in predicate

To make Prolog programming more practical, a number of predicates or are built into Prolog. These include some utility procedures, which *could* have been programmed by the Prolog user, and some which do non-logic programming things, like the input and output routines, debugging routines, and a range of others.

Bratko

This refers to the text used in COMP9414/9814 at the University of New South Wales, Australia: Bratko, I., *Programming in Prolog for Artificial Intelligence*, 4th Edition, Addison-Wesley, 2011

built-in functions, **`abs`**, **`atan`**, **`ceiling`**, **`cos`**, **`exp`**, **`float`**, **`floor`**, **`log`**, **`round`**, **`sign`**, **`sin`**, **`sqrt`**, **`tan`**, **`truncate`**

A small number of mathematical functions are built into Prolog, including:

Function	Meaning
`abs(Exp)`	absolute value of Exp : i.e. Exp if Exp ⩾ 0, −Exp if Exp < 0
`atan(Exp)`	arctangent (inverse tangent) of Exp : result is in radians
`cos(Exp)`	cosine of the Exp : Exp is in radians
`exp(Exp)`	e^{Exp} : *e* is 2.71828182845...
`log(Exp)`	natural logarithm of Exp : i.e. logarithm to the base* *e*
`sin(Exp)`	sine of the Exp : Exp is in radians
`sqrt(Exp)`	square root of the Exp
`tan(Exp)`	tangent of the Exp: Exp is in radians
`sign(Exp)`	sign (+1 or −1) of the Exp: sign(−3) = −1 = sign(−3.7)
`float(Exp)`	float of the Exp: float(22) = 22.0 – see also `float` the predicate
`floor(Exp)`	largest integer ⩽ Exp: floor(1.66) = 1
`truncate(Exp)`	remove fractional part of Exp: truncate(−1.5) = −1, truncate(1.5) = 1
`round(Exp)`	round Exp to nearest integer: round(1.6) = 2, round(1.3) = 1
`ceiling(Exp)`	smallest integer ⩾ Exp: ceiling(1.3) = 2

These functions should be used in a context where they will actually be evaluated, such as following `is` or as part of an arithmetic comparison operator like `=:=` or `>`.

Example:

```
?- X is sqrt(2).
X = 1.41421
```

Compare this with the following, where sqrt(2) is not evaluated, because `=` does not evaluate its arguments.

```
?- X = sqrt(2).
X = sqrt(2)
```

Another example:

```
?- X is log(3+2).
X = 1.60944.
```

These mathematical functions may correspond to arity 2 built-in predicates: for example, one can do this:

```
?- sqrt(2, X).
X = 1.41421
```

Some versions of SWI Prolog (e.g. 5.6.47) implement many of these arity 2 predicates, but not e.g. `exp/2`.

* High School Maths Reminder Service: if you want the logarithm to base *a*, divide log(Exp) by log(*a*). E.g. $\log_{10}$(X) = log(X)/log(10), and $\log_2$(X) = log(X)/log(2).

`call`

`call` is a built-in meta-predicate that allows its single argument to be called/invoked as a goal. For example, a program might create a goal (perhaps using `=..`) on the fly, and then, presumably later in the program, need to test the goal. Here are queries that perform these roles - in a real program, both the `assert` and the `call` would be built in to Prolog procedures written by the programmer.

```
?- assert(likes(mary, pizza)).

?- call(likes(Person, pizza)).
Person = mary

?- Goal =.. [likes, mary, What], call(Goal).
Goal = likes(mary, pizza)
What = pizza
```

clause

A clause in Prolog is a unit of information in a Prolog program ending with a full stop ("`.`"). A clause may be a fact, like:

```
likes(mary, pizza).
food(pizza).
```

or a rule, like:

```
eats(Person, Thing) :- likes(Person, Thing), food(Thing).
```

A clause may also be a query to the Prolog interpreter, as in:

```
?- eats(mary, pizza).
```

A group of clauses about the same relation is termed a procedure.

commenting your code

In Prolog, a the start of a comment is signalled by the character `%`. Comments are ignored by the Prolog interpreter; their purpose is to help a human reader understand the Prolog code. Examples:

```
% This is a full line comment.
% The next line contains a part-line comment.
member(Item, [Item|Rest]). % member succeeds if Item is 1st object in list.
```

Commenting Guidelines

As in other programming languages, comments should be used freely to explain the high-level significance of sections of code, to explain tricky sections of code, etc. Comments should *not* echo what the code already clearly says. Comments like that actually get in the way of understanding, for example because they are likely to make the reader ignore the comments. Where it is possible to make code understandable by using a meaningful functor name or variable name, this is preferable to a comment.

It is good practice to begin each Prolog procedure with a comment describing what it does, e.g.

```
% member(Item, List) - succeeds if the item is a member of the list
```

If the list needs to have some special properties, e.g. if it must be a list of numbers, or must be instantiated (that is, have a value) at the time the procedure is called, then this *header* comment should say so:

```
% member(Item, List) - succeeds if the item is a member of the list;
%   List must be instantiated at the time member is called. Item need not
%   be instantiated.
```

It can be a good idea for the header comments to indicate examples of the kind of data on which the procedure is intended to operate, and what the result should be, if this can be done reasonably briefly. E.g.

```
% Examples of use:
% ?- member(b, [a, b, c]).
% true.
%
% ?- member(X, [a, b, c]).
% X = a ;
% X = b ;
% X = c ;
% false.
```

Each file of Prolog code should begin with a (file) header comment, indicating who wrote the code, when, and what it is (overall) intended to do. This would be enough in a short Prolog assignment. In a "industrial strength" Prolog system, there would be details on the origins of algorithms used, and a revision history.

A good source of examples of Prolog commenting is example code made available in class, such as this one. Note however that this one is rather more heavily commented than usual, for instructional purposes. Note also that code examples presented on screen may be *under*-commented, because of the difficulty of fitting the comments and the code on the screen, and because the oral presentation accompanying the on-screen material replaces the comments to some extent.

Don't make your lines of comments (or code) too long - long lines can be hard to read, and really long lines may be folded, turning your neat formatting into a dog's breakfast. Stick to a maximum line length of 70 or 80 characters. Cute lines and/or boxes constructed of comment characters have the side effect of preventing the reader from seeing as much of your code at one time. The reader may then have to page up and down to figure out what your code does. They will not like you for this.

The use of cuts should normally be commented to explain why the cut is necessary.

It is a bad idea to comment (almost) every line, partly because of the clutter, and the distraction that this causes, and partly because most such comments are pointless duplications of the code (and/or comment things obvious except to a novice Prolog programmer):

Example of bad commenting (every line commented):

```
factorial(0, 1).                      % Factorial of 0 is 1.
factorial(N, FactN) :-
    N > 0,                            % N is positive
    Nminus1 is N - 1,                 % Calculate N minus 1
    factorial(Nminus1, FactNminus1),  % recursion
    FactN is N * FactNminus1.         % N! = N * (N - 1)!
```

Of these comments, only the first and last are even faintly justifiable, and even these are probably too obvious, as, particularly for the last comment, the variable names tell you everything you need to know. It looks pretty, with all the `%`-signs neatly lined up, but it forces the reader to check through all the unnecessary comments in case there is anything important there. This code needs a procedure header comment, and probably nothing else, though a beginning Prolog programmer might be justified in pointing out that the line `N > 0,` is there to make sure that the rule is only used in the case where `N` is positive. (If you're not sure why this is so important, try leaving out `N > 0,` and test the function on say `N = 2`.) So the comment would be "`% only use rule if N > 0.`"

How to write even worse comments!

It's easy – just write comments that are actually wrong. 🙁
Review your comments in a cool moment, and make sure that what they say is true.

The `+ - ?` convention for arguments to procedures

A convention often used to make the role of arguments to a procedure clearer is to tag them with `+`, `–`, and `?`, as follows:

Tag	Meaning
+	argument is expected to be *instantiated* (i.e. have a value rather than be a variable) when the procedure is called as a goal.
–	argument is expected to be a *variable to be bound* when the procedure is called as a goal.
?	argument may be either instantiated, or be an unbound variable, when the procedure is called as a goal.

Example:

```
% factorial(+N, -FactorialN).
%% supply a value for N, and FactorialN will be computed.

% member(?Item, ?List).
%% Item and List may either be instantiated, or a variable
%% (but in fact at least one of them must be instantiated!)
```

It's worth checking out, by the way, what happens if `List` is not instantiated ...

```
?- member(a, List).

List = [a|_G231] ;

List = [_G230, a|_G234] ;

List = [_G230, _G233, a|_G237]
```

and so on ... Prolog works its way through all list structures that contain `a`.

Spelling, grammar, etc.

It's no bad thing if all of these comments are spelled correctly and are grammatically phrased. (Sometimes it is appropriate to abbreviate, so perhaps your comments don't all need to be full sentences.) It makes life harder for folks trying to read your code, if they have to wade through poor spelling and grammar. Remember that the people reading your code will include people you will *want* to understand your comments easily – helpers, markers, colleagues, your boss, even yourself in a year's time when you've forgotten how you got the code to work ... So get into good commenting habits from the start.

See also indentation and white space.

comparison operators

(Only some of the operators below are relevant to COMP9414 at University of New South Wales – see green colouring below.)
Prolog has two main classes of comparison operators – arithmetic comparison operators (and similar

alphabetic comparison operators) and unification-style operators:

Comparison	Definition	Evaluates?
`X = Y`	succeeds if X and Y unify (match) in the Prolog sense	No
`X \= Y`	succeeds if X and Y do not unify; i.e. if `not (X = Y)`	No
`T1 == T2`	succeeds if terms T1 and T2 are identical; e.g. names of variables have to be the same	No
`T1 \== T2`	succeeds if terms T1 and T2 are not identical	No
`E1 =:= E2`	succeeds if values of expressions E1 and E2 are equal	Yes
`E1 =\= E2`	succeeds if values of expressions E1 and E2 are not equal	Yes
`E1 < E2`	succeeds if numeric value of expression E1 is < numeric value of E2	Yes
`E1 =< E2`	succeeds if numeric value of expression E1 is ⩽ numeric value of E2	Yes
`E1 > E2`	succeeds if numeric value of expression E1 is > numeric value of E2	Yes
`E1 >= E2`	succeeds if numeric value of expression E1 is ⩾ numeric value of E2	Yes
`T1 @< T2`	succeeds if T1 is alphabetically < T2	No
`T1 @=< T2`	succeeds if T1 is alphabetically ⩽ T2	No
`T1 @> T2`	succeeds if T1 is alphabetically > T2	No
`T1 @>= T2`	succeeds if T1 is alphabetically ⩾ T2	No

See also `is`. `is` is not a comparison operator, but is frequently confused with = by novice Prolog programmers. Briefly, you use `X is Exp` to *evaluate* an arithmetic expression, like `Y + 2`, that contains an arithmetic operator, like `+`, and bind the resulting value to the variable `X` to the left of the the operator `is`.

As an example of `@<` and its relatives,

```
?- likes(mary, pizza) @< likes(mary, plums).
true.
```

This succeeds because `likes` and `mary` are the same in both terms, and `pizza` alphabetically precedes `plums`.

Comparison of fractional numbers: When comparing two fractional numbers, problems can arise from the approximate nature of representations of fractional numbers in computers. Sometimes it will work as expected, and sometimes not. For example, the query `1.21 =:= 1.1 * 1.1.` fails with SWI Prolog on the computer where this article was written. You can and should work around this problem by, instead of testing fractional numbers for equality, doing something like the following:

```
?- abs(1.21 - 1.1 * 1.1) < 0.000001.
true.
```

abs signifies "absolute value": `abs(X) = X` if `X >= 0` and `abs(X) = -X` if `X =< 0`.
More generally, test for `abs(X - Y) < Tiny`, where `Tiny` is bound to small number (or *is* a small number, as in the example). How small to make `Tiny` above depends on the particular case – you might need to use a smaller number if `X` and `Y` need to be *very* close together to make your algorithm work correctly.

consult

These pre-defined pseudo-predicates allow one to load Prolog code into a running Prolog interpreter. Example:

```
?- consult('myprogram.pl').
```

This loads the contents of the Prolog program in the file `myprogram.pl` into the running Prolog's database. **Note** that in SWI Prolog, at least, before loading the new facts and rules into the database, Prolog first removes all facts and rules that relate to procedures in `myprogram.pl`. So if `myprogram.pl` contains, for example, the fact `likes(jane, pizza)` then all facts and rules about `likes` that have two arguments will be removed from the Prolog database before the new facts in `myprogram.pl` are loaded up. This is convenient when you are using `consult` to re-load a program after editing it (e.g. in another window, with the Prolog interpreter left running), but could be a little surprising if you were trying to load extra facts from a file.

It is possible to consult more than one file at a time, by replacing the single file name with a list of files:

```
?- consult(['file1.pl', 'file2.pl', 'file3.pl']).
% file1.pl compiled 0.00 sec, 524 bytes
% file2.pl compiled 0.01 sec, 528 bytes
% file3.pl compiled 0.00 sec, 524 bytes
true.
```

It is also possible to abbreviate a consult call, simply typing the list of (one or more) files as a goal for Prolog:

```
?- ['file1.pl', 'file2.pl', 'file3.pl'].
```

In some Prolog implementations, consulting `user` causes the Prolog interpreter to read facts and rules from the user's terminal:

```
?- [user].
|: likes(jane, pizza).
|: bad_dog(Dog) :-
|:    bites(Dog, Person),
|:    is_human(Person),
|:    is_dog(Dog).
|: <control-D>
% user://1 compiled 0.00 sec, 1,156 bytes
true.
?-
```

current input stream

At any given time, input to Prolog comes from the "current input stream". When Prolog starts up, the current input stream is the keyboard of the computer/workstation from which Prolog was started.

However, you might wish to get input/read from somewhere else during the execution of your Prolog program - for example, you might want to read from a file held on the computer on which the program is running, or on some local file server. To change the current input stream, you use the Prolog built-in extra-logical predicate `see`. If Prolog executes the goal `see('input.dat')`, then input will subsequently come from the file `input.dat`, in the current working directory of the workstation* that is running Prolog. If the specified file cannot be found in the current working directory, an error may be reported, as in this interaction with SWI Prolog:

```
?- see('nosuchfile.dat').
ERROR: see/1: source_sink `nosuchfile.dat' does not exist (No such file or directory)
```

Other errors are possible - you may not have the right to read the file in question, for example. If the file does exist and is readable, then subsequent read operations get their data from the file. The parameter to `see` can be just the file name, as illustrated above, or it could be a path to the file that is wanted: e.g. `see('/Users/billw/prologstuff/input.dat')` Prolog will continue to issue prompts for more queries while you are "seeing" a file, but any explicit read operations access the file. The built-in extra-logical predicate `seen` (with no argument) allows you to revert to reading data from the keyboard. Example (assuming `info.dat` starts with the line `hungry(jack).`):

```
?- see('info.dat'), read(X).
X = hungry(jack)

?- seen, read(Y).
|: full(jack).
Y = full(jack)
```

See also current output stream, input, output, files.

*Strictly speaking, `input.dat` will be expected to be in the current working directory of the command interpreter that started Prolog. The command interpreter will be running on the workstation/computer, and sending output to a window on that workstation or computer.

current output stream

At any given time, input to Prolog goes to the "current output stream". When Prolog starts up, the current output stream is the window or console of the computer/workstation from which Prolog was started. In other words, when your program writes things using the predicates described in the article on writing, they appear on your screen.

However, you might wish to write to somewhere else during the execution of your Prolog program – for example, you might want to write to a file held on the computer on which the program is running, or on some local file server.

To change the current output stream, you use one of the Prolog built-in extra-logical predicates `tell` and `append/1` *. If Prolog executes the goal `tell('output.dat')`, then output will subsequently go to the file `output.dat`, in the current working directory of the workstation* that is running Prolog.

If the specified file cannot be found in the current working directory, it will be created. If the file does exist, it will be overwritten. If you use `append/1`, subsequent write operations will add material to the end of the file, instead of overwriting the file. If you do not have permission to write files in the current directory, you will see an error message:

```
?- tell('/usr/bin/ztrash').
ERROR: tell/1: No permission to open source_sink `/usr/bin/ztrash' (Permission denied)
```

This means that either you do not have permission to write files in the directory `/usr/bin`, or if the file `ztrash` already exists in this directory, that you do not have permission to write to that file.

If the file is able to be written, then subsequent write operations send their data to the file. The parameter to `tell` can be a path to the file that is wanted, as in the example above, or it could be just the file name, e.g. `tell('output.dat')`. Prolog will continue to issue prompts for more queries and print bindings while you are "telling" or "appending" a file, but any explicit write operations access the file. The built-in extra-logical predicate `told` (with no argument) allows you to revert to writing data to the original window. Example:

```
?- tell('info.dat'), write(thirsty(jack)), nl.
true.
?- told, write(drunk(jack)).
drunk(jack)
true.
?- halt.
% cat info.dat # - # is Unix comment char, cat lists info.dat
thirsty(jack)
%
```

See also current input stream, input, output, files.

* `append/1` is not related to `append/3`, which in turn has nothing to do with output streams.

# Strictly speaking, `output.dat` will be expected to be in the current working directory of the command interpreter that started Prolog. The command interpreter will be running on the workstation/computer, and sending output to a window on that workstation or computer.

cut, !

The cut, in Prolog, is a goal, written as `!`, which always succeeds, but cannot be backtracked past. It is used to prevent unwanted backtracking, for example, to prevent extra solutions being found by Prolog. The cut should be used sparingly. There is a temptation to insert cuts experimentally into code that is not working correctly. If you do this, bear in mind that when debugging is complete, you should understand the effect of, and be able to explain the need for, every cut you use. The use of a cut should thus be commented.

Example: Suppose we have the following facts:

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

Then consider the following queries and their outputs:

```
?- teaches(dr_fred, Course), studies(Student, Course).

Course = english
Student = alice ;

Course = english
Student = angus ;

Course = drama
Student = amelia ;

false.
```

Backtracking is not inhibited here. `Course` is initially bound to `history`, but there are no students of `history`, so the second goals fails, backtracking occurs, `Course` is re-bound to `english`, the second goal is tried and two solutions found (`alice` and `angus`), then backtracking occurs again, and `Course` is bound to `drama`, and a final `Student`, `amelia`, is found.

```
?- teaches(dr_fred, Course), !, studies(Student, Course).

false.
```

This time `Course` is initially bound to `history`, then the cut goal is executed, and then `studies` goal is tried and fails (because nobody `studies history`). Because of the cut, we cannot backtrack to the `teaches` goal to find another binding for `Course`, so the whole query fails.

```
?- teaches(dr_fred, Course), studies(Student, Course), !.

Course = english
Student = alice ;

false.
```

Here the `teaches` goal is tried as usual, and `Course` is bound to `history`, again as usual. Next the `studies` goal is tried and fails, so we don't get to the cut at the end of the query at this point, and backtracking can occur. Thus the `teaches` goal is re-tried, and `Course` is bound to `english`. Then the `studies` goal is tried again, and succeeds, with `Student = alice`. After that, the cut goal is tried and of course succeeds, so no further backtracking is possible and only one solution is thus found.

```
?- !, teaches(dr_fred, Course), studies(Student, Course).

Course = english
Student = alice ;

Course = english
Student = angus ;

Course = drama
Student = amelia ;

false.
?-
```

In this final example, the same solutions are found as if no cut was present, because it is never necessary to backtrack past the cut to find the next solution, so backtracking is never inhibited.

Cuts in Rules

In practice, the cut is used in rules rather than in multi-goal queries, and some particular idioms apply in such cases. For example, consider the following code for `max1(X, Y, Max)`, which is supposed to bind Max to the larger of X and Y, which are assumed to be numbers (see note [1] below).

```
max1(X, Y, X) :- X > Y, !.
max1(_X, Y, Y). % See note [2]
```

This is a way of saying: "if the first rule succeeds, use it and don't try the second rule. (Otherwise, use the second rule.) We could instead have written:

```
max2(X, Y, X) :- X > Y.
max2(X, Y, Y) :- X =< Y.
```

in which case both rules will often be tried (unless backtracking is prevented by a cut in some other part of the code). This is slightly less efficient if `X` is in fact greater than `Y` (unnecessary backtracking occurs) but easier for people to understand, though regular Prolog programmers rapidly get to recognise this type of idiom. The extra computation in the case of `max2` is trivial, but in cases where the second rule involves a long computation, there might be a strong argument for using the cut on efficiency grounds.

Notes:

1. Note that although the version of `max` above with the cut in it works correctly if called with the first two arguments instantiated, and an uninstantiated variable as the third argument, as in

   ```
   ?- max1(6, 3, Max).
   Max = 6
   ?- max1(3, 6, Max).
   Max = 6
   ```

   if called with all three arguments instantiated, and the third argument instantiated to the wrong one of the first two arguments, then the code will succeed (incorrectly), since clause doesn't check anything:

   ```
   ?- max1(6, 3, 3).
   true.
   ```

   If the first two arguments are not instantiated, the code will also fail or generate an error message. E.g.

```
?- max1(X, 3, 0).
fail.
?- max1(0, X, 0).
ERROR: >/2: Arguments are not sufficiently instantiated
   Exception: (7) max1(0, _G188, 0) ?
```

So you need to think carefully about how your Prolog predicate is going to be used, and comment it to inform or remind readers about the situations in which it will and won't work. This is true about any code, but particularly Prolog predicates with cuts. In this case, a suitable comment would indicate that the first two arguments must be instantiated at the time when `max1` is called, and the third argument must be an uninstantiated variable.

Another way of dealing with this is to note that while the code for `max1` above with the cut works correctly if called as intended, there is in fact nothing wrong with adding the check `X =< Y` to the second rule of `max1` to get `max3`:

```
max3(X, Y, X) :- X > Y, !.
max3(X, Y, Y) :- X =< Y.
```

Because of the cut, the extra code (`X =< Y`) will only be used if `X > Y` fails (i.e. not as a result of inappropriate backtracking) so nothing is lost by doing this. The principle is: "A sound practice is to insert a cut in order to commit to the current clause choice, and also ensure as far as practically possible that clauses are written so as to stand independently as a correct statement about the predicate." (Quoted from p. 42 of *Clause and Effect: Prolog programming for the working programmer*, by William Clocksin.)

2. `_X` rather than just `X` is used in `max1` in order to avoid a singleton variable warning message.

debugging

means removing errors from program code. This is done by (1) observing the errors (i.e. testing); (2) locating the cause in the code; (3) correcting the errors.

(1) and (2) are often the hard part. Once you have observed or detected an error, tracing the code in question can help find the problem. Sometimes inserting calls to `write` into parts of the code where you think that the problem might be can help to localise the error.

Ultimately, reading your code carefully will be part of the task. So it would be a good idea to write the code carefully in the first place, in order to make it easy to understand. See also error and warning messages and commenting and white space.

declarative

A declarative programming language is one in which the relationships between the data are stated (or *declared* in a (usually logic-based) language, and then some automatic mechanism, e.g. a theorem-prover, is used to answer queries about the data. Prolog is a declarative programming language. Haskell, Miranda, Lisp, and Scheme are *functional* programming languages, in which all constructs are expressed using functions, function arguments, and function results, and C, C++, Java, Perl, Pascal, Modula-2, and Fortran are examples of (high-level) *procedural* programming languages, in which the program code expresses procedures to follow in manipulating the data.

`dynamic`

In Prolog, a procedure is either static or dynamic. A static procedure is one whose facts/rules are predefined at the start of execution, and do not change during execution. Normally, the facts/rules will be in a file of Prolog code which will be loaded during the Prolog session. Sometimes, you might want to add extra facts (or maybe even extra rules) to the procedure, during execution of a Prolog query, using assert/asserta/assertz", or maybe remove facts/rules using retract/retractall. To do this, you must declare the procedure to be dynamic.

You can declare a procedure to be dynamic by including in your code (normally adjacent to the facts/rules for that procedure) a suitable `dynamic` directive. Example – suppose you have a procedure called `likes`, with arity 2, and you have a "starter set" of facts/rules in your Prolog program, but you want to infer extra facts about `likes` during execution, and add them to the data base so that they don't need to be recomputed each time they are used. [You would normally only do this – add the new facts to the database – if the extra facts were slow to compute.] You need to declare `likes` (with arity 2) to be dynamic. You do this as follows:

```
:- dynamic likes/2.
```

[By the way, notice that this leaves open the possibility that a different version of `likes` (with arity 3, say) might not be dynamic.]

See also memoisation.

efficiency

There are many ways for code to be inefficient. This article concentrates on the issue of avoiding unnecessary recursive calls. Consider the following exercise: write a Prolog predicate `printOrAdd(IntegerList, Sum)` which binds the even numbers from its input list `IntegerList` to a list `EvensList`, and at the same time adds up the odd numbers in `IntegerList` and binds them to `Sum`. Here are two versions of the code, `printOrAdd` and `printOrAdd1`:

```
% printOrAdd is an inefficient solution to the task posed above:

printOrAdd([], [], 0).

% even numbers
printOrAdd([First | Rest], [First | RestOfEvens], Sum) :-
  printOrAdd(Rest, RestOfEvens, Sum),
  0 is First mod 2.

% odd numbers
printOrAdd([First | Rest], RestOfEvens, Sum) :-
  printOrAdd(Rest, RestOfEvens, SumOfRest),
  1 is First mod 2,
  Sum is SumOfRest + First.

% printOrAdd1 is a corrected version of printOrAdd which
% avoids the unnecessary recursive calls.

printOrAdd1([], [], 0).

printOrAdd1([First | Rest], [First | RestOfEvens], Sum) :-
  0 is First mod 2,
  printOrAdd1(Rest, RestOfEvens, Sum).

printOrAdd1([First | Rest], RestOfEvens, Sum) :-
  1 is First mod 2,
  printOrAdd1(Rest, RestOfEvens, SumOfRest),
  Sum is SumOfRest + First.
```

```
?- printOrAdd([1,2,3,4,5,6], Evens, SumOfOdds).
Evens = [2, 4, 6],
SumOfOdds = 9 ;
false.
```

The first version makes the recursive call in the two recursive rules before it tests to see if `First` is even or odd. If half the numbers are even and half odd, then clearly half of the recursive calls will be wasted. However, the situation is actually *much* worse than that, because of Prolog's automatic backtracking. The backtracking means that Prolog will eventually try all three rules for printOrAdd, and two of them will cause recursive calls, each of which will in turn generate two recursive calls, each of which ... Thus for a list of length 100, say, there will be more than 2^{100} recursive calls. If Prolog could manage a billion recursive calls per second, 2^{100} recursive calls would take about 2^{70} seconds, or about 30 trillion years.

The solution, of course, is to test before you make a recursive call, as is done in `printOrAdd1`. *Always test before you make a recursive call; never call before you test.*

error and warning messages

The messages from the Prolog interpreter when your program goes wrong can be difficult for a beginner to interpret. Little can be said in general, as the messages vary from system to system. You can find a guide for some of the messages produced by SWI Prolog here. See also underscore variables, debugging.

fact

A fact is a Prolog clause with a head with no variables in it, and no body, like

```
happy(fred).
likes(mary, pizza).
```

as opposed to this rule:

```
happy(Person) :-
    healthy(Person),
    enough_money(Person),
    has_friends(Person).
```

Bodiless clauses with variables, like ...

```
member(Item, [Item | RestOfList]).
```

... behave like rules in that they provide a general way of knowing that, for any `Item`, that `item` is a member of a list of which it is the first item (see also member). The clause above is likely to generate a SWI Prolog warning message like this:

```
        Singleton variables: [RestOfList]
```

to tell you that the variable `RestOfList` is only mentioned once in the code (so may be a spelling mistake, for example). You can suppress the warning by writing, instead:

```
member(Item, [Item | _]).
```

using a don't-care variable, although this may be more difficult for a human to follow, particularly a beginnner at Prolog.

`fail`

Built-in Prolog predicate with no arguments, which, as the name suggests, always fails. Useful for forcing backtracking and in various other contexts.

See also `true`, `repeat`, input schema.

`findall`

The built-in predicate `findall(+Template, +Goal, -List)` is used to collect a list `List` of all the items `Template` that satisfy some goal `Goal`. Example: assume

```
likes(mary, pizza).
likes(marco, pizza).
likes(Human, pizza) :- italian(Human).
italian(marco).
```

Then

```
?- findall(Person, likes(Person, pizza), Bag).
Person = _G180
List = [mary, marco, marco]
```

`findall` succeeds and binds `List` to the empty list, if `Goal` has no solutions. This can be convenient if you don't want your goal to fail just because the collection of solutions is empty. (In other cases, you *would* want the goal to fail if there are no solutions.)

Another difference between `bagof` and `findall` is the extent of backtracking done before binding the third parameter (`List`). For example, assume:

```
believes(john, likes(mary, pizza)).
believes(frank, likes(mary, fish)).
believes(john, likes(mary, apples)).
```

Then bagof and findall exhibit the following behaviour:

```
?- bagof(likes(mary, X), believes(_, likes(mary, X)), Bag).
X = _G188
Bag = [likes(mary, fish)] ;

X = _G188
Bag = [likes(mary, pizza), likes(mary, apples)] ;
false.

?- findall(likes(mary, X), believes(_, likes(mary, X)), Bag).
X = _G181
Bag = [likes(mary, pizza), likes(mary, fish), likes(mary, apples)] ;
false.
```

You can see that `bagof` is collecting `frank`'s beliefs about what `mary likes`, binding `Bag`, then backtracking and collecting `john`'s beliefs and re-binding `Bag`, while `findall` finds everybody's beliefs and binds them all to `Bag`, just once.

See also `setof`.

files, `tell, telling, told, see, seeing, seen, append/1`

When one writes a Prolog program, usually the facts and rules of the program are stored in a (text) file, and then loaded into the Prolog interpreter. Files have other uses in Prolog, too.

For example, we may wish to write out a table of results that have been computed for us by our Prolog program. One can use built-in predicates like `write, nl, putc, tab` and others to write out the table, but by default it will appear on the computer screen. To direct this output to a file, we use the `tell` built-in predicate. Suppose that we wish to write the table to a file called "mytable.data". By executing the (pseudo-)goal `tell("mytable.data")`, we tell Prolog that the new *current output stream* is to be the file "mytable.data". Subsequent writes will go to this file. When one wishes to stop writing to the file and resume writing on the screen, one uses the built-in predicate `told` (with no arguments). Also, the query *`?- telling(X).`* binds `X` to the name of the current output file. If the current output stream is not a file, then `X` will be bound to something that indicates that the current output stream is the screen - for example, in Unix, `X` may be bound to the atom `stdout` (standard output, which is normally the screen). Example:

```
?- tell('mytable.data'), write('***** Table of results *****'), nl, told.
% the file mytable.data should now contain a single line of text as above
```

The built-in Prolog predicate `append/1` is like `tell`, except that it arranges for subsequent write operations to add data to the end of the specified file, rather than overwriting the file with the first subsequent write operation. If `myotherfile.dat` initially contains, say, a single line, `This is the first line`, then `append/1` works as follows:

```
?- append('myotherfile.dat'), write('Here is another line'), nl.
true.
?- halt.
% cat myotherfile.dat # - # is Unix comment char, cat lists file contents
This is the first line
Here is another line
%
```

The situation for reading from a file is analogous to writing (except that there is no analogue for `append/1`). One can use built-in predicates like read, getc and others to read, by default from the keyboard. By executing the (pseudo-)goal `see('mydata.text')`, we tell Prolog that the new *current input stream* is to be the file `mydata.text`. Subsequent reads will come from this file. When one wishes to stop reading from the file and resume reading from the keyboard, one uses the built-in predicate `seen` (with no arguments). Also, the query `?- seeing(X).` binds `X` to the name of the current input file. If the current input stream is not a file, then `X` will be bound to something that indicates that the current output stream is the screen - for example, in Unix, `X` may be bound to the atom `stdin` (standard input, which is normally the keyboard). Example:

```
?- see('mydata.text'), read(X), seen, write(X), nl.
% the first Prolog term in the file mydata.text should now appear
% on the screen, having been read from the file with read(X), and then
% written to the screen with write(X) and nl.
```

What happens if you try to read from a file and there is nothing (left) to read, either because the file is empty, or you have previously read everything there was to read in this file? In this case, Prolog binds the variable that was the argument to `read` to the special atom `end_of_file`. Knowing this means that you can test after a `read` to make sure that you did not hit end of file. Example:

```
?- see('empty.dat'), read(Term).
Term = end_of_file
```

See also current input stream, current output stream, input, output. append/3 is unrelated to `append/1`.

functor, `functor`

In Prolog, the word functor is used to refer to the atom at the start of a structure, along with its arity, that is, the number of arguments it takes. For example, in `likes(mary, pizza)`, `likes/2` is the functor. In a more complex structure, like

```
persondata(name(smith, john), date(28, feb, 1963))
```

the top-level functor is termed the *principal functor* - in this case `persondata/2` - There is also a built-in predicate called `functor`, used to extract the name part and arity of a structure.

This built-in predicate takes three arguments: `functor(Term, Name, Arity)`. It succeeds if `Term` is a term with functor name `Name` and arity `Arity`. Examples:

```
?- functor(likes(mary, pizza), Name, Arity).
Name = likes
Arity = 2

?- functor(likes(X, Y), Name, Arity).
X = _G180
Y = _G181
Name = likes
```

```
Arity = 2

?- functor(likes, Name, Arity).
Name = likes
Arity = 0

?- functor(X, likes, 2).
X = likes(_G232, _G233)
```

Sometimes there are reasons to want to have the functor name somewhere other than at the start of the structure. For example, in the expression `X < Y`, "</2" is the functor, and so "<" is the functor name:

```
?- functor(2 < 4, Name, Arity).
Name = (<),
Arity = 2.

?- 2 < 4.
true.

?- <(2, 4).
true.
```

See op to find out how this works.

The term functor is used in a different sense in mathematics and in functional programming, and a different way again in philosophy.

goal

A query to the Prolog interpreter consists of one or more goals. For example, in

```
?- lectures(john, Subject), studies(Student, Subject).
```

there are two goals, `lectures(john, Subject)` and `studies(Student, Subject)`. A goal is something that Prolog tries to *satisfy* by finding values of the variables (in this case `Student` and `Subject`) that make the goal succeed. These value(s) are then said to be bound to the variable(s). If Prolog is unable to do this, the goal *fails* (and Prolog will print "false" in response to the query). If all the goals in a query succeed, Prolog prints the bindings necessary to make the query succeed. (If you then, in SWI Prolog, type a semicolon (`;`) Prolog will backtrack and look for another set of bindings that will satisfy the goals in the query.

Sometimes it is not necessary to bind variables in order to satisfy a goal. For example, there is no variable to bind, when the goal is `likes(mary, pizza)` and the Prolog database already contains `likes(mary, pizza)`. In this case, Prolog will print "true" in response to the query, rather than printing bindings.

Goals occur in rules as well as in queries. In

```
happy(Dog) :-
    is_dog(Dog),
    go_for_walk(Dog).
```

`is_dog(Dog)` and `go_for_walk(Dog)` are the two goals that form the body of the rule.

`halt`

This "goal", with no arguments, allows you to exit from Prolog in an operating-system-independent way. Typing the "end-of-file" character (control-D in Unix/Linux systems) will also get you out. Different operating systems may use different end-of-file characters. Example of `halt`:

```
% prolog
--- Welcome message from Prolog interpreter ---
```

```
?- halt.
%
```

The "`%`" signs in this example represent the operating system command interpreter (aka "shell") prompt, *not* a Prolog comment.

head

The first part of a Prolog rule. It is separated from the body by the neck symbol `:-`. It normally has the form of a functor (i.e. a relation symbol, followed by a comma-separated list of parameters, in parentheses. E.g. in the rule

```
sister_of(X,Y) :-
    female(Y),
    X \== Y,
    same_parents(X,Y).
```

`sister_of(X,Y)` is the head.

if-then-else, ->

The built-in infix predicate … `->` … `;` … functions as an if … then … else … facility. Example:

```
min(A, B, Min) :- A < B -> Min = A ; Min = B.
```

This version of `min` (which, like the one below, assumes that `A` and `B` are numbers) says "if `A < B` then unify `Min` with `A` otherwise unify `Min` with `B`". Possibly it is easier to understand a two-rule version of `min`:

```
min(A, B, A) :- A <= B.
min(A, B, B) :- B < A.
```

That is, the minimum of `A` and `B` is `A` if `A <= B`; the minimum is `B` if `B < A`. However, the `->` definition of `min` does illustrate the operation of `->`.

The code illustration has a rather procedural-programming feel to it that may comfort beginning users of Prolog. Possibly it should be avoided by them just for this reason! At least, they should avoid using it when the only reason for using it is the procedural feel. If use of `->` massively reduced the length of the code, thereby simplifying it, that might be an argument for using it.

The detailed semantics of … `->` … `;` … is relatively complex. The following description is quoted from the SWI Prolog help text on `->`:

> The `->/2` construct commits to the choices made at its left-hand side, destroying choice-points created inside the clause (by `;/2`), or by goals called by this clause. Unlike `!/0`, the choice-point of the predicate as a whole (due to multiple clauses) is not destroyed. The combination `;/2` and `->/2` acts as if defined by:
>
> ```
>     If -> Then  ; _Else :- If, !, Then.
>     If -> _Then ; Else  :-     !, Else.
>     If -> Then          :- If, !, Then.
> ```
>
> Please note that `(If -> Then)` acts as `(If -> Then ; fail)`, making the construct fail if the condition fails. This unusual semantics is part of the ISO and all de-facto Prolog standards.

Because of the cuts in this definition, the effect of `->` and … `->` … `;` … can be unanticipated. Beginning Prolog programmers should use it with as much care as a cut. Programmers who already know a programming language with an if-then-else construct (like C, C++, Java, ...) are likely to react to discovering … `->` … `;` … with little cries of joy and relief, and use it at every opportunity. A safer reaction is "if you don't fully understand the semantics, don't use it."

COMP9414 and COMP9814 students at UNSW are forbidden to use it in any situation in which cuts are forbidden, primarily because we want you to learn how to manage without it (and cuts) where possible.

indentation

Indenting your Prolog code in a standard way is a technique, along with commenting it, to make your code more easily understood (by humans). The standard way to lay out a Prolog procedure is exemplified below, using a procedure to compute the length of a list. Comments have been omitted, to allow you to focus just on the indentation.

```
listlength([], 0).

listlength([Head | Tail], Length) :-
    listlength(Tail, TailLength),
    Length is TailLength + 1.
```

The rule is to align heads of rules against the left margin, and to indent body clauses, normally all by the same amount. Sometimes further indentation is needed, for example if the code involves a complex term that won't fit on one line. In this case, you would indent the term to exhibit its structure.

```
    …
    book(
      title('The Strange Case of Dr Jekyll and Mr Hyde and Other Tales of Terror'),
      author(given_names(['Robert', 'Louis']),
             surname('Stevenson')
      ),
      publisher('Penguin Books')
    ), …
```

Sometimes the indentation rules can be bent: for example, it is not unusual to put on a single line, a rule with a body that contains just a single (short) clause.

```
happy(Person) :- rich(Person).
```

Comments on indented code should also be indented, if they can't be fitted on the same line as the code.

```
% sum_even_elements(+ListOfNumbers, -Sum): compute Sum of even items in ListOfNumbers
sum_even_elements([], 0). % base case
sum_even_elements([FirstNum | Rest], Sum) :-
    % check whether FirstNum is even
      0 is FirstNum mod 2,
    % if we get here it was even, so process rest of list and add FirstNum to result
      sum_even_elements(Rest, SumForRest),
      Sum is FirstNum + SumForRest.
sum_even_elements([FirstNum | Rest], Sum) :-
    % this rule handles the case where FirstNum is /not/ even
      0 =\= FirstNum mod 2,
      sum_even_elements(Rest, Sum).
```

This example is more heavily commented than necessary, in order to demonstrate the commenting convention.

See also comments and white space.

infix and prefix predicates/procedures

Most built-in constructs in Prolog, and, by default, the procedures and terms that you write yourself, use *prefix* syntax – that is, the name of the term/procedure (the functor) precedes the arguments (which are surrounded by parentheses and separated by commas). For example, when we use the built-in predicate `member`, we write something like `member(Item, [a, b, c])` – first the predicate name, `member`, then "(", then the first argument, `Item`, then a comma, then the second argument, `[a, b, c]`, then ")".

However, with built-in predicates like `=`, `<` and `>` that are usually written between their arguments, as in `First < Max`, we write, in Prolog as in mathematics, in *infix* notation. Another infix built-in "predicate" is `is`, which is used in evaluating arithmetic expressions.

It is possible in Prolog to define your own infix (and postfix) operators, and modify the syntactic handling of prefix operators – see `op`.

input in Prolog, `read`, `end_of_file`, `get`, `get_byte`, `getc`, `flush_output`

Input in Prolog is not always needed, as often the query process provides all the input that is needed. If explicit input is required, a set of extra-logical built-in predicates is available. These include:

- `read(X)` which reads the next term in the current input stream, which means the window on your workstation unless you have done something slightly fancy with files, and unifies it with the variable `X`.
- `end_of_file`: if there is nothing to read in the current input stream, `read(X)` causes `X` to be bound to the special symbol `end_of_file`. Unless you are absolutely sure you have some other way of knowing when there will be no more input data to read, you should check to make sure that the term that you have read is not `end_of_file`. Ways that you might be (almost) absolutely sure: the first term read might be a number indicating how many further terms are to be read.
- `get_byte(C)` which reads a single character from the current input stream, and binds the equivalent integer code, in the range 0 to 255. If there is no further character to read, `C` is bound to `-1`.
- `get(C)` is like `get_byte(C)`, except that it reads the first *non-blank* character, if any, from the current input stream.
- `flush_output` is actually an output goal, which ensures that any output which has been requested but not yet performed, is performed right now. Output might not be completed until there is enough to make it worthwhile, or until an operation like `flush_output` forces it.

Example 1: Assume that input is coming from the user's workstation window and that procedure `read_a_char` is defined as:

```
read_a_char(C) :-
   write('Type: '), flush_output,
   get_byte(C).
```

Then you can do the following – note that 43 is the character code for the character `+`.

```
?- read_a_char(Ch).
Type: +
Ch = 43
```

See also `atom_codes`, for conversion of a string of numeric character codes to an atom composed of the characters represented by those character codes. For example, 102, 105, 100, and 111 are the numeric codes for the letters f, i, d, and o. atom_codes can be used as follows:

```
?- atom_codes(A, [102, 105, 100, 111]).
A = fido
```

Example 2: Assume that there is a file called `inputdata`, which contains on its first line the term
`likes(mary, pizza).`
with a full stop at the end of the term.

```
?- see('inputdata'), read(Term), seen.
Term = likes(mary, pizza)
```

NB: *No* full stop at the end of Term's binding.

input schema

In any programming language, it is possible to specify schemata that how to code common programming tasks. The following is a schema for reading material from a file and processing it

```
dountilstop :-
  repeat,
  read(X),
  (X = stop, !
   ;
   process(X),
   fail
  ).
```

In this case, the code reads data, a term at a time, from the current input stream. To make that input stream a file called, say, inputdata, you would invoke the code as follows:

```
?- see('inputdata'), dountilstop, seen.
```

The call to `fail` is there to force backtracking. The call to `repeat` is there to force the code to endlessly repeat – that is, until something terminates the backtracking. This "something" is the cut after `X = stop` – if the term read is the atom `stop`, the goal `X = stop` succeeds, the cut is executed, and the rule terminates.

To use the schema, replace the read operation with the one you want (might be `getc` or `ratom` or ...), replace the call to `process` with a call to a procedure that does whatever *you* want to do to the data, and you're in business. Actually – you'll probably find a few other adjustments are needed, but at least you're on the right track.

See also `read`, `repeat`, `see, seen`, `;` (or), and `fail`.

`is`, evaluation

The `is` built-in predicate is used in Prolog to force the **evaluation** of arithmetic expressions. If you just write something like `X = 2 + 4`, the result is to bind `X` to the unevaluated term `2 + 4`, not to `6`. Example:

```
     ?- X = 2 + 4.
     X = 2+4
```

If instead you write `X is 2 + 4`, Prolog arranges for the second argument, the arithmetic expression `2 + 4`, to be evaluated (giving the result `6`) before binding the result to X.

```
     ?- X is 2 + 4.

     X = 6
```

It is only and always the second argument that is evaluated. This can lead to some strange-looking bits of code, by mathematical standards. For example, `mod` is the remainder-after-division operator, so in Prolog, to test whether a number `N` is even, we write `0 is N mod 2`, rather than the usual mathematical ordering: *N* mod 2 = 0.

What is the difference between `N = 1` and `N is 1`? In final effect, nothing. However, with `N is 1`, Prolog is being asked to do an extra step to work out the value of 1 (which, not surprisingly, is 1). Arguably, it is better to use `N = 1`, since this does not call for an unnecessary evaluation.

The message is: *use* `is` *only when you need to evaluate an arithmetic expression.*

Actually, you don't need to use `is` to **evaluate** arithmetic expressions that are arguments to the arithmetic comparison operators >, >=, < =<, =:= (equal), and =\= (not equal), all of which automatically evaluate their arguments.

Note the syntax of `is`: either
<*variable*> `is` <*expression*>
or
<*numeric constant*> `is` <*expression*>
Thus things like `X*Y is Z*W` do not work, as `X*Y` is an expression, not a variable – use either

```
0 is X*Y - Z*W
```

or

```
X*Y =:= Z*W
```

instead.

A common mistake, for people used to procedural programming languages like C and Java, is to try to change the binding of a variable by using an goal like `N is N + 1`. This goal will never succeed, as it requires `N` to have the same value as `N + 1`, which is impossible. If you find yourself wanting to do something like this, you could look at the code for `factorial` in the article on `tracing` for inspiration.

Another common mistake is to try a goal involving `is` before the variable(s) on the right-hand side of the `is` has/have been instantiated. Prolog cannot evaluate an arithmetic expression if it doesn't know the values of variables in the expression. This is why the following example fails:

```
?- X is Y + 1, Y = 3.
ERROR: is/2: Arguments are not sufficiently instantiated
```

Compare this with:

```
?- Y = 3, X is Y + 1.
Y = 3,
X = 4.
```

lists, [Head | Tail], .(Head, Tail)

A list in Prolog is written as a comma-separated sequence of items, between square brackets. For example, `[1, 2, 3]` is a list.

The empty list is written `[]`.

A list with just a single item, say the number 7, is written `[7]`.

Frequently it is convenient to refer to a list by giving the first item, and a list consisting of the rest of the items. In this case, one writes the list as `[First | Rest]`. Note that `Rest` *must* be a list, while Head need not be.

We have expressed this here using variables, but this need not be so, for example, we could write `[1, 2, 3]` as:

- `[1 | [2, 3]]`
- `[1 | Rest]`, where `Rest` is bound to `[2, 3]`
- `[First | [2, 3]]`, where `First` is bound to `1`
- `[First | Rest]`, where `First` is bound to `1`, and `Rest` is bound to `[2, 3]`

- `[1, 2 | [3]]`
- `[1, 2, 3 | []]`

and many more possibilities.

You should always write your Prolog list in the most compact reasonable format. So for example, while `[X | []]` is the same list as `[X]`, the second version is much easier to read, so you should use it.

It is possible to have lists of lists, or lists some of whose members are themselves lists:

- `[[1, 2], [3, 4], [5, 6]]` could be used to represent a 3 × 2 matrix;
- `[[a, b], c(d, e, f), 1]` is a 3-element list whose first element is the list `[a, b]`, second element is the term `c(d, e, f)`, and whose final element is the number `1`.

Lists can also be expressed using a normal term syntax, using the built-in predicate name `.` – that is, a full stop or period. In this case, the empty list atom (`[]`) must be used to terminate the list. However, this approach is more cumbersome, and in practice people use the `[1, 2, 3]`- style syntax. Example:

```
?- X = .(1, .(2, .(3, []))).
X = [1, 2, 3]
```

listing

The built-in meta-predicate listing can be used to check what is actually in the Prolog database during a Prolog session. It comes in two flavours: with no argument (`listing/0`) and with one argument (`listing/1`). With no arguments, it causes the contents of the Prolog database to be printed out. With one argument – the name of a procedure – it prints out just the rules and/or facts relating to that procedure. Examples, assuming you have started Prolog with a program file that contains the facts and rules `happy(ann). happy(tom). happy(X) :- rich(X). rich(fred).`

```
?- listing.
happy(ann).
happy(tom).
happy(A) :-
        rich(A).

rich(fred).
true.
?- listing(happy).
happy(ann).
happy(tom).
happy(A) :-
        rich(A).
true.
```

In the case of `listing/0`, Prolog may also print out some internal house-keeping information.

member built-in predicate

This predicate takes a item and a list as arguments, and succeeds if the item is a member of the list:

```
?- member(a, [b, a, c]).
true.

?- member(X, [b, a, c]).
X = b ;
X = a ;
X = c ;

false.
?- member(happy(fido), [angry(rex), happy(fido)]).
true.

?- member(a, [b, [a], c]).
false.
```

```
?- member([a], [b, [a], c]).
true.

?- member(a, Y).
Y = [a|_G310] ;
Y = [_G309, a|_G313] ;
Y = [_G309, _G312, a|_G316] ;
Y = [_G309, _G312, _G315, a|_G319] ;
```

... and so on for as long as you press ";". The response to this last query says that `a` is a member of an (uninstantiated) list `Y` if `a` is the first member or the second member or the third member or the fourth member or ...
The variables `_G309`, `_G310`, `_G313`, etc. represent bits of the list `Y` that we have no information about.

See also backtracking.

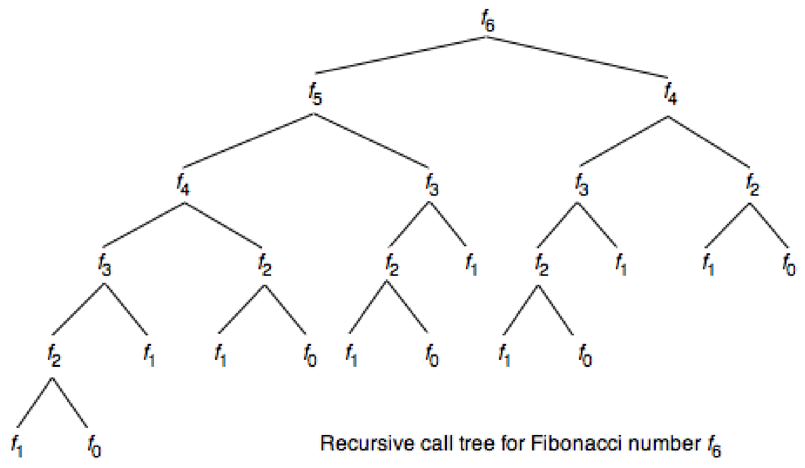
memoisation

Memoisation means storing a fact (e.g. using `asserta`), that has been inferred during the execution of your program, in the Prolog database, to avoid possibly having to infer the same fact again, later in the execution of the program.

Whether this is worth doing depends on the nature of the problem. If facts are regularly re-inferred (possibly as a consequence of a recursive solution), then memoisation could be a very good idea. Some algorithms perform double or multiple recursion (that is, they "recurse" on more than one variable), and both recursive branches can end up re-calculating the same thing, leading in some cases to exponential growth in the number of sub-cases as the size of the problem grows. The classic example is the recursive computation of the *n*–th Fibonacci number, which is (recursively) defined* by:

$f_0 = f_1 = 1$, and,
for $n > 1$, $f_n = f_{n-2} + f_{n-1}$.

* Note that a non–recursive expression for f_n is also known – something we ignore here since this example is about recursion.



Recursive call tree for Fibonacci number f_6

As can be seen (or rather, extrapolated) from the diagram, the number of calls grows exponentially, and the Fibonacci numbers rapidly become infeasible to calculate by the naive recursive program:

```
fib_naive(N, Result) :-
    N > 1,
    Nminus2 is N - 2,
    Nminus1 is N - 1,
    fib_naive(Nminus1, FibNminus2),
    fib_naive(Nminus1, FibNminus1),
    Result is FibNminus1 + FibNminus2.
```

However, by memoising, the computation becomes perfectly feasible for much larger values of *n*:

```
fib_memo(0, 1).
fib_memo(1, 1).
fib_memo(N, Result) :-
        N > 1,
        Nminus2 is N - 2,
        Nminus1 is N - 1,
        fib_memo(Nminus2, Fib2Nminus2),
        fib_memo(Nminus1, Fib2Nminus1),
        Result is Fib2Nminus1 +  Fib2Nminus2,
        asserta((fib_memo(N, Result) :- !)).
```

Note the use of `asserta` to store the memoised value at the *start* of the collection of facts for `fib_memo`, so that it will be checked *before* trying the rule. Note also the cut (`!`) at the end of the `assserta`-ed rule, which stops the algorithm from backtracking and trying the rule after it has found a solution using the memoised value(s). This cut is *necessary*, as otherwise, while the first solution would be found fast, the code would repeatedly find the same solution a huge number of times, taking a huge amount of computing time.

See also assert, dynamic.

`true.` vs `true`

The usual response to a Prolog query that does not have variables in it, or which only has underscores in it is `true.` or `false.`. However, in some versions of Prolog, including recent versions of SWI-Prolog another possible response can occur: `true` without a full-stop/period.
This means (1) that the query is true and (2) that there are other alternatives for Prolog to explore, that *might* allow it to conclude that the query is true in a different way. I.e. Prolog may be able to prove that your query is true in more than one way. For example, suppose that `likes(jane, pizza)` is in your Prolog program as a fact, but can also be inferred using a rule and some other facts.

```
?- likes(jane, pizza).
true
```

The `true` is a prompt to allow you to find out if this is so – i.e. that the query can be proven in more than one way. Type a semicolon if you want Prolog to go ahead and look for a different proof.

```
?- likes(jane, pizza).
true ;
true .

?-
```

Press "return" if you are happy with a single proof.

If you want to suppress this behaviour, you can insert cuts to prevent the backtracking that is finding the extra (possible) proofs, or you can, in SWI-Prolog, use this query:
`set_prolog_flag(prompt_alternatives_on, groundness).`
or alternatively, put the following directive in your Prolog source code:
`:- set_prolog_flag(prompt_alternatives_on, groundness).`

You may also be able to avoid responses of `true` in some cases by re-ordering relevant clauses in your program.

Here's some example code to demonstrate `true` :

```
happy(fido).
happy(rex).
happy(Dog) :- is_dog(Dog), walkies(Dog).
happy(Dog) :- is_dog(Dog), chase_car(Dog).
is_dog(fido).
is_dog(rex).
walkies(fido).
walkies(rex).
chase_car(fido).
```

There are three ways to prove `fido` is `happy`: from the fact, and from each of the two rules. So:

```
Prolog Dialogue              Commentary

?- happy(fido).
true ;          happy(fido) proven using fact
true ;          happy(fido) proven using "walkies" rule
true.           happy(fido) proven using "chase_car" rule

?- happy(rex).
true ;          happy(rex) proven using fact
true ;          happy(rex) proven using "walkies" rule
false.          happy(rex) not provable using "chase_car" rule
```

mutual recursion

Sometimes a procedure does not explicitly refer to itself (this would be (simple) recursion), but rather refers to a second procedure which in turn refers to the first. This sets up a two-step *mutual recursion*. Three- or more- step mutual recursion situations can also occur. The usual conditions applicable to recursion must occur - there must be a "trivial branch" somewhere in the cycle of mutually recursive procedures, and somewhere in the cycle something must happen to ensure that, each time around the cycle, a simpler version of the problem is being solved.

neck

the symbol `:-`, used in a Prolog rule to separate the head from the body. Usually read as *if*. Thus

```
a :- b, c.
```

is read as

`a` (is true) *if* `b` *and* `c` (are true).

negation, not, \+

The concept of logical negation in Prolog is problematical, in the sense that the only method that Prolog can use to tell if a proposition is false is to try to prove it (from the facts and rules that it has been told about), and then if this attempt fails, it concludes that the proposition is false. This is referred to as *negation as failure*. An obvious problem is that Prolog may not have been told some critical fact or rule, so that it will not be able to prove the proposition. In such a case, the falsity of the proposition is only relative to the "mini-world-model" defined by the facts and rules known to the Prolog interpreter. This is sometimes referred to as the *closed-world assumption*.

A less obvious problem is that, depending again on the rules and facts known to the Prolog interpreter, it may take a very long time to determine that the proposition cannot be proven. In certain cases, it might "take" infinite time.

Because of the problems of negation-as-failure, negation in Prolog is represented in modern Prolog interpreters using the symbol `\+`, which is supposed to be a mnemonic for *not provable* with the `\` standing for *not* and the `+` for *provable*. In practice, current Prolog interpreters tend to support the older operator `not` as well, as it is present in lots of older Prolog code, which would break if `not` were not available.

Examples:

```
?- \+ (2 = 4).

true.

?- not(2 = 4).
```

```
true.
```

Arithmetic comparison operators in Prolog each come equipped with a negation which does not have a "negation as failure" problem, because it is always possible to determine, for example, if two numbers are equal, though there may be approximation issues if the comparison is between fractional (floating-point) numbers. So it is probably best to use the arithmetic comparison operators if numeric quantities are being compared. Thus, a better way to do the comparisons shown above would be:

```
?- 2 =\= 4.

true.
```

number, `number(–)`

Numbers in Prolog can be whole numbers, like:

1 1313 0 –97 9311

or fractional numbers, like:

3.14 –0.0035 100.2

The range of numbers that can be used is likely to be dependent on the number of bits (amount of computer memory) used to represent the number. The treatment of real numbers in Prolog is likely to vary from implementation to implementation – real numbers are not all that heavily used in Prolog programs, as the emphasis in Prolog is on symbol manipulation.

`number` is also the name of a built-in predicate which tests its single argument and succeeds if that argument is a number. *Examples*:

```
?- number(2.1).
true.
?- number(-8).
true.
?- X = 7, number(X).
X = 7.
?- number(1+1).
false.
```

The last query fails because 1+1 is an unevaluated expression, not a number. See is, for more on evaluation.

`omnidirectional access`

... also known as *accessibility*, is not standard Prolog terminology, but it does describe a characteristic of many prolog predicates/relations. If we have a fact like `gives(mary, helen, book)`, then the information in this fact can be accessed in 8 ways (8 = 2^3, and the predicate `gives` has an arity of 3.)

```
?- gives(mary, helen, book).
true.

?- gives(X, helen, book).
X = mary

?- gives(mary, Y, book).
Y = helen

?- gives(mary, helen, Z).
Z = book

?- gives(X, Y, book).
X = mary,
Y = helen

?- gives(X, helen, Z).
X = mary,
Z = book

?- gives(mary, Y, Z).
Y = helen,
Z = book

?- gives(X, Y, Z).
X = mary,
Y = helen,
Z = book
```

This is an instance of omnidirectional access, or accessibility, and nothing like this occurs in procedural programming languages. Functional programming languages have what is termed *currying*, which is a related idea. For more on this, see Halford, Wilson, and Phillips (1998) Processing capacity defined by relational complexity: Implications for comparative, developmental and cognitive psychology, *Behavioral and Brain Sciences* **21**(6) 803–831, section 2.2.6.

once

The built-in Prolog extra-logical predicate `once` takes a single argument, which must be a "callable term" – one that makes sense as a goal – e.g. `happy(X)` makes sense as a goal, but `23` does not – and calls the term in such a way as to produce just one solution. It is defined as:

```
once(P) :- P, !.
```

See also `!`, `call`, `backtracking`.

op, infix, prefix, and postfix operators, precedence in Prolog

Syntax:

```
:- op(+Precedence, +Type, :Name).
```

The Prolog built-in predicate `op` serves to define the Type and Precedence of infix and postfix, and prefix operators in Prolog terms. Prolog terms normally begin with the functor (e.g. `likes`, in `likes(mary, pizza)`) but exceptions exist – for example, arithmetic expressions are written in the usual infix way (i.e. as `X + 1`, rather than `+(X, 1)`), and negation can be written without parentheses, as a prefix operator: `not P`.

The table below lists the predefined infix operators in SWI-Prolog. You may wish to add infix operators of your own. For example, you might wish to define an infix `and`. This can be done as follows:

```
:- op(700, xfy, and).
```

This declares `and` to be an operator with precedence 700 and type `xfy`.

Note two things: (1) the fact that these things are referred two as operators does *not* mean that an operation is performed when they are encountered. This is not usually the case; and (2) the declaration of an operator only affects the external appearance of the operator in your program – internally, the standard representation is used – for example `X + 1` really is internally represented by Prolog as though it was `+(X, 1)`.

`Precedence` is an integer between 0 and 1200. `Precedence` 0 removes the declaration. `Type` is one of: `xf`, `yf`, `xfx`, `xfy`, `yfx`, `yfy`, `fy` or `fx`. The `f` indicates the position of the functor, while `x` and `y` indicate the position of the arguments. Thus `xfx`, `xfy`, and `yfx` all indicate an infix operator. `y` should be interpreted as "in this position a term with precedence equal to or less than the precedence of the functor should occur". For `x` the precedence of the argument must be strictly less than that of the functor. The precedence of a term is 0, unless its principal functor is an operator, in which case the precedence is the precedence of this operator. A term enclosed in brackets (…) has precedence 0.

To see how this works, consider the arithmetic expression `a - b - c`. In normal maths, this is interpreted as `(a - b) - c`, rather than `a - (b - c)`. To achieve this, the binary infix operator `-` must be declared as type `yfx` so that the first argument has precedence over the second. Then, internally, `a - b - c` will be represented as `-(-(a, b), c)` rather than `-(a, -(b, c))`.

Built-in Operators (SWI-Prolog)

Prec.	Type	Operator
1200	xfx	-->, :-
1200	fx	:-, ?-
1150	fx	dynamic, discontiguous, initialization, module_transparent, multifile, thread_local, volatile
1100	xfy	;, |
1050	xfy	->, op*->
1000	xfy	,
954	xfy	\
900	fy	\+
900	fx	~
700	xfx	<, =, =.., =@=, =:=, =<, ==, =\=, >, >=, @<, @=<, @>, @>=, \=, \==, is
600	xfy	:
500	yfx	+, -, /\, \/, xor
500	fx	+, -, ?, \
400	yfx	*, /, //, rdiv, <<, >>, mod, rem
200	xfx	**
200	xfy	^

Note that all operators can be redefined by the user, most commonly accidentally and with disastrous results.

or, logical disjunction, ;

The logical conjunction (logical-and, ∧) of goals is achieved in Prolog by separating the goals by commas:

```
happy(X) :- rich(X), famous(X).
```

says that `X` is `happy` if `X` is rich and `X` is famous. What about logical-or, ∨, also called disjunction? If we want to say that `X` is `happy` if `X` is `rich` **or** if `X` is `famous`, we have two possibilities:

1. use two alternative rules
2. use the `;` operator (pronounced "or").

Option 1:

```
happy1(X) :- rich(X).
happy1(X) :- famous(X).
```

Option 2:

```
happy2(X) :- rich(X) ; famous(X).
```

Sometimes it is necessary, or at least advisable, to wrap parentheses `( )` around a disjunction in order to make clear what how the conjunctions and disjunctions interact. If you leave out the parentheses, Prolog has its own rules about the precedence of logical-and and logical-or, which might or might not correspond to what you intend. If you use parentheses, the things inside the parentheses are done first, so everything is crystal clear.

Option 3:

```
happy3(X) :- attractive(X), ( rich(X) ; famous(X) ).
```

This says `X` is `happy3` if `X` is attractive, and `X` is either rich or famous:
happy3(*X*) ⇐ *attractive*(*X*) ∧ (*rich*(*X*) ∨ *famous*(*X*)).*

* The Prolog Dictionary wishes to acknowledge that a range of world religions and philosophies would not agree that the criteria specified for `happy1`, `happy2`, and `happy3` are either necessary nor sufficient for happiness. The definitions given in this article, are, therefore, purely intended as Prolog programming examples, and should not be taken as a practical guide to life. Thank you.

output in Prolog

Output in Prolog is not always needed, as often the variable bindings return all the information that is required. If explicit output is required, a set of extra-logical built-in predicates is available.
These include:

- `write(X)` which writes the term X to the current output stream (which means the window on your workstation unless you have done something fancy).
- `print(X, …)` which writes a variable number of arguments to the current output stream. If an argument is a string (like `'Hello world!\n'`) then the string is printed without quotes, and any `'\n'` and `'\t'` are interpreted as newline and tab, respectively. A newline is printed at the end of the printing operation.
- `prin(X, …)` is like `print` except that it does not append a newline to the end of the output.
- `tab(N)` prints N spaces to output - `N` should be a number.
- `nl` starts a new line.
- `put(C)` prints a single character to the current output stream. C might be bound to a number in the range 0 to 255, in which case the number is treated as a character code, and the equivalent character is printed, or it might be bound to a character, e.g. `C = 'a'`, in which case obviously the character is printed.

predicate

Used to refer to the functionality afforded by a Prolog procedure. In this dictionary, we tend to talk about procedures when the code is written by the programmer, but predicates when the functionality is built-in, as with member.

procedure

A procedure in Prolog is a group of clauses about the same relation, for example:

```
is_a_parent(Parent) :- father_of(Parent, _Child).
is_a_parent(Parent) :- mother_of(Parent, _Child).
```

Here, two clauses together define the condition for someone to be a parent - together they form a procedure.

See also underscore for the reason for putting an underscore (_) character at the front of the variable `_Child`.

query

A query is a list of one or more goals typed to the Prolog interpreter's `?-` prompt, separated by commas, and terminated with a full stop (.). For example,
`?- lectures(john, Subject), studies(Student, Subject).`
is a query comprising two goals. Prolog tries to satisfy the goals, and if it manages to do this, the query is said to succeed. If not, the query fails.

If the query fails, Prolog types "`false.`". If it succeeds, Prolog either types the list of variable bindings it had to assume in order to make the query succeed, or, if no variable bindings were necessary, it types "`true.`".

If several variable bindings allow the query to succeed, then normally (i.e. in the absence of cuts) all the bindings will be typed out, one after the other, with the Prolog user typing a `;` to let the Prolog system know that they are ready to see the next set of bindings.

readability

See first the articles on commenting, indentation, and white space. Code needs to be readable so that readers, who might be you in a few days, or you in six months, or another programmer in six months or six years, can readily understand what you are trying to do.

Perhaps the first principle is to use *meaningful* and *accurate* names for variables and predicates. If your predicate is supposed to check if a number is even, you can call it `is_even`. If you later change it so that it also calculates the contribution, for the case of even numbers, to some total that you are calculating, then you need to change the name of the predicate to reflect this. Maybe you can call it `even_contribution`, and put a comment where the predicate starts, stating *exactly* what the predicate does, in more detail than you can fit into a predicate name.

Don't reinvent things already available in Prolog.
Here's an example of some code produced by a novice Prolog programmer:

```
mymember(First, [Second | _]) :-
        member(First, [Second]).
```

First of all, `member(First, [Second])` is identical to `First = Second`. So why not say so:

```
mymember(First, [Second | _]) :-
        First = Second.
```

But then, why have the explicit unification of `First` and `Second` – so instead:

```
mymember(First, [First | _]).
```

At this point, we can see that mymember is a misleading name for this predicate – which is actually checking whether its first argument is the same as the first member of the list that is the second argument. So the name of the predicate is misleading, as well as unclear. `member`, in Prolog, means check the whole list for the presence of the first argument, whereas this predicate only ever looks at the first element of the list. The work that this predicate is doing has no obvious name, and that suggests that it is not a useful abstraction of the problem being solved. The problem that the novice programmer was trying to solve was to find out if a list had successive elements that were the same, as happens for `b`, but not `a`. in the list `[a, b, b, c, d, a]`. So a better way would be like this:

```
has_repeats([]) :-
        fail. % unnecessary rule - see below
has_repeats([_OnlyOneItem]) :-
        fail. % unnecssary rule - see below
has_repeats([First, First | Rest]).
has_repeats([First, Second | Rest]) :-
```

```
        First \= Second,
        has_repeats([Second | Rest]).
```

In this code, the rule in red is the one that corresponds to what the novice programmer was trying to do with "`mymember`".
Notice also that the rules using `fail` can be omitted – they are only there to make it clear what happens with empty lists and lists with one member.

Don't have unused "junk" code. Obviously, it is worse still to have *used* junk code – that is, wrong code. However, unused junk code is confusing for the reader and demonstrates that the programmer who wrote it was confused. So how can you tell if code is unused junk? The best way, I suppose, is to have a thorough understanding of the code you are writing, and then you'll see that the code is junk. Failing that, when you test the code, you can put in calls to the built-in Prolog predicate `print`, at the start of any rule you are unsure about.

```
thing(X, Y) :-
        print('Entering rule 3 for predicate *thing*'),
        ... % rest of goals for this rule
        .
```

Then you try to generate a test case which will make use of your rule in finding it's first solution:

```
?- thing(A, [cat, dog, mouse]). % replace with actual parameters
    % that are supposed to test rule 3 of the predicate called thing
A = 3.
```

If, as in the example dialogue, the `print` never executes, or never executes as part of a successful search for a solution, then it's probably junk.

recursion

A recursive rule is one which refers to itself. That is, its body includes a goal which is an instance of the relation that appears in the head of the rule. A well-known example is the `member` procedure:

```
member(Item, [Item|Rest]).
member(Item, [_|Rest]) :- member(Item, Rest).
```

The second clause is clearly recursive – `member` occurs both in the head and the body. A recursive procedure can work because of two things:

1. the instance of the relation (like `member`) in the body is in some way simpler than that in the head, so that Prolog is being asked to solve a simpler problem. (In `member`, the list that is the argument to `member` in the body is shorter by one than that in the head.)
2. there must be a "trivial branch" "base case" or "boundary case" to the recursion – that is, a clause that does *not* involve recursion. (In the case of `member`, the first clause is the trivial branch – it says that `member` succeeds if `Item` is the first member of the list that is the second argument.)

A recursive **data structure** is one where the structures include substructures whose principle functor is the same as that of the whole structure. For example, the tree structure:
`tree(tree(empty, fred, empty), john, tree(empty, mary, empty))`
is recursive. Recursive data structures require recursive procedures to process them.

It can help, in understanding recursion, to separate the different depths of recursive invocation of Prolog rules by drawing boxes around the parts that correspond to a particular invocation, and giving separate (but systematic) names to the variables in each invocation. So if the rule involves a variable `X`, then in the first invocation, we refer to `X` as `X1`, while in the second (recursive) invocation, we refer to the new `X` as `X2`, and so on.

Let's try this out with the recursive procedure `sumlist`. In each recursive call of `sumlist`, there is a separate instance of the variables `First`, `Rest`, `Sum`, and `SumOfRest`, and these are distinguished by subscripts – so $\texttt{First}_1$ is the instance of `First` in the top-level call of `sumlist`, and $\texttt{First}_2$ is the instance of `First` in the first recursive call to `sumlist`.

```
% sumlist(NumList, SumOfList) - binds SumOfList to the sum
% of the list NumList of numbers.
% Rule 1:
sumlist([], 0).
% Rule 2:
sumlist([First | Rest], Sum) :-
   sumlist(Rest, SumOfRest), % Goal 2.1
   Sum is First + SumOfRest. % Goal 2.2
```

and the query

```
?- sumlist([5, 2, 1], Answer).
```

`sumlist[5, 2, 1], Answer).`
Choose Rule: only rule 2 matches, with
$\texttt{First}_1 = 5;\ \texttt{Rest}_1 = [2, 1];\ \texttt{Sum}_1 = \texttt{Answer}$
Goal 2.1: $\texttt{sumlist(Rest}_1\texttt{, SumOfRest}_1\texttt{),}$

> $\texttt{sumlist([2, 1], SumOfRest}_1\texttt{),}$
> Choose Rule: only rule 2 matches, with
> $\texttt{First}_2 = 2;\ \texttt{Rest}_2 = [1];\ \texttt{Sum}_2 = \texttt{SumOfRest}_1$
> Goal 2.1: $\texttt{sumlist(Rest}_2\texttt{, SumOfRest}_2\texttt{),}$
>
> > $\texttt{sumlist([1], SumOfRest}_2\texttt{),}$
> > Choose Rule: only rule 2 matches, with
> > $\texttt{First}_3 = 1;\ \texttt{Rest}_3 = [];\ \texttt{Sum}_3 = \texttt{SumOfRest}_2$
> > Goal 2.1: $\texttt{sumlist(Rest}_3\texttt{, SumOfRest}_3\texttt{),}$
> >
> > > $\texttt{sumlist([], SumOfRest}_3\texttt{),}$
> > > Choose Rule: only rule 1 matches, with
> > > $0 = \texttt{SumOfRest}_3$
> > > $\therefore \texttt{SumOfRest}_3 = 0$
> >
> > Goal 2.2: $\texttt{Sum}_3 \texttt{ is } \texttt{First}_3 + \texttt{SumOfRest}_3.$
> > $= 1 + 0$
> > $= 1$
> > $\therefore \texttt{SumOfRest}_2 = \texttt{Sum}_3 = 1$
>
> Goal 2.2: $\texttt{Sum}_2 \texttt{ is } \texttt{First}_2 + \texttt{SumOfRest}_2.$
> $= 2 + 1$
> $= 3$
> $\therefore \texttt{SumOfRest}_1 = \texttt{Sum}_2 = 3$

Goal 2.2: $\texttt{Sum}_1 \texttt{ is } \texttt{First}_1 + \texttt{SumOfRest}_1.$
$= 5 + 3$
$= 8$
$\therefore \texttt{Answer} = \texttt{Sum}_1 = 8$

Try doing a Prolog trace of the same query and procedure, and compare the results.

Here are some tips on writing recursive procedures in Prolog..

See also mutual recursion.

relation

The word *relation*, in Prolog, has its usual mathematical meaning. See relations and functions if you are not sure about what this is. A *unary* relation is a set of objects all sharing some property. Example:

```
is_dog(fido).
is_dog(rex).
is_dog(rover).
```

A *binary* relation is a set of pairs all of which are related in the same way. Example:

```
eats(fido, biscuits).
eats(rex, chocolate).
eats(rover, cheese).
eats(lassie, bone).
```

Similarly for *ternary* relations (triples) and so on. In the examples above, `is_dog` and `eats` are the functors for the relations. Prolog stores information in the form of relations, or more specifically relational instances, like those above, and also in the form of rules.

It is also possible in Prolog to have *nullary* relations (a relation with no arguments), though they are perhaps not often used. Example:

```
program_untested.
```

This could be used as a goal in a query:

```
?- program_untested.
true.
```

Possibly a better way to do this would be to use a unary relation:

```
program_status(untested).
```

However, the point is that nullary relations are there in Prolog, should you find a use for them.

See also the the next entry for more detail.

relations and functions in mathematics and in Prolog

(If you are looking for the Prolog implementation of mathematical functions like sin, cos, sqrt, log, exp, etc., look under built-in functions.)
A *binary* relation, in mathematics, is a set of ordered pairs. For example, if we are thinking about the 3 animals: *mouse*, *rabbit*, and *sheep*, and the "smaller" relation, then the set of pairs would be {(*mouse*, *rabbit*), (*mouse*, *sheep*), (*rabbit*, *sheep*)}. In mathematical notation, you would write, e.g., (*mouse*, *rabbit*) ∈ *smaller*, or *mouse smaller rabbit*. Or you might use a symbol instead of *smaller*, perhaps "<" – *mouse* < *rabbit*, or σ – *mouse* σ *rabbit*. This is referred to as an *infix* notation, because the name of the relation is written in-between the two objects (*mouse* and *rabbit*).

In Prolog, instead, by default we use prefix notation: `smaller(mouse, rabbit)`. See `op` if you want to define an infix operator in a Prolog program.

This view of a relation is sometimes called the *extensional* view – you can lay out the full "extent" of the relation, at least in the case of a relation over a finite number of items, like our "smaller" example. The alternative view is called *intensional*, where we focus on the meaning of the relation. In Prolog, this corresponds to a rule expressing the relation, such as:

```
smaller(X, Y) :-
    volume(X, XVolume),
    volume(Y, YVolume),
    XVolume < YVolume.
```

This, of course, only defines a meaning or intension for `smaller` relative to the unspecified meaning of the relation `volume(_,_)`, and the meaning of <, which is defined by the Prolog implementation.

When we come to non-binary relations, such as unary ones (like `is_a_dog(fido)`) or ternary ones (like `gives(john, mary, book)` or `between(rock, john, hard_place)`) the infix relation doesn't make any sense, so prefix notation is normal. (Postfix notation – `fido is_a_dog` can work for unary relations.)

While we often don't think of it this way, a *function* is a kind of relation. If you are thinking about the function $f(x) = x^2$, then the essential thing is that for each value of *x*, there is a value of $f(x)$ (namely x^2). In fact, there is a *unique* value of $f(x)$. So a function (of a single variable) can be viewed as a binary relation such that for every relevant first component *x*, there is exactly one pair in the relation that has that first component*: the pair is $(x, f(x))$. In the case of $f(x) = x^2$, the pairs are (x, x^2) for every applicable value of *x*. We need to specify what the applicable values of *x* are – the *domain* of the function. If the domain is {1, 2, 3}, then the pairs are {(1,1), (2,4), (3,9)}. If the domain is all natural numbers, then we can't write out the extension of the function in full, but we can use a set expression such as $\{(n, n\times n) \mid n \in N\}$ where *N* signifies the set of all natural numbers.

In Prolog, despite the fact that it uses a relational notation (except for arithmetic expressions), functions are common, but expressed using the relational notation that depends on the definition/convention in the previous paragraph. In our Prolog code defining `smaller(X, Y)`, above, the relation called `volume` is in fact a function written relationally. We are saying that for every relevant object `X` there is a value `XVolume` that is the volume of `X`. This is a function-type relationship.

The notion of domain applies to relations as well as functions: a more detailed definition of a binary relation *on a set A* says that such a relation is a subset of the set $A\times A$ of all pairs of elements of *A*. *A* is the *domain* of the relation. A binary relation between sets *A* and *B* is a subset of $A\times B$. And so on for ternary relations and beyond. A unary relation on *A* is just a subset of *A*. Thus *is_a_dog* is a subset of the set of all *Animals*. Or a subset of the set of all *Things*, depending on just how broadly you want to consider the concept of being a dog.

However, in Prolog, domains of relations are not explicitly addressed.# The notion of domain corresponds exactly to that of *type* in typed programming languages like, C, C++, Java, Haskell, etc. Prolog has no built-in way of confining the definition of a relation (whether extensionally or by a rule) to a particular domain. If you want to, you can build type-checking into your rules, e.g. by giving an definition of a type as a unary relation. For example, if you wanted to define the type of all popes, you could enumerate them all, though the list would be long:

```
pope(peter).
…
pope(john_paul_ii).
pope(benedict_xvi).
pope(francis).
```

Then the goal `pope(X)` would check if `X` was/is a pope.

Another example: when defining the relation `sister` in Prolog, you would usually write something like:

```
sister(Person1, Person2) :-
    female(Person1), female(Person2),
    mother(Person1, Mother), mother(Person2, Mother),
    father(Person1, Father), father(Person2, Father),
    Person1 \== Person2.
```

`female(Person1)` and `female(Person2)` are type-checks – without them, you are defining `sibling`, not `sister`. In practice, you can't enumerate all females, unlike all popes, but in many cases you would be able to enumerate all the relevant ones.

In some cases, you might be able to write rules to do type-checking. Prolog includes some built-in predicates for type-checking: `number(X)` succeeds if `X` is a number, while `integer(X)` succeeds only if `X` is a an integral (whole) number. Here is a rule to check if a number is divisible by 3.

```
div_by_3(X) :-
    X mod 3 =:= 0.
```

There is no special syntax for creating functions in Prolog (though there are special arrangements for built-in mathematical functions). To create a function in Prolog, you have to use the relation syntax and create a "relation that happens to be a function". You can do this extensionally, as in

```
beats(rock, scissors).
beats(paper, rock).
beats(scissors, paper).
```

or intensionally, with a rule, as with the function $f(x) = x^2$, which could be coded in Prolog like this (using the name `square_of`):

```
square_of(X, XSquared) :-
  number(X),
  XSquared is X * X.
```

used as in the next examples:

```
?- square_of(3.2, Y).
Y = 10.24
```

The `number(X)` test implicitly limits the domain of the function to numbers.

Note that while we defined `square_of` with a functional usage in mind, it can still be used in a relational sense if desired:

```
?- square_of(3, 9).
true.
```

There are limits to this – while some relations can be queried omnidirectionally:

```
?- beats(scissors, X).
X = paper
?- beats(Y, scissors).
Y = rock
```

`square_of` cannot, because the goal `XSquared is X*X` that is part of its definition cannot:

```
?- square_of(X, 4).
fail.
```

even though there are solutions (2 and —2).

Footnotes:
* if there is, not *exactly*, but *at most* one pair in the relation that has *x* as its first member, we say that the relation is a *partial function*. A partial function is like a function that has "holes" in it – *f*(*x*) never has more than one value, but for some *x* in the domain of the function, *f*(*x*) might not have any value. A real-life example is the partial function *eldest_child_of*, defined on the domain of all adult humans.

# there are a couple of areas of Prolog where there is some built-in type-checking – for example, the built-in predicate < will generate an error message if you try to use it compare a string like `'mouse'` with a

number like `9`.

```
?- mouse < 9.
ERROR: </2: Arithmetic: `mouse/0' is not a function
```

That is, the only way Prolog could make sense of this would be if `mouse` were a built-in function, like `sin`, `cos`, or `log`, but with no arguments. Compare

```
?- pi < 9.
true.
```

Here `pi/0` *is* a 0-argument function (i.e. a constant) whose value is an approximation to π, so Prolog can cope.

`repeat`

The built-in predicate `repeat` behaves as if defined by:

```
repeat.
repeat :- repeat.
```

Thus `repeat` succeeds when first called, thanks to the first clause. If the Prolog interpreter subsequently backtracks, the second clause (`repeat :- repeat.`) is tried. This initiates a new call to `repeat`, which succeeds via the first clause, and so on.

For a practical example of a use of `repeat`, see here.

`retract`, `retractall`

`retract` is a built-in meta-predicate used to remove facts or rules from the Prolog database while a program is executing. It is most often used in partnership with `assert` (or one of its relatives). For example, your program might have hypothesised that some fact or rule is correct, added it to the Prolog database using `assert` (or one of its relatives), then your program explores the consequences of that assumption, and concludes that the fact or rule was wrong. So then the program `retract`s the fact or rule.

More prosaically, you might simply have a query that runs repeatedly during a single prolog session, discovers some new facts in each run, but needs to get rid of them at the start of the next query. So again, `retract` can be used to clean the discovered facts out of the database.

```
?- assert(likes(mary, pizza)).
true.
?- likes(mary, pizza).
true.
?- retract(likes(mary, pizza)).
true.
?- likes(mary, pizza).
false.
?- assert((happy(X) :- rich(X), famous(X))).
X = _G180
true.
?- retract((happy(X) :- rich(X), famous(X))).
X = _G180
true.
```

Don't worry about the `X = _G180`, that's just SWI Prolog renaming the variable `X` with a unique name so it doesn't get confused with the (different) variable `X` that you might have used in some other rule. Note also the extra pair of parentheses `()` around the rule, as opposed to the fact.

What a call to `retract` actually does is to remove the *first* fact or rule that matches the argument to `retract`. If you want to remove, say, all the facts relating to `likes` with two arguments, it looks as though you might have to call `retract` repeatedly. Never fear, `retractall` is here! This meta-predicate, as its name suggests, retracts *all* facts or rules that match its single argument. For example:

```
?- assert(likes(mary, pizza)), assert(likes(john, beer)).
true.
?- listing(likes).
:- dynamic likes/2.
likes(mary, pizza).
likes(john, beer).
```

The "`:- dynamic/2`" tells us that `likes/2` is a built-in predicate that can be modified during program execution (see `dynamic`). This is to stop the program modifying unauthorised parts of itself, and becoming totally un-debuggable. Example continues:

```
?- retractall(likes(X, Y)).
X = _G180
Y = _G181
?- listing(likes).
:- dynamic likes/2.
true.
```

See also `assert`. Programs that use `retract` and `retractall` may be particularly difficult to debug.

Tip: In some versions of Prolog, `retractall` may fail if there is nothing to retract. This may seem reasonable (though it is probably a bug in the particular implementation) but can get in the way of a "prophylactic" `retractall` call to make sure there are no left-over facts/rules lying around from earlier computations. To get around this, `assert` a dummy fact of the right type (something `assert(likes(dummy, dummy))`, assuming `likes/2` is what you are trying to get rid of), before you call `retractall`. SWI Prolog does in fact succeed on a `retractall` call with no matching facts/rules.

rule

A rule in Prolog is a clause, normally with one or more variables in it. Normally, rules have a head, neck and body, as in:

```
eats(Person, Thing) :-
    likes(Person, Thing),
    food(Thing).
```

This says that a `Person` `eats` a `Thing` if the `Person` `likes` the `Thing`, and the `Thing` is `food`. Note that Prolog has no notion of the meanings of `eats`, `Person`, `Thing`, `likes`, and `food`, so for Prolog the rule is simply abstract symbol manipulation. However, presumably the author of the rule intended to have the meaning stated above. It would be possible to write the rule as `e(P, T) :- l(P, T), f(T).` or even `xx(X1, X2) :- xxx(X1, X2), xxxx(X2).` Obviously the rule is much more readable if meaningful procedure names and variable names are used.

Sometimes, in a "rule", the body may be empty:

```
member(Item, [Item|Rest]).
```

This says that the Item is a member of any list (the second argument) of which it is the *first* member. Here, no special condition needs to be satisfied to make the head true - it is always true.

Such rules can be re-expressed so that they have a body - e.g.

```
member(Item, [Head|Rest]) :- Head = Item.
```

This forces Prolog to perform another step (checking that `Head = Item`) so it is deprecated.

Sometimes in a rule, there might be no variables, or none in the head of the rule:

```
start_system :-
   initialise,
   invoke_system,
   finalise.
```

Such rules tend to have a very procedural style, effectively specifying a sequence of steps that need to be done in sequence. It's a look that is best avoided, or at least minimised.

See also fact.

schemata for list processing

Schemata (also called schemas) are program patterns which tell you, in a general way, how to write a program (or predicate) to deal with a particular class of data.

The simplest schema for processing a list computes a single result from the contents of a list, treating every member of the list in exactly the same way. For example, you might be adding up the numbers in a list, or multiplying them together. Here is such a schema – we'll call the predicate `listSchema`. If you use any of the schemata in this article, you'll need to change the name `listSchema` to something appropriate to the problem you are trying to solve:

Schema 1: doing the same thing to every member of a list

```
% first the base case - figure out what should happen to the
% empty list, and replace ResultForEmptyList with the
% value that your code should produce for the empty list.
listSchema([], ResultForEmptyList).

% second and last, the recursive case - if the list is not
% empty, then it has a first element, which is followed by the
% rest of the list:
listSchema([First | Rest], Result) :-
        listSchema(Rest, RestResult),
        compute Result given First, and RestResult.
```

The first rule will be used if the list is empty, and the second rule will be used if the list is not empty. Thus, in a particular case, there is only ever one rule that is applicable. For example, if you are adding up the items in a list of numbers, then replace *`ResultForEmptyList`* with `0`, and the last line of the recursive rule with `Result is RestResult + First`. So the final code in this case (renamed as `sumList`) would be

```
% sumList(List, Sum) adds up the numbers in a List that consists
% only of numbers and binds the result to Sum.

sumList([], 0).

sumList([First | Rest], Sum) :-
        sumList(Rest, RestResult),
        Sum is RestResult + First.
```

This all works provided the items in the list are indeed all numbers. Use the next schema for dealing with cases where this might not be so.

The schema above works provided the result being produced does not involve reconstructing the list as you process the items in it. Schema 2, below, deals with the case where the result is a new list.

Schema 2: transforming every member of a list, result is a new list

```
% First the base case - figure out what should happen to the
% empty list, and replace ResultForEmptyList with the
% value that your code should produce for the empty list.
% Often, the result for the empty list is the empty list again.
listSchema([], ResultForEmptyList).
```

```
% Second and last, the recursive case - if the list is not
% empty, then it has a first element, which is followed by the
% rest of the list:
listSchema([First | Rest], [NewFirst | RestResult]) :-
        listSchema(Rest, RestResult),
        compute NewFirst, the transformed version of First.
```

For example, if we are starting with a list of numbers, and producing as a result the list of squares of those numbers, then we would replace *compute NewFirst, the transformed version of First* with `NewFirst is First * First`.

So the final code in this case (renamed as `squareList`) would be

```
% squareList(List, ListOfSquares) binds ListOfSquares to a
% list consisting of the squares of the numbers in List.

squareList([], []).

squareList([First | Rest], [NewFirst | RestResult]) :-
        squareList(Rest, RestResult),
        NewFirst is First * First.
```

A slightly more complex schema is for processing a list, but testing each item in the list for some condition, and doing different things with the item depending on whether it does, or does not, satisfy the condition. This requires a base case, and *two* recursive cases:

Schema 3: doing different things to the members of a list

```
% First the base case - again, work out what should happen to the
% empty list, and replace ResultForEmptyList with the
% value that your code should produce for the empty list.
listSchema([], ResultForEmptyList).

% Second, the recursive case for when the item satisfies the
% condition. As before, if the list is not
% empty, then it has a first element, which is followed by the
% rest of the list:
listSchema([First | Rest], Result) :-
        goal to test for condition,
        listSchema(Rest, RestResult),
        compute Result given First, and RestResult.

% Third and last, the recursive case for when the item does
% not satisfy the condition. As usual, if the list is not
% empty, then it has a first element, which is followed by the
% rest of the list:
listSchema([First | Rest], Result) :-
        goal to test that condition is not satisfied,
        listSchema(Rest, RestResult),
        compute Result given First, and RestResult.
```

Once again, only one of these three rules can be applicable in a particular case. If the list is empty, the base rule is used. If the list is not empty and the condition is satisfied, the first recursive rule is used. If the list is not empty and the condition is not satisfied, the second recursive rule is used. For example, suppose that we want to add up the numbers in a list of items not all of which are numbers. Once again, the result for the empty list will be `0`. To check that the first item is a number, you could use the built-in Prolog predicate number, so your condition would be `number(First),`, and to compute the `Result` you would, as in the previous schema, use `Result is RestResult + First`. In the third rule, you'd use `not(number(First))`, and a first cut at computing the Result in this case would be `Result = RestResult`. However, this line could then be eliminated by simply putting `RestResult` in the result position in the head of this rule. So the final code in this case (renamed as `addNumbers`) would be

```
% addNumbers(List, Sum) adds up the numbers in the List (and ignores
% non-numbers) and binds the result to Sum.

addNumbers([], 0).

addNumbers([First | Rest], Result) :-
        number(First),
        addNumbers(Rest, RestResult),
        Result is First + RestResult.

addNumbers([First | Rest], RestResult) :-
        not(number(First)),
        addNumbers(Rest, RestResult).
```

More complicated cases, which need to look at the first *two* members of the list in order to decide how to handle the list, might need two base cases – one to handle the empty list, `[]`, and one to handle a list with exactly one item in it.

Another type of more complicated case might have more than two recursive rules, because there are three (or more) ways to proceed depending on what the next item is. For example, you might want to ignore non-numbers in the list, do something with positive numbers and zero, but do something different with negative numbers. You could do this with three recursive rules, which would use the three conditions

```
not(number(First)),
number(First), First >= 0,
number(First), First < 0,
```

respectively.

Yet another variant, see Schema 4 below, is where a list is being transformed into a new list, with members of the old list being transformed in different ways depending on whether they satisfy some condition.

Schema 4: transforming a list doing different things to the members, result is a list

```
% First the base case - work out what should happen to the
% empty list, and replace ResultForEmptyList with the
% value that your code should produce for the empty list, often [].
listSchema([], ResultForEmptyList).

% Second, the recursive case for when the item satisfies the
% condition. If the list is not empty, then it has a
% first element, which is followed by the rest of the list:
listSchema([First | Rest], [NewFirst | RestResult]) :-
        goal to test for condition,
        listSchema(Rest, RestResult),
        compute NewFirst given First, case where condition holds.

% Third and last, the recursive case for when the item does
% not satisfy the condition. As usual, if the list is not
% empty, then it has a first element, which is followed by the
% rest of the list:
listSchema([First | Rest], [NewFirst | RestResult]) :-
        goal to test that condition is not satisfied,
        listSchema(Rest, RestResult),
        compute NewFirst given First, case where condition does not hold.
```

For example, suppose you want to take a list of numbers and bind Result to the list obtained by squaring the non-negative numbers and adding 1 to the negative numbers in the list. In the first rule, `ResultfForEmptyList` would be `[]`. In the second rule, the condition would be `First >= 0`, and to compute `NewFirst` we would use `NewFirst is First * First`. In the third rule, the condition would be `First < 0` and to compute `NewFirst` we would use `NewFirst is First + 1`.

So the final code in this case (renamed as `squareOrAdd1List`) would be

```
% squareOrAdd1List(List, NewList) binds NewList to a
% list consisting of the squares of the non-negative numbers,
% and the successors of the negative numbers, in List.

squareOrAdd1List([], []).

squareOrAdd1List([First | Rest], [NewFirst | RestResult]) :-
        First >= 0,
        squareOrAdd1List(Rest, RestResult),
        NewFirst is First * First.

squareOrAdd1List([First | Rest], [NewFirst | RestResult]) :-
        First < 0,
        squareOrAdd1List(Rest, RestResult),
        NewFirst is First + 1.
```

Other variants on this include where you ignore elements if they don't satisfy the condition – e.g. to copy numbers and ignore non-numbers:

```
% squareOrIgnore(List, NewList) binds NewList to the result of copying
% numbers in List and ignoring non-numbers.

squareOrIgnore([], []).

squareOrIgnore([First | Rest], [First | NewRest]) :-
        number(First),
        squareOrIgnore([Rest, NewRest].

squareOrIgnore([First | Rest], NewRest) :-
        not(number(First)),
        squareOrIgnore(Rest, NewRest).
```

Plenty of other schemata are possible.

See also negation for why it might be preferable to use \+ rather than `not`.

See also efficiency for why it is important to do the test for *`condition`* or the test for *`not(condition)`* before making the recursive call.

See also notes on writing recursive predicates in Prolog.

`setof`

The built-in Prolog predicate `setof(+Template, +Goal, -Set)` binds `Set` to the list of all instances of `Template` satisfying the goal `Goal`.

For example, given the facts and rule:

```
happy(fido).
happy(harry).
happy(X) :- rich(X).
rich(harry).
```

it follows that

```
?- setof(Y, happy(Y), Set).
Y = _G180
Set = [fido, harry] ;
false.
```

Notice that:

- there are two ways to prove that `harry` is `happy`
    - from the fact `happy(harry)`.
    - from the rule and the fact `rich(harry)`.

However, `harry` only appears once in the binding for `Set`;
- the binding for `Set` is in sorted order.

Compare:

```
?- bagof(Y, happy(Y), Bag).
Y = _G180
Bag = [fido, harry, harry] ;
false.
```

and

```
?- findall(Y, happy(Y), Bag).
Y = _G180
Bag = [fido, harry, harry] ;
false.
```

The differences between `bagof` and `findall` on the one hand, and `setof` on the other hand, include the fact that with `setof`, you get sorted order and no repetitions. (For differences between `bagof` and `findall`, see findall.)

`setof` will fail (and so not bind `Set`) if the there are no instances of `Template` for which `Goal` succeeds – i.e. effectively it fails if `Set` would be empty. `bagof` also fails in these circumstances, while `findall` does not (it binds the variable that is its third argument to the empty list, or, if the third argument is instantiated, it succeeds if the third argument is the empty list).

side-effects

Prolog is supposedly a declarative language – that is, one which specifies relationships between concepts using facts and rules and then uses a logic engine to answer queries that relate to those facts and rules, typically by finding bindings for variables that allow queries to succeed.

However, in "real" Prolog, there are built-in predicates that do no comform to this pattern. These include input-output "predicates" like `see, seeing, seen, tell, telling, told`, `write, nl, prin, print, put`, and `read, get, get_byte, flush_output`. All of these predicates succeed or fail independent of any logical aspect of the problem being solved – normally, in fact, they will succeed unless there is an error relating to the external input/output system – e.g. it may not be possible to read from a file because the user does not have permission to access the file, or because the file does not exist.

Even more extra-logical are `assert, asserta, assertz` and `retract, retractall`, which even change the program that is running.

Side-effects take Prolog programs out of the world of strict logic (as does the cut, which changes the way the inference engine operates in solving a query). Programs with side-effects are harder to analyse that programs without side-effects. The practical impact of side-effects on the understandabilty of Prolog programs depends on how they are used. For example, using memoisation to record facts inferred in order to save re-calculation is probably harmless, but using `assert` to create new rules might in some cases make debugging a program next to impossible. Use side-effects with care.

singleton variables

SWI Prolog sometimes produces an warning message like this:

```
% cat example.pl
happy(Pet) :-
   dog(Pet),
   walkies(Oet).
% prolog -s example.pl
Warning: example.pl:1:
        Singleton variables: [Oet]
…
```

This is alerting you to the fact that the variable `Oet` has only been used once in the rule it appears in. Such variables are often (but not always) spelling mistakes. In this case, `Oet` should have been `Pet`.

Sometimes singleton variable reports are caused by the programmer trying to write helpful, readable code, like this version of `member`:

```
member(Item, [Item | Rest]).
member(Item, [NotItem | Rest]) :-
    \+ Item = NotItem,
    member(Item, Rest).
```

Prolog would report

```
        Singleton variables: [Rest]
```

for such code, because the helpfully-named variable Rest is only used once in `member(Item, [Item | Rest])`. The solution is to put an underscore (_) at the start of this instance of `Rest`, thus replacing `member(Item, [Item | Rest])`. with `member(Item, [Item | _Rest])`. SWI Prolog will treat `_Rest` as a don't-care variable and not report it.

spy

The built-in pseudo-predicate `spy` allows one to trace the execution of a query in a selective manner. If you want to find out about each time that a particular predicate is called, you `spy` that predicate: see example below.

```
?- listing(happy).
happy(A) :-
        rich(A).
true.
?- listing(rich).
rich(fred).
true.
?- spy(rich).
% Spy point on rich/1
true.
[debug]  ?- happy(fred).
   Call: (8) rich(fred) ? creep
   Exit: (8) rich(fred) ? creep
   Exit: (7) happy(fred) ? creep
true.
```

As you can see, the call to rich is traced, but not much else. See also `trace`, which traces the execution of everything.

Turn spying off with `nospy`:

```
[debug]  ?- nospy(rich).
% Spy point removed from rich/1

true.
```

static

A Prolog procedure is static if its facts/rules do not change during the execution of the program of which they are part. This is the default: most Prolog procedures are static. The opposite of static is dynamic.

structures

Structures in Prolog are simply objects that have several components, but are treated as a single object. Suppose that we wish to represent a date in Prolog - dates are usually expressed using a day, a month, and a year, but viewed as a single object. To combine the components into a single structure, we choose a functor, say `date`, and use it to group the components together – for the date usually expressed as 21 March 2004, we would probably write

```
date(21, mar, 2004)
```

Note that the order of the components is our choice – we might have instead written `date(2004, mar, 21)` The choice of functor is arbitrary, too.

The components between the parentheses are referred to as **arguments**. Thus in the date example, the arguments are `21`, `mar`, and `2004`.

Structures may be nested, too – the following example groups a name and a date, perhaps the person's date of birth:

```
persondata(name(smith, john), date(28, feb, 1963))
```

This term has two arguments, the first being `name(smith, john)` and the second being `date(28, feb, 1963)`

See also Bratko, section 2.1.3.

succeed

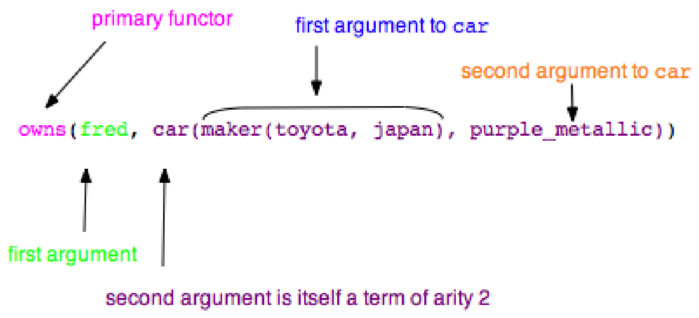
A Prolog goal succeeds if it is possible to either check it directly against the facts known to Prolog, or find bindings for variables in the goal that makes the goal true.

A query succeeds if each of its goals succeeds.

term

Syntactically, all data objects in Prolog are terms. For example, the atom `mar` and `date` and the structure `date(2004, mar, 21)` are terms. So are the numbers `2004` and `21`, for that matter. So in `date(2004, mar, 21)`, the primary functor is `date`, the arity is 3, and the arguments are `2004`, `mar`, and `21`.

In general, a term of arity *n* can be an atom (the functor) followed by *n* **arguments**, which will be enclosed in parentheses `( )` and separated by commas. Note that the arity might be 0, in which case there are no parentheses, commas, or arguments. A term can also be a number. Sometimes the term can use an infix operator like `<`, in which case the functor appears between the arguments (if the arity is 2) or before the argument but without parentheses (if the arity is 1).


primary functor
first argument to car
second argument to car
owns(fred, car(maker(toyota, japan), purple_metallic))
first argument
second argument is itself a term of arity 2

For `+` and `–` (and occasionally `?`) signs in front of arguments, in procedure header comments, see comments.

term type testing, `atom`, `atomic`, `compound`, `float`, `integer`, `nonvar`, `number`, `var`

It may be useful to be able to test if a term is an number or a variable or something else. Prolog provides several such tests:

Goal	succeeds if X is ...
`var(X)`	an uninstantiated variable
`nonvar(X)`	not a variable, or is an instantiated variable
`compound(X)`	a compound term – not an unbound variable, and not `atomic` (see below)
`atom(X)`	an atom, or bound to an atom
`integer(X)`	an integer, or bound to an integer
`float(X)`	a floating point (fractional) number such as 3.5 or 1.0, or bound to one – but, if `float(X)` is evaluated using `is`, then you get *this* `float`
`number(X)`	a number, or bound to a number (integer or float)
`atomic(X)`	a number, string, or atom, or bound to one

testing your code

Here are some tips on testing your Prolog code.
And here are some tips on writing recursive procedures in Prolog that include some tips on debugging (in example 3).

tracing

Tracing the execution of a Prolog query allows you to see all of the goals that are executed as part of the query, in sequence, along with whether or not they succeed. Tracing also allows you to see what steps occur as Prolog backtracks.

To turn on tracing in Prolog, execute the "goal"

```
?- trace.

true.
```

When you are finished with tracing, turn it off using the "goal"

```
?- notrace.
```

Here is an example of `trace` in action. First some code, which computes factorial N – the product of the numbers from 1 to N. By convention, factorial 0 is 1. Factorial N is often written N!, where the exclamation mark signifies the "factorial operator". Here is the code:

`factorial(0, 1).`	rule 1	factorial(0) = 1
`factorial(N, NFact) :-`	rule 2	
`    N > 0,`	rule 2, goal 1	if N > 0, then
`    Nminus1 is N - 1,`	rule 2, goal 2	compute N - 1,
`    factorial(Nminus1, Nminus1Fact),`	rule 2, goal 3	recursively work out factorial(N-1),
`    NFact is Nminus1Fact * N.`	rule 2, goal 4	then multiply that by N

The tables above and below may not display correctly if viewed in a narrow window. If the commentary doesn't seem to line up correctly with the text to the left of it, try making your browser window wider.

Prolog dialog/trace output	Commentary
`prolog -s factorial.pl`	Invoke prolog, loading code for factorial
`blah blah blah …`	Greeting from Prolog
`?- trace.`	Turn on tracing
`true.`	
`[trace]  ?- factorial(3, X).`	Call factorial
`   Call: (7) factorial(3, _G284) ? creep*`	Trace echoes query, replacing X with a unique variable
`^  Call: (8) 3>0 ? creep`	Rule 2, Goal 1 (N > 0) is invoked
`^  Exit: (8) 3>0 ? creep`	Goal 1 succeeds immediately

^ Call: (8) _L205 is 3-1 ? creep	Rule 2, Goal 2 invoked to compute 3 - 1
^ Exit: (8) 2 is 3-1 ? creep	and succeeds
Call: (8) factorial(2, _L206) ? creep	Rule 2, Goal 3 is invoked: level 2 call to factorial(2, …)
^ Call: (9) 2>0 ? creep	Goal 1 again for new call to factorial
^ Exit: (9) 2>0 ? creep	Goal 1 succeeds
^ Call: (9) _L224 is 2-1 ? creep	New version of goal 2
^ Exit: (9) 1 is 2-1 ? creep	succeeds
Call: (9) factorial(1, _L225) ? creep	Rule 2, Goal 3 is invoked: level 3 call to factorial(1, …)
^ Call: (10) 1>0 ? creep	Goal 1 again
^ Exit: (10) 1>0 ? creep	succeeds
^ Call: (10) _L243 is 1-1 ? creep	Goal 2 again
^ Exit: (10) 0 is 1-1 ? creep	successfully
Call: (10) factorial(0, _L244) ? creep	Rule 2, Goal 3 is invoked: recursive call to factorial(0, …)
Exit: (10) factorial(0, 1) ? creep	This time, Rule 1 succeeds at once.
^ Call: (10) _L225 is 1*1 ? creep	This is Goal 4 of level 3
^ Exit: (10) 1 is 1*1 ? creep	Compute 1 * 1 without trouble
Exit: (9) factorial(1, 1) ? creep	so the level 3 factorial call succeeds
^ Call: (9) _L206 is 1*2 ? creep	This is goal 4 of level 2
^ Exit: (9) 2 is 1*2 ? creep	Compute 1 * 2 without trouble
Exit: (8) factorial(2, 2) ? creep	so the level 2 factorial call succeeds
^ Call: (8) _G284 is 2*3 ? creep	This is goal 4 of level 1
^ Exit: (8) 6 is 2*3 ? creep	Compute 2 * 3 without trouble
Exit: (7) factorial(3, 6) ? creep	so the level 1 factorial call succeeds
X = 6 ;	… with this binding of X - type ";" to find more solutions
Redo: (10) factorial(0, _L244) ? creep	Prolog backtracks looking for another solution
^ Call: (11) 0>0 ? creep	
^ Fail: (11) 0>0 ? creep	
Fail: (9) factorial(1, _L225) ? creep	
Fail: (8) factorial(2, _L206) ? creep	
Fail: (7) factorial(3, _G284) ? creep	
false.	without success
[debug] ?- *notrace*. % turn off tracing	Turn off tracing.
true.	
[debug] ?-	

* What is this "creep" business? In SWI Prolog, the implementation of Prolog which this dictionary uses for the syntax of its examples, when you press return at the end of a line of tracing, Prolog prints "creep" on the *same* line, and then prints the next line of trace output on the next line. Pressing return again produces "creep" again and another line of tracing, and so on.

There are further tracing facilities in SWI Prolog. Do

```
?- help(trace).
```

to start to find out about them.

Built-in Prolog functions are not traced - that is, the internals of calls to things like `member` are not further explained by tracing them.

`true`

Built-in Prolog predicate with no arguments, which, as the name suggests, always succeeds.

See also `fail`, `repeat`.

See the article on don't care variables.

underscore, don't-care variable

The Prolog variable _ (underscore) is a "don't-care" variable, which will match anything (atom, number, structure, …). For example, the rule

```
bad(Dog) :- bites(Dog, _).
```

says that something (`Dog`) is bad if `Dog` bites anything. The "anything" is represented by _. Unlike other variables in Prolog, a bare _ can match different things in the same rule. So, for example, if `gives(From, To, Gift)` is a three-place procedure that is true if `From` gives `Gift` to `To`, then

```
giver(X) :- gives(X, _, _).
```

signifies that someone (`X`) is a "giver" if `X` gives something to anybody – the two _ s don't have to match the same thing. So if `gives(fred, john, black_eye).` is stored in the Prolog database, then the rule above allows us to infer that `fred` is a giver.

To be more explicit about the meaning of your rule, you could instead write:

```
bad(Dog) :- bites(Dog, _Anybody).
```

This indicates what the _ variable is actually representing, but has the feature that the binding of `_Anybody` typically doesn't get reported when Prolog finds a solution. In Prolog interpreters that report singleton variables (variables that are only used once in a rule – often these are caused by typographical errors in the code, which is why Prolog warns you about them) do not report singleton variables that begin with _. This is useful is you are trying to get rid of warning messages in your Prolog code.

The rule

```
bad(Dog) :- bites(Dog, _Anybody).
```

in fact has a bug, in the sense that the _Anybody will match things (like dogfood) that a dog might bite without being "bad". Maybe the rule should say instead:

```
bad(Dog) :-
    bites(Dog, Something),
    is_person(Something).
% add your own list of prohibitions - shoes, cats, …
```

Prolog systems make internal use of variable names beginning with an underscore, e.g. `_G157`. These are not "don't-care" variables, but are chosen to not clash with any variable name a Prolog programmer is likely to use.

Underscores may also be used in the middle of variable or atom names, as in `Intermediate_solution`, `likes_person` – again, these are not "don't-care" variables. The idea is to make the variable or atom name more readable.

unification, =, \=

When we write a goal like `X = Y` in Prolog, we are testing for more than simple equality in the mathematical sense. We are testing whether `X` (which might be a variable, an atom, or an arbitrarily complex term) *unifies* with `Y` (which might also be an atom, a variable, or a term). Two atoms, including numeric atoms, unify if they are the same. Two more complex terms unify if they have the same functor and their corresponding arguments unify. A variable always unifies with a term (provided that it is not previously unified with something different) by binding to that term. *Examples*:

`?- fred = fred.`
`true.`
	`fred` unifies with `fred` – they are the same atom.

`?- X = fred.`
`X = fred`
	`X` unifies with `fred` by binding `X` to `fred`.

`?- X = likes(mary, pizza).`
`X = likes(mary, pizza).`
	`X` unifies with `likes(mary, pizza)` by binding `X` to `likes(mary, pizza).`

`?- likes(`	

```
|         mary,
|         book(
|              title(Title),
|              author(
|                     given('Herman'),
|                     SurnameTerm)))
|   =
|   likes(
|         Who,
|         book(
|              title('Moby Dick'),
|              author(
|                     given('Herman'),
|                     surname('Melville')))).
Title = 'Moby Dick',
SurnameTerm = surname('Melville'),
Who = mary.
```

Note that the | characters at the start of each line are continuation prompts from Prolog because this query has been spread over more than one line.

`Title`, `SurnameTerm`, and `Who` are bound in order to achieve unification. The primary functors of the terms on either side of the `=` match, and unification proceeds by matching pairs of corresponding arguments, for example, by matching and unifying the first argument on the left, `mary`, with the first argument on the right, `Who`. Unification succeeds if *all* of the arguments match. This will include recursively matching the arguments of substructures such as the `author/2` substructures.

The goal `X \= Y` succeeds if `X = Y` would fail; it is the negated form of `=`.

Unnecessary Use of "`=`" to Force Unification

This is basically a style issue. Consider the two following alternative pieces of Prolog code for computing the maximum of two numbers.

```
max1(X, Y, X) :- X > Y.
max1(X, Y, Y) :- X =< Y.
```

versus

```
max2(X, Y, Max) :- X > Y, Max = X.
max2(X, Y, Max) :- X =< Y, Max = Y.
```

The first version is to be preferred, particularly for a novice Prolog programmer, because it reinforces how Prolog works. It also does not involve the unnecessary `Max = X` or `Max = Y`.

Whenever you write "`=`" in a Prolog procedure, review the code to see whether you can get rid of the "`=`" clause by replacing the item on the left of "`=`" by the item to the right of it, elsewhere in the procedure. If you do this with `max2`, you get `max1`. Sometimes you may have written the "`=`" with the variable on the right, in which case you need to instead replace the item on the right of the "`=`" with the item to the left of it.

It should be possible to do this whenever there is a variable (rather than a more complex term) on at least one side of the "`=`".

univ, `=..`

The `=..` meta-predicate, pronounced "univ", can be used to convert a list whose first item is a *non-numeric atom*, into a term, and vice-versa. For example,

```
?- Term =.. [likes, mary, pizza].
Term = likes(mary, pizza).
```

or

```
?- likes(mary, pizza) =.. List.
List = [likes, mary, pizza].
```

Here are some other examples:

```
?- Term =.. [this, predicate, works, with, longer, lists, too].
Term = this(predicate, works, with, longer, lists, too).
```

```
?- Term =.. [one].
Term = one.

?- Term =.. [one, two].
Term = one(two).
```

univ works with infix operators, too, though perhaps in a slightly unexpected way. Two examples:

```
?- op(700, xfy, and).

true.

?- Term =.. [and, p, q].

Term = p and q

?- Expression =.. [+, 2, 3], Value is Expression.

Expression = 2+3
Value = 5
```

variable

A variable in Prolog is a string of letters, digits, and underscores (_) beginning either with a capital letter or with an underscore. Examples:

`X`, `Sister`, `_`, `_thing`, `_y47`, `First_name`, `Z2`

The variable _ is used as a "don't-care" variable, when we don't mind what value the variable has. For example:

```
    is_a_parent(X) :- father_of(X, _).
    is_a_parent(X) :- mother_of(X, _).
```

That is, X is a parent if they are a father or a mother, but we don't need to know who they are the father or mother of, to establish that they are a parent.

Variables are used in more than one way:

- to express constraints in parts of a rule:
  `likes(A,B) :- dog(B), feeds(A,B).`
  Note that both `A` and `B` appear on both sides of the neck symbol – the appearance of the same variable in two (or more places) in effect says that when the variable is bound to a value, it must be bound to the same value in all the places that it appears in a given rule.
  Note that if the same variable appears in two or more different rules, it might be bound to different values in the different rules. This is achieved by Prolog renaming the variables internally, when it comes to use the rule – usually the names are things like `_1`, `_2`, `_3`, etc. This is why variables with names like these sometimes turn up in error messages when your Prolog program goes wrong.

- to formulate queries, as in the following example:
  `?- studies(X, 9311).` Prolog responds by finding all the values for `X` for which `studies(X, 9311)` is true, and successively listing them, e.g.

```
X = fred ;
X = jane ;
X = abdul ;
X = maria ;
```

white space

In writing code in Prolog or any other programming language, white space (blank lines, indentation, perhaps alignment of related data items) can be used to make the code easier to follow. Here are some suggestions:

- Put a blank line before the start of any prolog procedure.
- Group collections of facts using blank lines.
- Sometimes it helps to line up comments, but not at the expense of making lines longer than about 75–80 characters.
- Don't put helper procedures in the middle of other procedures: separate them out before or after the procedure they help. If your helper procedure is used in several other procedures, you might want to put it right at the end of all your code.

Example. In moderate size browser windows, the bad example will have long lines that are folded by the browser, and the good example will not. In practice, you could make the lines in the good example rather longer – up to 75–80 characters.

Good	Bad

`% smallest(+ListOfNumbers, -Min)`
`%  binds Min to the smallest item`
`%  in the non-empty ListOfNumbers.`
`%  ListOfNumbers should be instantiated`
`%  at time of call.`
`smallest([FirstNum], FirstNum).`
`smallest([FirstNum | Rest], Min) :-`
`      smallest(Rest, MinRest),`
`      smaller(FirstNum, MinRest, Min).`
 	
`% smaller(+First, +Second, -Smaller)`
`%  helper procedure for smallest.`
`smaller(A, B, A) :-`
`      A <= B.`
`smaller(A, B, B) :-`
`      B < A.`
	`% smallest(ListOfNumbers, Min) binds Min to the`
	`smallest item in the non-empty ListOfNumbers.`
	`ListOfNumbers should be instantiated at time of call.`
	`smaller(A, B, A) :- A <= B.`
	`smaller(A, B, B) :- B < A.`
	`% This is a helper procedure for smallest.`
	`smallest([FirstNum], FirstNum).`
	`smallest([FirstNum | Rest], Min) :- smallest(Rest,`
	`MinRest), smaller(FirstNum, MinRest, Min).`

See also indentation and comments.

+–? arguments

When you write a Prolog predicate, it is useful to document with a comment that indicates how the arguments are intended to be used. The `+–?` convention lets you do this succinctly. In

```
% factorial(+N, -NFactorial)
```

the `+` in front of `N` indicates that `N` is intended to be an input to this predicate. If when such a predicate is called, the `+` argument is not given a value (instantiated) the predicate is likely to fail. Some predicates (e.g. <) have only + arguments, because they are ended to be used to test some condition and produce a true/false (i.e. true/fail) result.
The `-` in front of `NFactorial` indicates that `NFactorial` is intended to be an output from this predicate. Thus it is OK to provide an (uninstantiated) variable as the second argument when calling this predicate. If when such a predicate is called, the – argument is given a value (instantiated), then the predicate is likely, in effect, to *calculate* the value for the the – argument, and if this is not the same as the supplied value, the predicate will then fail.
Some predicates have "?" arguments that can be either inputs or outputs. Such predicates can be used in multiple ways – with all arguments instantiated, or with some instantiated. If not all are instantiated, then Prolog will try to find bindings for the variables given for the uninstantiated arguments. Here is a trivial but hopefully understandable example:

```
% twist(?Pair, ?TwistedPair)
% - invert the order of items in a 2-member list
twist([X, Y], [Y, X]).
```

```
?- twist(X, [a, b]).
X = [b, a]

?- twist([c, d], Y).
Y = [d, c]
```

Either the second argument can be instantiated, and twist calculates the appropriate value for the first argument, or else the first argument can be instantiated, and twist calculates the appropriate value for the second argument.
Another, perhaps unexpected, example occurs with the built-in predicate `member`:

```
?- member(X, [a, b]). % finds both possible bindings for X
X = a;
X = b.

?- member(a, X). % finds expressions for all sets containing "a"
X = [a|_G312] ;
X = [_G311, a|_G315] ;
X = [_G311, _G314, a|_G318] ;
X = [_G311, _G314, _G317, a|_G321]   and so on for ever...
```

In the first solution to `member(a, X)`, `a` is the first member of the solution. In the second solution, `a` is the second member, and so on. So member's header comment would include something like

```
% member(?Item, ?List)
```