Backwards reasoning is from (potential) conclusions to facts instead of from facts to conclusions.
Prolog uses it because the space of possible conclusions grows too quickly in the number of premises.

You can use the `consult` and `reconsult` predicates, or use bracket notation.
Bracket notation allows re-consultation of several files at once:

```
['file1.pl', file2.pl']
```

A Prolog database knows everything it needs to know.

A name followed by zero or more arguments. Parens are omitted if there are no arguments.

A structure terminated by a period. It represents a simple fact.

A structure followed by a turnstile and a list of structures separated by commas. It represents a rule.

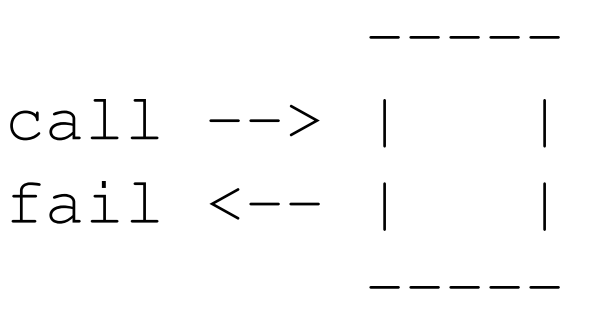
A **collection** of clauses with the same *functor* (name) and arity.
This is similar to the collection of overloaded versions of a function in an imperative language.

A collection of predicates, in any order.

Use single quotes, which does **not** make a string.

They indicate a list of ASCII values.

Use double quotes or an escaped single quote to use a single quote.
Other escapes use backslash as in other languages.



`exit` ports connect to `call` ports.
`fail` ports connect to `redo` ports.

Facts can be added at any time using the `assert` predicate.
Facts can be removed at any time using the `retract` predicate.
Such rules are *dynamic*.

To force a rule to be interpreted as a single argument, since rules contain commas.

```
assert((loves(chuck, X) :- female(X), rich(X))).
```

- Output can't be undone.
- `assert` and `retract` can't be undone either.

`write` predicate outputs its single argument to stdout.
`nl` writes a newline.

```
listing(predicate)

listing
```

It's an anonymous variable and can represent any term.

`fail` doesn't force other the entire predicate to fail. Other clauses will be tried.

Using a cut creates a commit point, preventing backtracking past the commit point and preventing attempts on other clauses.

The predicate as a whole fails.

To tell Prolog you are only going to use it once, but don't wish to use the anonymous variable.

For onymous variables, the single clause in which it appears.

The same order in which they were defined.

- Predicates with multiple clauses that have "tests" in them.
- Recursion.

Use a fail loop.

```
my_func(X) :-
    Some(),
    Imperative(),
    Calls(),
    fail.
my_func(_)
```

It's generally bad style to do this.

A parameter (conventionally the final one) of a parameter list can be used for output.

```
?- assert((first([Head | Tail], X) :- X = Head)).
true.
?- first([1,2,3], X).
X = 1.
```

You can store state in the database as facts.

```
bump_count :-
    retract(count(X)),
    Y is X + 1,
    assert(count(Y)).
```

It's generally bad style to do this.

The "univ" operator. It converts between structures and lists.

```
loves(chuck, X) =.. [loves, chuck, X]
```

```
?- member(1, [1, 2]).
true .

?- append([1], [2], [1, 2]).
true.
```

Make the final case a "dummy" with no body that uses anonymous variables for all the parameters.

```
wrapper :-
    potentially_failing_call.
wrapper.
```

Now any call to wrapper will succeed.

They are like `assert`, but `asserta` guarantees the added clause will come before any clauses with the same functor. Likewise `assertz` guarantees the added clause will be the last case in its predicate.

It removes ``all` clauses of the predicate with the given functor and arity.

```
abolish(somePred, arity).
```

`+`, `-`, `*`, `/`, and `mod` have their normal meanings, but *only when evaluated*.

They may not be evaluated when you want, leading to strange outcomes like:

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

Static clauses are the default and cannot be later modified using `assert/retract`.
Marking clauses dynamic (before they are defined) allows you to change the definition during program execution.

```
:- dynamic somePredArityTwo/2, somePredArityOne/1.
```

To actually force arithmetic to be performed you can use `is` or comparison operators like `=:=`, `=/=`, `>`, `>=`, `<`, `<=`.