Prolog is a “logic programming language.” Instead of expressing a sequence of actions to perform, Prolog programmers can (to a certain extent) describe the logical properties of their problem and let the computer search for a suitable solution.

The key concepts in logic programming are: terms, logic variables, unification, rules, negation, backtracking, and recursion.

### Terms

A term in Prolog is a named group of related values. Terms look like Python function calls, but they are completely different things: Prolog terms simply associate related pieces of data.

For example, `city(uppsala, sweden)` is a term, which we can use to express the fact that Uppsala is a city in Sweden. With a few more terms, we can assemble a database of cities:

```
city(uppsala, sweden).
city(stockholm, sweden).
city(helsinki, finland).
city(oslo, norway).
city(london, england).
city(paris, france).
city(paris, texas).
city(amsterdam, netherlands).
city(hague, netherlands).
```

Given these facts about cities, we can ask Prolog some questions. Prolog displays the prompt “?- ” when it is ready to receive a question. The question (text you need to type) is shown in a box.

```
?– city(paris, france). % Is Paris a city in France?
yes
?– city(paris, england). % Is Paris a city in England?
no
```

### Logic variables

In Prolog, variables behave much like variables in mathematics. If you write `X = uppsala`, then a subsequent statement `X = stockholm` makes as much sense as writing `uppsala = stockholm`. Also, `X = uppsala` and `uppsala = X` mean exactly the same thing.

Prolog uses the convention that variable names must start with an Upper Case letter (or an underscore character). Any name that starts with a lower-case letter (or is enclosed in single quotes) is a constant.
That’s why our city and country names were written in lower case in the city database above. If we wanted them in Upper Case, we would have to put single quote marks around each name.

Logic variables mean we can ask Prolog “Which cities are in Sweden?”. Here, Prolog displays the first answer it finds, and then waits to be told whether to look for more solutions. You can type ? to ask Prolog to keep looking, or Enter to make it stop.

?- city(City, sweden).
City = uppsala ? ;
City = stockholm ? [Enter]
no

We can also ask Prolog to find all the solutions to a question in one go:

?- findall( City, city(City, sweden), Cities ).
Cities = [uppsala, stockholm] ? ;
no

Questions in Prolog are called “goals.”

---

Unification

Unification in Prolog means making two terms equal, if possible. Terms that don’t have any variables are either identically equal or not. If they are identically equal, then the unification succeeds; if there is any difference, then the unification fails. So, uppsala = uppsala succeeds (“is true”) because the term uppsala is identically equal to itself.

Unification becomes more interesting when variables are involved. Unifying the terms city(X, paris) and city(france, paris) succeeds, and in doing so it makes the variable X have the value france. This is known as “binding” the variable.

---

Rules

Prolog programs can include rules of the form “conclude (something) is true if (something else) is true”. For example, we can conclude a city is in Scandinavia if that city is in Denmark. This rule of geography can be written in Prolog as:

city( X, scandinavia ) :- city( X, denmark ).

The conclusion is written to the left of the :-, and the condition for the conclusion is written on the right.

We can also add further rules to the geographic definition:
city( X, scandinavia ) :- city( X, norway ).
city( X, scandinavia ) :- city( X, sweden ).

If a rule requires several things to all be true, the conditions can separated by commas. For example, we might add some information about which cities are capitals:
capital(stockholm).
capital(wellington).
capital(oslo).
capital(helsinki).
capital(paris).
capital(washington).
capital(edinburgh).
capital(amsterdam).
capital(hague).

A rule for a capital city in Scandinavia can then be written:
scandinavian_capital( City ) :-
city(City, scandinavian).
capital(City).

These two conditions could also have been written in the opposite order.

---

**Negation**

Sometimes it is useful to ask whether something is not true. You can do this in Prolog by writing `\+` in front of a goal. For example, to ask “is Paris not a city in Texas” you can write `\+ city(paris, texas)`. This goal fails if `city(paris, texas)` succeeds, and succeeds if `city(paris, texas)` fails.

For example, we might have a rule for countries that have more than one capital. The first attempt below is wrong, since it does not say that City\_1 and City\_2 have to be different.

has\_more\_than\_one\_capital\_WRONG( Country ) :-
city(City\_1, Country).
capital(City\_1).
city(City\_2, Country).
capital(City\_2).

We can fix this problem by adding a negation:

has\_more\_than\_one\_capital( City ) :-
city(City\_1, Country).
capital(City\_1).
city(City\_2, Country).
capital(City\_2).
\begin{itemize}
\item City.1 = City.2.
\end{itemize}

**Backtracking**

The goal \texttt{city(X, paris)} can succeed in two different ways: by binding \texttt{X} to \texttt{france}, or by binding \texttt{X} to \texttt{texas}. Prolog copes with such choices by taking the first alternative that works, but it also remembers that an alternative choice remains. If a subsequence goal fails, Prolog returns to the most recent "choice point", undoing any variable bindings that were made in between, and tries the alternative instead.

Choice points accumulate. If we ask Prolog to find countries in Scandinavia, it first encounters the rule

\texttt{city(X, scandinavia) :- city(X, denmark).}

This rule is only valid if a solution can be found for \texttt{city(X, denmark)}. As the database has no cities for Denmark, the rule fails and Prolog backtracks to try the rule for \texttt{city(X, norway)}. This succeeds, and unifies \texttt{X} with \texttt{oslo}. This is returned as Prolog's first answer. If you ask for another solution (by typing \texttt{;}), Prolog returns to the most recent choice point, the rule for \texttt{city(X, sweden)}. This time, the backtracking will undo the binding for \texttt{X}, so this variable is able to be reused in finding a match for \texttt{city(X, sweden)}. The rule for Sweden in turn generates two choices: \texttt{uppsala} and \texttt{stockholm}. Prolog selects \texttt{uppsala} first, binding \texttt{X}, and comes back to select \texttt{stockholm} if required.

**Recursion**

The idea that geographic units can contain other geographic units is a powerful one: this type of structure arises in many different situations, such as classifications of species, lines of management, the Dewey Decimal classification of library books, etc.

Here are some geographical relationships:

\begin{itemize}
\item contains(scandinavia, denmark).
\item contains(scandinavia, norway).
\item contains(scandinavia, sweden).
\item contains(europe, scandinavia).
\item contains(europe, france).
\item contains(europe, germany).
\item contains(europe, britain).
\item contains(britain, england).
\item contains(britain, scotland).
\item contains(britain, wales).
\end{itemize}
We can include all our existing city facts in this model of geography by adding the rule:

```
contains(Country, City) :-
    city(City, Country).
```

We often want to know if one geographical unit is contained within another. Mathematicians call this relationship “transitive closure”. We will look at how a transitive closure predicate, `contains_t`, can be written in Prolog.

What facts do we need to define `contains_t`? We can start writing down each case one at a time, and then see how to write just two rules that capture all these cases (and more) succinctly.

The first rule is that a city “contains” itself:

```
contains_t(City, City).
```

For our second rule, we add that a region contains any city in that region:

```
contains_t(Region, City) :-
    contains(Region, City).
```

A third rule to express that a country contains any city in any region in that country:

```
contains_t(Country, City) :-
    contains(Country, Region),
    contains(Region, City).
```

Next, a geographic area contains any city in any region in any country in that area:

```
contains_t/Area, City) :-
    contains/Area, Country),
    contains(Country, Region),
    contains(Region, City).
```

These rules are now enough to let us ask for everything in Europe:

```
?- contains_t(europe, X).
```

```
X = europe.
X = scandinavia.
X = france.
X = germany.
X = britain.
% etc.
```

In principle, this approach could be extended indefinitely by adding more and more rules. However, it is possible to generalise these rules in a way that copes with every possible nesting of geographic structures.
Observe that each rule for contains \(_T\) is nearly identical to the preceding one, but adds another contains. We can express this transitive closure pattern using these two rules:

\[
\begin{align*}
\text{contains}_{\text{T}}( \text{Object, Object} ).
\text{contains}_{\text{T}}( \text{Bigger, Smaller} ) :&= \\
&\text{contains}( \text{Bigger, Intermediate} ). \\
&\text{contains}_{\text{T}}( \text{Intermediate, Smaller} ).
\end{align*}
\]

The first rule is the same as the first rule for contains \(_T\). This is sometimes called the "base case", as it can be answered immediately. The second rule says to find an intermediate value, and see if the "smaller" object is contained within that. If there are several possible intermediate objects (as with Scandinavia), then Prolog will try each in turn.

---

**Lists**

Lists are a convenient way of organising, well, lists of things. In Prolog, a list is written using square brackets, with the items in the list separated by a comma. For example, a list of the ten most popular baby names in Sweden in 2009 is written:

\[\text{[alice, elias, ella, elsa, emma, hugo, lucas, maja, oscar, william]}\]

This list happens to be sorted alphabetically, but this is not required.

A list can be empty, in which case it is written \([\text{}]\). If the list is not empty, then it can be divided into a “head” (the first element in the list) and a “tail” (a list of all the remaining elements, if any). If the list has only one element, then the tail is the empty list.

When working with lists, it often helps to think of just two types of list: an empty list, and a non-empty list. Every possible list must be one or the other. A non-empty list can be identified using the pattern \([\text{Head}|\text{Tail}]\). This is a pattern that matches a non-empty list. This pattern can be used both to build a list, and to peek inside one.

For example, to peek inside a list (and extract the first element), you can use:

\[
| \text{?-}\{\text{First}|\text{Rest}\} = [1,2,3].
| \text{First} = 1.
| \text{Rest} = [2,3]
\]

To make a list, you can use

\[
| \text{?-}\{\text{ExistingList} = [1,2],\text{NewList} = [0|\text{ExistingList}]\}.
| \text{ExistingList} = [1,2]
| \text{NewList} = [0,1,2]
\]
One useful predicate for working with lists is \texttt{member}:

\begin{verbatim}
| ?- member( Name, [alice, elias, ella] ).
Name = alice ;
Name = elias ;
Name = ella ;
no
\end{verbatim}

\texttt{member} can also be used to test if an element is present in a list:

\begin{verbatim}
| ?- member( john, [alice, elias, ella, elsa, emma, hugo, lucas, maja, oscar, william] ).
no
\end{verbatim}

We might like to compare the list of Swedish baby names to a list of the ten most popular baby names in New Zealand in 2009:

\texttt{[ella, isabella, jack, james, joshua, oliver, olivia, ruby, sophie, william]}

Clearly the lists are not the same, but we can find names that appear in both lists using the \texttt{member} predicate:

\begin{verbatim}
common( Name ) :-
    member( Name, [alice, elias, ella, elsa, emma, hugo, lucas, maja, oscar, william] ).
    member( Name, [ella, isabella, jack, james, joshua, oliver, olivia, ruby, sophie, william] ).
\end{verbatim}

This can be read as “a name is common to the two lists if it appears in the Swedish baby name list and also appears in the New Zealand baby name list.” The predicate can be used to either generate common names, or to test if a particular name is common:

\begin{verbatim}
| ?- common(Who).
Who = ella ;
Who = william ;
no
| ?- common(john).
no
\end{verbatim}

Observe how the first occurrence of \texttt{member} enumerates the names one by one, and the second occurrence is a test to see if the currently selected name is present.

\texttt{member} is so useful that Prolog provides it as a built-in predicate. If it didn’t, you could write it yourself as follows:

\begin{verbatim}
member( Element, [Element|Tail] ).
member( Element, [Head|Tail] ) :-
    member( Element, Tail ).
\end{verbatim}

— John Hamer, 19 June 2012