**A gentle introduction to Prolog**

Our language of choice will be Prolog, the most popular logic programming language. A program in Prolog is initiated by running a query and seeing if it can be proven using the relations defined. In a sense, logic programming isn’t much different from the database query language [SQL](http://en.wikipedia.org/wiki/SQL).

**Prolog Relations**

Relations are defined by means of clauses and lucky us, we only have two types of clauses: facts and rules!

**Prolog Facts**

Facts are nothing more than statements, they are the truths of our program and have a very simple syntax. They always **start with a lowercase letter and end with a period**.

|  |
| --- |
| sunny.  logic\_programming\_is\_cool.  tomorrow\_will\_rain. |

So far everything that Prolog knows is that it’s sunny, that logic programming is cool and that tomorrow will rain. Facts can also have arguments in the form relation(argument1, argument2, ... argumentN).

|  |
| --- |
| likes(alice,bob).  likes(bob,carol).  likes(james,mary).  likes(mary,james). |

The first line is for example a relationship that links Alice and Bob. We are free to pick the interpretation of our relationships, as long as we are consistent. This means you can’t change the interpretation from one line to another! In our case we will read it as  ‘Alice likes Bob’, ‘Bob likes Carol’, etc.

**Prolog Shell and Queries**

So, to start doing something interesting, we need to learn how to start writing our queries. If you write a file called *facts.pl*containing the four clauses above, you can load it on your shell by calling

|  |
| --- |
| consult('facts.pl'). |

We can ask Prolog if alice likes bob.

|  |
| --- |
| ?- likes(alice,bob). */\* our first query! \*/*  true. */\* Prolog matches it with the known fact that alice likes bob. \*/* |

where ?- is the Prolog Shell. Let’s make some more queries.

|  |
| --- |
| ?- likes(bob,alice).  false. */\* poor alice isn't liked by bob :( \*/*  ?- likes(mary,john).  false. */\* we don't even have a John on our list! \*/* |

Simple, right?

**Prolog queries in a terminal**

It is important to note that [on the shell you can only make queries](http://www.swi-prolog.org/FAQ/ToplevelMode.html), you are not allowed to declare new facts or rules. The best way to load your relations is using the consult command, as mentioned above.

**Variables**

Before we get to rules, let’s introduce variables. How do we know who Alice likes? One idea would be to run the following query

|  |
| --- |
| ?- likes(alice,who).  false. */\* well, we did ask if alice liked a person named who \*/* |

who will not match bob because who is not a variable. Variables in Prolog **start with an uppercase letter**.

|  |
| --- |
| ?- likes(alice,Who). */\* Who does alice like? \*/*  Who = bob. */\* yes! we got it! \*/* |

Finally we did something useful with Prolog! Did you notice the key difference in how a variable works between the logic paradigm and the other paradigms? On the imperative, OO and functional paradigms we always have to say exactly how a variable is defined. On logic programming we are allowed to pass un-instantiated arguments and the interpreter will try to instantiate the variables for us respecting the facts previously defined. This powerful process of matching variables with items is known as [unification](http://en.wikipedia.org/wiki/Unification_%28computer_science%29) and is exactly where logic programming shines. I’m not going to explain on this post how unification works as this is a complex topic on its own, so for now let’s just say Who was unified and is now bound to bob.

**Rules**

Now that we have learned how to express facts and how to query them using variables, we can take a look at rules. Rules are a key concept in Prolog and allow us to make conclusions about our world. A rule has the form

conclusion(arg1, arg2, ... argN) :- relation1, relation2, ... relationN.

The conclusion is only valid if all the relations are also true. Commas work exactly like the logical and. So this can be read as conclusion is true if everything that comes after the :- can also be proven true. We call what comes before the :- head, and what comes after, body. The next examples will help understanding this new concept, so don’t worry if you didn’t get this.

Suppose we want to create a matchmaking agency, so let’s write a nice rule called love\_compatible using the facts we already defined above

love\_compatible(X, Y) :- likes(X, Y), likes(Y, X).

This can be read as: X and Y are a love\_compatible **if** X likes Y **and** Y likes X. An equivalent intepretation would be: to prove that X and Y are love\_compatible, prove that X likes Y and that Y likes X. Now let’s make some queries

|  |
| --- |
| ?- love\_compatible(james,Who). */\* Is james compatible with someone? \*/*  Who = mary. |

In the example above we forced the X argument of love\_compatible to be james, but it is important to note that we don’t have to instantiate all arguments.  In fact, we don’t have to instantiate any parameters. We can pass variables to all arguments and Prolog will work its magic. It will not only find the first match, but all matches that exist!

|  |
| --- |
| ?- love\_compatible(X,Y). */\* Prolog, please find me all love pairs with the facts you know! \*/*  X = james,  Y = mary |

Because of inherent symmetry in our rule love\_compatible, Prolog will actually find two matches for our world.

|  |
| --- |
| ?- love\_compatible(X,Y).  X = james,  Y = mary ; */\* typing semicolon causes Prolog to find the next match \*/*  X = mary,  Y = james. */\* our love pairs are (X=james, Y=mary) and (X=mary, Y=james) \*/* |

You may be wondering how this is even working. As I’ve mentioned before, Prolog builds in Backtracking automatically for you and will do an exhaustive search until it finds a match. I’ll repost our clauses and rules below to easen the comprehension.

|  |
| --- |
| likes(alice,bob).  likes(bob,carol).  likes(james,mary).  likes(mary,james).  love\_compatible(X, Y) :- likes(X, Y), likes (Y, X). |

When we call love\_compatible(X,Y)., the first goal is to match likes(X, Y). As X and Y weren’t instantiated yet, Prolog will simply pick the first clause of likes, likes(bob,carol).. This binds X to bob and Y to carol. The second goal is likes(Y, X). This time X and Y are already instantiated to bob and carol respectively. Prolog then checks in it’s world of knowledge if carol likes bob, that is, if it has a clause in the form likes(carol, bob). This is not the case, so it fails and Prolog backtracks automatically to the first goal, X and Y become uninstantiated again. Prolog then tries the second clause of likes and binds X to james and Y to mary. The second goal then also succeeds because likes(mary,james)., resulting in our first match.

Let’s take a look at a more complex example. First we define some facts

|  |
| --- |
| mother(alice,lea).  mother(john,julia).  mother(lea,alberta).  father(james,alfred).  father(lea,john). |

The first line can be read as alice‘s mother is lea and the fourth as james‘ father is alfred. Now let’s define some rules.

|  |
| --- |
| parent(X, Y) :- father(X, Y).  parent(X, Y) :- mother(X, Y). |

Prolog has no problem having multiple definitions of a rule, so if the first clause of parent fails, it will try the second one.  We can also define a rule in term of other rules. In fact, for Prolog there isn’t any difference between a fact and a rule. A fact is simply a rule that is always true. Let’s define a rule using other rules.

|  |
| --- |
| grandparent(X, Y) :- parent(X, Z), parent(Z, Y). |

This can be read as X‘s grandparent is Y. Did you follow the logic of grandparent? Get a piece of paper and you’ll be able to easily follow this. Z is X‘s parent and Y is Z‘s parent, therefore Y is X‘s grandparent. Time to use our new rule on the shell!

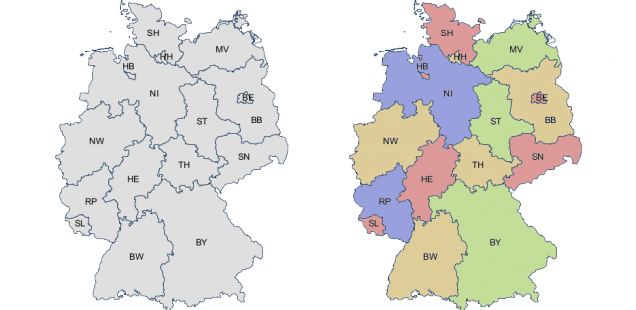
|  |
| --- |
| ?- grandparent(X, Y).  X=alice,  Y=alberta ; */\* alberta is alice's grandma \*/*  X=alice,  Y=john. */\* john is alice's grandpa \*/* |

Isn’t it amazing that we didn’t even have to tell Prolog how to solve this? We only defined very intuitively what family relations are and it finds the matches for us.

We now have all the tools required to start solving real world problems, let’s get cracking!

**The Graph Coloring Problem**

Given a map divided into regions, can you color the map using a defined amount of colors such that no two adjacent regions have the same color? In the image below, we start with the left map, uncolored, and try to find a map coloring using only four different colors. The right map is one of the possible solutions.

[](https://bernardopires.com/wp-content/uploads/2013/09/map_coloring_germany.png)

The map coloring problem applied to Germany and it’s states. Click on it for a larger version.

First things first, let’s get our facts straight (pun intended)!

|  |
| --- |
| color(red).  color(green).  color(blue).  color(yellow). |

Now Prolog knows that the four colors red, green, blue and yellow exist. So how do we express that no two adjacent states in our map can share the same color? We create a rule.

|  |
| --- |
| neighbor(StateAColor, StateBColor) :- color(StateAColor),  color(StateBColor),  StateAColor \= StateBColor. */\* \= is the not equal operator \*/* |

Alright, so what’s happening here? The first two clauses, color(StateAColor) and color(StateBColor) simply associate colors to our variables StateAColor and StateBColor. The interesting part is the third clause, StateAColor \= StateBColor, that forces the two states to have different colors.

The only thing we’re missing now is a relation that defines the topology of our map, that is, which states are adjacent. Looking at the bottom of our map we can see for example that the states BW and BY are adjacent. So we start writing

|  |
| --- |
| germany(BW, BY) :- neighbor(BW, BY). |

BW is however also adjacent to RP and HE. Let’s expand germany.

|  |
| --- |
| germany(BW, BY, SL, RP, HE) :- neighbor(BW, BY), neighbor(BW, RP), neighbor(BW, HE). |

When we add in all adjacencies we have

|  |
| --- |
| germany(SH, MV, HH, HB, NI, ST, BE, BB, SN, NW, HE, TH, RP, SL, BW, BY) :-  neighbor(SH, NI), neighbor(SH, HH), neighbor(SH, MV),  neighbor(HH, NI),  neighbor(MV, NI), neighbor(MV, BB),  neighbor(NI, HB), neighbor(NI, BB), neighbor(NI, ST), neighbor(NI, TH),  neighbor(NI, HE), neighbor(NI, NW),  neighbor(ST, BB), neighbor(ST, SN), neighbor(ST, TH),  neighbor(BB, BE), neighbor(BB, SN),  neighbor(NW, HE), neighbor(NW, RP),  neighbor(SN, TH), neighbor(SN, BY),  neighbor(RP, SL), neighbor(RP, HE), neighbor(RP, BW),  neighbor(HE, BW), neighbor(HE, TH), neighbor(HE, BY),  neighbor(TH, BY),  neighbor(BW, BY). |

Alright, so how do we get a valid map coloring of germany now? Simple!

|  |
| --- |
| ?- germany(SH, MV, HH, HB, NI, ST, BE, BB, SN, NW, HE, TH, RP, SL, BW, BY). |

Just execute the query above on your Prolog shell! There you go, a solution to the graph coloring problem using only 6 lines (4 facts, 2 rules).

**So how do we test if we can get a coloring with three colors?**