Logic Programming in Prolog

By CNMPeralta, Revised 1st Semester, A.Y. 2020-2021

1 What is Prolog?

Prolog (short for **Pro**gramming **log**ic) is a high-level programming language conceived by Alain Colmerauer and Phillipe Roussel, both from the University of Aix-Marseille, and Robert Kowalski from the University of Edinburgh, and was finalized in 1972. Its main application is in artificial intelligence.

Unlike most programming languages, it is *declarative*, that is, it *specifies a goal*, but *not how to achieve* it. In contrast, *imperative languages* (like C) specify a *sequence of steps* to achieve a goal.

To install Prolog on your computer, look for the `swi-prolog` package:

```
$ sudo apt-get install swi-prolog
```

And to run the Prolog interpreter, open a terminal and enter:

```
$ swipl
```

To terminate a Prolog interpreter, enter:

```
?- halt.
```

The `(.)` at the end of the command, serves as the terminating symbol.

2 What are Prolog programs?

Prolog programs are made up of *facts* and *rules* that make up a *knowledge base*. They are used by posing *queries*, which are answered using information in the knowledge base.

For example, we have the following first knowledge base, which we will call `icsfacts1.pl` [1]:

```
1  female(kat).
2  female(dja).
3  female(berna).
4  female(monina).
5  male(regi).
6  male(clinton).
7  male(perico).
8  male(jm).
9  playsViolin(dja).
10 party.
```

In `icsfacts1`, lines 1-10 contain *facts*.

To load a knowledge base, enter the command `[KnowledgeBaseName]` in the interpreter shell:

```
?- [icsfacts1].
```

Once a knowledge base is loaded, you can pose *queries* to the Prolog interpreter, which it will then answer using the information in the loaded knowledge base.

```
?- female(kat).
true.
```

`female(kat)` is true because it is a *fact*.

```
?- female(jm).
false.
```

`female(jm)` is false because there is no indication that JM is a woman.

[1]Note that Prolog and Perl files have the same file extensions

```
?- playsViolin(berna).
false.
```

`playsViolin(berna)` is false because there is no indication that Berna plays violin.

```
?- playsViolin(chris).
false.
```

`playsViolin(chris)` is false mainly because Prolog does not know who Chris is; he is not in the knowledge base that was loaded.

```
?- party.
true.
```

`party` is true because it is a fact in the knowledge base.

```
?- foamParty.
ERROR: Undefined procedure: foamParty/0 (DWIM could not correct goal)
```

`foamParty` produces an error because it is not defined in the knowledge base.

```
?- sings(kat).
ERROR: Undefined procedure: sings/1 (DWIM could not correct goal)
```

`sings()` produces an error because it is not defined in the knowledge base, even if kat is.

Now, let's take a look at another knowledge base, which we will name `icsfacts2.pl`:

```
1 happy(regi).
2 listensToMusic(kat).
3 listensToMusic(regi):- happy(regi).
4 lipSyncs(kat):- listensToMusic(kat).
5 lipSyncs(regi):- listensToMusic(regi).
```

In `icsfacts2`, there are only two *facts* on lines 1 and 2: `happy(regi)` and `listensToMusic(kat)`. Lines 3-5 contain *rules*.

Rules are statements that are conditionally true. The rule on line 3 means, "if regi is happy, then regi listens to music". Generally, `:-` should be read as "if" or "is implied by". The *left-hand side* of the `:-` is called the *head*, and *right-hand side* is called the *body*.

Now, if we query:

```
?- lipSyncs(kat).
```

Prolog will answer true even though there is no fact in `icsfacts2` that says Kat lip syncs. However, `listensToMusic(kat)` is a fact, and `lipSyncs(kat):- listensToMusic(kat)` is a rule in `icsfacts2`. Since the body of the rule is true, then the head is true by *modus ponens*.

Now, what about the query:

```
?- lipSyncs(regi).
```

Prolog will again answer true, even though neither `lipSyncs(regi)`, nor `listensToMusic(regi)` are facts in the knowledge base. This is because `happy(regi)` is a fact, and it is the body of the rule `listensToMusic:- happy(regi)`, allowing Prolog to deduce that `listensToMusic(regi)` is true. In turn, this is the body of the rule `lipSyncs(regi):- listensToMusic(regi)`, allowing Prolog to conclude that `lipSyncs(regi)` is true.

In plain English, the chain of reasoning is: Regi is happy (fact), so he listens to music (inferred from line 3 rule). Since Regi listens to music, Regi lip syncs (inferred from line 5 rule).

Knowledge bases are made up clauses; both facts and rules can be clauses in a knowledge base. Actually, we can think of facts as rules that have no conditions; that is, they do not depend on anything else to be true.

Now make another knowledge base, `icsfacts3.pl`, and enter the following clauses:

```
1 happy(dja).
2 playsDota(clinton).
3 winsDotaGame(dja):- happy(dja),
4                     playsDota(dja).
5 winsDotaGame(clinton):- happy(clinton).
6 winsDotaGame(clinton):- playsDota(clinton).
```

In `icsfacts3`, there are five clauses (two facts and three rules).

Something new in this knowledge base is the use of a *comma* in the rule spanning lines 3 and 4. This is how Prolog expresses *logical conjunctions (and)*. That is, the rule should be read as, “Dja wins a Dota game if she is happy and she plays Dota”. Therefore, if we ask Prolog:

```
?- winsDotaGame(dja).
```

Prolog will answer false. In order for `winsDotaGame(dja)` to be true, both `happy(dja)` and `playsDota(dja)` must be true, but though `happy(dja)` is a fact, we can’t infer `playsDota(dja)` anywhere.

To express a *logical disjunction* (or), two rules with the same head must be specified. Looking at the rules on lines 5 and 6, we can see two rules with the head `winsDotaGame(clinton)`. The first such rule has a body, `happy(clinton)`, while the second has, `playsDota(clinton)`. Thus, in plain English, the logically disjuncted rule is, “Clinton wins in Dota if he is happy or he plays Dota”. If we then query:

```
?- winsDotaGame(clinton).
```

Prolog will answer true because even though Clinton is not stated to be happy, he is playing Dota, as per line 2, allowing Prolog to use the rule on line 6.

An alternate way to express logical disjunction is:

```
winsDotaGame(clinton):- happy(clinton);
                        playsDota(clinton).
```

Instead of using a comma (used for logical conjunction), use a semicolon to denote a logical disjunction within one rule.

Now make another knowledge base, name it `icsfacts4.pl`:

```
1 animal(cat).
2 animal(dog).
3 animal(hamster).
4 loves(kat, dog).
5 loves(berna, cat).
```

In `icsfacts4`, the knowledge base does not have any rules, but it does have facts that uses a *relation*, `loves`, involving two entities. We can read line 4 as “Kat loves dog”, and line 5 as “Berna loves cat”.

However, our main object of interest is the set of queries that we can use on this knowledge base:

```
?- animal(X).
```

The query uses a *variable*, `X`, and means, “which of the entities in the knowledge base are animals?” Prolog will then look at its clauses and see the first fact, `animal(cat)`, and *unifies* `cat` with `X`. It outputs:

```
X = cat
```

But since `cat` is not the only entity declared to be an animal, you can ask Prolog to show you other possible values that can be unified with `X` by entering a semicolon (`;`) like:

```
X = cat ;
X = dog ;
X = hamster.
```

The first semicolon will prompt Prolog to say `X` can also be `dog`, and the second semicolon will allow it to say `X` can also be a `hamster`. Thus, there are three entities that can be unified with the variable `X`.

But what if we make our query more complicated:

```
?- loves(berna, X), animal(X).
```

This asks Prolog what entity, `X`, does Berna love, that is also an animal (remember, commas are logical conjunctions!). Looking through the database, it will see `loves(berna, cat)`. It will then check if `cat` is an animal, which it finds to be true in line 1. Thus, Prolog will say:

```
X = cat.
```

No other entities can be unified with `X`, as Berna does not love dogs nor hamsters.

The last knowledge base, for now, will be named (dun, dun, duuuuun...) `icsfacts5.pl`:

```
loves(kei, bulacs).
loves(ippo, bulacs).
jealous(X,Y):-loves(X,Z),loves(Y,Z).
```

In `icsfacts5`, we have two facts using the `loves` relation, stating "Kei loves Bulacs", and "Ippo loves Bulacs". We also have a rule defining the concept of jealousy, which states that if entity `X` and entity `Y` love `Z`, `X` will be jealous of `Y`. The use of variables makes this rule *general*; that is, there is no specific individual that the rule applies to.

We can then ask Prolog:

```
?- jealous(ippo, W).
```

And Prolog will say:

```
W = kei
```

But that's not all! Ippo is jealous of Kei because they both love Bulacs, but, if you enter a semicolon, you will find that Prolog thinks that Ippo is also jealous of himself. Why is that?

```
W = kei ;
W = ippo.
```

This is because Prolog exhausts all possible ways to answer a query.

In order to force Prolog to stop after the first match, use a `!` in the query.

```
?- jealous(ippo, W), !.
W = kei.
```

3 Prolog Syntax

We have already programmed some knowledge bases in Prolog and posed queries on those knowledge bases using the Prolog interpreter. But what exactly are Prolog's rules when it comes to defining facts, rules, and queries?

All of Prolog's facts, rules, and queries are made up *terms*, and there are four kinds of terms: atoms, numbers, variables, and complex terms (or structures). We will take a look at each of them and how they are used.

A. Atoms

An *atom* can be defined using three ways:

1. Start with a lower-case letter, followed any combination of other letters (upper- and lower-case), digits, and underscores. Examples are from our previous knowledge bases are: `kei`, `kat`, `loves`, `animal`.
2. A sequence of characters enclosed in single quotes. Spaces are allowed in these kinds of atoms. Examples are: `'Juan Miguel'`, `'!@#$%^&'`, and `' '`.
3. A string of special characters, like `:-`, `@=`, and `=====>`. Some atoms of this kind have special meanings, like `:-`.

B. Numbers

Prolog supports both real numbers and integers, though real numbers are not often used. Either way, numbers are represented as usual, like 43, 13, -153, and so on.

C. Variables

Variables start with an upper-case letter or underscore, and may be followed by letters (upper- and lower-case), digits, and underscores. Examples of variables are `X`, `Y`, `Z`, `_tag_`, `_head`, and so on.

D. Complex Terms

Complex terms are made up of atoms, numbers, and variables, and are also called *structures*. They are made up of a *functor* and a list of arguments enclosed in parentheses like:

```
functor(arg1, arg2, ...)
```

Functors are always atoms; variables and numbers cannot be used as functors. However, arguments may be any kind of term, that is, they can be atoms, variables, numbers, or other complex terms. There can be *no space* between the functor and its list of arguments.

We have used numerous complex terms in our knowledge bases, like `female(kat)`, `animal(dog)`, `male(jm)`, and `loves(berna, cat)`. The number of arguments a functor takes is called its *arity*. Thus, the functors `female`, `animal`, `male` all have arity 1, while `loves` has arity 2. The combination of a functor and its arity, commonly notated as `functor/arity` is usually referred to as a *predicate*. Thus, we have previously defined the predicates `female/1`, `animal/1`, `male/1`, and `loves/2`, among others. It is quite possible to use an existing functor with a different arity, like defining `loves/3` will not conflict with `loves/2`.

Recursive structures can be achieved by nesting complex terms within each other:

```
animalia(felis(silvestris(catus(cat)))).
```

4 Unification

Unification is one of Prolog's main programming mechanisms. There are three clauses to consider when two terms are to be unified, and one catch-all clause:

1. If `term1` and `term2` are constants (not variables), then they unify if and only if they are the exactly the same. For example, two atoms, `kei` and `kei`, unify, as do two numbers, `124` and `124`.
2. If `term1` is a variable and `term2` is a constant, then `term1` is instantiated to `term2`, and they can be unified. This works both ways.
3. If `term1` and `term2` are complex terms, then they unify if and only if:
   (a) They have the same functor;
   (b) All *corresponding* arguments unify; and
   (c) Variables are instantiated only once (if it is instantiated to unify for a term, it can't be instantiated for another term).
4. If two terms can't be unified by the first three ways, then they can't be unified.

To test if two terms unify, the predicate `=/2` can be used:

```
% Example for Clause 1
?- =(kei, kei).
true.

?- =(124, 124).
true.
```

```
% Example for Clause 2
?- =(X, kei).
X = kei.

?- =(X, Y).
X = Y.
```

```
% Example for Clause 3
?- =(woman(X), woman(kei)).
X = kei.

?- =(loves(X, Y), loves(kei, bulacs)).
X = kei,
Y = bulacs.
```

```
% Example for Clause 4
?- =(X, kei), =(X, kat).
false.

?- =(loves(X, X), loves(kei, bulacs)).
false.
```

Prolog also allows the usage of `=` as an infix operator:

```
?- kei = kei.
true.

?- 124 = 124.
true.

?- X = kei.
X = kei.

?- X = Y.
X = Y.

?- woman(X) = woman(kei).
X = kei.

?- loves(X, Y) = loves(kei, bulacs).
X = kei,
Y = bulacs.

?- X = kei, X = kat.
false.

?- loves(X, X) = loves(kei, bulacs).
false.
```

NOTE: Variables, though not exactly the same, will usually unify with each other, but cannot be unified with two different constants in the same query.

Prolog will also sometimes make deductions based on information in your query:

```
?- X = Y, X = Z.
X = Y, Y = Z.
```

What if we use complex terms like:

```
?- k(s(g), Y) = k(X, t(k)).
Y = t(k),
X = s(g).
```

`Y` is instantiated to `t(k)`, and `X` is instantiated to `s(g)`, allowing Prolog to unify `k(s(g), Y)` and `k(X, t(k))`.

Now, take another example of using unification by creating another knowledge base, call it `lines.pl`:

```
vertical(line(point(X, Y), point(X, Z))).
horizontal(line(point(X, Y), point(Z, Y))).
```

This knowledge base contains no rules, only two facts. The first fact, `vertical`, is a complex term that takes another complex term, `line` as an argument. In turn, `line` takes two complex terms as inputs, both of the point/2 predicate. The second fact, horizontal, is similar; the two rules differ in the variables that are passed as parameters to the various points in their definition. For vertical lines, the first arguments of the point should be the same, but for horizontal lines, the second arguments of the points should be the same.

If we then ask:

```
?- vertical(line(point(1, 1), point(1, 3))).
true.

?- vertical(line(point(1, 1), point(3, 2))).
false.

?- horizontal(line(point(1, 2), point(3, 2))).
true.

?- horizontal(line(point(1, 1), point(1, 3))).
false.
```

Prolog can tell if a line is vertical or horizontal based on the rule.

```
?- horizontal(line(point(1, 1), point(2, Y))).
Y = 1.
```

Here, we left the second argument of the second `point` predicate as a variable, Prolog figured out that the only way to make the line horizontal is to unify `Y` with `1`.

```
?- horizontal(line(point(2, 3), P)).
P = point(_G1614, 3).
```

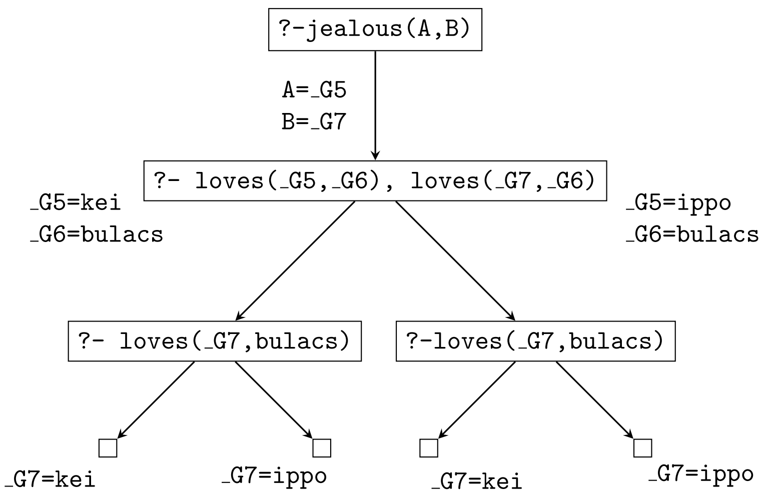
Lastly, we left the second `point` predicate out completely, instead using the variable `P`, Prolog returns with `point(_G1614, 3)`. This is Prolog's way of telling us that any `point`, as long as the second argument is 3, will do. `_G1614` is a variable that Prolog uses internally, and your own execution of the query may use a different variable.

5 Proof Search

Let's now look at how Prolog comes to its conclusions when it answers queries. Say we take `icsfacts5` and make a query:

```
?- jealous(A, B).
A = B, B = kei;
A = kei,
B = ippo;
A = ippo,
B = kei;
A = B, B = ippo.
```

The search tree would look like:



Prolog comes up with four possible answers to `jealous(A, B)`:

1. `kei` is jealous of `kei`.
2. `kei` is jealous of `ippo.`
3. `ippo` is jealous of `kei.`
4. `ippo` is jealous of `ippo`.

But how? First, it produces `loves(_G5, _G6), loves(_G7, _G6)` by the `jealous` rule. Remember that `_G5`, `_G6`, and `_G7` are all internal variables used by Prolog. It then attempts to resolve `loves(_G5, _G6)` by looking at the existing facts: `loves(kei, bulacs)`, and `loves(ippo, bulacs)`. It can use the *second unification clause* and sees two possibilities: (1) where `_G5` is `kei` and `_G6` is `bulacs`, and (2) another where `_G5` is `ippo` and `_G6` is `bulacs`, hence the branching in the search tree. It applies `_G6` as `bulacs` to the second `loves` predicate, thus, `loves(_G7, _G6)` → `loves(_G7, bulacs)`. It then looks through the knowledge base for something to unify with `_G7`. Since `_G7` is the first argument of a `loves` predicate, it looks at possible first arguments for `loves` and finds two: `kei` and `ippo`. The tree then branches out from each of the existing branches again, with two possibilities: (1) `_G7` is `kei` and `_G7` is `ippo`.

References

- Blackburn, P., Bos, J., and K. Striegnitz. 2001. *Learn Prolog Now!* Retrieved from `http://www.learnprolognow.org/lpnpage.php?pageid=online` on 16 October 2015.