COMP304 Tutorial 7
09/09/2016

1 Getting Started

There are several implementations of Prolog, such as GNU Prolog and SWI-Prolog. SWI-Prolog would probably be the best to use, as it comes with a lot of built-in predicates and an interactive interpreter. We'll run our programs by writing them in `.pl` files and loading them into SWI-Prolog.

Note: `.pl` is often used as an extension for Perl files (another programming language), so your text editor might try to do Perl highlighting instead of Prolog highlighting. If you're using Gedit you can fix this by going View → Highlight Mode → Sources → Perl.

We'll add some facts to our file.

```
1 bird(kiwi).
2 bird(tui).
3 bird(sparrow).
4 bird(flamingo).
5 flightless(kiwi).
6 pink(flamingo).
```

The fullstops are important! They mark the end of predicat definitions (a bit like how semi-colons mark the end of statements in imperative languages). Also, whitespace inside brackets is significant. The atom ' tui' is not the same as 'tui'.

From inside SWI-Prolog we can load the file and put forward some queries:

```
1 ?- [birds].
2 % birds compiled 0.00 sec, 5 clauses
3 true.
4
5 ?- bird(sparrow).
6 true.
```

To solve this query, Prolog attempts to match `bird(sparrow)`. to the facts in its knowledge-base, top-to-bottom. Here's what happens if we pose something not in its knowledge-base.

```
1 ?- bird(kookaburra).
2 false.
```

At its heart, Prolog is a theorem-prover. It proves a predicate is true by constructing an example which satisfies that predicate. If it can't, it considers it to be false. We can ask Prolog for things which are birds by asking it for `bird`(X). X is an unbound/free

variable (these start with upper-case letters or an underscore). It will then try to find values to X, which satisfy the predicate `bird/1`.

```
?- bird(X)
X = kiwi ;
X = tui ;
X = sparrow.
```

Each solution will be listed one-by-one, in order that you specified in the file (because Prolog works top-to-bottom). To see the next solution, you have to enter ";". The solutions are terminated in a full-stop (if you want to terminate early, you can enter a full-stop instead of a semi-colon).

We can ask for something which is pink, or ask for something which is flightless.

```
?- pink(X)
X = flamingo.

?- flightless(X)
X = kiwi.
```

If we want to ask for something which is pink OR flightless, we can use a semi-colon to join the two.

```
?- pink(X); flightless(X).
X = kiwi ;
X = flamingo.
```

We can use a comma to ask if something is both pink AND flightless. Prolog tells us no.

```
?- pink(X), flightless(X).
false.
```

2 Predicates & Relations

Recall that an m-ary relation is a set of tuples of size m (over some sets). For example, $\{(2,0),(1,2),(2,3)\}$ is a relation over pairs of numbers. This is the same as a Prolog predicate. In the previous example, `bird` is a 1-ary (unary) relation over atoms. Its members are `kiwi`, `tui`, `sparrow`, `flamingo`. When we asked Prolog for `bird`(X) it tried to answer by producing a member in set.

Here's an example of a 2-ary (binary) predicate.

```
mother(mary, amber).
mother(mary, rachel).
mother(mary, aj).
mother(tuhi, mary).
```

`mother/2` is a 2-ary predicate (Prolog predicates are always described by their name and arity). The members in the `mother` relation are $\{$(`mary`, `amber`), (`mary`, `rachel`), (`mary`, `aj`) (`tuhi`, `mary`)$\}$

Some notes and cautionaries.

1. The order in pairs is significant. `mother(mary, aj)` is NOT the same as `mother(aj, mary)`.

2. Beware of spelling mistakes! If you type in `mohter(aj, mary)`, it will register a new relation `mohter`. It doesn't know you made a typo.

If we wanted to explicitly list all the (mother, child) relations we can do this.

```
?- mother(X, Y).
X = mary,
Y = amber ;
X = mary,
Y = rachel ;
X = mary,
Y = aj ;
X = tuhi,
Y = mary.
```

Notice this time it returns pairs of results, because it's trying to bind an atom to X (the mother) and an atom to Y (the child). Every successful way of doing this yields the relation.

If we used X twice instead of using X and Y it would try to find an X satisfying `mother(X, X)`. In other words, a person who is their own mother. Prolog will tell you that's false. Sometimes it makes sense to be a relation with yourself though. For example, equality on numbers: every number is equal to itself.

Lastly, even though we asked for solutions X and Y, the fact that we used different names doesn't mean that X and Y have to be distinct. Sometimes $X = Y =$ `something` is a solution (but not here).

We can also partially specify the arguments and let Prolog fill in the rest. Doing this, we can see who Mary's children are, and ask who Mary's mother is.

```
?- mother(mary, Y).
Y = amber ;
Y = rachel ;
Y = aj.

?- mother(X, mary).
X = tuhi.
```

3 Predicate Bodies

Let's say we wanted to add the relation `grandmother/2` relation. One way to do it is to enumerate all the possible pairs, but that's tedious (and can't always be done if there are infinitely many solutions). Here's a better way.

```
grandmother(X,Z) :- mother(X, Y), mother(Y, Z).
```

`grandmother/2` is our first non-trivial predicate. The `bird/1` and `mother/2` predicates were just defined by explicitly listing the members of the relation, but here we've given

a rule the members in the relation satisfy.

The :- symbol separates the name and arguments of the predicate (left-hand side) from the body (right-hand side). The name of a predicate is called its *functor*. So the functor of `grandmother/2` is `grandmother`.

(Warning: if googling for information on Prolog, beware that the word functor has a more general meaning in category theory and Haskell).

Here's how `grandmother/2` works: it tries to bind X, Y, and Z to values which satisfy both `mother(X, Y)` and `mother(Y, Z)`. This is done left-to-right, so it first finds an X and Y such that `mother(X, Y)`, and then finds a Z such that `mother(Y, Z)`. When Prolog finds such a triple it spits out an answer.

If you ask Prolog for another answer it unbinds the last variable that was bound and tries to find another value (in this case, that's Z). If it can't do that, it unbinds the second-last variable and tries to find another value for that, and so on. This is why, when we query `mother(X, Y)` it lists all the pairs `(mary, Y)` before listing any of the pairs `(tuhi, Y)`.

4 More Predicates

Let's add some more facts to our file and write some more fancy predicates.

```
father(john, aj).
father(john, rachel).
father(john, amber).
mother(edith, john).
```

Our `grandmother/2` is really the "mother's mother" relation. `grandmother(X, Z)` should be true if X is the mother of a parent of Z (not just the mother of the mother of Z). So let's define a more general `parent` predicate, by ORing together `mother` and `father`. Then we can give a better `grandmother` predicate.

```
parent(X, Y) :- mother(X,Y) ; father(X,Y).
grandmother(X, Z) :- mother(X, Y), parent(Y, Z).
```

```
?- grandmother(X, aj).
X = tuhi ;
X = edith.
```

What about siblings? Two people should be in the sibling relation if there is an X which is a parent of both of them.

```
sibling(X, Y) :- parent(M, X), parent(M, Y).
```

```
sibling(X, aj).
X = aj ;
X = amber ;
X = rachel ;
false.
```

That's weird: I am my own sibling. This is because there's no constraint saying X and Y have to be distinct. We can add that on the end like this.

```
sibling(X, Y) :- parent(M, X), parent(M, Y), X \= Y.
```

```
?- sibling(X, aj).
X = amber ;
X = rachel ;
false.

?- sibling(X, mary).
X = lance ;
false.
```

5 Types and Terms

Prolog has a very minimal type-system. There are four types:

1. Numbers.
2. Atoms. These are things such as `kiwi`, `tui`. This is a sequence of characters starting with a lower-case letter. You can kind of think of it as like a string, but it's NOT a string. It's a sequence of characters, or a name. You can also specify atoms with single quotes.
3. Variables. A sequence of characters which starts with an underscore or a capital letter. Variables are either bound or unbound.
4. Predicates (and also other compounds, like lists).

Prolog is pretty implicit about types, and unlike Haskell the types of things won't really dominate your thoughts.