COMP304 Tutorial 4
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.

```
bird(kiwi).
bird(tui).
bird(sparrow).
bird(flamingo).
flightless(kiwi).
pink(flamingo).
```

We can note a few things already. The fullstops are important! They mark the end of predicates (a bit like semi-colons 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. We do that like so:

```
?- [birds].
\% birds compiled 0.00 sec, 5 clauses
true.

?- bird(sparrow).
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.

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

At its heart, Prolog is a theorem-prover. It tries to show things are true. 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 variable. These start with upper-case letters. It will then try to bind values to X, which satisfy the predicate `bird`.

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

Each solution will be listed one-by-one. 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 the semi-colon to join the two predicates.

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

In the same way, we can use a comma to join two predicates and ask if something is both pink AND flightless, Prolog tells us no.

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

If we want to check if something is false, we use the `not` predicate (there are other ways which help manipulate the search-tree, but `not` will do for now).

```
?- not(pink(X)).
false.
```

Does this seem weird? Sparrows aren't pink, so you might think Prolog could bind X to `sparrow`. Recall what negation means in Prolog: not P is true iff P cannot be proven true. Because we can prove `pink(X)`, that means `not(pink(X))` is false.
We can also feed a compound predicate into `not`. If want to check if not (flightless $\wedge$ pink) is "false" you might try this:

```
?- not(flightless(X), pink(X)).
ERROR: Undefined procedure: not/2
ERROR: However, there are definitions for:
ERROR: not/1
false.
```

Prolog is trying to pass in `flightless(X)` and `pink(X)` as two separate predicates to `not`. The comma here means "separate the predicate arguments". If we want to pass them in as a single, conjunct predicate, we need an extra pair of brackets around them.

```
?- not((flightless(X), pink(X)))
true.
```

This is the expected result, because Prolog cannot prove that anything is flightless and pink.

2 Predicates & Relations

Recall that an $m-ary$ relation is a set of tuples of size m (over some set). For example, $\{(2,0),(1,2),(2,3)\}$ is a relation over pairs of numbers. Prolog predicates are the same. We specify predicates by their name and arity. For example:

```
1 mother(mary, amber).
2 mother(mary, rachel).
3 mother(mary, aj).
4 mother(tuhi, mary).
```

`mother/2` is a 2-ary predicate/relation. The functor is `mother` and the arity is 2. If we wrote this using mathematical notation, we can define the relation as the following set: $\{(\texttt{mary},\texttt{amber}),(\texttt{mary},\texttt{rachel}),(\texttt{mary},\texttt{aj}),(\texttt{mary},\texttt{tuhi})\}$.

Some notes and cautionaries.

1. The order in pairs is significant. $\texttt{mother}(\texttt{mary},\texttt{aj})$ is NOT the same as $\texttt{mother}(\texttt{aj},\texttt{mary})$.
2. Beware of spelling mistakes! If you type in $\texttt{mohter}(\texttt{aj},\texttt{mary})$, it will register a new relation `mohter`. It doesn't know you made a typo.
3. The word "functor" means something different in category theory and Haskell, so if you google for "functor" while looking to learn Prolog you might get some confusing results.

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

```
1 ?- mother(X, Y).
2 X = mary,
3 Y = amber ;
4 X = mary,
5 Y = rachel ;
6 X = mary,
7 Y = aj ;
8 X = tuhi,
9 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 $\texttt{mother}(\texttt{X},\texttt{X})$. In other words, a person who is their own mother. Prolog will tell you that's false (unless we added in as a fact that someone was their own mother).

Sometimes it makes sense to be a relation with yourself though. For example, equality on numbers: every number is equal to itself (the functor is equality, the arity is 2). 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 $X = Y = \texttt{something}$ is a solution (not here though).

We could partially specify the arguments, and leave the rest to be filled in. 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.
```

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 listing pairs, but `grandmother/2` is defined in terms of other predicates (a relation between relations, I guess). The :- symobl separates the name and arguments of the predicate (left-hand side) from the body (right-hand side).

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 tries `mother(X, Y)` and then `mother(Y, Z)`. It then looks for other solutions by "backtracking", unbinding the last bound variable (Z in this case) and then trying to find a new solution. This is why, when we query `mother(X, Y)`, it lists all the pairs `(mary, Y)` before listing any of the pairs `(tuhi, Y)`.

3 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.

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