COMP304 Tutorial 10
30/09/2016

1 Negation

Something is "true" in Prolog if it can prove that it exists from its knowledge-base. Otherwise, it is false. Negation works a bit weird on this definition. Take the following:

```
1 bird(kiwi).
2 bird(albatross).
3 flightless(kiwi).
```

If we want to negate something we may use the `not/1` predicate. To ask if an albatross is not flightless, we might try this:

```
1 ?- not(flightless(albatross)).
2 true.
```

But if we left the argument to flightless as an unbound variable, this is how Prolog responds:

```
1 ?- not(flightless(X)).
2 false.
```

Does this seem weird? It did to me at first. Recall truth in Prolog: something is false if you cannot prove it true. But Prolog can prove `flightless(X)` true, by binding $X =$ `Kiwi`. You might try to read `not(flightless(X)` as "give me an X for which `not(flightless(X))` is true", but what it really means in Prolog is "show that no X is flightless".

There's a reason for this. If you think about the constructive nature of Prolog, when you ask it to find a predicate $P(X)$ you may think of it as solving the logical equation $\exists X \mid P(X)$. The negation of this is $\neg(\exists X \mid P(X)) \equiv \forall X \mid \neg P(X)$. That means `not(flightless(X))` is true if `flightless(X)` is false, for all X. The mistake comes from thinking that Prolog will interpret it as $\exists X \mid (\neg P(X))$.

There's another way to do the same thing, using the \+ operator. As far as I know, \+ and `not` are the same thing.

```
1 ?- \+ flightless(X).
2 false.
```

2 Cuts

When Prolog attempts to prove something we've seen how it builds a search-tree of solutions. The cut operator ! lets us manipulate that search tree. It essentially says, "don't try to find alternative solutions for whichever variables have been bound already". Here's an example. Suppose we want to write a function to determine if a state machine is non-deterministic. Here's a first attempt:

```
edge(1, a, 2).
edge(1, a, 3).
edge(2, b, 3).
edge(2, c, 3).

ndfa(Edges) :-
   member(edge(Start, Event, End1), Edges),
   member(edge(Start, Event, End2), Edges),
   End1 \= End2.
```

Let's go ahead and check what we get.

```
?- ndfa.
true ;
true ;
false.
```

It works, but it's kind of annoying that it gives us multiple solutions. This is because there are two pairs of edges which make it non-deterministic. Probably we only care *if* the machine is deterministic, not *how*. This distinction might matter if we query `ndfa` as part of a big query with lots of solutions, in which case Prolog is going to do a lot of unnecessary backtracking. We can prevent that by cutting as soon as we've found a solution.

```
ndfa(Edges) :-
   member(edge(Start, Event, End1), Edges),
   member(edge(Start, Event, End2), Edges),
   End1 \= End2, !.
```

The cut says, "whatever we've bound so far, freeze its value and don't look for any more solutions for those variables anymore". That means it won't try and find alternative solutions for `edge(Start, Event, End1)` and so on. Because we put this at the end, when it would have found a solution for every free variable, this effectively means "don't look for anymore solutions".

```
?- ndfa.
true.
```

3 Negation, Part 2

It might be nice to know if a machine is deterministic too. An easy way to describe that is to say, "it's deterministic if it's not non-deterministic". There are a few ways we can do that. The standard way is to use the `not` predicate.

```
dfa :- not(ndfa).
```

In this case negation works like we expect, because non-deterministic means "there exists two edges going to same node", so deterministic would mean "for every pair of edges, they don't go to the same node", which is exactly the meaning of negation given by Prolog.

4 Cuts, Part 2

One extra thing about cuts is it means you commit to the current goal being proved. There may be multiple cases for some predicate in a Prolog file.

```
1 P1 :- G1, !, G2.
2 P1 :- G3.
```

To prove $\texttt{P}_1$ we'd first try to prove the goal $\texttt{G}_1, !, \texttt{G}_2$. If we succeed in proving $\texttt{G}_1$, the cut means "don't find anymore solutions for $\texttt{G}_1$", but it also means "don't try to prove $\texttt{P}_1$ by using a different goal". To illustrate, take this example of a family tree and a grandmother predicate (which works on both sides of the family).

```
1 father(john, aj).
2 mother(edith, john).
3 mother(mary, aj).
4 mother(tuhi, mary).
5
6 grandma(G, X) :- mother(G, M), mother(M, X).
7 grandma(G, X) :- mother(G, F), father(F, X).
```

This works as we'd expect.

```
1 ?- grandma(G, aj).
2 G = tuhi ;
3 G = edith ;
4 false.
```

But if we add in extra cuts, we run into a problem.

```
1 grandma(G, X) :- mother(G, M), mother(M, X), !.
2 grandma(G, X) :- mother(G, F), father(F, X), !.
```

```
1 ?- grandma(G, aj).
2 G = tuhi.
```

What happens is that Prolog attempts the first goal for $\texttt{grandma}(\texttt{G}, \texttt{aj})$, which is to prove $\texttt{mother}(\texttt{G}, \texttt{M}), \texttt{mother}(\texttt{M}, \texttt{X}), !$. It's able to do this. When it reaches the cut, it will never go back up the search-tree (meaning it's never going to try the other case of `grandma/2` and thus misses the other solution).

5 Cuts, Part 3

Consider our earlier predicate to determine if a graph is non-deterministic. It won't work on a graph with parallel edges. A solution is straight-forward: we define another case of `ndfa` which succeeds if there is a pair of parallel edges. Here's a first attempt at that.

```
1 ndfa :-
2    edge(a, e, b),
3    edge(a, e, b).
```

Unfortunately this will succeed on any (non-empty) graph, because anything solving the first `edge(a, e, b)` will also solve the second `edge(a, e, b)`. We want two distinct edges which are syntactically equivalent. Expressing that is a bit tricky. Here's one solution: an edge is parallel if it exists in the list of edges, and the number of times it occurs is more than once.

A I don't think SWI-Prolog has a built-in `count` predicate, so let's implement that (what would a tail-recursive version look like?)

```
count(_, [], 0).

count(X, [X|Xs], Count) :-
   count(X, Xs, Count2),
   Count is Count2 + 1.

count(X, [Y|Xs], Count) :-
   X \= Y,
   count(X, Xs, Count).
```

Furthermore we want to gather up the edges in the graph. We can do that using `findall`.

```
edges(List) :-
   findall(edge(A, E, B), edge(A, E, B), List).
```

Finally we can implement `parallel/1`, which holds if the argument is an edge.

```
parallel(E) :-
   edges(Edges),
   member(E, Edges),
   count(E, Edges, Count),
   Count > 1.
```

Finally we can implement the more robust version of `ndfa` which can deal with parallel edges.

```
ndfa :- parallel(_).

ndfa :-
   edge(Start, Event, End1),
   edge(Start, Event, End2),
   End1 \= End2, !.
```

It's not perfect though. Let's test it on the following graph.

```
edge(1, event, 2).
edge(1, event, 2).
edge(1, event, 4).
```

```
?- ndfa.
true ;
true ;
true.
```

This happens because there are three derivations which show our graph is non-deterministic.

1. Show that `edge(1, event, 2)` is parallel.

2. Show that the other `edge(1, event, 2)` is parallel.

3. Show that a pair of edges go to the same destination node.

If we didn't have that cut on the second case of `ndfa`, that third one would turn into even more derivations (by finding all the pairs which go the same node, not just a single pair). A quick way to prune this down is to use cuts.

```
1 ndfa :- parallel(_), !.
2
3 ndfa :-
4    edge(Start, Event, End1),
5    edge(Start, Event, End2),
6    End1 \= End2, !.
```

Now it will only search for one solution and stop when we find it. Another strategy would be to put the cut inside the `parallel` predicate (if you only care about whether the graph *contains* parallel edges, but not what they are).

The use of cuts in this manner is a bit like the way you pattern-match in Haskell. Recall that when you have multiple definitions for a function f, when you call f Haskell will go down the list, matching what you've passed in as arguments to the patterns defined. When it sees one that matches, it executes that definition of the function and ignores the rest. This is a little bit like what our program does when you pose `ndfa` to it: it's trying to find the first case of `ndfa` applicable, and when it finds a successful one it ignores the others.