COMP304 Tutorial 7
16/09/2016

1 Debugging

From inside SWI Prolog you can type "trace" to see how it attempts to unify terms with free variables. First, here are some facts.

```
1 lecturer(alex).
2 tutor(james).
3 tutor(aj).
4 supervisor(alex, aj).
```

We can activate trace by typing in "trace" at the console. Then if we pose a query, we can push enter to go through it step-by-step.

```
1  [trace] ?- tutor(X), supervisor(alex, X).
2     Call: (7) tutor(_G337) creep
3     Exit: (7) tutor(james) creep
4     Call: (7) supervisor(alex, james) creep
5     Fail: (7) supervisor(alex, james) creep
6     Redo: (7) tutor(_G337) creep
7     Exit: (7) tutor(aj) creep
8     Call: (7) supervisor(alex, aj) creep
9     Exit: (7) supervisor(alex, aj) creep
10 X = aj.
```

It first attempted to solve the query with the solution `X = james`, but failed because `supervisor(alex, james)` is not true. It then backtracked, and rebound `X` to `aj`, and tried again. This time it succeeded.

You can end the debugger with `notrace`. Also, you can provide arguments to `trace` so that it will only stop on certain predicates, only stop on failures, etc. More information is here: http://www.swi-prolog.org/pldoc/man?section=debugger

2 Lists

The list syntax in Prolog is a little bit different. If you want to match multiple elements at the head of the list, use a comma. Here's a showcase of that in the below predicate, which inserts a number into a list so that it remains ordered.

```
1 % insert(X, L1, L2) holds if L2 is the result of inserting X into L1,
2 % and it is inserted in ascending order.
3
```

```
% inserting into empty list gives a singleton.
insert(X, [], [X]).

% if X is less than or equal to head of list, insert X followed by the rest
% of the list.
insert(X, [Z|Rest], [X,Z|Rest]) :-
   X < Z ; X == Z.

% otherwise X > head of list, so keep searching.
insert(X, [Z|List], [Z|Rest]) :-
   X > Z,
   insert(X, List, Rest).
```

Here's another example. We can define `zip` as a predicate `zip/3`, which holds if the third argument is the result of zipping together the first two lists.

```
zip([], _, []).
zip(_, [], []).
zip([X|Xs], [Y|Ys], [X:Y|Rest]) :-
   zip(Xs, Ys, Rest).
```

Note we're using : to delimit the elements of a pair. There's nothing special in our use of this. : is a (predefined) infix operator symbol, and we're overloading it to mean "pair" in this case. We could also use ?, if we wanted.

Lastly, [a, b, c] is the same thing as [a|[b|[c|[]]]]

3 Unification

Unification is the process of equating two terms by substituting free variables for concrete terms. For example, the query `supervisor(alex, Y)` can be solved by the substitution $\{Y \mapsto aj\}$. We would then say that the terms `supervisor(alex, Y)` and `supervisor(alex, aj)` can be unified.

Here's a cool example of using unification, from learnprolognow.org. A line can be defined as a pair of points, and a point can be defined as a predicate on two numbers. A line is vertical iff the x-component of the two points is the same. A predicate for that is:

```
vertical(line(point(X, _), point(X, _))).
```

The behaviour of this predicate can be illustrated with the following queries:

```
?- vertical(line(point(3, 5), point(3, B))).
true.

?- vertical(line(point(3, 5), point(5, 3))).
false.
```

You can think of this as like pattern-matching on trees. The way in which Prolog unifies is by depth-first search.

4 Semantics

In Prolog the distinction between syntax and semantics is blurred quite a bit. In Haskell, for instance, $f(X)$ means the function f applied to the expression X. The syntax is the sequence of tokens "f", "(", "X", ")", but the semantics of $f(X)$, the meaning of it, is the value you get when you evaluate the expression $f(X)$, based on how f has been defined.

In Prolog, syntax and semantics are kind of blurred. If you see $t_1 = t_2$ in Prolog, it is true iff the two things have the same structure. For instance, `wellington` = `wellington` is true, but `1 + 1 = 2` is not true, because `1 + 1` and `2` are syntactically distinct terms.

In the theory of arithmetic, `1 + 1 = 2` is true though, but that's another sense of "arithmetic equality". We can get that notion of arithmetic equality by using `is`. For instance, `2 is 1 + 1` will give us true (the order is important; the argument on the left of `is` should be the value that the expression on the right evaluates to).

A term like $p(X)$ is "true" if $p(X)$, with some concrete value of X, can exist inside the universe of your program. What that means is you've either specified $p(X)$ for some concrete value of p (e.g. `p(Wellington).`) or $p(X)$ can be unified with t_1, where t_1 coheres with the body of p.

(This interpretation of Prolog terms "as themselves" is called the Herbrand interpretation in logic).

5 Cyclic Expressions

Because two terms are equal if their syntactic structures are the same (or can be made the same), you have to be careful not to define cyclic structures. Consider the following examples.

```
1 weird1(X, f(X)).
```

`weird1(X, X)` should unify iff $X = f(X)$. But if you pick any value of X the right-hand side will have an extra 'f' at the front. For example, `weird1(wellington, wellington)` holds iff `wellington` = `f(wellington)`, but these two terms do not have the same syntactic structure. `weird1(X, X)` should be false then.

The issue boils down to this: if you try to unify $t_1 = t_2$, and t_1 occurs inside t_2, then it should fail. Checking that t_1 does not occur inside t_2 is called the occurs check. For matters of efficiency, a lot of Prolog implementations (including SWI) don't do the occurs check. As a result, trying to unify terms where one occurs inside the other gives you the wrong answer, when really it should fail.

```
1 ?- weird1(X, f(X)), print X.
2 @(S_1,[S_1=f(S_1)])
3 X = f(X).
```

This issue can manifest if you have recursion that never reaches a base-case. Here's a second example:

```
1 weird2(X, X:X).
```

`weird2(X, X)` should unify if `X = X : X`, but these two terms have distinct syntactic structures. Another way to state the equality is, "X is the pair containing itself twice". This should be false, because no such term exists, but Prolog will say otherwise. Be cautious.

6 Graph Algorithms

Here's a graph:

```
edge(a, b).
edge(a, c).
edge(c, d).
edge(b, d).
edge(e, f).
edge(f, g).
```

Let's say we want to find the paths between two nodes. We'll represent a path as a list of edges. We'll use `edge(X, Y)` as the data representation of an edge. `path/3` holds if the third argument is a path from the node in the first argument, to the node in the second argument.

```
% Any node has an empty path to itself.
path(X, X, []).

% An edge between source and goal is a path, if it exists.
path(X, Z, [edge(X, Z)]) :- edge(X, Z).

% Otherwise if you can find an edge from X to Y, and then a path from Y to Z,
% you have a path from X to Z.
path(X, Z, [edge(X, Y) | Rest]) :-
   edge(X, Y),
   path(Y, Z, Rest).
```

That second one is a little tricky. The predicate body is necessary, because we want that edge to actually be an edge in our knowledge-base (our knowledge-base being the representation of the collection of edges in the graph). We can query the paths between a pair of nodes like so.

```
?- path(a, d, P).
P = [edge(a, b), edge(b, d)] ;
P = [edge(a, b), edge(b, d)] ;
P = [edge(a, c), edge(c, d)] ;
P = [edge(a, c), edge(c, d)] ;
false.
```

7 Tests

You can test your predicates by playing around with them in the interpreter, or by writing predicates which test other predicates. You can also write nameless predicates which are automatically checked on entry. For example:

```
:- path(a, d, _).
```

As soon as you load Prolog, it will try to prove there is a path from `a` to `b`, and give you a warning if there isn't.