6. tutorial in Prolog

May 17, 2016

1 Practice with assert and retract

Task 1: Learn how to use dynamically modifiable predicates using *assert* and *retract* meta-predicates. First, try some basic commands:

1. Declare the predicate `closed` as dynamic using "`:- dynamic my_dyn/1.`" in your program.
2. Try calling "`?- assertz(my_dyn(city1)).`". A next query "`?- closed(X)`" should give you the single answer `X = city1`.
3. Another assertion "`?- assertz(my_dyn(city2)).`" will add yet another clause into your program.
4. Remove the first clause by calling "`?- retract(my_dyn(city1)).`". The query "`?- my_dyn(X)`" should give you a single answer `X=city2`.
5. Remove *all* assertions using "`?- retractall(my_dyn(_)).`". The query "`?- my_dyn(X)`" should give no answer.

Next, choose one of the remaining tasks. You don't have to finish all of them, the assert and retract techniques are really straightforward.

Task 2: Implement a Floyd-Warshall algorithm in the graph of *European cities* (see previous tutorials). Its implementation should be very short:

1. Declare a *dynamic* predicate `floydwarshall(From, To, Distance)`. First, declare (programatically) `floydwarshall` to be true for journeys of length 1. The `floydwarshall` should be implied by the `europe` predicate.

2. Write a `run_fw` procedure, which finds an extension of an existing journey and saves it using `assertz(floydwarshall(...))`.

3. If `run_fw` finds a shorter route between existing cities, it removes the *longer* journey (using `retract`) and saves the *shorter* one (using `assertz`).

Task 3: Change the representation of the graph of *European cities* (see previous tutorials). Define a new predicate `into_adjacency`, which reads the graph in the *edge-list* representation from previous tutorials:

```
europe(barcelona, madrid, 504).
europe(belehrad, bukurest, 447).
europe(belehrad, budapest, 316).
...
```

and stores them into the *adjacency-list* representation using a dynamic predicate `europe/2`:

```
europe(barcelona, [madrid]).
europe(madrid, [barcelona]).
europe(belehrad, [bukurest, budapest, ...]).
...
```

Finally, update the bredth-first-search procedure to use this representation. Your predicates should look simpler.

Task 4: Modify the depth-first-search procedure so that the closed list is not implemented as a separate argument, but as a global dynamic predicate.

2 Advanced algorithms in Prolog

Task 5: Implement the *merge-sort* algorithm. First, create a `split` predicate, which divides a list into 2 lists of roughly equal size:

```
?- split([5, 3, 1, 2, 9, 5, 7], X, Y).
X = [5, 1, 9, 7],
Y = [3, 2, 5].
```

Next, implement the merge predicate, which joins two (sorted) lists into a larger sorted list:

```
?- merge([1, 5, 7, 9], [2, 3, 5], X).
X = [1, 2, 3, 5, 5, 7, 9].
```

Voilà, the merge sort is almost done!

Task 6: Implement a quick-sort algorithm. Modify the `split` predicate, so that

1. the first item becomes the *pivot*,
2. the list is divided into items
   (a) smaller than the pivot and
   (b) larger or equal than the pivot.

```
?- split([5, 3, 1, 2, 9, 5, 7], X, Y).
X = [3, 1, 2],
Y = [5, 9, 5, 7].
```

The rest is almost the same as in the merge sort!