Constructive Logic (15-317), Fall 2018
Assignment 8: Practicing Prolog

Course Staff*

Due: Friday, November 2, 2018, 11:59 pm

Submit your homework as a **tar** archive containing the files: `g4ip.pl`, and `coloring.pl`.

After submitting via Autolab, please check the submission's contents to ensure it contains what you expect. No points can be given to a submission that isn't there.

1 Implementing a theorem prover (one more time)

Now that you are experts in implementing **G4ip** in Standard ML, it is time to try doing so in Prolog.

Task 1 (15 points)**.** Implement a theorem prover for **G4ip** in Prolog. You must define the predicate predicate `prove/1` for proving a formula, and use the predefined logical operators (see accompanying `g4ip.pl` file). This means that, given a valid *ground* formula a, the query `prove`(a) should succeed (with *true* or *yes*).

For your convenience, we have provided you with a shell script to test your implementation. You can invoke it by going

```
$ ./test_g4ip.sh
```

2 Colouring maps

Graph colouring is an interesting problem in graph theory. A graph colouring is an assignment of colours to each vertex such that no two adjacent vertices have the same colour. Of particular interest is a colouring using a minimum number of colours; this number is called the *chromatic number* of the graph. The four-colour theorem states that any planar graph[1] can be coloured using at most four colours. The theorem was proved in 1976 using a computer program, and has caused much controversy (is a computer proof really a proof?). It has since been formally verified using the Coq theorem prover in 2005.

As a consequence of this theorem, any map can be coloured with at most four colours such that no adjacent regions have the same colour. This is because every map can be represented by a planar graph, with one vertex for each region, and an edge between two vertices if and only if their corresponding regions are adjacent.

Consider, for example, Australia's map in Figure 1. Observe that this map uses more colours than necessary, although this might make it more visually appealing.

Task 2 (15 points)**.** Implement a predicate `color_graph`$(nodes, edges, colours)$ that associates with the graph $(nodes, edges)$ all of the valid 4-colourings of the graph. Submit your implementation in a file named `coloring.pl`.

*Based on an assignment by Giselle Reis.

[1] A graph that can be drawn on the plane with no crossing edges.

Figure 1: Australia (more colourful than necessary)

The predicate `color_graph` should find all valid colourings via backtracking. For efficiency reasons, you may prefer to find all valid colourings without repetition, but we will not be checking this. Once all valid solutions have been found via backtracking, the predicate should fail. You may assume the graph is finite and planar, and your implementation should satisfy the following requirements:

1. You should define a `color/1` predicate with four colours.
2. Assume there are predicates `node/1` and `edge/2`.
3. In `color_graph/3`, the first parameter is a list of `node/1` terms, the second parameter is a list of `edge/2` terms, and the third parameter is a list of pairs `(a,c)`, where `a` is a node and `c` is a colour.
4. In the terminology of Task 3, the predicate `color_graph` should be multisolution for the mode `color_graph`$(+nodes, +edges, -colouring)$. (Indeed, the four-colour theorem tells us that we will always be able to find a 4-colouring for a planar graph, and the graph's finiteness guarantees there are only finitely many such colourings.)

To clarify the terminology, consider the predicate `childOf`(P, Q), which we claim is multisolution for the mode `childOf`$(+person, -person)$:

```
person(alice).
person(bob).
person(eve).
person(mallory).
childOf(eve, alice).
childOf(eve, bob).
childOf(alice, eve). % Yes, this family tree has a cycle...
childOf(bob, eve).
childOf(mallory, alice).
childOf(mallory, bob).
% Repeated for the sake of contrasting findall and setof below.
childOf(mallory, bob).
```

We can ask Prolog to backtrack and find additional solutions by entering ";" when prompted:

```
| ?- childOf(eve, Parent).

Parent = alice ? ;
```

```
Parent = bob

yes
```

Observe that `childOf`$(+person, -person)$ is multisolution because it will always terminate with at least one solution. In contrast, `childOf`$(-person, +person)$ is *not* multisolution, because for no term P does `childOf`$(P, mallory)$ hold.

The built-in Prolog predicates `findall/3` and `setof/3` may be useful in debugging your implementation. You can use the `findall/3` predicate to return a list of all solutions (including repetitions):

```
| ?- findall(P, childOf(mallory, P), Parents).

Parents = [alice,bob,bob]

yes
```

We can also ask Prolog to return a set of all solutions (in list form) using the `setof/3` predicate:

```
| ?- setof(P, childOf(mallory, P), Parents).

Parents = [alice,bob]

yes
```

For your convenience, we have provided you with a shell script to test your implementation. You can invoke it by going

```
$ ./test_coloring.sh
```

Submitting your assignment

Please generate a tarball containing your solution files by running

```
$ tar cf hw8.tar coloring.pl g4ip.pl
```

and submit the resulting `hw8.tar` file to Autolab.