Constructive Logic (15-317), Fall 2019
Assignment 10: Practicing Prolog and Focusing

Instructor: Karl Crary
TAs: Avery Cowan, David Kahn, Siva Somayyajula*

Due: Friday, November 15, 2019, 11:59 pm

Submit your homework in two parts: (1) a **tar** archive containing the files: `g4ip.pl`, and `coloring.pl` to Autolab as well as (2) your written solutions in a PDF to Gradescope.

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 Practicing Prolog

1.1 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. **Remember, do not translate implementations from Standard ML directly to Prolog! You'll make life harder for yourself, because these problems are designed to be directly expressible as logic programs.**

Task 1 (15 points)**.** Implement a theorem prover for **G4ip** in Prolog. You must define the 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
```

1.2 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.

*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)

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`.

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. The predicate `color_graph` should be a *multisolution* for the mode `color_graph`$(+nodes, +edges, -colouring)$. That is, a call to this predicate that agrees with its mode ought to return *at least* one solution if it terminates, which it should! (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. For your convenience, we have provided you with a shell script to test your implementation. You can invoke it by going

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

2 Focusing and Chaining

A major theme of this course has been the discovery of theory through practice: strategies for efficient proof search in the concrete conditions of real-world implementations are transformed into razor-edged intellectual weapons, entirely new logics which sharpen the principal contradiction of proof theory: the dialectic of the *positive* and *negative* (polarity).

The decomposition of *truth* into *verification* and *use* was our first encounter with the scientific law, "One Divides Into Two". By studying invertibility in the context of the sequent calculus (when does a conclusion imply its premises?), we were able to achieve a firmer grasp of the fault-lines at play, summarized in a dangerously over-simplified[2] form below:

	LEFT RULE	RIGHT RULE
POSITIVE	invertible	**non-invertible**
NEGATIVE	**non-invertible**	invertible

Inversion Invertible rules can always be applied without any need for backtracking: since the conclusion of an invertible rule implies its premises, the "future truth" of the goal is preserved under free application of such rules. This practical insight, which is crucial for implementing a performant proof search engine, can be codified by sharpening the logic to include deterministic inversion phases $\Gamma; \Omega \longrightarrow_{\mathsf{L}} C$ and $\Gamma; \Omega \longrightarrow_{\mathsf{R}} C$ (where Ω is an ordered context of propositions).

Chaining While the above gives a clear and deterministic account of invertible rules, the non-invertible ones beg for something similar. In this week's lecture, we began to study *chaining*, which fixes a dynamics for the non-invertible rules based on two forms of judgment, $\Gamma \longrightarrow [A^+]$ and $\Gamma; [A^-] \longrightarrow C$. Chaining is a technique to minimize backtracking by applying a sequence of non-invertible rules in one go.

2.1 Practicing focusing

Task 3 (10 pts)**.** Construct a derivation in focused logic for the following sequent:

$$\cdot; \cdot \longrightarrow_{\mathsf{R}} \downarrow\big(\big(a^+ \supset b^-\big) \wedge \big(a^+ \supset c^-\big)\big) \supset \big(a^+ \supset \big(b^- \wedge c^-\big)\big)$$

[2] In *structural* or *persistent* logic, some rules which ought to be non-invertible turn out to be invertible; polarity arises properly from the proof search dynamics of *linear logic*, and casts an imperfect shadow in persistent logic.

Task 4 (20 pts)**.** Consider the following depolarized formula:

$$\neg(a^+ \vee b^-) \supset \neg a^+ \wedge \neg b^-$$

Come up with two *distinct* polarizations of the formula, adding shifts in the appropriate places; you do not need to prove them. Hint: remember that in depolarized constructive logic, negation $\neg A \equiv A \supset \bot$; in your solution, you must choose a polarization for negations.

2.2 Saturation

Consider the following grammar of ground terms representing binary numbers:

$$n \quad ::= \quad \epsilon \mid \mathsf{b0}(n) \mid \mathsf{b1}(n)$$

In class, we learned to write forward logic programs using inference rules; a forward logic programming engine will apply these inference rules until saturation is reached, and then the result of our program can be read from the saturated proof state. In the tasks that follow, you are free to introduce any auxiliary predicates that you require. You need to ensure that your rules *saturate* when new facts of the indicated form are added to the database.

In the problems that follow, you are required to implement forward logic programs by writing down systems of inference rules. You may find it useful to experiment with **DLV**, an implementation of forward logic programming which can be downloaded here: `http://www.dlvsystem.com/dlv/`. **DLV** can be used to test your ideas on specific cases and quickly determine if they are likely to work; but it is not required.

Task 5 (5 pts)**.** Implement a forward logic program $\mathsf{std}(n)$ which derives the atom `no` iff it is not the case that n is in standard form. You may assume that n is ground (i.e. not subject to unification).

Task 6 (5 pts)**.** Next, implement a forward logic program $\mathsf{succ}(m, n)$ which derives `no` when it is not the case that $m + 1 = n$. For the purpose of this exercise, you may assume that m and n are ground. You may also assume that m and n are in standard form.

Submitting your assignment

- Please generate a tarball containing your solution files by running

  ```
  $ tar cf hw10.tar coloring.pl g4ip.pl
  ```

  and submit the resulting `hw10.tar` file to Autolab.

- Submit your written solutions as `hw10.pdf` to Gradescope.