Constructive Logic (15-317), Spring 2020
Assignment 8: Past the Prolog Prologue

Instructor: André Platzer
TAs: Avery Cowan, Cameron Wong, Carter Williams, Klaas Pruiksma

Due: Tuesday, March 31, 2020, 11:59 pm

Submit your homework as a **tar** archive containing the files: `g4ip.pl`, and `coloring.pl`. Submit `hw8.pdf` to Gradescope, as usual.

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 Mean, Median...

For each of the following Prolog predicates, give all possible modalities relative to the described intended behavior. For the sake of this problem, negatively-moded parameters *must* eventually terminate when backtracked, but do *not* need to generate all correct possibilities. You may assume that predicates are passed "sane" values (i.e., all arguments are of the correct form). If a predicate is incorrect when a parameter moded positively, give a counterexample (one counterexample per predicate is fine). Otherwise, you do not need to prove your modalities correct.

Task 1 (3 points). `lookup(Map,Key,Value)` is satisfiable when `(Key, Value)` occurs in `Map`.

```
lookup([],_,_) :- false.
lookup([(Key, Value) | _], Key, Value).
lookup([_ | Ms], Key, Value) :- lookup(Ms, Key, Value).
```

Task 2 (3 points). `sublist_sum(L,N,S)` is satisfiable when S is a sublist of L and sums to N.

```
sublist_sum(_,0,[]).
sublist_sum([],_,_) :- false.
sublist_sum([_ | Xs], N, Ss) :- sublist_sum(Xs, N, Ss).
sublist_sum([X | Xs], N, [X | Ss]) :- N2 is N-X, sublist_sum(Xs, N2, Ss).
```

Task 3 (3 points). `subset_sum(L,N,S)` is satisfiable when S is a permutation of some sublist of L and sums to N.

```
subset_sum(_,0,[]).
subset_sum([],_,_) :- false.
subset_sum([_ | Xs], N, Ss) :- subset_sum(Xs, N, Ss).
subset_sum([X | Xs], N, [X | Ss]) :- N2 is N-X, subset_sum(Xs, N2, Ss).
```

For the following two tasks, the predicate `std/1` is defined as in class (reproduced below for your convenience):

```
std(e).
std(b1(N)) :- std(N).
std(b0(b0(N))) :- std(b0(N)).
std(b0(b1(N))) :- std(b1(N)).
```

Task 4 (3 points)**.** `plus(A,B,C)` is satisfiable when C is the sum of the binary numbers A and B. You may assume `std(N)` for any input N.

```
plus(A,e,A).
plus(e,B,B).
plus(b0(A),b0(B),b0(C)) :- plus(A,B,C).
plus(b0(A),b1(B),b1(C)) :- plus(A,B,C).
plus(b1(A),b0(B),b1(C)) :- plus(A,B,C).
plus(b1(A),b1(B),b0(C)) :- plus(A,B,C2), plus(C2,b1(e),C).
```

Task 5 (3 points)**.** `plus(A,B,C)` is satisfiable when C is the sum of the binary numbers A and B. You may assume `std(N)` for any input N.

```
plus(A,e,A).
plus(e,B,B).
plus(b0(A),b0(B),b0(C)) :- plus(A,B,C).
plus(b0(A),b1(B),b1(C)) :- plus(A,B,C).
plus(b1(A),b0(B),b1(C)) :- plus(A,B,C).
plus(b1(A),b1(B),b0(C)) :- plus(b1(e),A,A2), plus(A2,B,C).
```

2 Implementing G4IP (one more time)

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

Task 6 (10 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
```

3 Colouring maps

Graph coloring is an interesting problem in graph theory. A graph coloring is an assignment of colors to each vertex such that no two adjacent vertices have the same color. Of particular interest is a coloring using a minimum number of colors; this number is called the *chromatic number* of the graph. The four-color theorem states that any planar graph[1] can be colored using at most four colors. 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 colored with at most four colors such that no adjacent regions have the same color. 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.

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

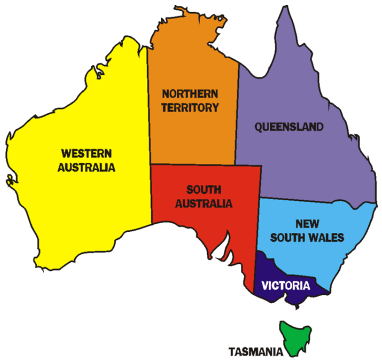

Figure 1: Australia (more colorful than necessary)

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

Task 7 (10 points)**.** Implement a predicate `color_graph`$(nodes, edges, colors)$ that associates with the graph $(nodes, edges)$ all of the valid 4-colorings of the graph. Submit your implementation in a file named `coloring.pl`.

The predicate `color_graph` should find all valid colorings via backtracking. For efficiency reasons, you may prefer to find all valid colorings 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 colors.
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 color.
4. The predicate `color_graph` should be multisolution for the mode `color_graph`$(+nodes, +edges, -coloring)$ – that is, it should return all possible colorings if queried. (Indeed, the four-color theorem tells us that we will always be able to find a 4-coloring for a planar graph, and the graph's finiteness guarantees there are only finitely many such colorings).

Task 8 (5 points)**.** Give the inference rules corresponding to the program written for task 7.

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

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. Solutions to the written portion can be submitted to Gradescope, as usual.