CS152 Homework: Prolog

Due Friday, May 6, at 11:59PM

This assignment has several purposes:

- To give you practice with Prolog's programming model, which may be the most unusual model you ever encounter.
- To show you the power of exhaustive search −−− you can solve interesting problems with very few lines of code.
- To show you the limitations of exhaustive search −−− if you are not careful, you can write solutions that take too much time or space even given the very small problems below. Even a good solution to some of the problems below can take a surprisingly long time.

Do Exercises 6, 20, and 21 from Ramsey and Kamin [35 points] as well as problems A−−E [50 points] and J [15 points] below.

Source code for uProlog is available in several places, but *only* the version on the FAS servers, in `~cs152/software/bare/uprolog`, gives you code for substitution and unification, which is not in the book. If you have already made a private copy of the book software, you will need to go back to the FAS server and get the current version, so that you won't have to re−implement substitution and unification. Otherwise, the source code should be reasonable, although it is still a bit of a mess in spots; you should be especially careful to ignore redundant ``readers'' functions.

Note: ***do not write a lot of code for Exercises 20 and 21.*** I added `not` by adding 4 lines to the interpreter, and I added the cut by changing about 7 lines of existing code. Focus on test cases that convince us you have got the correct semantics.

Background for problems A−E

Consider the game ``peg solitaire'' as played on a board of ten holes arranged in the following layout:

```
   _
  o o
 o o o
o o o o
```

where _ represents an empty hole and `o` represents a hole with a peg in it. A ``move'' results when one peg jumps over another to land in a hole. The two pegs and hole must be colinear, and the stationary peg that was jumped over is removed from the board. So after a legal first move of the 1st peg on the third row (peg 4) we have:

```
   o
  _ o
 _ o o
o o o o
```

and after moving the last peg on the same row (peg 6) we have:

```
   o
  _ o
 o _ _
o o o o
```

and so on. When no peg can jump over any adjacent peg to land in a hole, the game is over. The object of the game is to leave a single peg, preferably in a designated hole. After my first attempt, I left this configuration:

```
   _
  o o
 _ _ _
_ _ _ o
```

If you want to play the game yourself, try it with small coins, or you can run `~cs152/bin/peg10` or `~cs152/bin/peg`.

For the problems below, number the pegs from 1, i.e., number the 10−hole layout like this:

```
   1
  2 3
 4 5 6
7 8 9 10
```

Peg−Solitaire problems

Solve the following problems:

A. Write Prolog rules such that the query `cansolve(n)` succeeds if and only if 10−hole peg solitaire has a solution leaving `n` or fewer pegs. You can assume that `n` will always be passed in, e.g., we should expect `cansolve(3)` to succeed always. [15 points]
B. Add new rules for `minleaving` such that querying `minleaving(N)` puts in `N` the minimum number of pegs that can be left on the board. *Hint: this is much easier with the cut.* [10 points]

Now, switch to a 15−hole layout for peg solitaire:

Or more schematically,

```
    _
   o o
  o o o
 o o o o
o o o o o
```

C. Solve problem B over the new layout. [5 points]

D. Number the holes from top to bottom, left to right, and write Prolog rules such that `solution(n, M)` either produces in `M` a list of moves leaving a single peg in hole `n`, or fails if there is no such sequence. Represent a single move by the term `move(Start, Finish)`, so for example the two possible initial moves would be represented as `move(4,1)` and `move(6,1)`. [10 points]

E. We don't always have to start with the top hole empty. Write Prolog rules such that `moves(S, F, M)` produces a sequences of moves `M` that takes the board from a configuration in which all holes *except* `S` have pegs to a configuration in which only hole `F` has a peg. Using these rules,
   - Write a query that shows where you can put a hole initially in order to be able to leave a single peg in hole 5.
   - Time how long it takes to answer this query.
   - Explain how you would speed it up.

   [10 points]

Hints:

- *Think about a predicate that means ``move M takes the board from configuration B to configuration BB."*
- *It might be easier to do the* ***Generality*** *extra credit and treat the problems above as special cases.*
- *The board has a symmetry group composed of threefold rotational symmetry plus reflection symmetry.*

Logic problem

J. The following scenario is adapted from a problem by Raymond Smullyan, who has made a career out of this sort of nonsense.

> Someone has stolen the jam! The March Hare said he didn't do it (naturally!). The Mad Hatter proclaimed one of them (the Hare, the Hatter, or the Dormouse) stole the jam, but of course it wasn't the Hatter himself. When asked whether the Mad Hatter and March Hare spoke the truth, the Dormouse said that one of the three (including herself) must have stolen the jam.
>
> By employing the very expensive services of Dr. Himmelheber, the famous psychiatrist, we eventually learned that not both the Dormouse and the March Hare spoke the truth. Assuming, as we do, that fairy–tale characters either always lie or always tell the truth, it remains to discover who *really* stole the jam.

Write a Prolog program to solve this logical puzzle. In particular, write rules for a predicate `stole` such that the query `stole(X)` succeeds if and only if `X` could have stolen the jam. The query should work even if `X` is left as a variable, in which case it should produce *all* the suspects who could possibly have stolen the jam.

- Your program should be as brief and well structured as possible.
- Your predicates should be clearly connected to relevant concepts and relationships; do not include information that is clearly irrelevant.
- It is most likely that one of the three characters is the culprit, but the culprit could be an outsider.

Approaches to the problem. There are two ways to approach this kind of problem.

1. The first approach uses exhaustive search of the entire state space to found all the possibilities that are consistent with the facts as given. You will explore that approach in section and get a handout on it.
2. The second approach is to explore a much smaller state space and use logical implication to get

the rest. This technique won't be covered in section, but you'll have a chance to attack it in your own.

You can earn at most 14 points on this problem by using the first approach (exhaustive search). To earn the last, lousy point, you'll have to use the second approach (logical implication).

Hints:

- The full state space for this problem should encompass who's lying, who's telling the truth, and of course who stole the jam.
- A restricted state space might involve only who stole the jam−−−and you could deduce everything else from that.
- You may get stuck if you work only with simple predicates such as ``the Hare is telling the truth'' or ``the Dormouse stole the jam.'' It might help to consider such compound predicates as ``if the Dormouse stole the jam, then the Hare is telling the truth.''
- It's unwise to use the Prolog `not` predicate on anything except a ground term.
- You should assume that Dr. Himmelheber is telling the truth.

[15 points]

Efficiency

Part of this assignment is to come up not just with solutions, but with reasonably efficient solutions. Here are some problems I've seen in several students' Prolog code, which if not corrected may lead to unnecessary inefficiency:

- A predicate that means ``these three holes are in a row'' but that works only when all arguments are `in' mode, i.e., when all arguments are instantiated to integers. This is not a very useful tool.
- A `jump` or `move` predicate (somewhat like the tic−tac−toe `move` predicate) that does too much searching. Be very careful here.
- Too much counting of pegs.

  For problems where you're trying to reach particular final position, it is *never* necessary to count pegs −−− you know what configuration you want, so just look for that configuration. Don't count.

  Even for problems where you do have to count pegs, it's rarely necessary to count pegs on individual boards, because you know that each move takes away one peg.

Extra Credit

Symmetry. In the 10−hole game, there is no real choice for the first two moves−−−all positions that can be reached in two moves are identical under the symmetry group. Use this fact to speed up your solution to problem B, and *measure the speedup*.

More symmetry. Use symmetry to speed up your solution to problem D. *Measure the speedup*.

Generality. Solve one or more of problems C−−E, but make the number of holes in the triangle a parameter to the problem. So for example, I would try to solve the board in the introduction by `solution(4, 1, M)` where 4 is the number of holes along one side of the triangle, 1 is the desired final hole, and M is the desired sequence of moves. Measure the performance cost of this generalization.
Hint: The tough part is figuring out what's the numbering for a potential move. Think about shearing the board to form a lower−triangular matrix. What are the rules then for the permissible directions of motion? You may find it

useful to number by row and column instead of just numbering the individual holes. This is perfectly OK.

Complementarity. The ``complementarity problem'' for peg solitaire asks for which holes H we can start with a peg in every hole except H, then finish with a board that is empty except for a single peg in H. Write Prolog rules for `complements(HS)` that leaves in `HS` a list of holes for which the complementarity problem can be solved. *Exploit symmetry to reduce searching time.* Which holes do you actually have to check?

Types. This programming–language stuff all fits together. In Prolog, write a type checker for the first–order typed lambda calculus with products (`pair`, `fst`, `snd`) and integer literals. The core of your type checker should be a relation `has_type(Gamma, Term, Type)` where you supply the environment and the term and Prolog computes the type. For even more extra credit, add polymorphism.

For the simplest possible type system, a checker in Prolog should take about a dozen lines of code. Adding sums, products, and polymorphism will more than double that. Here's a sample from my code, running on an old homework problem:

<sample run of a type checker in Prolog>=
```
| ?- has_type([],
       tylambda(alpha, tylambda(beta,
           lambda(p, cross(alpha, beta), pair(snd(var(p)), fst(var(p)))))), T).

T = forall(alpha,forall(beta,arrow(cross(alpha,beta),cross(beta,alpha))))
```

Can you ``run it backwards'' and get the engine to exhibit a term with a particular type? If not, why not? Can you modify your code to produce a derivation as well as a type? If not, why not?

Pegs as games. Implement peg solitaire as a game, and use the AGS from the Standard ML Modules homework to solve problem E and the **Complementarity** extra credit. Hint: you'll find it easiest if you make the solitaire itself a functor, which you can then parameterize by the initial and final positions desired.

<peg solitaire sketch>=
```
signature PEG_BASICS = sig ... end
structure PegBasics : PEG_BASICS = struct
  type config
  ...
end

signature PEG_PARMS = sig
  structure Basics : PEG_BASICS
  val initial_hole : int
  val final_peg    : int
end

functor PegGame(Parms : PEG_PARMS) : GAME =
  struct
    open Parms.Basics (* justified because all these names are re-exported *)
    fun who_won conf =
       if pegs_left conf = 1 andalso contents(conf, Parms.final_peg) = Peg then
         Player.WINS Player.X
       else
         Player.TIE
    ...
  end
```

and so on... You might start out simply by implementing a game like `~cs152/bin/peg`, in which you win if you clear the board.

What to submit

Using submission script `submit-prolog`, submit these files:

- `README`, which should contain a brief discussion of your solutions as well as your answer to Exercise 6. Your `README` file should also contain any short transcripts and test cases that demonstrate your implementation of the cut and of `not`, as well as a discussion of any false starts or bugs you uncovered in these codes.
- File `uprolog.sml`, which should contain your solutions to Exercises 20 *and* 21 (the cut and `not`).
- Files `pegA.P` through `pegE.P`, containing your solutions to peg–solitaire problems. Don't worry if these files have a lot of duplicate code–––we'll sort out the differences.
- File `jam.P`, containing your solution to problem J.

You can use either uProlog (`~cs152/bin/uprolog`) or XSB Prolog (`~cs152/bin/xsb -i` or `/usr/local/XSB-2.5/2.5/bin/xsb -i`). XSB Prolog is faster, and this may be helpful for some of the peg problems. For extra credit, you may also turn in `pegBx.P`, `pegDx.P`, `general.P`, `complement.P`, `types.P`, and more.

- *<peg solitaire sketch>*: D1
- *<sample run of a type checker in Prolog>*: D1