CS 81: Computability and Logic, Fall 2023

Homework 5: Prolog

Due: 11:59pm, Tuesday, October 3, 2023

The usual collaboration rules in the CS 81 Syllabus still apply, even for source code. Anything you submit should be generated by yourself on your own, and not just transcribing someone else's answer (or even a grutor's answer).

This assignment will be autograded, so be very careful to get your interfaces correct! If we're looking for a one-argument predicate `hasYS`, you will get a zero on that part if you submit a correct one-argument predicate named `hasys` or `hasYoungerSister`, or if you submit a two-argument predicate `hasYS`; the autograder won't even see your code.

Getting Prolog

For this assignment you need to use the SWI-Prolog (`swipl`) interpreter.

- The `swipl` command is already installed on all HMC CS and CIS machines.
- Installers for MacOS and Windows are available at `https://www.swi-prolog.org/download/stable`
- If you have a Mac that uses the Homebrew package manager (`https://brew.sh`) then you can install the interpreter using the command `brew install swi-prolog`
- Linux machines may be able to install `swipl` with their own package managers (e.g., `apt-get install swi-prolog`)

You can get the starter Prolog files as `hw5files.zip` from the Files section on Canvas.

Editing

You may use any text editor to modify the Prolog source files. **Perl and Prolog use the same `.pl` file extension, so if you use an editor that provides syntax highlighting, make sure that it is in Prolog mode (or Plain Text mode) rather than in Perl mode!** (For Visual Studio Code, there is a `Prolog` extension to install before you can switch to Prolog syntax highlighting.)

Loading and Testing Your Rules

- Rules are read only from external files; the Prolog prompt only lets you type queries.
- To load rules from a file `foo.pl` **the first time**, type `[foo].` (with a period)
- **But** if you modify the contents of a previously-loaded file (because you've fixed a bug, or have added new rules), enter `make.` to reload the updated rules into Prolog. Then you can enter queries to test your changes.
- Once rules are loaded, you can type (or paste in) queries for testing. Hitting up-arrow and down-arrow lets you scroll back through previous queries.
- Remember, if Prolog finds an answer you can type a semicolon `;` to make it continue searching for alterate answers.
- You can turn on tracing with `trace.` You can turn off tracing with `nodebug.`

1 The Simpsons' family tree [20 points]

Recall that `simpsons.pl` contains rules for a (very fictional) Simpsons' family tree. These rules define, from scratch, the predicates `person`, `parent`, `female`, `male`, and `age`. In addition, there are example rules that define `child`, `mother`, and `anc`(estor) relationships, using on the above primitive rules. This part asks you to write predicates that answer questions about the Simpsons' family tree—or, at least, the version that appears in `simpsons.pl`.

There are four placeholders where you should replace the defeatist definition `fail` with useful rules to define each of the predicates described in detail below. (In Prolog, `fail` aborts that branch of the proof search) You may also add any additional helper predicates that you wish, including anything that we looked at in class.

Don't change any other part of `simpsons.pl`, particularly the base facts about parenthood and relationships!

1. `grandparent(X, Y)` should be true if and only if `X` is a grandparent of `Y`.

   You can test this out with

   ```
   grandparent( john, bart ).   % OK
   grandparent( lisa, bart ).   % false.
   grandparent( X, bart ).      % four answers if you hit ; enough
   ```

   In fact, you can check all four of the answers to the last query above using the `setof` predicate. Try this; it will be useful many times.

   ```
   setof( P, grandparent( P, bart ), Answer ).

   % true if Answer is the set { P | grandparent(P, bart) }
   % specifically, Answer = [homericus, jackie, john, matilda]
   ```

2. `cousins(X, Y)` should be true if and only if `X` and `Y` are first cousins. Note that for the sake of this problem (and perhaps in general), a person cannot be their own cousin. Similarly, do not consider brothers and sisters to be cousins for this problem.

   You can test this out with

   ```
   cousins( bart, lisa ).           % false
   cousins( bart, terpsichore ).    % OK

   setof( C, cousins(C,lisa), Answer ).    % lisa has 2 cousins
   % [millhouse, terpsichore]
   ```

3. `hasDaughterAndSon(P)` should be true when `P` is a parent of a daughter and of a son.

   For example, you can test this out with

   ```
   hasDaughterAndSon( homer ).           % OK
   hasDaughterAndSon( esmerelda ).       % false

   setof( P, hasDaughterAndSon(P), Answer ).
   % [cher, glum, homer, jackie, john, marge]
   ```

4. hasOlderSibling(X) should be true if and only if X has an older sibling or half-sibling. Consider "older" to mean strictly greater-than in age.

   You should check this with some queries of your own devising; in addition, you can check for all of the people with older siblings with

   ```
   setof( X, hasOlderSibling(X), Answer ).
   % [atropos,glum,homer,homericus,lachesis,lisa,maggie,marge]
   ```

2 Lists in Prolog [30 points]

Here, you will write a few Prolog predicates that express relationships about lists. Please submit your work by extending the file `lists.pl`

1. `removeOne(E, List, NewList)` should be true if and only if `E` is an element of list `List` one or more times, and the list `NewList` should be identical to List except that the element `E` is removed once. Note that the element may appear more than once, and so it can be removed from different parts of the list `List`.

   ```
   removeOne( a, [a,b,c,a], R ).   % [b, c, a] or [a, b, c]
   removeOne( d, [a,b,c,a], R ).   % false

   setof( [E,R], removeOne( E, [a,b,c,a], R ), Answer ).
   % [[a,[a,b,c]],[a,[b,c,a]],[b,[a,c,a]],[c,[a,b,a]]]
   ```

   For full credit, your `removeOne` predicate must be able to work when either the first or last argument, or both, are variables, as demonstrated above. You can assume that the middle argument is already a list value (that is, it's not a variable). Keep in mind the requirement that `E` be a member of `List`!

   You should need no more than two lines (rules) for `removeOne`. If you're writing more than two, take a step back and reconsider! In particular, remember that in Prolog, negation is expressed through failure of its search.

   For example, you don't want a base case involving the empty list as the second argument (because there's no way to successfully remove an element from the empty list and get a result)! If you don't write a rule where the second argument can be empty, the proof search automatically fails (as desired) when the second argument is passed in as an empty list (either directly or through recursion).

   So, what rules should there be? For this one, you will want one base-case rule that handles the case when `E` and the first element of `List` are identical. Then, you'll want a recursive rule, using the rest of `List`. That's it!

2. One-dimensional pattern matching is an important task in many applications such as cryptography and DNA sequencing. For this problem we encode a 1d pattern as a list of items (named `Pattern`) and we'll search for that pattern within another list called the `Target`.

   For this problem write rules for `find(Pattern, Target, Index)` that takes two lists `Pattern` and `Target` and a non-negative integer `Index` as input. This find predicate should be true if and only if `Pattern` occurs inside list `Target` beginning at position `Index`. The find predicate should not be true in the case that `Pattern` is the empty list – so your base case will need to be bigger than the empty list!

For full credit, your find predicate should work when either or both of `Pattern` and `Index` are variables. We will only test your code when `Target` is bound to a list value (it's a non-variable).

Hint 1: The `find` predicate requires only three rules. Here is a rough guide:

- a base case when the Pattern is a list of a single element and when that same element is the first element of a potentially longer Target list. In this case, Index should be 0.
- a recursive case when the Pattern has a first and a rest (and so does the Target), but when the Index is still 0.
- a recursive case when the Pattern is any list, the Target has a first and a rest, and the Index is a variable (and will end up being something greater than 0).

Hint 2: If you have already computed a number `A` and you want `B` to be one larger, you can say `B is A+1`. The alternative `A is B-1` won't work (because we can't evaluate `B-1` until we know `B`), and similarly `B = A+1` and `A = B-1` won't work (because Prolog doesn't calculate the arithmetic without `is`).

For example,

```
find([1, 2], [1, 2, 1, 2, 1], 0).   % true
find([], [1, 2, 1, 2, 1], N).  % false

setof( N, find( [1,2], [1,2,1,2,1], N ), Answer ). %  [0, 2]

setof( [N,P], find( P, [a,c,a,c], N ), Answer ).
      %  [[0, [a]], [0, [a, c]], [0, [a, c, a]], [0, [a, c, a, c]],
      %   [1, [c]], [1, [c, a]], [1, [c, a, c]], [2, [a]],
      %   [2, [a, c]], [3, [c]]]
```

3 Trees (encoded as lists), and a fun puzzle [50 points]

In an untyped language like Prolog, Racket, or Python, we can use lists to represent tree-structured data, which includes tree-based representations of arithmetic expressions. Please submit your work by extending the file trees.pl

Trees don't have to contain just numbers in binary-tree order. We can represent arbitrary arithmetic operations on integers as trees, with the operation to perform at the root, and the two subtrees being the sub-expressions of the operation, with numbers only at the leaves). More specifically,

- a leaf node with value r is represented by just the number r itself
- an interior node (operation on two sub-expressions) is represented by the three-element list [r, *subtree*$_1$, *subtree*$_2$] where r is the operation and *subtree*$_1$ and *subtree*$_2$ are encodings of the left and right subtrees.

For example:

The value of the tree `[+, 3, 5]` is 8.	"the sum of 3 and 5"
The value of the tree `[-, 3, 5]` is -2.	"the difference of 3 and 5"
The value of the tree `[*, [*, 3, 2], [-, 9, 2]]` is 42	"the product of (3*2) and (9-2)"
The value of the tree `[/, [*, 2, [+, 1, 5]], 3]` is 4	"the quotient of (2*(1+5)) and 3"

Write a predicate `eval(Tree, N)` such that `N` is the numeric value of the arithmetic expression tree `Tree`. You can assume that `Tree` is a valid encoding of an arithmetic expression (as described above), and is a given list rather than a variable.

Important:

- Your code should have five cases (if the tree is just a number, if the tree represents a sum of two arbitrary subtrees, if the tree represents a difference of two arbitrary subtrees, ...).
- If your code has already computed numbers `N` and `M`, you ensure a variable `X` is their sum by saying `X is N+M`.
- Although we are using `/` in our trees to represent division, the actual Prolog operator for integer divisions is written `//` (e.g., `X is N//M`).
- Evalation should fail to find a numeric answer instead of crashing with ERROR if the given expression contains division by zero.

```
eval(3, 3).                          % true
eval([+, 3, 5], 8).                  % true
eval([+, 3, 5], -2).                 % false
eval([-, 3, 5], X).                  % X = -2
eval([*, [*, 3, 2], [-, 9, 2]], X).  % X = 42
eval([/, [*, 2, [+, 1, 5]], 3], 4).  % OK
eval([/, [*, 2, 2], [-, 3, 3]], N).  % false (not ERROR)
```

3. Define a predicate `atree(Ops, L, Tree)` where

   - `Ops` is a list of arithmetic operators—some subset of `[+, *, -, /]`
   - `L` is a nonempty list of integers,
   - `Tree` is an arithmetic expression tree.

   The predicate should be true when `Tree` uses only the operations from `Ops` (it's OK if the same operation is used repeatedly) and the numbers in `L` are used exactly once in `Tree`, in order from left to right. For example,

```
atree([+,-], [3,4,5], [+, [+, 3, 4], 5]).   % true!
atree([+,-], [3,4,5], [-, 3, [+, 4, 5]]).   % true!
atree([+,-], [3,4,5], [+, [*, 3, 4], 5]).   % false (uses *)
atree([+,-], [3,4,5], [+, [+, 5, 4], 3]).   % false (out of order)
atree([+,-], [3,4,5], [-, 3, 4]).           % false (wrong values)
atree([+,-], [3,4,5], [+, [-, 3, 99], 4]).  % false (wrong valuea)
atree([+,-], [3,4,5], T).                   % generates 8 possible trees
```

   You may assume that `Ops` and `L` are specific lists, but if the third argument is a variable then your code should be able to generate all valid trees with the given operations and the given leaves.

   If there is only one number in `L` then the only possible tree is just a leaf (encoded as the number itself). If there is more than one number in `L`, then the root is a member of the list of valid operations; the two subtrees are valid arithmetic trees, with numbers from some prefix of `L` are used in the left subtree, and the numbers in the remaining suffix of `L` being used in the right subtree.

   Warning: we can say `append(L1, L2, L)` to divide a list `L` into a prefix `L1` and a suffix `L2`, but if you don't forbid `L1 = []` and/or `L2 = []` then your code is likely to go into an infinite loop when you try to generate trees.

4. Now it's easy to write a solver for the famous "Four fours" puzzle and more!

   Define the predicate `solve(Ops, L, X, Tree)` where

   - `X` is a target number,
   - `Ops`, `L`, `Tree` are the same as the previous part.

   The predicate should be true when `Tree` uses only the operations from `Ops` (it's OK if the same operation is used repeatedly) and uses each occurrence of a number in `L` exactly once in any order, and the tree evaluates to the target value `X`. You may assume that `Ops`, `L`, and `X` are known inputs, but your code should work whether the final input is either a specific tree or a variable.

   Here are some examples:

```
solve([+, *], [1, 2, 3], 7, [+, 1, [*, 3, 2]]).   % true

solve([/], [6, 0], 7, X).   % false, not ERROR!

solve([+, -, *, /], [4, 4, 4, 4], 24, X).   % Four fours!
% lots of trees, including  [+, 4, [+, 4, [*, 4, 4]]]
%   and  [+, [*, 4, 4], [+, 4, 4]]

solve([+, *, /], [5, 2, 1], 7, X).
% lots of trees, including  [+, 5, [*, 2, 1]]
% and [+, 5, [/, 2, 1]] and [*, [+, 5, 2], 1]
% and [*, 1, [+, 2, 5]].
```

   You can implement `solve` with one rule using a "generate-and test" strategy.

   Specifically, we can define `solve(...)` to be true when there is a permutation of the given integers, and there is a valid tree using those permuted integers (in order), and the tree evaluates to the right number.

   Prolog's DFS search strategy will consider all possible permutations of the given numbers, for each permutation consider all possible trees with those leaves, and then keep only the trees with the correct final value!