CS-UY 3314 — Homework 9

Jeff Epstein

Introduction

Complete the following exercises in Prolog.

Before starting, please make sure that you're using a suitable programming environment, as discussed in previous homeworks.

SWI-Prolog should already be installed on the Vital machines. If you're using another Debian-ish Linux distriution, you can install it with the command `sudo apt-get install swi-prolog`.

You can start the Prolog interactive session with the command `swipl`, followed by the name of your source file. Execute the `halt.` predicate to exit the session, or type Ctrl-D.

I recommend the following materials to help you get started with Prolog:

- SWI-Prolog reference manual – The official reference of the Prolog implementation that we use.
- Prolog syntax – A summary of the standardized syntax.
- The history of Prolog – The origins of the language and its development
- `help` predicate – From the SWI-Prolog query prompt, you can use the `help` predicate to view the documentation on any built-in feature; for example, type `help(maplist).` to learn about `maplist`. Type `q` to exit the help viewer.
- `apropos` predicate – From the SWI-Prolog query prompt, you can use the `apropos` predicate to search the documentation for any entries matching some text; for example, type `apropos("reverse").` to show all help topics containing the word "reverse."
- `tspy` and `trace` predicates – These features of SWI-Prolog can help you debug your code.

For all questions in this assignment, follow the instructions carefully. You may not use any modules not loaded by default, and you may not use any extensions to the language (such as DCG or CLP(X)), except as specified. You may not use the `->` and `;` operators, unless otherwise specified.

Submission

Unless otherwise specified, put your solutions into a file named `hw`n`.pl`, where n is the number of this assignment.

Test your code thoroughly before you submit. The test cases provided should guide you in verifying your solution.

Make sure that you name your predicates and/or facts exactly with the names specified in the assignment. For questions whose solution is not code, provide the answer within a comment. In addition, mark each solution with a single-line comment indicating the question number. Single-line comments in Prolog begin with percent-sign (`%`).

Make sure that each file is syntactically correct and can be compiled. If one of your solutions is partial or incomplete, please comment it out (using `/*` and `*/`) so that the remainder of the file can be compiled. If we can't compile your file, we can't grade it, and your grade may be penalized.

Submit all your solution files on Gradescope.

1 Horn clauses

A *Horn clause* is a subset of first-order logic of the form:

$$L_1 \wedge ... \wedge L_n \Rightarrow L$$

where L, L_1, ..., L_n are *predicates*, i.e. logical propositions that are either true or false. The $\wedge$ symbol means *logical and*. The arrow indicates *implication*. Therefore, we can read the above Horn clause to mean "if L_1, ..., and L_n are true, then L is true." Here, L is the *conclusion*, and L_1, ..., L_n are the *conditions*. Note that if one or more of the conditions is not true, then we will not be able to determine that the conclusion is true.

A condition of one Horn clause may be a conclusion of another Horn clause:

$$\begin{aligned} L_1 \wedge L_2 &\Rightarrow L \\ J_1 \wedge ... \wedge J_n &\Rightarrow L_1 \\ K_1 \wedge ... \wedge K_n &\Rightarrow L_2 \end{aligned}$$

In the above case, L is true when L_1 and L_2 are true. L_1 is true when $J_1, ..., J_n$ are true, and L_2 is true when $K_1, ..., K_n$ are true. Therefore, L is true when $J_1, ..., J_n, K_1, ..., K_n$ are true.

A single conclusion may appear as a conclusion in multiple Horn clauses:

$$\begin{aligned} L_1 \wedge ... \wedge L_n &\Rightarrow L \\ L_{n+1} \wedge ... \wedge L_{n+m} &\Rightarrow L \end{aligned}$$

In the above case, L will be true when $L_1, ..., L_n$ are true. L will also be true when $L_{n+1}, ..., L_{n+m}$ are true. Therefore L will be true when either all of $L_1, ..., L_n$, or all of $L_{n+1}, ..., L_{n+m}$ are true.

Some Horn clauses have no conditions, i.e. the conclusion is always true. Such clauses are called *facts*. For example:

$$\Rightarrow F$$

1. Consider the following set of Horn clauses:

$$\begin{aligned} A \wedge B &\Rightarrow C \\ C \wedge D &\Rightarrow E \\ G &\Rightarrow E \\ C \wedge F &\Rightarrow G \\ &\Rightarrow A \\ &\Rightarrow B \\ &\Rightarrow D \end{aligned}$$

   Based on the above clauses, can we determine that E is true? Explain your answer.

2. Prolog is based on Horn clauses. In Prolog, we write the conditions and the conclusion in reverse order relative to the usual mathematical notation, but the logic remains the same. This Horn clause:

$$C_1 \wedge ... \wedge C_n \Rightarrow P$$

   would be expressed in Prolog like this:

```
p :- c1, ..., cn.
```

In this case, the meaning is that the predicate `p` succeeds when `c1`, ..., `cn` *all* succeed.

A *fact* (i.e. a predicate that always succeeds) is written like this:

```
f.
```

In Prolog, we can also use `false` (or `fail`), a special predicate that *never* succeeds.

Consider the following set of Prolog predicates:

```
a.
b.
c :- f, b.
f :- a, b.
h :- false.
q :- a, b, false, c.
d :- e, c.
k :- e, c.
k :- a, b, f.
j :- k, f, c, q.
e :- a, f, c, h.
e :- h.
```

Without using a computer, determine whether each of the following predicates will succeed:

(a) `a`

(b) `c`

(c) `h`

(d) `d`

(e) `k`

(f) `e`

(g) `j`

2 Family relations

A Prolog file, typically with a `.pl` suffix, can contain predicates and facts. Such a file is called a *knowledge base*. We can load it in our Prolog interpreter by providing it on the command line when we start Prolog, like this:

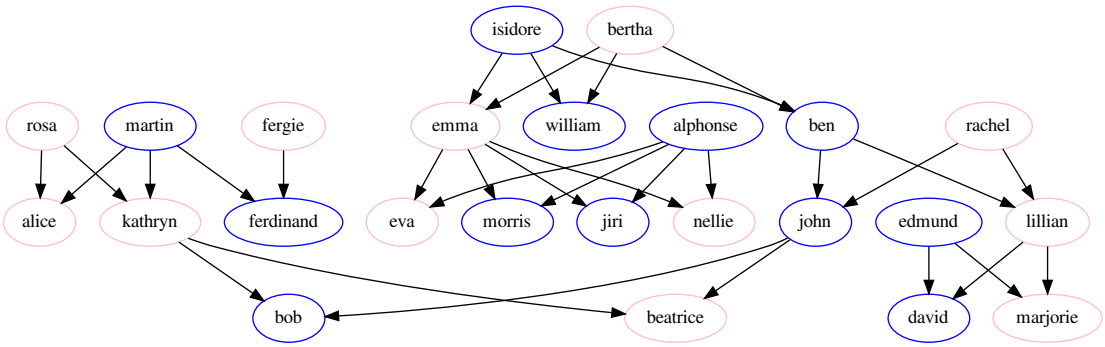
```
swipl myfile.pl
```

Download the file `family.pl` and load it as a knowledge base. It contains three kinds of facts:

- `male(X)` – X is a male person
- `female(X)` – X is a female person
- `child(A,B)` – A is a child of B

Note that most people in the knowledge base are the first argument of two `child` facts, since most people have two parents.

To help you understand the knowledge base and verify your solutions to the exercises below, here is a graphical representation of the knowledge base's data, in the form of a directed graph. An arrow represents a parent-child relationship. Each node is color-coded according to gender.



The knowledge base file contains *facts* and *predicates* which are consulted during queries. Queries are executed by typing them in to the Prolog interpreter at the `?-` prompt. Although queries are syntactically similar to facts and predicates, they are fundamentally different: inside a file, the fact `male(jiri)` asserts that `jiri` is male; whereas the query `male(jiri)` is asking Prolog to determine whether `jiri` is male.

Load the knowledge base by running the command `swipl family.pl`. Once you've loaded the knowledge base, you can issue a query from the Prolog `?-` prompt. For example:

```
% emit all of bob's parents
child(bob, X).
```

(If this query is confusing to you, keep in mind that `child(A,B)` means that A is a child of B. In this case, we are asking Prolog to find all X, such that `bob` is a child of X.)

If you type in the above query, you should see the following result:

```
X = john _
```

This indicates that Prolog has found one solution to your query, namely that the variable X can be instantiated to value `john`, indicating that `john` is a parent of `bob`. And the blinking cursor will appear after the word `john`, indicating that Prolog is waiting for you to tell it what to do. Your options at this point are either to continue searching for solutions, by pressing semicolon (`;`); or to quit searching, by pressing period (`.`). If you press semicolon, you should see another result appear:

```
X = john ;
X = kathryn.
```

Prolog found another solution, and then finished searching, indicating that there are no more solutions available. This makes sense: if you inspect the knowledge base or the above graph, you'll see that `bob` has exactly two parents.

If there are no solutions, Prolog will say so with `false`. Try this query:

```
child(X, bob).
```

The knowledge base contains no information about `bob`'s children, so the query fails.

You may sometimes get a situation where a query initially succeeds, but after you continue searching, Prolog responds `false`. That doesn't mean that the query failed; it just means that Prolog was unable to find any *further* solutions, beyond those that it already showed to you.

Here are some more queries you can try out:

```
% emit all of john's children
child(X, john).

% emit all of emma's sons
child(X, emma), male(X).

% is alice a daughter of rosa? (yes)
child(alice, rosa), female(alice).

% does isidore have a child rosa? (no)
child(rosa, isidore).
```

Comma-separated terms in a query must all be fulfilled for the search to succeed. Note that symbols beginning with an uppercase letter designate *variables*, which a query will attempt to find values for. On the other hand, symbols beginning with a lowercase letter are *atoms*, literal identifiers.

Now, edit the `family.pl` file and add the following predicate:

```
parent(A,B) :- child(B,A).
```

The above code defines the predicate `parent/2` (so named because it takes two parameters), such that `parent(A,B)` will succeed when `child(B,A)` succeeds. Reload the file (either by restarting the Prolog interpreter, or by running `make.` at the query prompt) and try it out:

```
% is rosa a parent of kathryn? (yes)
parent(rosa, kathryn).

% find all people of whom rosa is the parent
parent(rosa, X).
```

Complete the following tasks by adding appropriate predicates in your `family.pl` file.

3. Write a predicate `father/2` such that `father(A,B)` succeeds when A is the father of B. Write a predicate `mother/2` such that `mother(A,B)` succeeds when A is the mother of B.

   For example:

```
?- father(john, X).
X = bob ;
X = beatrice.

?- mother(rosa, X).
X = kathryn ;
X = alice.

?- mother(john, X).
false.
```

```
?- mother(X, john).
X = rachel.
```

Hint: use the `parent/2` predicate that we defined above, combined with the gender facts.

4. Write a predicate `sibling/2` such that `sibling(A,B)` succeeds when A and B are siblings (brothers or sisters) of each other. For the purposes of this exercise, siblings are defined as distinct people who have at least one parent in common.

   Below are some examples of the use of the `sibling` predicate.

```
?- sibling(john, X).
X = lillian ;
X = lillian.

?- sibling(X, ben).
X = william ;
X = william ;
X = emma ;
X = emma ;
false.

?- sibling(ferdinand, alice).
true.
```

   Note that in some cases we get duplicate results: for example, Lillian is indicated as being John's sibling not once, but twice. The reason is that these people are siblings in two ways: they have not one, but two parents in common. In this assignment, we aren't going to worry about duplicate results. If you really want to remove duplicates, you can use a query like `setof(X,sibling(X,ben),Y)`.

   Hint: note that a person is not their own sibling. Use the inequality operator (`\=`) to prevent oneself being included in the search results.

5. Write a predicate `uncle/2`, such that `uncle(A,B)` succeeds when A is the uncle of B. Your uncle is a male sibling of your parent.

   Write a predicate `aunt/2`, such that `aunt(A,B)` succeeds when A is the aunt of B. Your aunt is a female sibling of your parent.

   For example:

```
?- aunt(X, john).
X = emma ;
X = emma ;
false.

?- uncle(X, john).
X = william ;
X = william ;
false.

?- uncle(william, X).
X = john ;
```

```
X = john ;
X = lillian ;
X = lillian ;
X = morris ;
X = morris ;
X = nellie ;
X = nellie ;
X = eva ;
X = eva ;
X = jiri ;
X = jiri.
```

6. Write a predicate `ancestor/2` such that `ancestor(A,B)` succeeds when A is an ancestor of B. Your parents are your ancestors, as are your parents' parents, your parents' parents' parents, etc.

   For example, to show all of `bob's` ancestors:

```
?- ancestor(X, bob).
X = john ;
X = kathryn ;
X = ben ;
X = rachel ;
X = isidore ;
X = bertha ;
X = rosa ;
X = martin ;
false.
```

   We can also use it to find all descendants of X, i.e. all people for whom X is an ancestor. In the below example, we find all of `ben`'s descendants.

```
?- ancestor(ben, X).
X = john ;
X = lillian ;
X = bob ;
X = beatrice ;
X = marjorie ;
X = david ;
false.
```

   Hint: your predicate will be recursive.

7. Write a predicate `first_cousin/2` such that `first_cousin(A,B)` succeeds when A is a first cousin of B. A first cousin is a child of your aunt or uncle; in other words, a person with whom you share at least one grandparent.

   For example:

```
?- first_cousin(david, X).
X = bob ;
X = beatrice ;
```

```
X = bob ;
X = beatrice ;
false.

?- first_cousin(jiri, X).
X = john ;
X = lillian ;
X = john ;
X = lillian ;
false.
```

Submit your complete `family.pl` family.

3 Lists and recursion

For problems in this section, put your solution in a file named `hw`n`.pl`, where n is the number of this assignment.

Recall the Collatz sequence from a previous homework. Here is a possible implementation in Prolog:

```
collatz(N, N).
collatz(N0, N) :-
  0 is mod(N0, 2),
  N1 is N0 / 2,
  collatz(N1, N).
collatz(N0, N) :-
  1 is mod(N0, 2),
  N1 is 3 * N0 + 1,
  collatz(N1, N).
```

Read the code and make sure you understand it. Note that we need the first clause, `collatz(N, N)`, because each element of the Collatz sequence should be output by this predicate, before we calculate the next number in the sequence.

8. Write a non-recursive predicate `inboth/3` such that `inboth(A,B,X)` will succeed for all values `X` that are in both the list `A` and the list `B`. For example:

   ```
   ?- inboth([green,red,orange],[apple,orange,pear],orange).
   true ;
   false.

   ?- inboth([1,2,3,4],[2,5,6,1],X).
   X = 1 ;
   X = 2 ;
   false.
   ```

   Note the confusing result in the first query above, where `inboth` returns `true` and then `false`. This means that the query succeeded, but upon continued searching failed to find any *further* results. We can interpret this as meaning that `orange` appears in both lists only once. If you want to prevent this behavior, use the `once/1` predicate.

   Hint: use the built-in `member/2` predicate to determine if an item is in a list.

9. Write a recursive predicate `increment_all/2` that, given a list of integers, will produce a list with each number incremented by one. For example:

```
?- increment_all([12,99,4,-7], X).
X = [13, 100, 5, -6].
```

Hint: recall the syntax `[H|T]` to match on list's head and tail, similar to Haskell's `(h:t)` syntax.

Hint: remember to use `is` for arithmetic.

10. Write a recursive predicate `merge/3` such that `merge(A,B,X)`, when given sorted lists `A` and `B`, will produce a sorted list `X` consisting of all the elements in `A` and `B`.

    You must use an $O(N)$ algorithm that can merge the lists in one pass. You may assume that the first two parameters are lists of non-decreasing integers. You may not use any of the built-in `sort` predicates.

    For example:

```
?- merge([1,4,5,10,11,13],[3,4,1000],X).
X = [1, 3, 4, 4, 5, 10, 11, 13, 1000] .
```

4 Unification

11. Consider the following two Prolog terms:

```
T1 = [[a,f(b),c,d],g(X1,[e|X2]),X3,X4|X5]
T2 = [Y1,g(d,Y2)|Y1]
```

    Without using your computer, determine if `T1` will unify with `T2`. If so, give the post-unification values of `X1`, `X3`, `X4`, `X5`, and `Y1`. If not, indicate where unification failed.

5 Negation

12. Write a recursive predicate `all_diff/1` such that `all_diff(X)` will succeed if each element in the list `X` is different, i.e. there are no repeated elements.

    For example:

```
?- all_diff([a,b,c]).
true.

?- all_diff([a,b,c,b]).
false.
```

    For a further example, consider the following query, which will emit all the unique orderings of the numbers 1 to 3. It uses the built-in `between(Low,High,X)` predicate, which succeeds when `X` is between `Low` and `High`, inclusive. The first ordering is `X=1, Y=2, Z=3`; the second ordering `X=1, Y=3, Z=2`, etc.

    `

```
?- between(1,3,X), between(1,3,Y), between(1,3,Z), all_diff([X,Y,Z])
   ↪ .
X = 1,
Y = 2,
Z = 3 ;
X = 1,
Y = 3,
Z = 2 ;
X = 2,
Y = 1,
Z = 3 ;
etc, etc
```

Hints:

- Use the built-in `member/2` predicate. The call `member(X,L)` will succeed when item `X` is in list `L`.
- Use the built-in `not/1` predicate. Its argument is another predicate. For example:

```
?- X=[q,y,z,w], not(length(X, 4)).
false.

?- X is 3+4, not(X=99).
X = 7.
```

Note, however, that `not(X)` does not mean that `X` is false; rather, it means that `X` cannot be proven true. This can sometimes lead to unintuitive and illogical results: For example, in the following queries, the only difference is the order of the predicates, so the result should logically be the same. However, we get a different result depending on the order, due to the extra-logical nature of `not/1`.

```
% Is it provable that some value X is not in the list? No.
?- not(member(X,[a,b,c])), X=f.
false.

% Is it provable that the atom f is not in the list? Yes.
?- X=f, not(member(X,[a,b,c])).
X = f.
```

13. Write a predicate `splitAt(Pos, List, FirstPart, SecondPart)` that will split the list `List` at position `Pos`, resulting in two parts `FirstPart` and `SecondPart`, such that `FirstPart` contains exactly `Pos` elements, and `SecondPart` contains the rest. For example:

```
?- splitAt(3, [a,b,c,d,e,f,g,h], A, B).
A = [a, b, c],
B = [d, e, f, g, h] ;
false.
```

For full credit, your solution must *not* be recursive.

Hint: you need only `length` and `append`. Recall that `append` works backwards as well as forwards:

```
?- append([a,b,c], [d,e,f], X).
X = [a, b, c, d, e, f].
```