Logic Programming
Tutorial 4: Nonlogical features

For discussion in week 6 (Oct. 20-24)

This tutorial relies on material covered in programming lectures 4 and 5. These concepts are also covered in LPN chapter 10-11.

1. **Cut & negation** Consider the following facts:

   ```
   r(1). r(2).
   s(1). s(3).
   ```

   Draw the depth-first proof search trees for the following queries, showing each solution as well as each failing branch, and indicating which branches are discarded by cuts.

   (a) `r(X), !, s(Y)`

   (b) `r(X), s(Y), !`

   (c) `r(X), \+(s(X))`

   (d) `r(X), !, \+(s(X))`

   (e) `r(X), \+(r(X))`

   (f) `\+(\+(r(X)))`

   (g) `\+(\+(r(3)))`

2. **Atom and list manipulation** A *substitution cipher* is a simple encryption method in which. For example, the Caesar cipher replaces shifts each letter in a string by 3:

   ```
   abcdefghijklmnopqrstuvwxyz
   xyzabcdefghijklmnopqrstuvw
   ```

   so that a simple message such as `hello world` would be encoded as `ebiil tloia`. Assume messages are atoms that may contain spaces, lower or uppercase letters, numbers, or punctuation. Define a predicate `caesar/2` that relates a message with its encoding.

   For this problem you will need to use a special predicate `atom_chars/2` that relates an atom with the list of characters of that atom.

3. **Input/output** Building on the solution to the previous part, define a predicate `encode/0` that repeatedly reads an atom from the input, encodes it using the Caesar cipher, and writes it to the output.

4. (*) **Using cut & negation** Consider the fibonacci program:

```
fib(0,0).
fib(1,1).
fib(N,P) :- N >= 2,
            M is N-1, fib(M,F),
            L is M-1, fib(L,G),
            P is F+G.
```

(a) Rewrite the program to avoid unnecessary backtracking by adding (meaning-preserving, or "green") cuts.

(b) Rewrite the program to avoid the explicit `N >= 2` test, using cuts. The resulting program should still work properly when called with a ground number $N \geq 0$, but does not have to work if $N < 0$.

(c) Rewrite the program to use negation-as-failure (covered in Tutorial 1) instead of the inequality test. Is this a good idea from the point of view of efficiency?

5. (**) **Collecting solutions** This problem uses the `setof/3`, `bagof/3` and `findall/3` predicates (discussed in Lecture 3 and LPN chapter 11) and the Simpsons database from Tutorial 1.

   (a) Use `setof` to define a predicate that calculates:

      i. the set of all descendants of a person.
      ii. the set of all people that have two or more descendants.
      iii. the set of all people that have no descendants.

      For part (iii), you may find the `person/1` predicate (defined in Tutorial 1) useful.

   (b) Using `findall` define a predicate `flatten/2` that takes a list of lists and flattens it to a list, so that on the success of `flatten(Xs,Ys)` each element of `Ys` is an element of an element of `Xs`.

      Experiment with `bagof` and `setof` instead of `findall`, with different inputs (and quantifiers) to see the differences in behavior.

6. (**) **Assert/retract** In this problem we will use the `assert/1` predicate. This is covered in Lecture 5, and LPN chapter 11. Essentially, `assert/1` adds a fact or clause to the program dynamically.

   Many Prolog implementations use *first-argument indexing*, meaning that it is often a lot faster at finding the next rule to apply if the first argument of the predicate is known.

   Use `assert/1` to define a goal `buildcaesar/0` that always succeeds by building a dynamic predicate `table/2` that tabulates the Caesar cipher table, and use this relation instead of `caesarchar/2`.

   Note: You will need to add a line `:- dynamic table/2.` to your Prolog file to declare `table/2` as a dynamic predicate.