1 Project 3

Due Date: 04/16 by 11:59p

Important Reminder: As per the course Academic Honesty Statement, cheating of any kind will minimally result in receiving an F letter grade for the entire course.

1.1 Aims of This Project

The aims of this project are as follows:

- To give you an introduction to programming in Prolog.
- To expose you to using pattern matching for both constructing and accessing data-structures.
- To familiarize you with using backtracking in Prolog.

1.2 Project Specification

Update your github repository with a directory `submit/prj3-sol` which contains a file `prj3-sol.pro` which contains implementations of the Prolog procedures documented in the skeleton file prj3-sol.pro.

1.3 Provided Files

The <./prj3-sol> directory contains the following:

prj3-sol.pro A skeleton file containing the specifications of the Prolog procedures you need to implement.

README A template README; replace the `XXX` with your name, B-number and email. You may add any other information you believe is relevant to your project submission.

1.4 Hints

You may choose to follow the following hints (they are not by any means required).

- First work on your solutions using Prolog's declarative semantics. Write facts and rules for a problem which states something about the problem domain.

Then think about the procedural semantics. It may be a good idea to include facts/rules corresponding to base cases before facts/rules corresponding to recursive cases (this should not affect correctness, but may result in different orders of solutions or a solution versus an infinite loop).

- Review the Prolog Programming Heuristics given in the course slides.

Hints for the individual exercise follow:

1. Write out a fact for the sum of a `leaf(V)`. Then write out a rule for sum of a `tree(L, V, R)`: recursively compute the sum of `L` and `R` and then compute the overall sum in terms of those results and `V` using `is/2`.
2. Write out a fact for the naive flatten of a `leaf(V)`. Then write out a rule for the naive flatten of `tree(L, V, R)`: recursively flatten the trees `L` and `R` and then compute the overall flatten in terms of those results and `V` using `append/3`.
3. Define an auxiliary Prolog procedure which accumulates the flatten of the tree in reverse order. Call the auxiliary procedure with this accumulator empty and reverse the result of the auxiliary procedure to get the overall flattened list.
4. Proceed as follows:
   (a) Review the material on DCGs from the slides.
   (b) The provided grammar is left recursive; transform it to replace the left recursion with the Kleene closure operator.
   (c) When using an imperative programming paradigm, the Kleene closure operator was implemented using loops. That will need to be implemented in Prolog using recursion.

   Introduce new non-terminals in your grammar, say `exprLoop` and `termLoop`. Write out right-recursive rules for each of these, using a separate rule for each operator. So you will have 3 rules for `exprLoop`: one for `'+'`, one for `'-'` and one for the base-case recognizing empty; similarly for `termLoop`.
   (d) Translate the transformed grammar from the previous step to DCG notation. For now, ignore constructing the AST.
   (e) Connect your DCG rules to the required `parse_arith/2`, leaving the AST variable uninstantiated.
   (f) Test until your `parse_arith/2` works as a recognizer, correctly accepting or rejecting different token lists.
   (g) Add an AST argument to your DCG. Note that the AST in the head of a rule will be constructed using the AST's constructed in the body; it may be a good idea to review the solution to the previous project.

(h) Iterate until you build the correct AST.

5. This problem is extremely straight-forward and illustrates the power of Prolog (my solution consists of 5 lines of code). The solution will depend on the backtracking behavior of `member/2`. To see how that would work, run the following goal:

```
member(s - 1 - S, [x - 0 - y, x - 1 - y,
                   s - 0 - x, s - 1 -x, s - 1 -t]).
```

   Once you understand how `member/2` can be used to drive the simulation, simply write out your `nfa_sim/2`, recursing on the list of input symbols. You will need one rule for the base case, succeeding if the input is empty and the current state is a final state, and one rule for the recursive case when the input is non-empty.