Lab 13: Logic Programming and Exam Review

Learning objectives:

- Gain experience solving problems in Prolog
- Review some topics in preparation for the final exam

Use the following commands to download and unpack the distribution code:

```
$ wget https://eecs390.github.io/lab/lab13/starter-files.tar.gz
$ tar xzf starter-files.tar.gz
```

1. *Logic puzzle.* Write a Prolog program that finds a solution to this logic puzzle:

   There are four different fathers: Smith, Baker, Carpenter, and Tailor. Each of them also has a son: Smithson, Bakerson, Carpenterson, and Tailorson.

   Assume you know the following:

   1. Each person works as either a smith, baker, carpenter, or tailor.
   2. No two of the parents share the same profession.
   3. No two of the sons share the same profession.
   4. No individual has a name that reflects their profession (e.g., Smith is not a smith, Bakerson is not a baker).
   5. No son has the same profession as his father.
   6. Baker has the same profession as Carpenter's son.
   7. Carpenter has the same profession as Tailor's son.
   8. Smith's son is a baker.

   What is each person's profession?

   Complete the following Prolog program to solve this puzzle:

   ```
   % professions(Professions).
   %
   % True if Professions is the list [smith, baker, carpenter, tailor].
   professions([smith, baker, carpenter, tailor]).

   % solve_puzzle(Smith, Baker, Carpenter, Tailor,
   %               Smithson, Bakerson, Carpenterson, Tailorson).
   %
   % Solves the puzzle described above.
   %
   % All parameters are output parameters.
   solve_puzzle(Smith, Baker, Carpenter, Tailor,
                Smithson, Bakerson, Carpenterson, Tailorson) :-
       Fathers = [Smith, Baker, Carpenter, Tailor],
       Sons = [Smithson, Bakerson, Carpenterson, Tailorson],
       % rule 1
       professions(Professions),
       fail. % replace with your solution
   ```

   Hint: Use the built-in `permutation` predicate to generate permutations of the professions.

2. *Logic queries.* For each of the following Prolog predicates and set of queries, do the following:

- Predict whether the query succeeds, fails, or results in a runtime error, **without running the code**.
- Run the code and queries. How do the results compare to your predictions?

a)
```
sum([], 0).

sum([First|Rest], Total) :-
    sum(Rest, RestTotal),
    Total is First + RestTotal.
```

Query	Succeeds	Fails	Error
`sum([1, 2], 3).`	◯	◯	◯
`sum([1, 2], 4).`	◯	◯	◯
`sum([1, A], 3).`	◯	◯	◯
`sum([1 | 2], Total).`	◯	◯	◯
`sum([1, 2], Total).`	◯	◯	◯

b)
```
contains([Item], Number) :-
    Item =:= Number.

contains([_|Rest], Number) :-
    contains(Rest, Number).
```

Query	Succeeds	Fails	Error
`contains([], 3).`	◯	◯	◯
`contains([3], 3).`	◯	◯	◯
`contains([-1], 3).`	◯	◯	◯
`contains([A], 3).`	◯	◯	◯
`contains([-1, 3, 7], 3).`	◯	◯	◯

3. *Review of functional ADTs.* Implement a stack functional data abstraction in Scheme that is created by a call to `make-stack`:

```
> (define stack (make-stack))
```

The resulting object should respond to the following messages:

- `'push`, with a subsequent item argument, pushes the item onto the top of the stack
- `'pop` removes the item that is on top of the stack and returns it
- `'top` returns the item on top of the stack without removing it
- `'size` returns the number of items in the stack

You do not have to do any error checking. The behavior of the stack is undefined if it receives a message that does not match an item in the list above, or if it receives the `'pop` or `'top` messages when it is empty.

The following is an example of using the stack created above:

```
> (stack 'size)
0
> (stack 'push 3)
> (stack 'push -1)
> (stack 'size)
2
> (stack 'top)
-1
> (stack 'pop)
-1
> (stack 'top)
3
```

Write your implementation in `stack.scm`. To test your implementation, run the included tests:

```
$ plt-r5rs stack.scm
```

4. *Review of uC.* The *sieve of Eratosthenes* is a method for computing prime numbers. Given a set that initially contains all the natural numbers starting from 2, the algorithm is as follows:

   1. Let k be the smallest number still in the set. It must be the case that k is prime, so add k to the result.
   2. Eliminate all multiples of k from the set.
   3. If any numbers remain in the set, go to step 1.

   In this problem, we will implement the sieve of Eratosthenes in uC. Our implementation will be *out of place*, meaning that we will produce a new array as output, without modifying the original array. We will work with arrays of integers (`int[]`). Write your code for all three parts in `sieve.uc`.

   a) First, implement the `range()` function. Given a start and stop value, the function returns an array that contains the values in the range `[start, stop)` in increasing order. Use the skeleton below.

   ```
   int[] range(int start, int stop) {
     return null; // replace with your code
   }
   ```

   b) Next, implement the `filter_not_multiple()` function. Given a factor and an array of integers, it returns a new array that contains all the integers from the input array that are **not** multiples of the given factor, preserving their relative order. For example, `filter_not_multiple(3, new int[] { 2, 3, 4, 5, 6, 7 })` returns a new array containing `2, 4, 5, 7`. Use the skeleton below.

   ```
   int[] filter_not_multiple(int factor, int[] numbers) {
     return null; // replace with your code
   }
   ```

   c) Finally, use `range()` and `filter_not_multiple()` to implement the `sieve()` function. Given an upper bound `limit` such that `limit >= 2`, the function returns all primes in the range `[2, limit)`, computing them using the sieve algorithm. For instance, `sieve(11)` returns an array containing `2, 3, 5, 7`. Use the skeleton below.

   ```
   int[] sieve(int limit) {
     return null; // replace with your code
   }
   ```