Lab 13: Template Metaprogramming and Exam Review

Learning objectives:

- Understand how template substitution works in C++
- Understand how function overloading works in combination with templates in C++
- 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. *Function templates.* In Python, a string can be multiplied by a non-negative integer *N*, which evaluates to a new string with the original string repeated *N* times:

   ```
   >>> 'abc' * 3
   'abcabcabc'
   >>> 3 * 'z'
   'zzz'
   ```

   Write a set of overloads in `mult.hpp` for a `mult()` function in C++ that performs this string repetition when the one argument is a string and the other a non-negative integer. On the other hand, if the two arguments both have numerical type, then `mult()` multiplies the two arguments and returns the result. The following are examples of calling `mult()`:

   ```
   std::cout << mult(3, 4.1) << std::endl;   // prints 12.3
   std::cout << mult("abc", 3) << std::endl; // prints abcabcabc
   std::cout << mult(3, "z") << std::endl;   // prints zzz
   ```

   Run the test cases as follows:

   ```
   $ g++ --std=c++17 mult_test.cpp -o mult_test.exe
   $ ./mult_test.exe
   ```

2. *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
```

3. *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)(/* add code here */) {
     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)(/* add code here */) {
     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)(/* add code here */) {
     return null; // replace with your code
   }
   ```

4. *Review of Prolog.* For this question, you must write your code in SWI-Prolog 9.

   - You may use the built-in unification (=), negation (\+ and \=), and list operations (`[`, `]`, `|`).
   - You may **not** use disjunction (`;`), nor any built-in predicates.
   - You may write any helper predicates you want.
   - Your predicates do **not** have to be "tail recursive".
   - Your code must **not** have singleton variables that do not begin with an underscore.

   Write your code in `remove.pl`.

a) Write a Prolog predicate `remove` that relates an item, a list, and another list that is the same as the first one but with all occurrences of the item removed. Examples:

```
?- remove(3, [3, 1, 3, 2, 3], X), !.
X = [1, 2].
?- remove(4, [3, 1, 3, 2, 3], X), !.
X = [3, 1, 3, 2, 3].
```

Your code must not produce incorrect solutions if the user asks for more solutions. Recall that the `findall` predicate finds all solutions to a query. Then `remove` should be defined such that:

```
?- findall(X, remove(3, [3, 1, 3, 2, 3], X), Solutions).
Solutions = [[1, 2]].
```

Hint: You will need to use negation in your solution.

```
% remove(Item, List, Result)
%
% True if removing all occurrences of Item in List results in Result.
%
% Item: in
% List: in, must be a list
% Result: in/out
% your code below
```

b) Now write a predicate `remove_all` that relates a list of items, a second list, and a third list that is the same as the second but with all occurrences of the items in the first list removed. Example:

```
?- remove_all([3, 1], [3, 1, 3, 2, 3], X), !.
X = [2].
```

You **must** use `remove` in your solution. You will not need negation in your code for `remove_all`.

```
% remove_all(Items, List, Result)
%
% True if removing all occurrences of the elements in Items from List
% results in Result.
%
% Items: in, must be a list
% List: in, must be a list
% Result: in/out
% your code below
```