CS 131 HW 1 Assessment

Set Functions:
I made these functions with ample use of the List.mem and List.filter functions. This allowed me to avoid using excessive recursive helper functions. The subset function was used to create the equal_sets function.

Computed Fixed Point Function:
The computed_fixed_point function simply applies the function f to the input x, then compares that result to the input x. If they're equal (through the eq function), then it returns x. If not, then it returns the result of running the computed_fixed_point function on the result.

Filter Reachable Function:
The filter_reachable function has multiple recursive parts. I originally wanted to encapsulate the entire thing in one function, but soon realized that I had too many dimensions to iterate through to do in one recursive function (multiple rules must be applied to multipled symbols per state, and all of it must be done for an increasing list of states). I first created a recursive function which attempts to apply a rule to one state, recursively iterating through all the symbols in that state. This function adds the newly created states into a growing list of new states (which are passed into the next level of the function, and eventually returned). Next, I created a recursive function which attempts to apply 1 rule to multiple states, maintaining both a list of new states and a list of used rules. Both of these are returned in a tuple. Predictibly, the next thing to do was to create a recursive function to apply a list of rules to a list of states. Since the last function adds to a list of used rules, this function can maintain a similar list, before eventually returning the used rules and the new set of states. Lastly, I created the filter_reachable function, which does a little bit of processing on the grammar input, before running compute_fixed_point on the start state, with a function which applies all the rules to all the given states. The equality function for compute_fixed_point simply checked whether the usedRules list for the input and output was the same. If it was, then that means that the search tree has been exhausted in terms of reachable rules.

I then realized that this entire approach was unnecessary, overcomplicated, and far too slow.
I deleted it all, and remade it with a new approach. This approach simply maintains a list of the reachable nonterminal symbols, starting with the start state symbol. I created a function which would take in all accessible nonterminal symbols and the list of rules, and then for every rule whose left-hand-side was a nonterminal symbol in the list of accessible nonterminal symbols, any nonterminal symbols in the right-hand-side of the rule would be added to the list of accessible nonterminal symbols.
In filter_reachable, the computed_fixed_point function is applied to this updateAccessibleNonT(erminalSymbols) function. When this stabilizes, the filter_reachable function then simply filters the list of rules for those whose left-hand-side symbol is included in the list of accessible nonterminal symbols, then returns a grammar with this new list of rules.
This approach was far faster and far simpler than the previous one.