Programming with relations, functions, and rules --- set and recursive queries  
  
We will practice writing set and recursive queries, the core of database and functional programming, respectively, and the core of logic and constraint programming.  You can find provided files at (folder attached: a3-qry)   
  
A. Sets and relations, as in Database programming  
  
Implement the following functions in Python.  In a while-loop that retrieves a witness, you may use Python's assignment expression, introduced in Python 3.8.  
  
  
1. Successors map.  Given a set of edges e, as a set of pairs, compute and return a Python dictionary, mapping each vertex that has successors to its set of successors.  This is exactly the adjacency list representation of e.  
  
  
2. Graph reachability.  Given a set of edges e, as a set of pairs, and a set of vertices s, compute and return the set of vertices r reachable from those in s by following a sequence of edges in e.  This is as discussed in class, with the slides on "Sets: incrementalize and implement".  
  
a. Define function "reach\_iter" that takes e and s as arguments, and returns r computed by precisely following the algorithm on Slide 46.  
  
b. Define function "reach\_inc\_chain" that takes e and s as arguments, and returns r computed by precisely following the algorithm on Slide 53.  
  
c. Define function "reach\_inc\_direct" that takes e and s as arguments, and returns r computed by precisely following the algorithm on Slide 55.  
  
Note that for b and c, you need to use the successors map from 1 to retrieve each successor in constant time.  
  
  
3. Testing and analyzing performance for graph reachability.  
  
a. Write a script to test your programs on the given data in file reach.in.1000 as well as on a number of randomly generated inputs (report what kinds of inputs you generated, include size and number) and compare the results of the three functions from A.2.  
  
b. Write a script to measure the running times of your programs on a number of inputs of different sizes (report how you did timing, include what interval you are timing, e.g., from before loading in input to after writing output), plot the results, and conclude what kind of curves you are getting.  
  
  
B. Recursive and higher-order functions, as in functional programming  
  
Implement the following functions in SML.  You may install SML/NJ (from <http://www.smlnj.org/>), or use the online environment of SML at <https://sosml.org/editor>.  
  
  
1. Recursive functions.  
  
List cut. A cut of a list l is a pair l1 and l2 such that l = l1@l2.  For example, the cuts of [1,2] are ([],[1,2]), ([1],[2]), and ([1,2],[]).  Write a function "cuts" that takes a list as input and returns the list of all its cuts.  
  
  
2. Higher-order functions as control abstraction  
  
Give expressions not involving explicit recursion (or iteration, which is equivalent to tail recursion) for the following functions.  You may use given functions map, reduce, accumulate, exists, forall, and through in file a3rec.ml (their definitions are recursive), but don't any extra recursion (or iteration, for that matter).  Actually, your code could be no more than two lines for each of these.  
  
a. Append.  This takes two lists, and returns a list that is the concatenation of the two lists.  This is also built in as operator @, but don't use @.  
  
fun append [] ys = ys  
  | append (x::xs) ys = x::(append xs ys);  
  
- append [1,2,3] [];  
val it = [1,2,3] : int list  
  
- append ["Spring", "Summer", "Fall"] ["Winter"];  
val it = ["Spring","Summer","Fall","Winter"] : string list  
  
b. Replicate.  This takes a value and a number, and returns a list with the value replicated n times.  
  
fun replicate x 0 = []  
  | replicate x n = x::(replicate x (n-1));  
  
- replicate "Help" 5;  
val it = ["Help","Help","Help","Help","Help"] : string list  
  
- replicate () 10;  
val it = [(),(),(),(),(),(),(),(),(),()] : (unit) list  
  
c. Unique.  This takes a list, and returns it with duplicates removed.  Your output don't need to return exactly the same list as shown.  
  
fun member x [] = false  
  | member x (y::ys) = (x=y) orelse member x ys;  
  
fun unique [] = []  
  | unique (x::xs) = let val u = unique xs  
                     in if (member x u) then u else x::u end;  
  
- unique ["u", "n", "i", "q", "u", "e"];  
val it = ["u","n","i","q","e"] : string list  
  
- unique [1,2,3];  
val it = [1,2,3] : int list  
  
- unique [1,1,1,2,2,2,3,3,3,2,1,3,2,4,1,2,3,1,4,3,2,3,4,3];  
val it = [1,2,3,4] : int list  
  
d. Prime.  This takes an integer, and returns whether it is a prime number.  
  
fun prime n =  
  let fun prime2 n m = if m>=n then true  
                              else if n mod m = 0 then false  
                       else prime2 n (m+1)  
  in (n>1) andalso prime2 n 2 end;  
  
- map prime (1 through 10);  
val it = [false,true,true,false,true,false,true,false,false,false] : bool list  
  
- prime 329;  
val it = false : bool  
  
(Note: To understand ML types in practical uses, see <https://www.smlnj.org/doc/Conversion/types.html>)  
  
  
C. Rules, as in logic programming  
  
Implement the following rules and queries in Prolog.  You may install either XSB (from <http://xsb.sourceforge.net/>) or SWI Prolog (from <https://www.swi-prolog.org/>), or use the online environment of SWI Prolog at <https://swish.swi-prolog.org/>.  Reminder: your README file must say which one you used, as part of how to run and test your code.  
  
  
1. Append.  You are given a definition of predicate append(Xs,Ys,Zs) meaning that the concatenation of Xs and Yu equals Zs.  You are asked to use it to define two predicates.  
  
a. Predicate suffix/2, such that querying suffix with two lists Xs and Ys will give false if Xs does not equal a prefix of Ys, and otherwise will print out the suffix of Ys after the prefix that equals Xs, followed by true.  For example, querying   
suffix([1,2], [1,2,3,4]) will print out [3, 4] followed by true.  
  
b. Predicate cut/1, such that querying cut with a list Xs will print out the list of all cuts of Xs followed by true.  For example, querying cut([1,2,3]) will print out the following followed by true.  
[], [1, 2, 3]  
[1], [2, 3]  
[1, 2], [3]  
[1, 2, 3], []  
  
  
2. Graph reachability.  Implement the graph reachability problem (take predicates source/1 and edge/2, and define predicate reach/1 to be true for all vertices reachable from the sources following the edges), including input and output functionalities (output all reachable vertices).  
  
You task is to try different ways of writing the rules, run test and analysis as in A.3, and understand and describe how termination and efficiency are affected by the order of rules, the order of hypotheses, and the use of tabling.  
  
  
3. Transitive closure and cycle detection.  Consider given edge/2 as in C.2.  
  
a. Define predicate path(a,b) to be true iff there is a sequence of edges leading from a to b.  
  
b. Define predicate cycle(a) to be true iff a is in a cycle.  
  
  
D. Constraints, as in constraint programming and mathematical programming  
  
Implement what we call the n-queens-plus problem in IDP and Clingo.  You may install them or use their online environments, at <http://verne.cs.kuleuven.be/idp/> and <https://potassco.org/clingo/run/>, respectively.  
  
We start with the well-known n-queens problems, where n queens are to be placed on an n-by-n board so that no two queens are in the same row, or same column, or same diagonal.  
  
We added a distance constraint: any two queens must be at least k units apart, where a unit is one horizontal or vertical square distance.  For example, two queens at (3,6) and (4,2) is 5 units apart, because 4-3 is 1 unit, 6-2 is 4 units, and 1 unit + 4 units is 5 units.  
  
We implement this n-queens-plus problem by modifying existing implementation of the n-queens problem in IDP and Clingo to add the distance constraint 2:  
  
1. With IDP, at <http://verne.cs.kuleuven.be/idp/>, (1) click "File" in the menu at the top and select the last entry "Problem Library", and (2) select "N-Queens Problem" in the list on the left.  You can click "Run" in the menu at the top to see the output displayed on the right.  
  
2. With Clingo, at <https://potassco.org/clingo/run/>, (1) select "n-Queens" in the box for "Examples" at the top, and (2) select "enumerate all" in the box for "reasoning mode" below the program.  You can click "Run!" just below to see the output displayed below that.  
  
You task is to try different ways of adding the distance constraint, run with different number of queens and different values of k, and determine, for 20 queens, the maximum value of k where the problem have solutions.  
  
  
E. Writing and running your program  
  
You may start with the provided files: a3sets.py for part A, a3rec.ml for part B, and a3rules.pl for part C (and a3extra.da for extra credits, describe below).  a3nqueens.idp and a3nqueens.lp contain the n-queens program from the online environments for IDP and Clingo, respectively.  
  
Your main programs must use these names.  
  
Your Python program (and DistAlgo program for extra credit) must run under Python 3.9, and must run with a command like the following and output the desired results:  
  
python.exe a3sets.py  
  
python.exe -m da a3extra.da  
  
  
F. Things to think about  
  
1. Can I make my program clearer?  
  
2. What are the pros and cons of different languages for implementing the same problem?  
  
3. Which language allows me to write which programs in the clearest and easiest way?  
  
4. Why in the world do we use all of DistAlgo/Python, SML, XSB or SWI Prolog, IDP, and Clingo to program these problems?  How about creating a single language to allow all problems to be expressed in the clearest and easiest way?  
  
  
G. Submission  
  
1. Prepare your submission by adding all your files (a3sets.py for part A, a3rec.ml for part B, a3rules.pl for part C,  a3queens.idp and a2queens.lp for part D, your README.txt, and any other files you used) to a zip file.  
  
2. Name your file exactly as "First\_Last\_ID\_Ax.zip", where First is your first name, Last is your last name, ID is your Stony Brook student ID, and x is the homework number.  
  
3. Submit the single zip file in Google classroom, and add no other files or comments.  
  
  
------------  
Extra credit I (10%)  
  
Do part A.2 in DistAlgo, an extension of Python that support proper quantification, comprehension, and aggregation, all with pattern matching.  In a while-loop that retrieves a witness, you may use DistAlgo's "some", not Python's assignment expression.  Then do part A.3 on the resulting DistAlgo functions.  
  
You can install DistAlgo using the following command with Python 3.9 and see additional information at <http://distalgo.cs.stonybrook.edu>.  
  
pip install --pre pyDistAlgo  
  
DistAlgo modules (files with suffix ".da) currently support Python 3.7 features, but allow import of Python modules (files with suffix ".py") that use Python 3.9 features.  
  
  
------------  
Extra credit II (10%)  
  
1. Write a function in Python, that generates the infinite sequence of natural numbers 0,1,2,...  
  
2. Do part B.2 using DistAlgo/Python, by defining your own map, reduce, and accumulate in DistAlgo/Python, and using DistAlgo's some and each and Python's range.  
  
  
------------  
Extra credit III (20%)  
  
Do parts C.2 and C.3 using DA-rules, an extension of DistAlgo with Datalog rules.  
  
You can request to do this only if you finished all parts before this, and I will give you access to DA-rules.  
  
  
------------  
Extra credit IV (20%)  
  
Do part D using DA-rules, which also contains a very preliminary extension for constraints.  
  
Do this only if you finished all parts before this.  
  
  
------------  
Extra extra credit (% TBD, based on effort and results)  
  
1. Study more problems that are programmed using sets, recursion, rules, and constraints, analyze and test these programs to show how to understand their correctness and efficiency, and/or improve the clarity and/or efficiency of the programs.   
  
2. Use programming with sets, recursion, rules, and constraints for applications.  An example: analysis of Python programs.  
  
3. Try new implementations.  Two examples:  
  
(1) do parts C.2 and C.3 using XSB's new Python interface that's blazing fast (link to XSB 4.1 Beta on <http://xsb.sourceforge.net/>)  
  
(2) do part D.1 using IDP's new implementation with z3 solver (<https://pypi.org/project/idp-engine/>)