**Preface / Readme**

All solutions were tested on the <http://swish.swi-prolog.org/> prolog interpreter. Solutions have all resulted in the desired results, as exemplified by the homework file. Only rules are given for the programs, as it is suggested by the homework that there is no particular dataset of facts to be used for all problems. Some solutions span more than one page - please evaluate the full solutions. Thank you for your time.

**Part One - Knowledge Base Manipulation**

**Question 1:** Write a Prolog predicate uncle(X,Y) that is true if X is Y’s uncle. Note that we are not considering uncles “by marriage”, meaning that for X to be Y’s uncle the two must be related by blood.

**Solution 1:**

sibling(X, Y) :- parent(Z, X), parent(Z, Y), X \= Y.

uncle(X, Y) :- male(X), sibling(X, PARENT), parent(PARENT, Y).

**Question 2:** Write a Prolog predicate halfsister(X,Y) that is true if X is Y’s half-sister.

**Solution 2:**

sibling(X, Y) :- parent(Z, X), parent(Z, Y), X \= Y.

full\_sibling(X, Y) :- parent(V, X), parent(V, Y), parent(W, X), parent(W, Y), X \= Y, W \= V.

half\_sibling(X, Y) :- sibling(X, Y), \+ full\_sibling(X, Y), X \= Y.

half\_sister(X, Y) :- half\_sibling(X, Y), female(X).

halfsister(X, Y) :- half\_sister(X, Y).

**Question 3:** By using the following database of facts, define new rules (using male/1, female/1 and parent/2) for the following family relations: (a) father (b) sister (c) grandmother (d) cousin.

**Solution 3:**

%% NOTE :

%% Sisters includes half sisters

%% Cousins only includes first-cousins, second cousins are not included

sibling(X, Y) :- parent(Z, X), parent(Z, Y), X \= Y.

full\_sibling(X, Y) :- parent(V, X), parent(V, Y), parent(W, X), parent(W, Y), X \= Y, W \= V.

half\_sibling(X, Y) :- sibling(X, Y), \+ full\_sibling(X, Y), X \= Y.

parent\_sibling(X, Y) :- sibling(X, PARENT), parent(PARENT, Y).

uncle(X, Y) :- male(X), parent\_sibling(X, Y).

aunt(X, Y) :- female(X), parent\_sibling(X, Y).

grandparent(X, Y) :- parent(X, PARENT), parent(PARENT, Y), Y \= X, PARENT \= X, PARENT \= Y.

father(X, Y) :- parent(X, Y), male(X), X \= Y.

sister(X, Y) :- sibling(X, Y), female(X).

grandmother(X, Y) :- grandparent(X, Y), female(X).

cousin(X, Y) :- parent\_sibling(Aunt\_OR\_Uncle, Y), parent(Aunt\_OR\_Uncle, X), X \= Y.

**Part Two - List Manipulation**

**Question 1:** Write a Prolog predicate sublist(X,Y) that is true if list X is a sublist

of list Y. A sublist is defined as the original list, in the same order, but in which some elements

may have been removed.

**Solution 1:**

sublist([], []).

sublist([A|B], [A|C]) :- sublist(B, C).

sublist(B, [\_|C]) :- sublist(B, C).

**Question 2:** Write a Prolog predicate has duplicates(X) that is true if list X contains duplicated elements (that is at least 2 copies of an element).

**Solution 2:**

sublist([], []).

sublist([A|B], [A|C]) :- sublist(B, C).

sublist(B, [\_|C]) :- sublist(B, C).

has\_duplicates([A|B]) :- sublist([A], B), !; has\_duplicates(B), !.

**Question 3:** Write a Prolog predicate has triplicate(X) that is true if list X contains triplicated elements (that is at least 3 copies of an element).

**Solution 3:**

sublist([], []).

sublist([A|B], [A|C]) :- sublist(B, C).

sublist(B, [\_|C]) :- sublist(B, C).

remove\_an\_instance(X, [X|Z], Z).

remove\_an\_instance(X, [A|Z], [A|R]) :- remove\_an\_instance(X, Z, R).

has\_triplicates([A|B]) :-

sublist([A], B),

remove\_an\_instance(A, B, R),

sublist([A], R), !;

has\_triplicates(B), !.

**Question 4:** Write a Prolog predicate remove third(X,Y) that is true if list Y is just list X with its third element removed. If X has no third element, then the predicate should fail.

**Solution 4:**

concat\_lists([], List, List).

concat\_lists([Elem | List1], List2, [Elem | List3]) :- concat\_lists(List1, List2, List3).

remove\_third([A, B, \_ | Z], L) :- concat\_lists([A, B], Z, L).

**Question 5:** Write a Prolog predicate remove nth(N,X,Y) that is true if list Y is just list X with its Nth element removed. If X does not have an Nth element then the predicate should fail. You can assume that N is strictly greater than 0.

**Solution 5:**

concat\_lists([], List, List).

concat\_lists([Elem | List1], List2, [Elem | List3]) :- concat\_lists(List1, List2, List3).

remove\_nth(N, List, NewList) :-

dont\_concat\_nth(N, 1, List, NewList).

dont\_concat\_nth(N, Index, [\_ | List1], NewList) :-

Index =:= N,

concat\_lists(List1, [], NewList).

dont\_concat\_nth(N, Index, [Elem | List1], [Elem | NewList]) :-

Index =\= N,

NewIndex is Index + 1,

dont\_concat\_nth(N, NewIndex, List1, NewList).

**Question 6:** Write a Prolog predicate remove every other(X,Y) that is true if list Y is just list X with every other element removed (the two lists should have the same first element).

**Solution 6:**

is\_odd(X) :- X /\ 1 =:= 1, !.

remove\_every\_other(List, NewList) :-

dont\_concat\_eventh(1, List, NewList).

dont\_concat\_eventh(\_, [], []).

dont\_concat\_eventh(Index, [\_ | List1], NewList) :-

\+ is\_odd(Index),

NewIndex is Index + 1,

dont\_concat\_eventh(NewIndex, List1, NewList).

dont\_concat\_eventh(Index, [Elem | List1], [Elem | NewList]) :-

is\_odd(Index),

NewIndex is Index + 1,

dont\_concat\_eventh(NewIndex, List1, NewList).

**Question 7:** Write a Prolog predicate to compute the length of a list.

**Solution 7:**

len\_list(List, Length) :-

gen\_list\_len(List, 0, Length).

gen\_list\_len([], Length, Length).

gen\_list\_len([\_|Rest], Counter, Length) :-

NewCounter is Counter + 1,

gen\_list\_len(Rest, NewCounter, Length).

**Question 8:** Provide a Prolog predicate to check if a list of numbers is in ascending order.

**Solution 8:**

is\_sorted([X|List]) :- is\_sorted(X, List), !.

is\_sorted(\_, []).

is\_sorted(X, [Y|List]) :- X < Y, is\_sorted(Y, List).

**Question 9:** Write a Prolog predicate mean/2 to compute the arithmetic mean of a given list of numbers.

**Solution 9:**

calc\_len(List, Length) :- len\_list(List, Length).

len\_list(List, Length) :-

gen\_list\_len(List, 0, Length).

gen\_list\_len([], Length, Length).

gen\_list\_len([\_|Rest], Counter, Length) :-

NewCounter is Counter + 1,

gen\_list\_len(Rest, NewCounter, Length).

calc\_sum(List, Sum):-

gen\_calc\_sum(List, 0, Sum).

gen\_calc\_sum([], Sum, Sum).

gen\_calc\_sum([This\_Val|Rest], Partial\_Sum, Sum) :-

New\_Partial\_Sum is Partial\_Sum + This\_Val,

gen\_calc\_sum(Rest, New\_Partial\_Sum, Sum).

mean(List, Mean) :-

calc\_sum(List, Sum),

calc\_len(List, Length),

Mean is Sum / Length.

**Question 10:** Write a Prolog predicate minimum/2 to find the smallest number within a given list of numbers.

**Solution 10:**

minimum(List, Global\_Minimum) :-

find\_minimum(List, [], Global\_Minimum), !.

first\_is\_minimum(\_, []).

first\_is\_minimum(X, Y):- X < Y.

find\_minimum([], Minimum, Minimum).

find\_minimum([X|List], This\_Minimum, Global\_Minimum) :-

first\_is\_minimum(X, This\_Minimum),

find\_minimum(List, X, Global\_Minimum).

find\_minimum([X|List], This\_Minimum, Global\_Minimum) :-

\+ first\_is\_minimum(X, This\_Minimum),

find\_minimum(List, This\_Minimum, Global\_Minimum).

**Part Three - Number Manipulation**

**Question 1:** Write a Prolog program factorial/2 to calculate a factorial.

**Solution 1:**

factorial(N, Factorial):-

calc\_factorial(N, 1, Factorial), !.

calc\_factorial(0, Factorial, Factorial).

calc\_factorial(N, Partial\_Factorial, Factorial) :-

New\_Partial\_Factorial is Partial\_Factorial \* N,

New\_N is N - 1,

calc\_factorial(New\_N, New\_Partial\_Factorial, Factorial).

**Question 2:** Write a Prolog predicate fibonacci/2 to compute the nth Fibonacci number.

**Solution 2:**

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% Calculates Number at index Number of fibonacci sequence

%%%%

%Indices : -1, 0, 1, 2, 3, 4, 5, 6 ,

%Sequence : 0, 1, 1, 2, 3, 5, 8, 13, ...

%%%%

fibonacci(0, 1) :- !.

fibonacci(1, 1) :- !.

fibonacci(N, Number):-

N =\= 0,

N =\= 1,

New\_N is N,

calc\_fibonacci(New\_N, 0, 1, Number), !.

calc\_fibonacci(0, \_, Number, Number).

calc\_fibonacci(N, Num\_Oldest, Num\_Old, Number) :-

New\_Number is Num\_Oldest + Num\_Old,

New\_Oldest is Num\_Old,

New\_Old is New\_Number,

New\_N is N - 1,

calc\_fibonacci(New\_N, New\_Oldest, New\_Old, Number).

**Question 3:** Write a predicate factor/2 to compute the prime factorisation given integer > 2.

**Solution 3:**

%% NOTE : Prime factorization is not in ascending order as it was was not asked for.

%%%%%%%%%%%%%%%%%%%%%%%%

%% Evaluate if Prime

%%%%%%%%%%%%%%%%%%%%%%%%

is\_prime(Number) :-

Smaller\_Number is Number - 1,

has\_no\_divisors(Number, Smaller\_Number).

has\_no\_divisors(\_, 1).

has\_no\_divisors(Number, Divisor) :-

Remainder is Number mod Divisor,

Remainder =\= 0,

New\_Divisor is Divisor - 1,

has\_no\_divisors(Number, New\_Divisor).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% Append List

%%%%%%%%%%%%%%%%%%%%%%%%%%%%

concat\_list([], L, L).

concat\_list([H|T], L2, [H|L3]) :-

concat\_list(T, L2, L3).

%%%%%%%%%%%%

%%% Eval if Factor

%%%%%%%%%%%%%

is\_a\_factor\_for(X, Factor):- (X mod Factor) =:= 0.

%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% Build Factors List

%%%%%%%%%%%%%%%%%%%%%%%%%%%%

factor(X, L):-

Number\_To\_Try is X - 1,

build\_factors\_list(X, Number\_To\_Try, [], L), !.

build\_factors\_list(\_, 1, List, List).

build\_factors\_list(X, Number\_To\_Try, Current\_List, Final\_List) :-

is\_a\_factor\_for(X, Number\_To\_Try),

is\_prime(Number\_To\_Try),

New\_Number is Number\_To\_Try - 1,

concat\_list(Current\_List, [Number\_To\_Try], New\_Current\_List),

build\_factors\_list(X, New\_Number, New\_Current\_List, Final\_List).

build\_factors\_list(X, Number\_To\_Try, Current\_List, Final\_List) :-

(\+ is\_a\_factor\_for(X, Number\_To\_Try) ; \+is\_prime(Number\_To\_Try)),

New\_Number is Number\_To\_Try - 1,

build\_factors\_list(X, New\_Number, Current\_List, Final\_List).

**Question 4:** Implement Euclids algorithm to compute the greatest common divisor(GCD) of two non-negative integers. This predicate should be called gcd/3 and, given two non-negative integers in the first two argument positions, should match the variable in the third position with the GCD of the two given numbers.

**Solution 4:**

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% GCD, GCD(A,B) = GCD(B,R)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

gcd(A, B, GCD) :-

A < B, find\_GCD(A, B, GCD), !.

gcd(A, B, GCD) :-

B < A, find\_GCD(B, A, GCD), !.

find\_GCD(0, X, X).

find\_GCD(Smaller, Larger, GCD) :-

Smaller =\= 0,

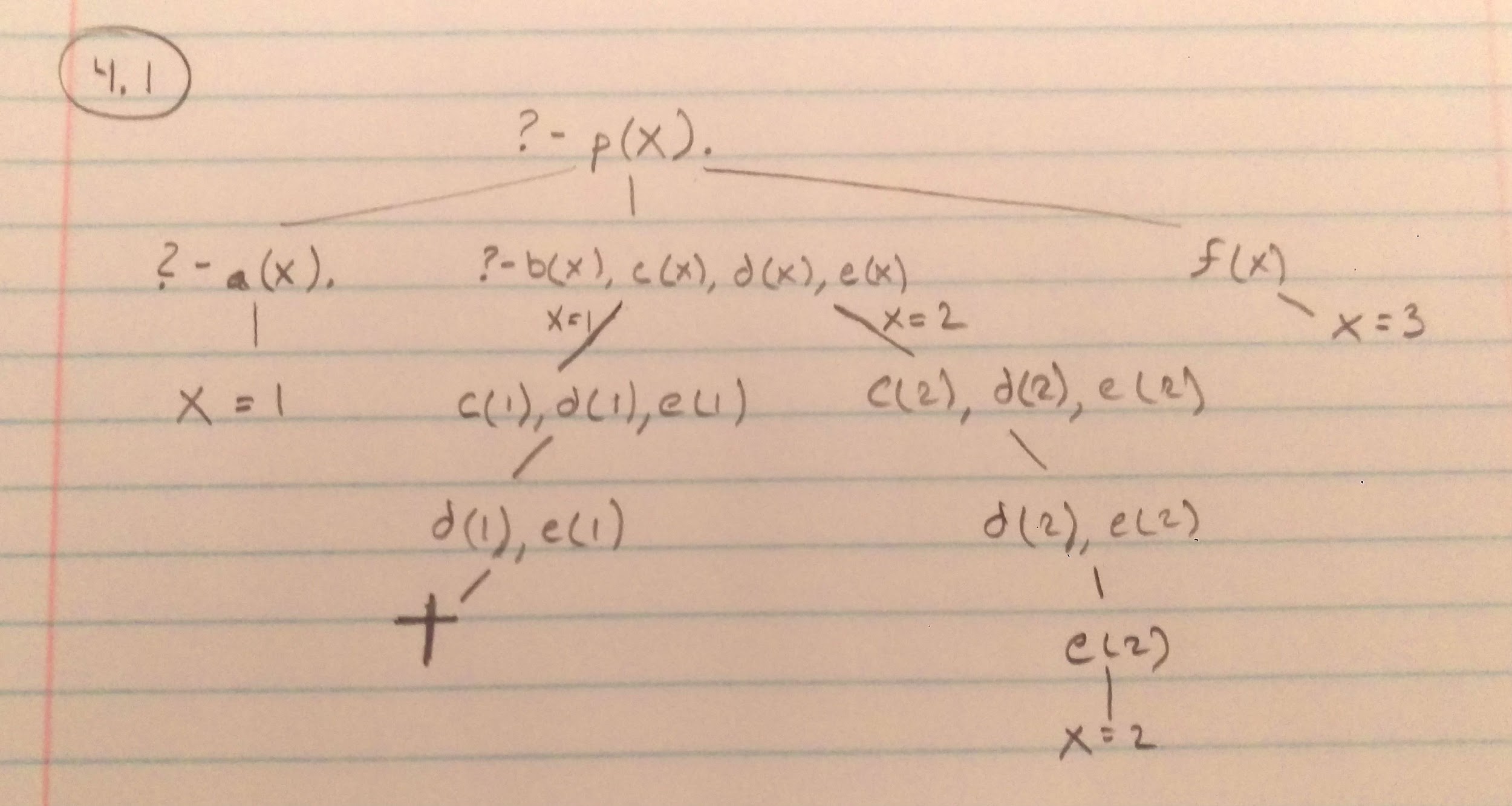
R is Larger mod Smaller,

find\_GCD(R, Smaller, GCD).

**Part Four - Cuts, Negation and Logic**

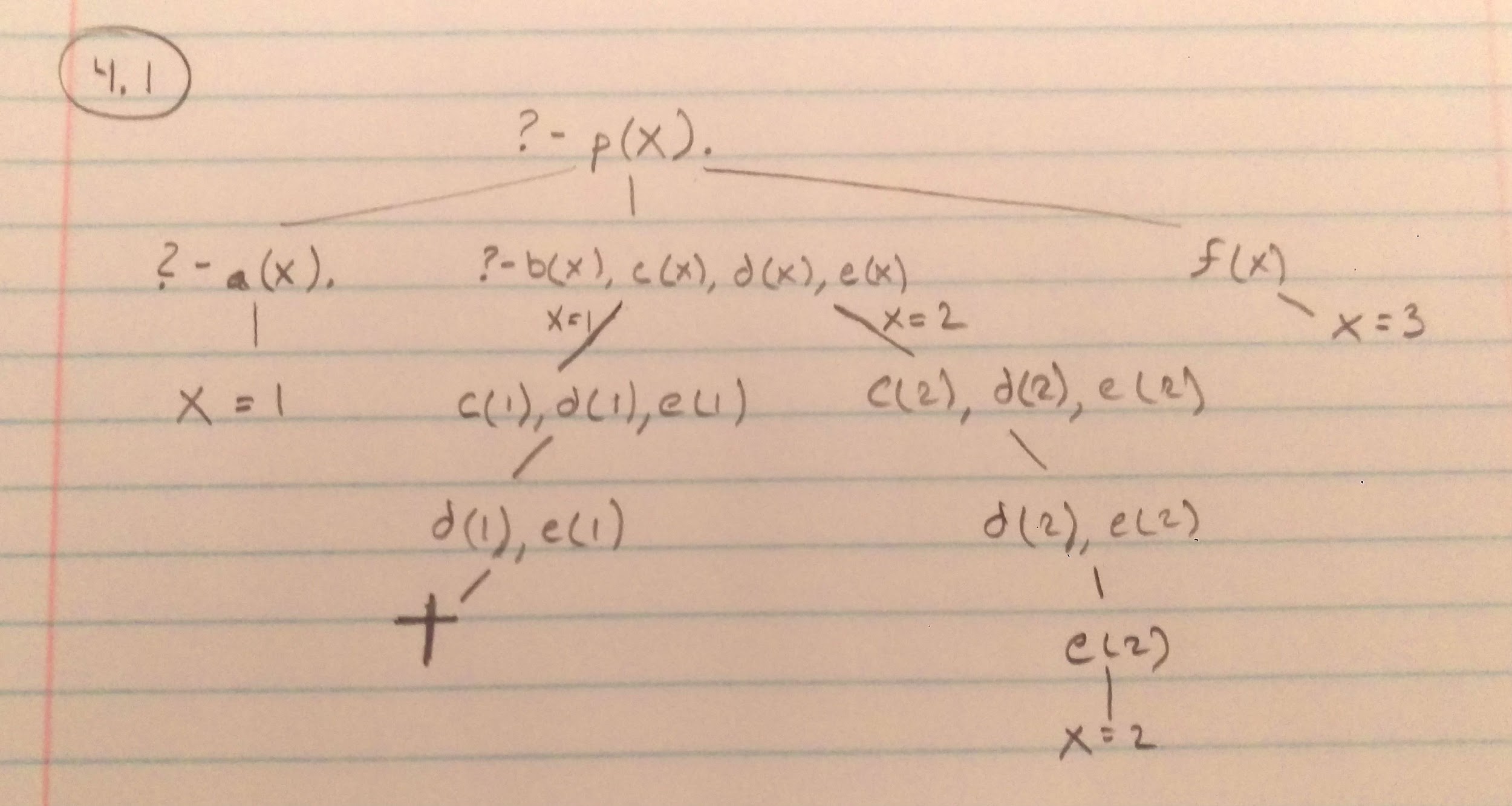
**Question 1:** Given the following Prolog program, draw the Execution Trace as a tree structure (Only draw the final tree).

**Solution 1:**



**Question 2:** Given the following Prolog program, draw the Execution Trace as a tree structure (Only draw the final tree).

**Solution 2:**



**Question 3:** Implement a Prolog predicate occurrences/3 to count the number of occurrences of a given element in a given list. Make sure there are no wrong alternative solutions.

**Solution 3:**

occurrences(Element, List, Occurences) :-

count\_occurences(Element, List, 0, Occurences), !.

count\_occurences(\_, [], Occ, Occ).

count\_occurences(Element, [This\_Element|Rest], Counter, Occ) :-

Element == This\_Element,

NewCounter is Counter + 1,

count\_occurences(Element, Rest, NewCounter, Occ).

count\_occurences(Element, [This\_Element|Rest], Counter, Occ) :-

\+ (Element == This\_Element),

count\_occurences(Element, Rest, Counter, Occ).

**Question 4:** Write a Prolog predicate divisors/2 to compute the list of all divisors for a given natural number. Make sure your program doesnt give any wrong alternative solutions and doesnt fall into an infinite loop.

**Solution 4:**

%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% Append List

%%%%%%%%%%%%%%%%%%%%%%%%%%%%

concat\_list([], L, L).

concat\_list([H|T], L2, [H|L3]) :-

concat\_list(T, L2, L3).

%%%%%%%%%%%%

%%% Eval if Factor

%%%%%%%%%%%%%

is\_a\_factor\_for(X, Factor):- (X mod Factor) =:= 0.

%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% Build Factors List

%%%%%%%%%%%%%%%%%%%%%%%%%%%%

divisors(X, L):-

Number\_To\_Try is X,

build\_factors\_list(X, Number\_To\_Try, [], L), !.

build\_factors\_list(\_, 0, List, List).

build\_factors\_list(X, Number\_To\_Try, Current\_List, Final\_List) :-

is\_a\_factor\_for(X, Number\_To\_Try),

New\_Number is Number\_To\_Try - 1,

concat\_list(Current\_List, [Number\_To\_Try], New\_Current\_List),

build\_factors\_list(X, New\_Number, New\_Current\_List, Final\_List).

build\_factors\_list(X, Number\_To\_Try, Current\_List, Final\_List) :-

\+ is\_a\_factor\_for(X, Number\_To\_Try) ,

New\_Number is Number\_To\_Try - 1,

build\_factors\_list(X, New\_Number, Current\_List, Final\_List).