CS-6374: Computational Logic

Final Exam

**Abhijit Balaji - AXB190013**

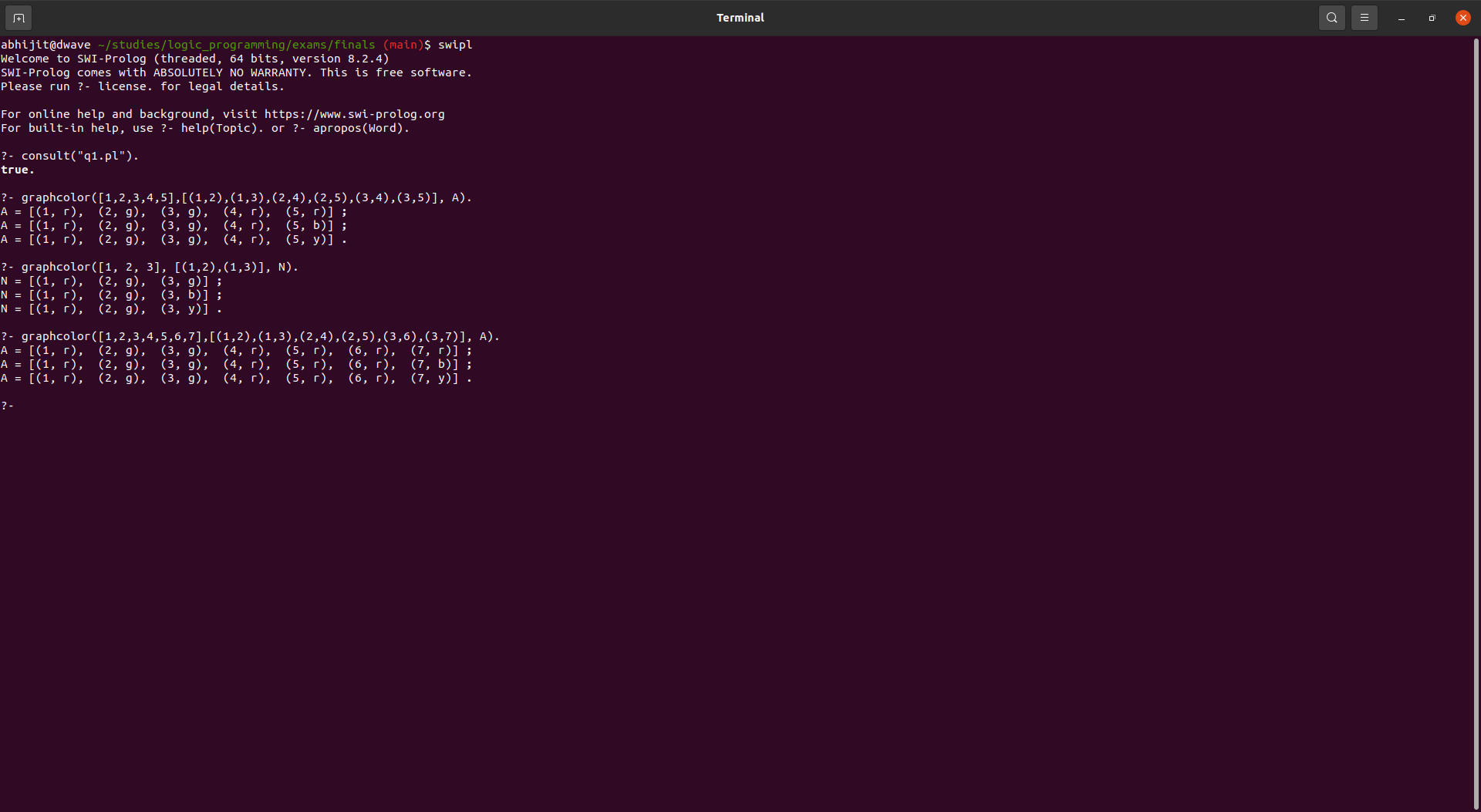
# Problem 1: Graph coloring using CLPFD

## Solution:

* I used the Test and generate paradigm in CLPFD to solve this problem
* First I wrote a predicate generate\_node\_colors which returns a list of all possible combinations of nodes and colors of format [(Node1, Color1), (Node2, Color2), ...]
* Then I bind the colors to domain 10,11,12,13 (10 for ‘r’, 11 for ‘g’) etc.
* Then I wrote another predicate **color\_different** which is False if adjacent nodes have the same color.
* Then I wrote another predicate **map\_index2color** to map 10 to “r”, 11 to “g”, 12 to “b”, 13 to “y”.

|  |
| --- |
| :- use\_module(library(clpfd)).  % modified append to return a list when empty list is given  append([],L,[L]).  append([X|T], Y, [X|Z]) :- append(T,Y,Z).  graphcolor(Nodes, Edges, NodeColorsOutput) :-  generate\_node\_colors(Nodes, Colors, NodeColors),  Colors ins 10..13, % create all domains  color\_different(Edges, NodeColors),  labeling([ff], Colors),  map\_index2color(NodeColors, [], NodeColorsOutput).    % Generate all possible node colorings  generate\_node\_colors([],[],[]).  generate\_node\_colors([N|Nodes], [C|Colors], [(N, C)|NodeColors]) :-  generate\_node\_colors(Nodes, Colors, NodeColors).  % Test if adjacent colors are different  color\_different([], \_).  color\_different([(N1, N2)|Edges], Colors) :-  member((N1, C1), Colors),  member((N2, C2), Colors),  C1 #\= C2,  color\_different(Edges, Colors).  % maps the integer colors to letters  map\_index2color([], Acc, Output) :- Output = Acc.  map\_index2color([(N, C)|NodeColors], Acc, Output):-  (  C == 10 -> append(Acc, (N, 'r'), Output1);  C == 11 -> append(Acc, (N, 'g'), Output1);  C == 12 -> append(Acc, (N, 'b'), Output1);  append(Acc, (N, 'y'), Output1)  ),  map\_index2color(NodeColors, Output1, Output). |

## Output:



# Problem 2:

|  |
| --- |
| sub([],X).  sub([X|L], L1) :- sff([X|L2], L1), sub(L, L2).  sff(L, L).  sff(L1, [X|L]) :- sff(L1, L).  comm(X,Y) :- sub([X,Y] ,[a,t,s]), sub([X,Y] ,[a,s,t]). |

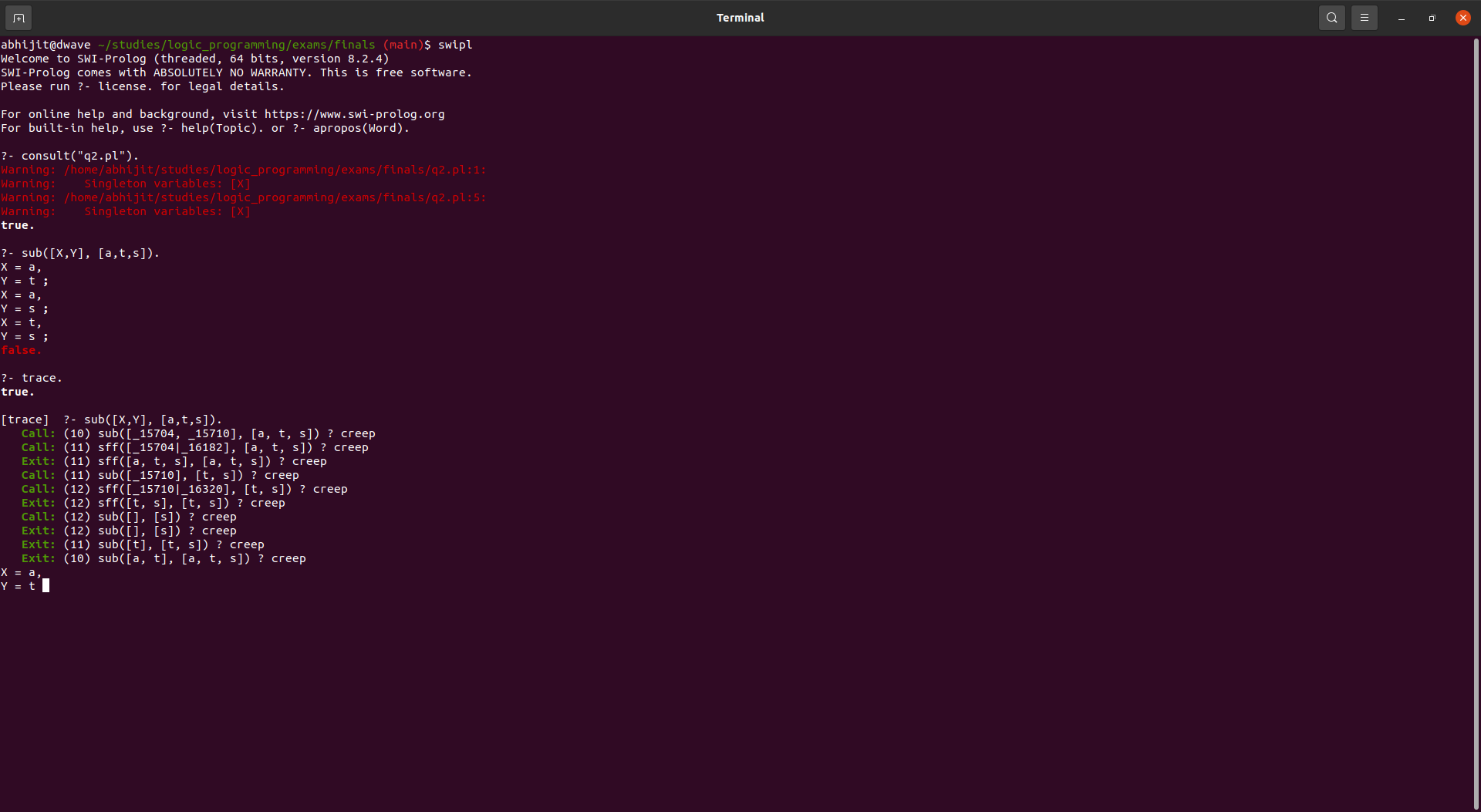
## What does the predicate sff do?

Ans: The predicate sff(A, B) returns the common tail between A and B where len(B) >= len(A). Thus

* sff([11,12,13,14,15],[1,2,3,4,5,11,12,13,14,15]) is **True.**
* *sff([11,12,13,14,15],[1,2,3,4,5,11,12,13,14]) is* **False**

## Show the search tree for query ?- sub([X,Y], [a,t,s]).

Ans:



Basically, The solutions are:

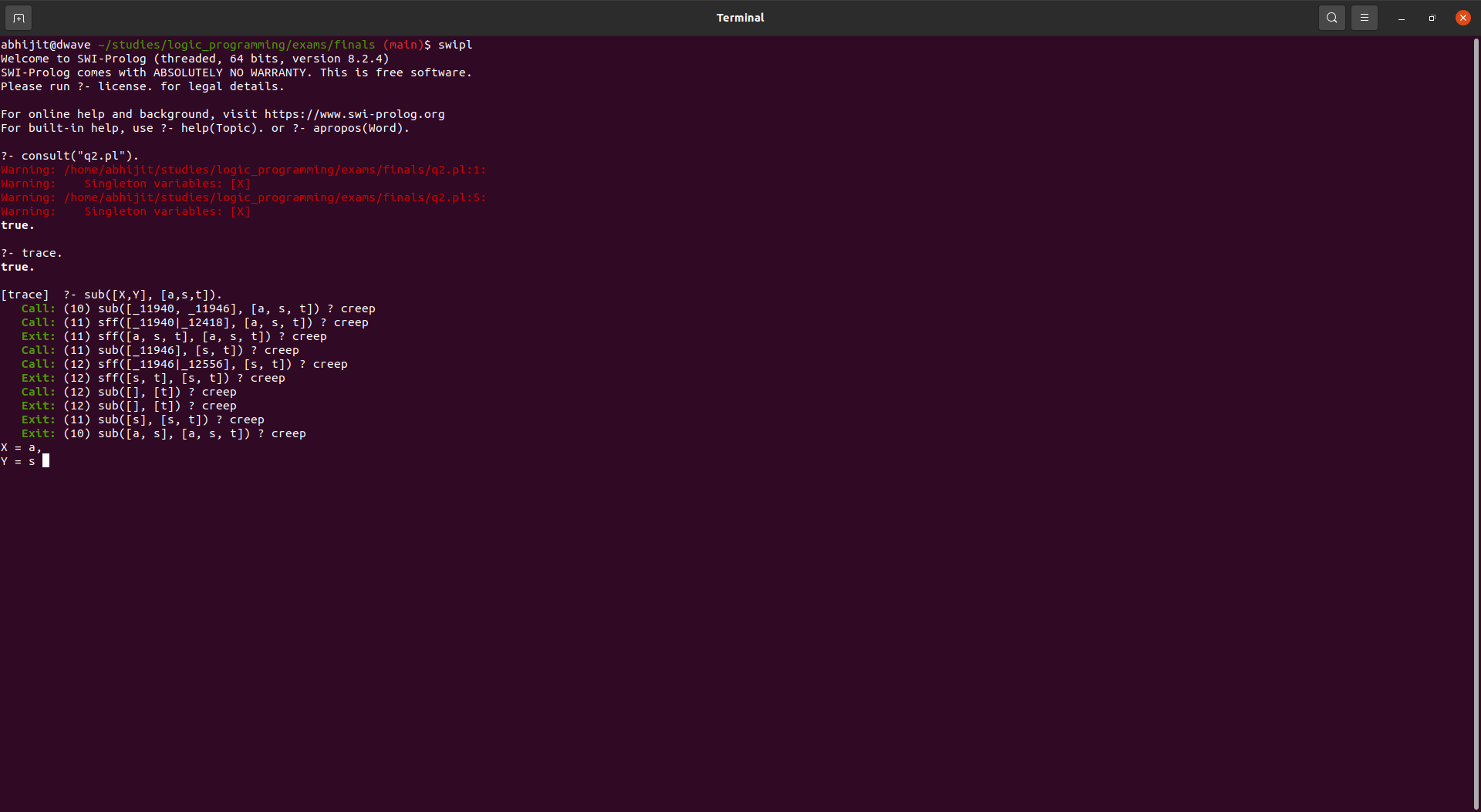
* X=a, Y=t;
* X=a, Y=s;
* X=t, Y=s;

Basically it returns all combinations of tuples of length 2.

I.e len(given \_list)C2 (C is the combinations formula)

## Show the search tree for query ?- sub([X,Y], [a,t,s]).

Similar to above:



Similar to above the solutions are:

* X=a, Y=s;
* X=a, Y=t;
* X=s, Y=t.

## Output of ?- comm(A,B).

Ans: This finds the common tuples of length 2 which is generated by the predicate sub using [a,t,s] and [a,s,t].

Thus outputs are:

* A = a, B = t;
* A = a, B = s.

## 

# Problem 3: Define the following terms

## Definite Clause Grammar:

Ans:

* It is basically a notation to represent Context-Free Grammar in Prolog.
* Example:
* sentence --> noun\_phrase, verb\_phrase.
* noun\_phrase --> determiner, noun.
* Basically DCG is just a syntactic sugar for normal definite clauses.

For example:

The DCG:

**sentence --> noun\_phrase, verb\_phrase.**

Is the syntactic sugar representation of:

**sentence(S1,Output) :- noun\_phrase(S1,X), verb\_phrase(X,Output)**

## Coinductive Logic Programming

Ans:

* Coinduction is basically the dual of induction.
* Inductive definition has 3 components: initiality, iteration and minimality.
* Coinduction eliminates the initiality condition (base case) and replaces the minimality condition with maximality.
* Basically Co-inductive logic programming is an extension of logic programming where the proofs may be of infinite length.
* Basically it allows predicates such as:

generate\_integer([H|T]) :- integer(H), generate\_integer(T).

Without defining the base case thereby resulting in the proof being of infinite length and also the answer being of infinite length.

## Gelfond-Lifschitz:

Ans: It is the method for finding Answer sets / worlds especially in cyclical programs

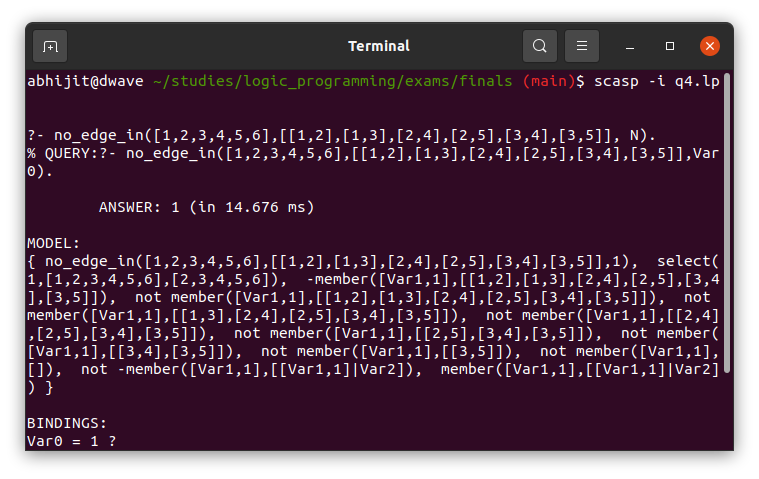
1. Given an answer set S (We guess it), for each p in S, delete all rules whose body contains “not p”
2. Delete all goals of the form “not q” in remaining goals
3. Compute the least fixed point, L, of the residual program
4. If S=L then S is an answer set.

# Problem 4: No Edges In

* For every node in the given nodes, check if it is **not** present as the second element of the tuple in the edges list

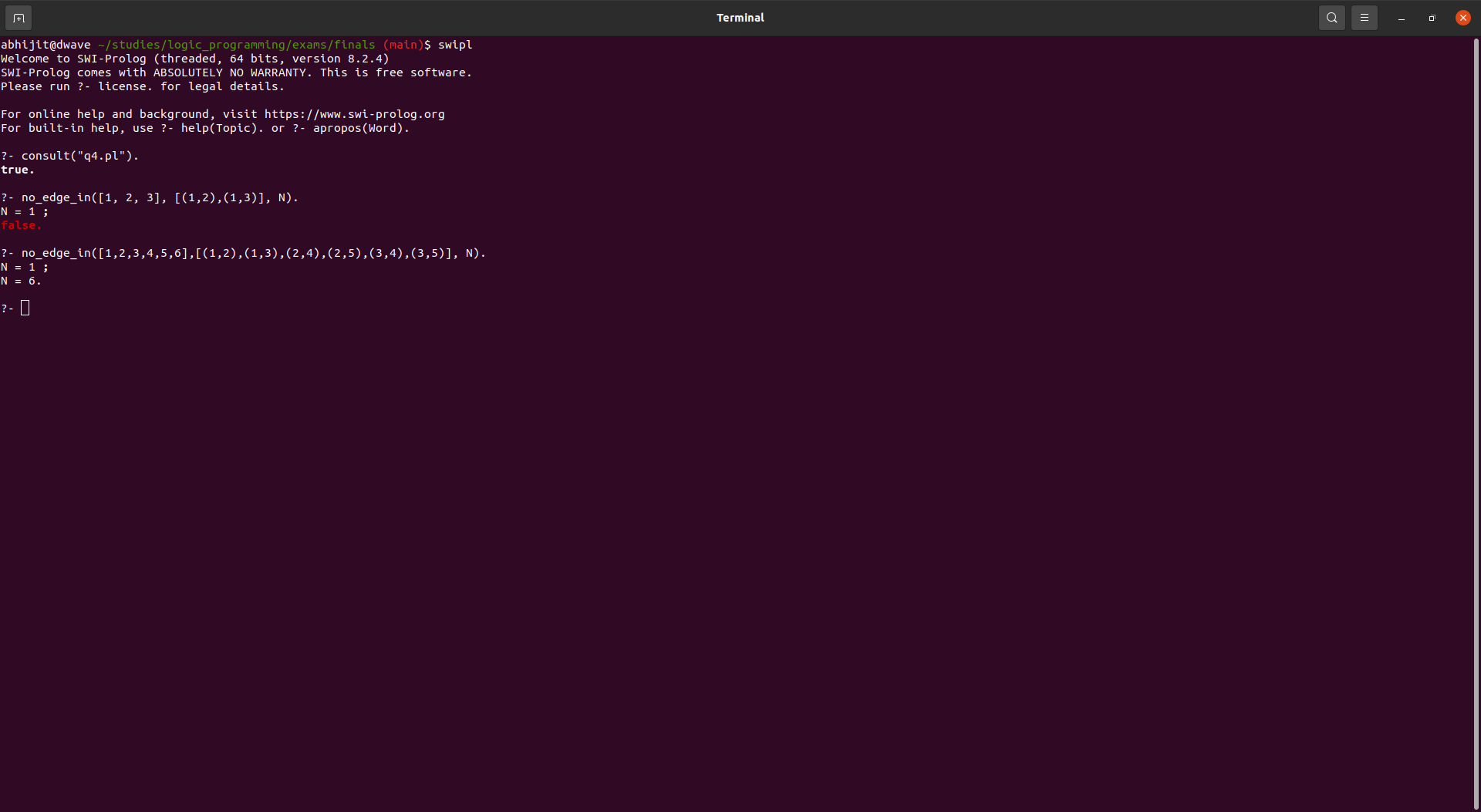
|  |
| --- |
| no\_edge\_in(Nodes,Edges,N):-  select(N,Nodes,\_),  -member([\_,N],Edges).  member(X,[X|\_]).  member(X,[\_|Tail]):- member(X,Tail).  % CWA  -member(X,L) :- not member(X,L).  select(X,[X|Xs],Xs).  select(X,[Y|Ys],[Y|Zs]):- select(X,Ys,Zs). |





# Problem 4: Alternate solution - This is much easier to write in prolog

|  |
| --- |
| no\_edge\_in(Nodes,Edges,N):-  select(N,Nodes,\_),  \+ member([\_,N],Edges). |



# 

# 

# Problem 5

## Define Abduction

Ans: In a nutshell abduction is assumption based reasoning. From the example given in the class, deduction is basically for every X if it is a swan(X) => it is white(X). Abduction is the reverse of this. I.e when we have white(10), then we can abduce that it is a swan(10). Thus abduction allows us to do assumption based reasoning. Basically, it means forming logical conclusions using what is known.

**swan(X) :- not -swan(X).**

**-swan(X) :- not swan(X).**

**white(X) :- swan(X)**

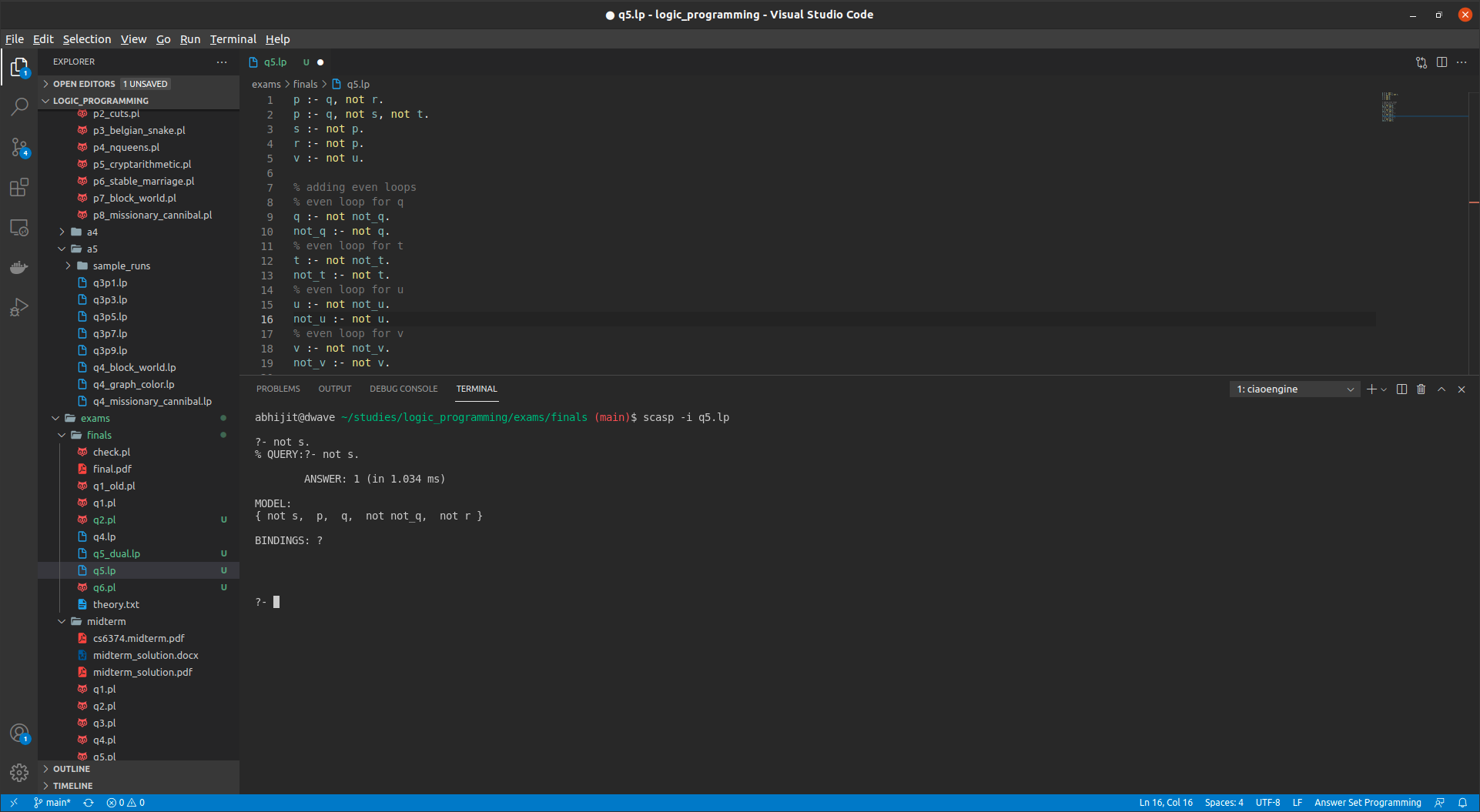
**The above rules say that either the given swan(X) is either True/False.**

**Thus, when we query: white(10), In a world where swan(10) is True we can abduce that white(10) is True.**

## Dual Rules

|  |
| --- |
| % Question  p :- q, not r.  p :- q, not s, not t.  s :- not p.  r :- not p.  v :- not u.  % Compute the dual rules for p, s and r.  %------------------------------------------------  %% Answer  % Dual for p  p :- (q, not r) ; (q, not s, not t).  % negating p using Demorgan's law  not\_p :- (not q ; r), (not q ; s ; t).  % Splitting into multiple rules  not\_p :- not q. % rule 1  not\_p :- not q, s. % rule 1 makes this rule redundant.  not\_p :- not q, t. % rule 1 makes this rule redundant.  not\_p :- r, not q.  not\_p :- r, s.  not\_p :- r, t.  % Dual for s  not\_s :- p.  % Dual for r  not\_r :- p. |

## Abductibles in sCasp



# Problem 6: Lights out

## Algorithm: Chasing the lights - For 5X5

* Look at the first row and see which positions are turned on
* Toggle the switches in the 2nd row where the 1st row light is on
* Repeat this until the last row - This is called as chasing the lights.
* This will result in a grid where only the lights are turned on only in the last row
* For a 5X5 board we will arrive at these 7 cases.

|  |  |
| --- | --- |
| Columns on in the bottom row | Column to push on the top row |
| 1,2,3 | 2 |
| 1,2,4,5 | 3 |
| 1,3,5 | 5 |
| 1,5 | 1,2 |
| 2,3,5 | 1 |
| 2,4 | 1,4 |
| 3,4,5 | 4 |

* Then repeat the same chasing the light process.
* But this time when you do it at the end you will have all the lights off.

|  |
| --- |
| % modified append to return a list when empty list is given  append([],L,[L]).  append([X|T], Y, [X|Z]) :- append(T,Y,Z).  % This solves only the 5X5 puzzle  % everything using 1 indexing  % removes the element  remove\_at(X,[X|Xs],1,Xs).  remove\_at(X,[Y|Xs],K,[Y|Ys]) :- K > 1,  K1 is K - 1, remove\_at(X,Xs,K1,Ys).  % insert an element X at position K in the list L and the result is R  insert\_at(X,L,K,R) :- remove\_at(X,R,K,L).  % Modify the element at position K with X in the list L and the result is R  modify\_at(X,L,K,R) :- remove\_at(\_, L, K, R1), insert\_at(X, R1, K, R).  % index into 2D list  % Uses 1 indexing.  get\_index2D(Grid, (RowIndex, ColIndex), Element):-  nth1(RowIndex, Grid, Row),  nth1(ColIndex, Row, Element).  % modifies elements in a given 2D index.  % This Modifies the element at position (R,C) with X in the 2D Grid and the result is OutGrid  modify\_index2D(X, Grid, (RowIndex, ColIndex), OutGrid):-  remove\_at(Row, Grid, RowIndex, RemovedGrid), % get the row  modify\_at(X, Row, ColIndex, ModifiedRow), % modify the row  insert\_at(ModifiedRow, RemovedGrid, RowIndex, OutGrid).  flip\_element(Grid, (R,C), FlippedGrid):-  % does not do any check for index. So check if the index is valid before calling this function  get\_index2D(Grid, (R,C), Element),  (Element == 0 -> modify\_index2D(1, Grid, (R,C), FlippedGrid); modify\_index2D(0, Grid, (R,C), FlippedGrid)).  check\_index(Grid, (R, C)) :-  length(Grid, MyLen),  R >= 1, R =< MyLen,  C >= 1, C =< MyLen.  flip\_one\_switch(Grid, (R,C), FlippedGrid):-  check\_index(Grid, (R,C)) -> flip\_element(Grid, (R,C), FlippedGrid); FlippedGrid = Grid.  % This flips the Grid as given in problem definition  % The given R,C and all the 4 adjacent cells are flipped  flip\_switch(Grid, (R,C), FlippedGrid) :-  % writeGrid(Grid),  % flip that position  flip\_one\_switch(Grid, (R,C), FG1),  R1 is R - 1, R2 is R + 1,  C1 is C - 1, C2 is C + 1,  % left flip  flip\_one\_switch(FG1, (R,C1), FG2),  % right flip  flip\_one\_switch(FG2, (R,C2), FG3),  % top flip  flip\_one\_switch(FG3, (R1,C), FG4),  % bottom flip  flip\_one\_switch(FG4, (R2,C), FlippedGrid).  % nl,  % writeGrid(FlippedGrid).  % driver predicate  get\_on\_switch\_in\_row(RowList, OutList):- get\_on\_switch\_in\_row(RowList, 1, [], OutList).  get\_on\_switch\_in\_row(RowList, C, Acc, OutList):-  length(RowList, MyLen),  C =< MyLen,  nth1(C, RowList, Element),  (Element == 1 -> append(Acc, C, Acc1); Acc1 = Acc),  C1 is C+1,  get\_on\_switch\_in\_row(RowList, C1, Acc1, OutList).  % base case  get\_on\_switch\_in\_row(RowList, C, Acc, OutList):-  length(RowList, MyLen),  C > MyLen,  OutList = Acc.  % base case  % flips the switches in the given row of the grid using a list of columns  flip\_row(Grid, \_, [], FlippedGrid) :- FlippedGrid = Grid.  flip\_row(Grid, R, [C|ColList], FlippedGrid):-  writeMove((R,C)),  flip\_switch(Grid, (R,C), FlippedGrid1),  flip\_row(FlippedGrid1, R, ColList, FlippedGrid).  % driver code  solve\_until\_last\_row(Grid, FlippedGrid):-  solve\_until\_last\_row(Grid, 1, FlippedGrid).  solve\_until\_last\_row(Grid, 5, FlippedGrid) :- FlippedGrid = Grid. % stop when you reach the last row  solve\_until\_last\_row(Grid, RowIndex, FlippedGrid):-  RowIndex<5,  % get all columns with switch on in that row  nth1(RowIndex, Grid, Row),  get\_on\_switch\_in\_row(Row, ColList),  RowIndex1 is RowIndex + 1,  flip\_row(Grid, RowIndex1, ColList, FG),  solve\_until\_last\_row(FG, RowIndex1, FlippedGrid).  solve\_last\_row(Grid, FinalAns) :-  % The seven cases are listed above putting an if else for the cases  nth1(5, Grid, LastRow),  get\_on\_switch\_in\_row(LastRow, ColList),  % Flip the corresponding first row  (  ColList == [1,2,3] -> flip\_switch(Grid, (1,2), FlippedGrid), writeMove((1,2));  ColList == [1,2,4,5] -> flip\_switch(Grid, (1,3), FlippedGrid), writeMove((1,3));  ColList == [1,3,4] -> flip\_switch(Grid, (1,5), FlippedGrid), writeMove((1,5));  ColList == [1,5] -> flip\_switch(Grid, (1,1), FlippedGrid1), writeMove((1,1)), flip\_switch(FlippedGrid1, (1,2), FlippedGrid), writeMove((1,2));  ColList == [2,3,5] -> flip\_switch(Grid, (1,1), FlippedGrid), writeMove((1,1));  ColList == [2,4] -> flip\_switch(Grid, (1,1), FlippedGrid1), writeMove((1,1)), flip\_switch(FlippedGrid1, (1,4), FlippedGrid), writeMove((1,4));  ColList == [3,4,5] -> flip\_switch(Grid, (1,4), FlippedGrid), writeMove((1,4));  FlippedGrid = Grid  ),  % repeat solve until last Row  solve\_until\_last\_row(FlippedGrid, FinalAns).  % The predicate to use  lights\_out(Grid):-  write("Given Grid: "), nl,  writeGrid(Grid),  write("The (Rows, Cols) to flip are given below: "), nl,  solve\_until\_last\_row(Grid, FlippedGrid),  solve\_last\_row(FlippedGrid, FinalAns),  write("The grid after following the above procedure: "), nl,  writeGrid(FinalAns).  % write the moves  writeMove((R,C)):- write(R), write(", "), write(C), nl.  % writes a given 2D grid to console  writeGrid([R1, R2, R3, R4, R5|\_]):-  writeRow(R1),  writeRow(R2),  writeRow(R3),  writeRow(R4),  writeRow(R5).  % writes a given row to the console. This is used by write grid  writeRow([A,B,C,D,E|\_]):-  write(A), write(" "), write(B), write(" "), write(C), write(" "), write(D), write(" "), write(E),nl. |

## Output:

## 

