# Problem 1: Rabbits and Chickens

## 1.A: Find the number of chickens and rabbits given total number of heads and legs

* Given the number of heads and number of legs, first find the number of rabbits.
* This can be found by solving the equations: 4\*R + 2\*C = Legs and R+C = Heads.
* Thus, I first found the number of rabbits using the first equation and found the number of chickens using the second equation.
* I Check if both the counts make sense. That is, check if both of them are integers and >=0.

## Code:

|  |
| --- |
| % Farm yard. Chicken and rabbits  countRabbits(Heads, Legs, C) :-  H is 2\*Heads,  C1 is Legs - H,  C is C1 / 2.  % solution A  find\_rabbits\_and\_chickens(Heads, Legs, RabbitCount, ChickenCount):-  % First find the number of rabbits  countRabbits(Heads, Legs, RabbitCount),  RabbitCount>=0, integer(RabbitCount),  ChickenCount is Heads - RabbitCount,  ChickenCount>=0, integer(ChickenCount). |

## Output:

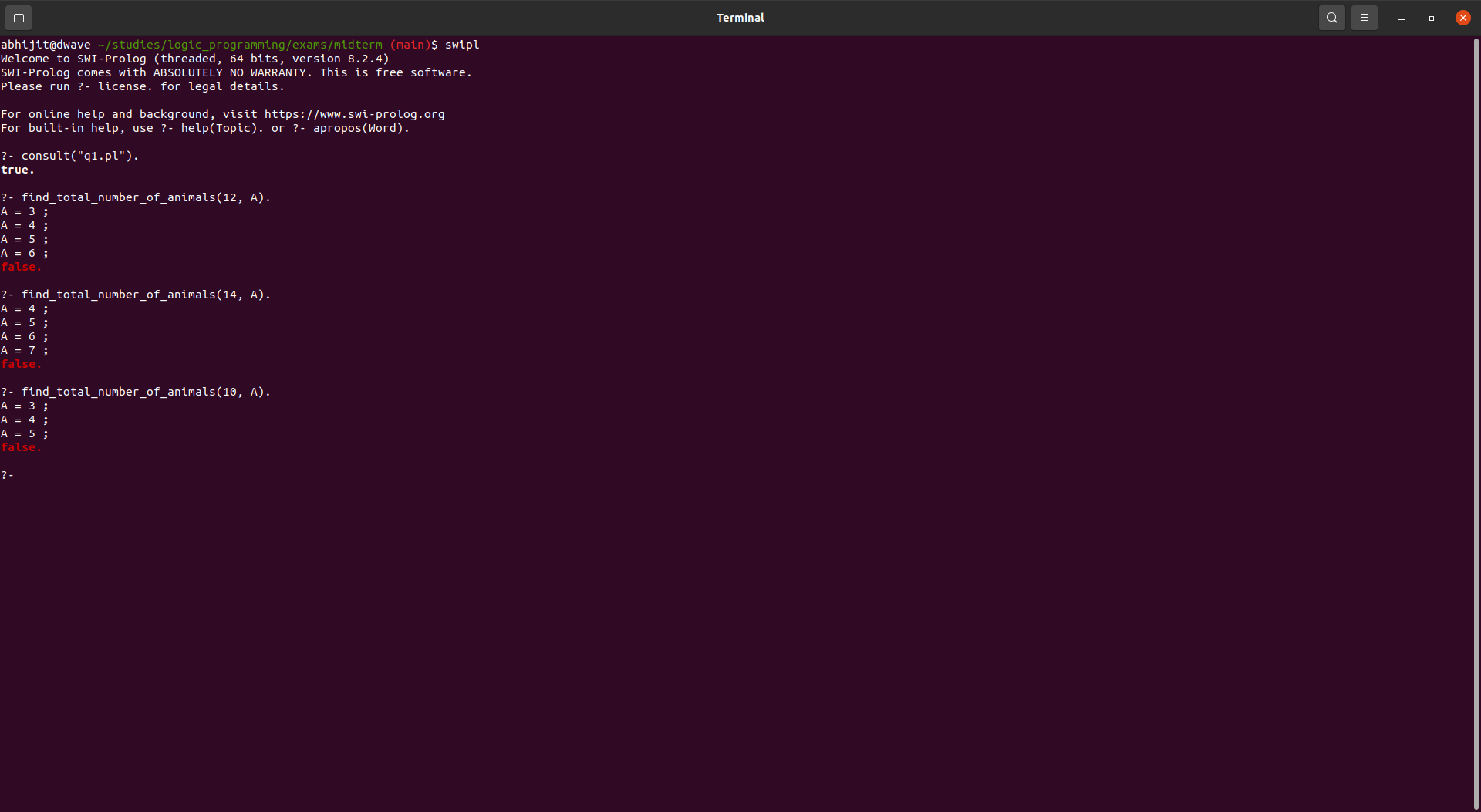
## 1.B: Find the total number of animals with just the number of legs given

* I used the **generate and test** paradigm for the second part.
* First generate n heads in range [1, 40] and get the counts of rabbits and counts of chickens given by the 1.A function are valid.That is, check if both of them are integers and >=0.
* If yes, then add the chicken and rabbit counts to get the total counts.

## Code:

|  |
| --- |
| % generate a list of numbers between M and N  range(M,N,[M|Ns]):- M<N, M1 is M+1, range(M1, N, Ns).  range(N,N,[N]).  % select  select(X, [X|Xs], Xs).  select(X, [Y|Ys], [Y|Zs]):- select(X,Ys,Zs).  % solution B  find\_total\_number\_of\_animals(Legs, TotalCount):-  % generate a list of heads from 1 to 40  range(1, 40, HeadList),  % select each head: This is the generation part  select(Heads, HeadList, \_),  % below is the test part  % this is from the first part of the code  find\_rabbits\_and\_chickens(Heads, Legs, RabbitCount, ChickenCount),  % make sure the Rabbit and chicken counts make sense  integer(RabbitCount), integer(ChickenCount), RabbitCount>=0, ChickenCount>=0,  TotalCount is RabbitCount + ChickenCount. |

## Output:



# 

# Problem 2: Weighted and Colored Blocks world

Approach:

* As given in the question I defined a new structure: **weight(block, weight, color)**.
* The only predicate that needs to be modified from the code given in the textbook is: **legal\_action(to\_block(Block1,Y,Block2),State).**
* In that predicate I added new predicates which checks if the weight of block1 > block2 and if it is, it checks if the weight difference is less than 3 ounces. I also made sure that the colors of block1 and block2 are different.

## Code:

|  |
| --- |
| % block's world with weights  /\*  transform(State1,State2,Plan) :-  Plan is a plan of actions to transform State1 into State2.  \*/  block(a).  block(b).  block(c).  place(p).  place(q).  place(r).  weight(a, 2.2, green).  weight(b, 6.1, red).  weight(c, 8.2, blue).  % on(X,Y) -> where X is block and Y is block/place, means X is on Y  % initial states are: [on(a,b) ,on(b,p) ,on(c,r)]  % final states are: [on(a,b) ,on(b,c) ,on(c,r)]  % Actions is a list of [to\_place(block,Y,Place)|to\_block(block,Y,block)]  transform(State1,State2,Plan) :-  transform(State1,State2,[State1],Plan).  transform(State,State,Visited,[]).  transform(State1,State2,Visited,[Action|Actions]) :-  choose\_action(Action,State1, State2),  update(Action,State1,State),  \+ member(State,Visited),  transform(State,State2,[State|Visited],Actions).  % two possible actions: moving to a place and moving to a block  % move block from Y(either a block or place) to place  legal\_action(to\_place(Block,Y,Place),State) :-  on(Block,Y,State), % block must be on Y  clear(Block,State), % block must be clear to move  place(Place), % Place must be a place  clear(Place,State). %  % check if moving Block1 to Block2 from Y is a legal action  % The predicate that needs to be modified  legal\_action(to\_block(Block1,Y,Block2),State) :-  on(Block1,Y,State), % Block 1 is on Y  clear(Block1,State), % Block 1 is clear to move  block(Block2), % Block2 is a block  Block1 \== Block2, % block1 and block2 are not same  % incorporating the weight and color constraints  weight(Block1, Weight1, Color1),  weight(Block2, Weight2, Color2),  % block 1 is going on top of block2  % Thus weight of block1 must not exceed the weight of block1 by 3  (Weight1>Weight2 -> WeightDiff is Weight1 - Weight2, WeightDiff < 3; true),  Color1 \= Color2,  clear(Block2,State). % Block2 is clear  % True if nothing is on top of A  clear(X,State) :- \+ member(on(A,X),State).  % True if X is on Y  on(X,Y,State) :- member(on(X,Y),State).  update(to\_block(X,Y,Z),State,State1) :-  substitute(on(X,Y),on(X,Z),State,State1).  update(to\_place(X,Y,Z),State,State1) :-  substitute(on(X,Y),on(X,Z),State,State1).  % substitute(X,Y,L1,L2), where L2 is the result of substituting Y for all occurrences of X in L1, e.g., substitute(a,x,[a,b,a,c],[x,b,x,c])  % is true, whereas substitute (a, x, [a,b, a, cl , [a,b ,x, c] ) is false  substitute(X,Y,[X|Xs],[Y|Xs]).  substitute(X,Y,[X1|Xs],[X1|Ys]) :- X \== X1, substitute(X,Y,Xs,Ys).  choose\_action(Action,State1,State2) :-  suggest(Action,State2), legal\_action(Action,State1).  choose\_action(Action,State1,State2) :-  legal\_action(Action,State1).  suggest(to\_place(X,Y,Z), State) :-  member(on(X,Z),State), place(Z).  suggest(to\_block(X,Y,Z), State) :-  member(on(X,Z),State), block(Z). |

## Output:



# Problem3: Change Maker

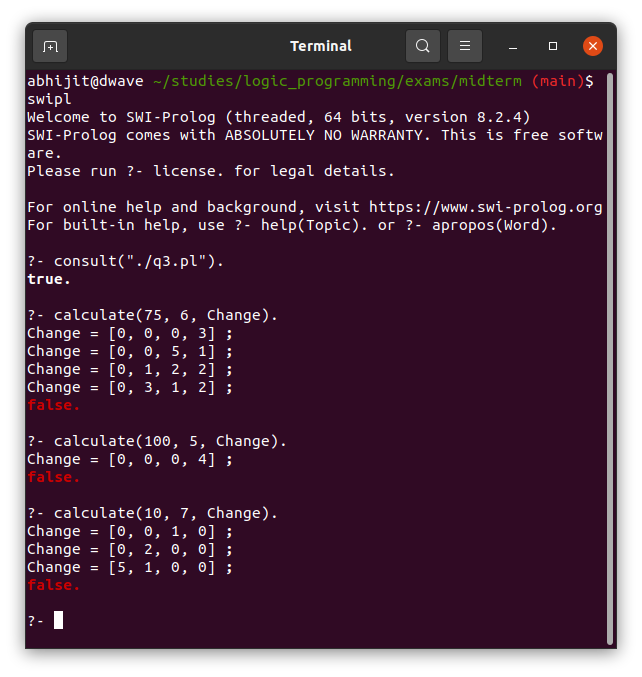
Approach:

* I used the generate test paradigm to solve this problem.
* First I generate a list of **[penny, nickel, dime, quarter]** with each value capped at the given max number of coins.
* Then for every list, I test if **P + 5\*N + 10\*D + 25\*Q=Amount**.
* If yes, I accept this solution and print them in order **[penny, nickel, dime, quarter].**

## Code:

|  |
| --- |
| /\*  Change Maker  P=1, N=5, D=10, Q=25  \*/  % generate a list of numbers between M and N  range(M,N,[M|Ns]):- M<N, M1 is M+1, range(M1, N, Ns).  range(N,N,[N]).  % select  select(X, [X|Xs], Xs).  select(X, [Y|Ys], [Y|Zs]):- select(X,Ys,Zs).  % create a list  generateChangesList(MaxCoins, ChangesList):-  range(0, MaxCoins, PennyList), select(P, PennyList, \_),  range(0, MaxCoins, NickelList), select(N, NickelList, \_),  range(0, MaxCoins, DimeList), select(D, DimeList, \_),  range(0, MaxCoins, QuarterList), select(Q, QuarterList, \_),  P+N+D+Q=<MaxCoins,  ChangesList = [P, N, D, Q].  validateChange([P,N,D,Q|\_], Amount):-  CurrAmount is P + 5\*N + 10\*D + 25\*Q,  CurrAmount == Amount.  % Generate Test Paradigm for finding the changes  calculate(Amount, MaxCoins, CoinList):-  generateChangesList(MaxCoins, CoinList),  validateChange(CoinList, Amount). |

## Output:



# Problem 4: MiniSudoku

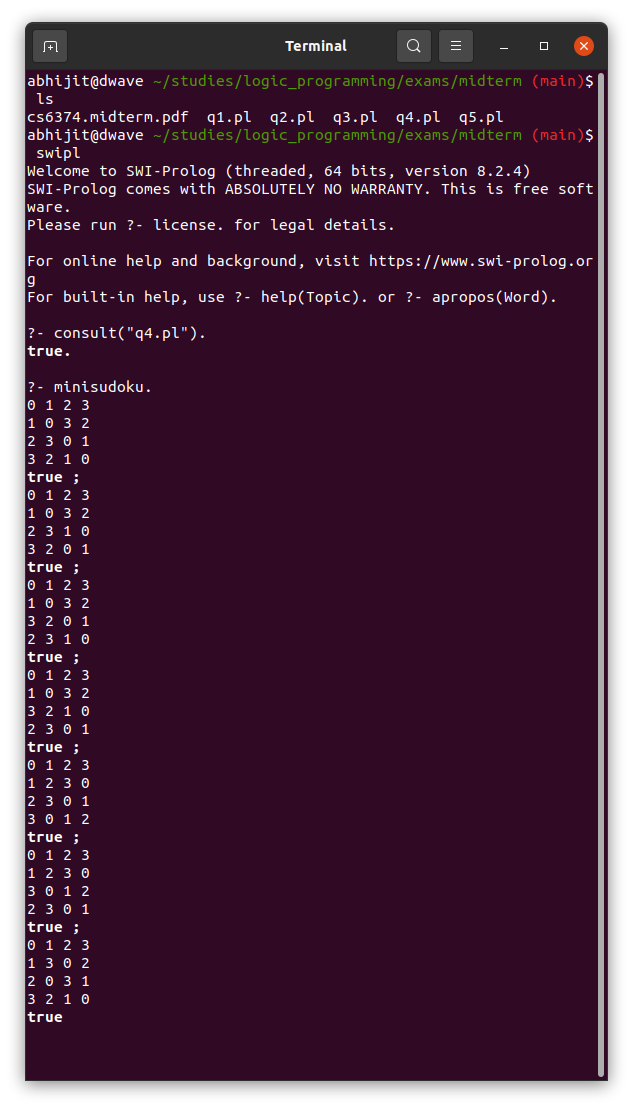
Approach:

* I used the **generate and test** paradigm to solve this problem.
* I generate a list of lists for eg: **[[0,1,3,2], [2,3,1,0], [1,2,0,3], [3,0,2,1]]** to model minisudoku. I call this list of list as grid
* I made sure that the rows are generated using **permutations** of **[0,1,2,3].** This **ensures that the rows have no repeating elements**.
* I created helper predicates: **getColumnVectors** which when given thegrid and the column index, it gives that corresponding columns as a list.
* I check if all the **4 column vectors are unique**.
* I accept the grid as a sudoku if the above test passes.
* I use write/1 to print each row line by line.

## Code:

|  |
| --- |
| % Sudoku  % True if a list is unique  allDiff(L) :- \+ (select(X,L,R), member(X,R)).  % index into 2D list  index2D(Grid, RowIndex, ColIndex, Element):-  nth0(RowIndex, Grid, Row),  nth0(ColIndex, Row, Element).  % Given a column index, get the column vector as a list  getColVector(Grid, ColIndex, ColVector):-  index2D(Grid, 0, ColIndex, A),  index2D(Grid, 1, ColIndex, B),  index2D(Grid, 2, ColIndex, C),  index2D(Grid, 3, ColIndex, D),  ColVector = [A, B, C, D].  % generate all possible permutations of the rows  generateRow(Row):-  permutation([0,1,2,3], Row).  % generate all possible grids  generateGrid2X2(Grid):-  generateRow(R1),  generateRow(R2),  generateRow(R3),  generateRow(R4),  Grid = [R1, R2, R3, R4].  % test if the given grid is a valid 2X2 sudoku  testGrid(Grid):-  getColVector(Grid, 0, A), allDiff(A),  getColVector(Grid, 1, B), allDiff(B),  getColVector(Grid, 2, C), allDiff(C),  getColVector(Grid, 3, D), allDiff(D).  % solution using generate test paradigm  minisudoku:-  generateGrid2X2(Grid), testGrid(Grid), writeGrid(Grid).  % writes a given 2D grid to console  writeGrid([R1, R2, R3, R4|\_]):-  writeRow(R1),  writeRow(R2),  writeRow(R3),  writeRow(R4).  % writes a given row to the console. This is used by write grid  writeRow([A,B,C,D|\_]):-  write(A), write(" "), write(B), write(" "), write(C), write(" "), write(D), nl. |

## Output:



# Problem 5: Decimal to AnyBase

Approach:

* The main **convert** predicate uses two predicates to solve the problem which are discussed below
* I separated the problem into two:
  + - One predicate for Decimal to Base 1
    - Another for Decimal to Base [2,16]
* For Decimal to Base n where n in [2, 16]:
  + Remainder = Decimal Number mod Base
  + I append the Remainder to a list
  + I recurse this process using Decimal=Decimal // 2.
  + The recursion base case is when the Decimal is < Base.
  + Now I append the Decimal as such to the list
  + I reverse the list
  + This reversed list is the final answer
  + And for printing a...f instead of 10...15, I use if else predicates as shown in the code.
* For Decimal to base 1, The idea is simple. I append 1’s to a list until the length of list = Decimal

## Code:

|  |
| --- |
| % Decimal base conversion  % modified append to return a list when empty list is given  append([],L,[L]).  append([X|T], Y, [X|Z]) :- append(T,Y,Z).  % tail recursive rev  rev(L1, L2) :- rev(L1, [], L2).  rev([], P, P).  rev([H|T], P, R) :- rev(T, [H|P], R).  % handle base 1 separately and all other bases separately  convert(N, Base, FinalAns):- Base > 1, convertToBaseN(N, Base, [], FinalAns).  convert(N, 1, FinalAns):- convertToBase1(N, [], FinalAns).  % handle base 1 separately  % recursive case  convertToBase1(N, CurrList, FinalAns):-  length(CurrList, N1),  N1<N,  append(CurrList, 1, CurrList2),  convertToBase1(N, CurrList2, FinalAns).  % base case  convertToBase1(N, FinalAns, FinalAns):- length(FinalAns, N1), N1 is N.  % handle all other bases here  % recursive case  convertToBaseN(N, Base, CurrList, FinalAns):-  N >= Base,  N1 is N // Base,  Value is N mod Base,  (  Value is 10 -> append(CurrList, 'a', CurrList2);  Value is 11 -> append(CurrList, 'b', CurrList2);  Value is 12 -> append(CurrList, 'c', CurrList2);  Value is 13 -> append(CurrList, 'd', CurrList2);  Value is 14 -> append(CurrList, 'e', CurrList2);  Value is 16 -> append(CurrList, 'f', CurrList2);  append(CurrList, Value, CurrList2)  ),  convertToBaseN(N1, Base, CurrList2, FinalAns).  % base case  convertToBaseN(N, Base, CurrList, FinalAns):- N<Base, append(CurrList, N, Ans), rev(Ans, FinalAns). |

## Output:

