Variable Unification

Unification is Prolog's matching technique.
1 Unbound variables unify with anything.
2 Constants or integers only unify with itself.
3 Structures only unify with other structures if: (a) same name and number of arguments, (b) all the corresponding arguments unify.

1) 100=10*10 → No, case 2 ("is" would work)
2) struct(A, b(C, d), e) = struct(X, X, Y) → Yes, case 3
3) [a,b,c] = [a|[b,c]] = [a,b|[c]] = [a,b,c|[]] → Yes, case 3
4) [1|[2, 3]] = [Last|First] → Yes, Last = 1, First = [2, 3]
5) [1, Y, 3] = [A|B], 25 = Y. → Yes
6) [[[a, b]] |c] = [H|T] → Yes, H = [[a, b]], T = c
7) [a(25, b), c(B), B|T] = [X, c(400), D, D] → Yes, B = 400, T = [400], X = a(25, b), D = 400
8) oh(MY) = MY → infinite recursion, unfies as MY = oh(**)
9) Ans = 25*4 → Yes, Ans = 25*4
10) [_] = [X|Y] → Yes, X = _, Y = []
11) Y is (6+6)/3 → Yes, Y = 4.0
12) 6+6 is 3*4 → No, 6+6 is a structure
13) Tallgeese is mobilesuit(0) → Error, "is" means it tries to evaluate mobilesuit as a mathematical operation
14) assert(match(stick)),retract(match(X)) → X=stick
15) assert(match(stick)),retract(match(X)),retract(match(X)) → False, last retract fails to find a match, whole query fails

List Manipulation

```
?- count([1,2,3],N). N = 3
count([],0).   % initializes Sum to 0 (when list has been traversed)

count([_|T],N) :-
count(T,Sum),% recursively traverse the list
        N is Sum+1. % increase counter during backtracking

?- tally([a,b,b,c],b,N). N = 2
tally(L,Q,_) :-
  count(L,Q,0). % call count with N initialized to 0

count([Q|T],Q,N) :-% if Q matches current element (i.e. head),
!,% cut to not search exhaustively
N1 is N+1,% increase counter
count(T,Q,N1). % recursively traverse the list with the new count

count([_|T],Q,N) :-% else Q does not match current element, (because above rule already checked for match)
%not(H = Q) % can add a line such as not(H = Q) or H \= Q to be safe, but not really needed
count(T,Q,N). % recursively traverse the list with the same count

count([],_,N) :-% if finished traversing the list, (i.e. list is currently empty)
        write('N = '),
        write(N),% print out value of N. if this isn't done, then backtracking causes N to go back to "someth
        nl.

?- removeAll(b,[a,b,b,c],M). M = [a,c]
removeAll(A,[A|Tail],NewList) :-
removeAll(A,Tail,NewList).

removeAll(A,[Head|Tail],[Head|NewTail]) :-
%not(Head = A),
removeAll(A, Tail, NewTail).

removeAll(_,[],[]).

?- removeItem(b,[b,a,b],M). M = [a,b]; M = [b,a]
removeItem(A,[A|L],L). % base case -- when match found, co

removeItem(A,[N|M],[N|L]) :- % adds non matching elements
  removeItem(A,M,L).

?- removeGen(all,b,[a,b,b],M). M = [a]
removeGen(one,A,L,M) :-  % remove one item
removeItem(A,L,M).

removeGen(all,A,L,M) :- % remove all items
removeAll(A,L,M).

?- writeList([hello,world]). 1 hello (nl) 2 world
writeList(L) :-
  writeList(L,0). % need to initialize counter to 0, only

writeList([H|T],N) :-
N2 is N+1,% increase counter
write(N2),% write counter
write(' '),% write white space
write(H),nl,% write current element
writeList(T,N2).% recursively traverse the list with the r

?- reverseItA([1,2],New). New = [2,1]
reverseItA(Old,New):- flip(Old,[],New).
flip([OldHead|OldTail],WorkingList,FinalList):-
  append([OldHead],WorkingList,NewWorkingList),
  flip(OldTail,NewWorkingList,FinalList).
flip([],CompleteWorkingList,CompleteWorkingList).

%#2 final list as *something*, set to [] at end of tree
reverseItB([OldHead|OldTail],New):-
  reverseItB(OldTail,ReturnedNew),
  append(ReturnedNew,[OldHead],New).
reverseItB([],[]).

%#3 identical to #1, but doesn't use append/3 predicate
reverseItC(Old,New):- flipC(Old,[],New).
flipC([OldHead|OldTail],WorkingList,Final):-
  flipC(OldTail,[OldHead|WorkingList],Final).
flipC([],CompleteWorkingList,CompleteWorkingList).

% #4
reverse(List,Reversed):- rev(List,[],Reversed).
rev([],Rev,Rev).
rev([A|T],L,Rev):- rev(T,[A|L],Rev).
```

```
% append two lists
join([],L,L).
join([X|R],S,[X|T]):- join(R,S,T).

intersection([],_,[]).
intersection([X|R],Y,[X|Z]):-
  member(X,Y), intersection(R,Y,Z).
intersection([X|R],Y,Z):-
  not(member(X,Y)),
  intersection(R,Y,Z).

?- union([1,2],[2,3],U). U = [1,2,3]
union([], X, X).
union([X|R],Y,Z):- member(X,Y), union(R,Y,Z).
union([X|R],Y,[X|Z]):- not(member(X,Y)), union(R,Y,Z)

member(X,[X|_]).
member(X,[_|T]):- member(X,T).

?- is_palindrome([b,o,b]). true
is_palindrome(L):- reverse(L,L). % reverse is built in

even([]).
even([_,_|T]):- even(T).

odd([_]).
odd([_,_|T]): -odd(T).

Create a list of integers from 0 to N
intlist(N,L):- N > 0, make_intlist(0, N, L).
make_intlist(M,N,[]):- M > N. % exit case
make_intlist(M,N,L):- M =< N, M2 is M + 1,
  make_intlist(M2,N,L2), % M+1,...,N
  append([M],L2,L).

item(bag). item(pc). item(pc).
?- setof(Stuff,item(Stuff),Set).
-> Set = [bag, pc].
?- bagof(Stuff,item(Stuff),Bag).
-> Bag = [bag, pc, pc].

delete_all(A,[A|T],T2):- delete_all(A,T,T2).
delete_all(A,[B|T],[B|T2]):- delete_all(A,T,T2).
delete_all(_,L,L).

Swap first and last elements in a list.
?- transfer([a,b,c],X). X = [c,b,a].
transfer([H|T],[H2|T2]):- swap(T,H,H2,T2).
swap([First,Second|Rest],H,H2,[First|T2]):-
  swap([Second|Rest],H,H2,T2).
swap([Last],H,Last,[H]).

Raise each element to a power of itself.
?-raise([1,2,3,4],X). X = [1, 4, 27, 256].
raise([],[]).
raise([H|T],[H2|T2]):- raise(T,T2), H2 is H**H.
```

```
% Find last element of a list
last(X,[X]). % base case
last(X,[_|L]):- last(X,L).

?- element_at(X,[a,b,c,d,e],3). X = c
element_at(X,[X|_],1). % base case
element_at(X,[_|L],K):- K > 1, K1 is K - 1,
  element_at(X,L,K1).

?- flatten([a,[[b],c]],X). X = [a, b, c]
flatten(X,[X]) :- \+ is_list(X).
flatten([],[]).
flatten([X|Xs],Zs):-
flatten(X,Y), flatten(Xs,Ys), append(Y,Ys,Zs).

factorial(0,1).
factorial(X,Y) :- X1 is X - 1, factorial(X1,Z),
  Y is Z*X,!.

?-remove_dups([1,2,2,3],L). L = [1,2,3]
remove_dups([],[]).
remove_dups([A|R],S):- % do not add if member
  member(A,R), remove_dups(R,S), !.
remove_dups([A|R],[A|S]):- remove_dups(R,S).
OR
compress([],[]).
compress([X],[X]).
compress([X,X|Xs],Zs):- compress([X|Xs],Zs).
compress([X,Y|Ys],[X|Zs]):- X\=Y, compress([Y|Ys],Zs).

?- is_prime(7). true.
is_prime(2). is_prime(3). % base cases
is_prime(P):- integer(P), P > 3, P mod 2 =\= 0,
  \+has_factor(P,3).
has_factor(N,L):- N mod L =:= 0.
has_factor(N,L):- L*L<N, L2 is L+2, has_factor(N,L2).

?- odd_calc([1,2,3,4,5],Ans). Ans = 3
odd_calc(List,Answer):- odd_add(List,0,Answer).
odd_add([LHead|LTail],Working,Answer):-
  NewWorking is Working+LHead,
  odd_sub(LTail,NewWorking,Answer).
odd_add([],Answer,Answer).
odd_sub([LHead|LTail],Working,Answer):-
  NewWorking is Working-LHead,
  odd_add(LTail,NewWorking,Answer).
odd_sub([],Answer,Answer).

?- duplicate([a,b],X). X = [a,a,b,b]
duplicate([],[]).
duplicate([X|Xs],[X,X|Ys]):- duplicate(Xs,Ys).

?- pow(2,3,X). X = 8
pow(_,0,1).
pow(X,Y,Z) :- Y1 is Y-1, pow(X,Y1,Z1), Z is Z1*X.
```

Debugging

trace - step-by-step view of execution
CALL: is about to invoke a new goal/ match a new clause
RETRY: is doing backtracking(match another clause)
EXIT: a predicate call is successful, and is returning
FAIL: a goal cannot be satisfied, backtracking goes to another part of tree

```
forestDwelling(Animal) :-
  habitat(Animal,_), % add this line
  \+ tropical(Animal). %uses habitat
% Assign Animal to a habitat to work with variables

is_dog(lab). is_dog(poodle). is_dog(bulldog).
animal(monkey). animal(guinea_pig). animal(lab).

count_dogs([ListHead|ListTail],Count):-
  is_dog(ListHead), NewCount is Count+1,
  count_dogs(ListTail,NewCount).
count_dogs([ListHead|ListTail],Count):-
  \+ListHead, count_dogs(ListTail,Count).
% Doesn't actually *return* the final count

non_dog(A):-
  \+is_dog(A),
  animal(A).
% Doesn't get past \+is_dog term

grandfather(Gramps,Kid):-
  parent(Gramps,parent),
  parent(Parent,Kid).
% Note the 'parent' instead of 'Parent'

double([], []).
double([N | T], [N2|T2]) :-
  N2 is N*2,
  double(T, Soln2).
% Doubles all elements in list, but Soln2\=T2

% Move head of one list to head of another
movehead([ListAHead|ListATail],ListB,NewList):-
  append(ListAHead,ListB,NewList).
% ListAHead should be [ListAHead]
```

Stacks & Queues

```
%push(Item,Stack,NewStack)
push(Item,Stack,[Item|Stack]).

%pop(Item,Stack,NewStack)
pop(Item,[Item|Rest],Rest).

%top(Top,Stack). Top = top_of_stack
top(Top,[Top|_]).

is_empty([]).

%Queue
shove(Item,Queue,New):-
  append(Queue,[Item],New).
yank(Item,[Item|Stack],Stack).
```

Built-in Functions

random(X), write(X), nl,
read(X) ← reads next term from input stream,
get(X) ← read single character,
tab(X) ← print X spaces,
put(X) ← write single character,
see(X) ← opens input stream to come from file X (default 'user'),
seeing(X) ← indicates current input stream,
seen ← closes input stream, resets input as user,
tell(X) ← opens file X as target for output stream,
telling(X) ← where you are writing to,
told ← closes output stream, resets to user,
sort(L,S) ← sorts the list, but not nested lists,
setof(T,G,Set) ← unique elements and sorted

Cuts (!)

1. All the clauses before the first clause with a cut are executed with normal backtracking.
2. If the goals before the cut never succeed, the cut does not activate, and the subsequent clause is used, as normal.
3. If the goals before the cut succeed, the cut activates:
a) backtracking back to goals before the cut cannot occur
b) backtracking to subsequent clauses after the one with the cut cannot occur → that clause with the activated cut is committed
c) the goals after the cut are executed with normal backtracking

Testing and Manipulation

var(X) ← succeeds if argument X is an uninstantiated variable,
nonvar(X) ← succeeds if argument X is instantiated to something,
atom(X) ← succeeds if X is a non-numeric constant,
integer(X) ← succeeds if X is an integer,
ground(X) ← succeeds if X is instantiated to something that has no uninstantiated variables eg. ground(s(1,X)) fails, but ground(s(1,2)) succeeds,
functor(T, F, N) ← succeeds if term T has functor name F and arity N,
name(A, X) ← converts atom A into list of ascii values (integers); works backwards too

Equality

= unifies (eg: A = cat → true)
== does not unify (eg: A == cat → false) they have to be identically equal

Binary Tree

Binary trees are very useful when they are sorted: keys on the left branch are less than the node, and keys on right are greater

```
% add_bintree(Key, OldTree, NewTree)...
add_bintree(Key, nil, tree(nil, Key, nil).
add_bintree(Key, tree(L, Key, R), tree(L, Key, R)).
add_bintree(Key, tree(L, K, R), tree(L2, K,R)) :-
     Key < K,
     add_bintree(Key, L, L2).
add_bintree(Key, tree(L, K, R), tree(L, K, R2) :-
     Key > K,
     add_bintree(Key, R, R2).
```

Definite Clause Grammar (DCG)

nonterminal→nonterminals
nonterminal→terminals
nonterminal→nonterminals & terminals
Terminals terminate when called. Eg) letter→[a]

Example 1:

```
s-->[a],[b].
s-->[a],s,[b].

?- s(X,[]). X = [a, b] ; X = [a, a, b, b] ; etc

?- s([a],[]). false.

?- s([a,b],[]). true.

?- s([a,b,b],[]). false.

?- s([a,b,b],X). X = [b].
% [a,b] is consumed and [b] is left over

?- s([a,b],X). X = [].

?- s([],X). false. % add s-->[] to be true
```

Example 2:

```
sentence-->noun_phrase(subject), verb_phrase.
verb_phrase-->verb, noun_phrase(object).
noun_phrase(_)-->determiner,noun. % eg: 'the cat'
noun_phrase(X)-->pronoun(X). % eg: 'he', 'him'

determiner-->[a]. determiner-->[the].
noun-->[cat]. noun-->[mouse].
verb-->[scares]. verb-->[hates].
pronoun(subject)-->[he]. pronoun(subject)-->[she].
pronoun(object)-->[him]. pronoun(object)-->[her].
OR
determiner-->[Word],{lex(Word,det)}.
noun-->[Word],{lex(Word,noun)}. % Word = noun
verb-->[Word],{lex(Word,verb)}.
pronoun(subject)-->[Word],{lex(Word,pro_sub)}.
pronoun(object)-->[Word],{lex(Word,pro_ob)}.
lex(a,det). lex(the,det).
lex(cat,noun). lex(mouse,noun).
lex(scares,verb). lex(hates,verb).
lex(he,pro_sub). lex(she,pro_sub).
lex(him,pro_ob). lex(her,pro_ob).
```

Constraint Logic Programming (CLP)

CLP (or DCG) question has something evil in it!

Example 1: List the possible values for X, Y, and Z
Note: \/ is a unification of domains. Thus, 1..3\/5..7 means "1 to 3, or 5 to 7".

```
[X,Y,Z] ins 1..3\/5\/7..9,
Y#=X+1,
Z#=Y+2.
```

X: 2
Y: 3
Z: 5

```
[X,Y,Z] ins 1..3\/5\/7..9,
Y#=X+1,
Z#>Y.
```

X: 1..2\/7
Y: 2..3\/8
Z: 3\/5\/7..9

Example 2: Calculate temparature

```
:- use_module(library(clpr)).
temperature(Celsius,Fahrenheit):-
  {Celsius*9/5+32=Fahrenheit}.
```

Assignment 3

Question 1: Tokenizer

```
?- tokenize.
|: hi there
[hi, there]

tokenize:-
  read_line_to_codes(user_input,String),
  parse(String,[],[]).

parse([],ListOfTokens,CurrentToken):- % base case
  reverse(CurrentToken,CompleteToken),
  name(Token, CompleteToken), % convert to chars
  Tmp = [Token|ListOfTokens], % add to list
  reverse(Tmp,Output), % reverse complete list
  write(Output), !.
parse([H|T], ListOfTokens, CurrentToken):-
  H \= 32, % add char to current token
  parse(T, ListOfTokens, [H|CurrentToken]);
  H = 32, % else add current token to list
  reverse(CurrentToken,CompleteToken),
  name(Token, CompleteToken),
  parse(T, [Token|ListOfTokens], []).
  % insert token into list, and reset current token

reverse(List,Reversed):- rev(List,[],Reversed).
rev([],Rev,Rev).
rev([A|T],L,Rev):- rev(T,[A|L],Rev).
```

```
% This version uses a built-in predicate
tokenizer(String):-
  concat_atom(Output,' ',String), write(Output).
```

Question 2: Postfix calculator
Uses the same tokenizer as Q1, but instead of
"reverse(Tmp,Output), write(Output), !."
It has "reverse(Tmp,List), !, calculate(List, [])." and...

```
push(Item,Stack,[Item|Stack]).
pop(Item,[Item|Rest],Rest).
top(Top,[Top|_]).

% Examples:
% ?- push(2,[],Stack), push(3,Stack,NewStack).
% Stack = [2],
% NewStack = [3, 2].

% ?- pop(H, [a,b,c],Stack).
% H = a,
% Stack = [b, c].

calculate([],Stack):- % base case
  top(Top, Stack), write('Solution: '), write(Top).
calculate([H|T], Stack):-
  % First check for operators
  H = +, % ADD the top 2 values
  pop(Operand2, Stack, NewStack), % pop 2 values
  pop(Operand1, NewStack, NewerStack),
  Result is Operand1 + Operand2, % apply operator
  push(Result, NewerStack, NewestStack), !,
  calculate(T, NewestStack); % continue
  .
  . similar for Subtract, Multiply, Divide
  .
  % Else push the operand onto the stack
  push(H, Stack, NewStack),
  calculate(T, NewStack).
```

Question 3: 0-1 Knapsack

```
knapsack(Capacity):-
  abolish(sack/1),
  asserta(sack([])), % contents of best sack
  abolish(capacity/1),
  asserta(capacity(Capacity)),
  abolish(maxW/1),
  asserta(maxW(0)), % best weight
  abolish(maxV/1),
  asserta(maxV(0)), % best profit
  abolish(item/3), % cleanup any traces
  readFile(input), % Linux
  writeln('Data loaded.'),
  writeln('Finding optimal loot...'),nl,
  bruteForce,
  maxV(V), maxW(W), sack(S),
  writeln(V), writeln(W), writeln(S),!.

readFile(X):-
  see(X), seeing(InStream),nl,
  repeat, read_line_to_codes(InStream, String),
  tokenize(String),
  String = end_of_file, !, seen.

tokenize(end_of_file). % terminal condition
tokenize(String):-
  String \= end_of_file, % don't tokenize eof
  parse(String,[],[]).

% parse and reverse are the same, except
% addToKB(Output) added before cut, delimiter is 9

addToKB([Object, Weight, Value |_]):-
  assert(item(Object,Weight,Value)),
  write(Object),
  write(': W='), write(Weight),
  write(', V='), write(Value),nl.

count([],0).
count([_|T],N) :- count(T,Sum),N is Sum+1.

perm(List,[H|Perm]):-
  delete(H,List,Rest), perm(Rest,Perm).
perm([],[]).

delete(X,[X|T],T).
delete(X,[H|T],[H|NT]):- delete(X,T,NT).

/* Create all variations */
modified_varia(0,_).
modified_varia(N,X):-
  Z is N-1,
  bagof(L, varia(N,X,L), LL),
  bruteForce(LL),
  modified_varia(Z,X).

varia(0,_,[]).
varia(N,L,[H|Varia]):-
  N > 0, N1 is N-1,
  delete(H,L,Rest), varia(N1,Rest,Varia).

bruteForce:-
  findall([X,Y,Z],item(X,Y,Z),Items),
  count(Items, N), modified_varia(N, Items).

/* Evaluate every variaton to find the optimal loot */
bruteForce([]).
bruteForce([H|T]):-
  computeLoot(H,0,0,H),
  bruteForce(T).

/* Compute the current loot variation */
computeLoot([], TotalW, TotalV, Copy):-
  maxV(MaxV),
```

```
  TotalV > MaxV ->
  retract(maxV(MaxV)), retract(maxW(_)),
  retract(sack(_)),
  asserta(maxV(TotalV)), asserta(maxW(TotalW)),
  asserta(sack(Copy)); !.
computeLoot([[_,ItemW,ItemV]|T], SackW, SackV, Copy):-
  (NewW is SackW + ItemW, capacity(C), NewW =< C) ->
  NewW is SackW + ItemW,
  NewV is SackV + ItemV,
  computeLoot(T, NewW, NewV, Copy); !.
```

Assignment 4

Question 1: Logic puzzle

```
:- use_module(library(clpfd)). % Finite Domains
:- writeln('Type "solve." to solve the logic puzzle').

% NOTE: #=, #\=, #<, #>, #>=, #=< to compare values

solve:-
  % Set appropriate domain values to variables
  [Alicia, Heidi, Matilda, Rhonda, Tabitha] ins 1..5,
  [Anderson, Copperfield, Goldsmith, Silverstein, Weaver] ins 1..5,
  [GQ, CW, EE, W, SD] ins 1..5,
  [Br, Ch, Cu, Po, Pr] ins 1..5,

  % Puzzle indicates each value in the list is unique
  all_different([Alicia, Heidi, Matilda, Rhonda, Tabitha]),
  all_different([Anderson, Copperfield, Goldsmith, Silverstein, Weaver]),
  all_different([CW, EE, GQ, SD, W]),
  all_different([Br, Ch, Cu, Po, Pr]),

  % Fill in the clues
  Alicia #\= Anderson,
  Alicia #= GQ,
  Alicia #\= Cu,
  Tabitha #\= Silverstein,
  Tabitha #\= Pr,
  EE #= Pr,
  EE #\= Weaver,
  Rhonda #= Ch,
  CW #\= Tabitha,
  Heidi #= Weaver,
  Heidi #\= Br,
  Matilda #\= Cu,
  Rhonda #\= CW,
  Goldsmith #= W,
  Goldsmith #\= Po,
  Copperfield #\= Matilda,
  Copperfield #\= Br,
  Br #= SD,

  % Search for solutions (will be in raw form)
  label([Alicia, Heidi, Matilda, Rhonda, Tabitha]),

  % Sort the solutions
  sort([Alicia, Heidi, Matilda, Rhonda, Tabitha],
    SortedNames),
  sort([Anderson, Copperfield, Goldsmith,
    Silverstein, Weaver], SortedSurnames),
  sort([GQ, CW, EE, W, SD], SortedMovies),
  sort([Br, Ch, Cu, Po, Pr], SortedSnacks),
  % Write the solutions
  report(SortedNames,SortedSurnames,
  SortedMovies,SortedSnacks,Alicia,Heidi,
  Matilda,Rhonda,Tabitha,Anderson,Copperfield,
  Goldsmith,Silverstein,Weaver,GQ,CW,EE,W,SD,
  Br, Ch, Cu, Po, Pr), !.

report([], [], [], [],
  _,_,_,_,_, _,_,_,_,_, _,_,_,_,_, _,_,_,_,_).
report([Name_H|Name_T], [Surname_H|Surname_T],
  [Movie_H|Movie_T], [Snack_H|Snack_T],
  Al, He, Ma, Rh, Ta,
  An, Co, Go, Si, We,
  GQ, CW, EE, W, SD,
  Br, Ch, Cu, Po, Pr):-
  writeName(Name_H, Al, He, Ma, Rh, Ta),
  writeSurname(Surname_H, An, Co, Go, Si, We),
  writeMovie(Movie_H, GQ, CW, EE, W, SD),
  writeSnack(Snack_H, Br, Ch, Cu, Po, Pr),
  report(Name_T, Surname_T, Movie_T, Snack_T,
    Al, He, Ma, Rh, Ta,
    An, Co, Go, Si, We,
    GQ, CW, EE, W, SD,
    Br, Ch, Cu, Po, Pr).

writeName(Name, A, H, M, R, T):-
  Name == A, write('Alicia ');
  Name == H, write('Heidi ');
  Name == M, write('Matilda ');
  Name == R, write('Rhonda ');
  Name == T, write('Tabitha ').
writeSurname(Surname, A, C, G, S, W):-
  Surname == A, write('Anderson: ');
  Surname == C, write('Copperfield: ');
  Surname == G, write('Goldsmith: ');
  Surname == S, write('Silverstein: ');
  Surname == W, write('Weaver: ').
writeMovie(Movie, G, C, E, W, S):-
  Movie == G, write('Galaxy Quest, ');
  Movie == C, write('Cat Woman, ');
  Movie == E, write('Ella Enchanted, ');
  Movie == W, write('Willow, ');
  Movie == S, write('Stardust, ').
writeSnack(Snack, Br, Ch, Cu, Po, Pr):-
  Snack == Br, writeln('Brownies');
  Snack == Ch, writeln('Chocolates');
  Snack == Cu, writeln('Cupcakes');
  Snack == Po, writeln('Popcorn');
  Snack == Pr, writeln('Pretzels').
```

Question 2: Palindromes with DCG

```
palindrome:-
  read_line_to_codes(user,String),
  palindrome(String,[]).

% Check for invalid characters
palindrome-->[32],palindrome.
palindrome-->[Char],palindrome,[Char],[32].
palindrome-->[_],[32].

palindrome-->[39],palindrome.
palindrome-->[Char],palindrome,[Char],[39].
palindrome-->[_],[39].

% There are 3 cases of what a palindrome is
palindrome-->[Char],palindrome,[Char].
palindrome-->[_]. % single letter
palindrome-->[]. % an empty string
```

ASCII Table

```
32 (white space)
33 !
34 "
35 #
36 $
37 %
38 &
39 ’
40 (
41 )
42 *
43 +
44 ,
45 -
46 .
47 /
48 0
49 1
50 2
51 3
52 4
53 5
54 6
55 7
56 8
57 9
58 :
59 ;
60 <
61 =
62 >
63 ?
64 @
65 A
66 B
67 C
68 D
69 E
70 F
71 G
72 H
73 I
74 J
75 K
76 L
77 M
78 N
79 O
80 P
81 Q
82 R
83 S
84 T
85 U
86 V
87 W
88 X
89 Y
90 Z
91 [
92 \
93 ]
94 ^
95 _
96 ‘
97 a
98 b
99 c
100 d
101 e
102 f
103 g
104 h
105 i
106 j
107 k
108 l
109 m
110 n
111 o
112 p
113 q
114 r
115 s
116 t
117 u
118 v
119 w
120 x
121 y
122 z
123 {
124 |
```

```
125 }
126 ~
```

Arbitrary

•Procedural programming is "how to" programming: Understand program in terms of goals to solve and in which order to solve them.
•Declarative programming is "what to" programming: Understand each program predicate at a high level.
•Backtracking causes execution to revert back to last place a clause was successful, and move on to the next one.
•Prolog is made up of (1) Facts, (2) Queries, (3) Rules
•Green cuts: prune tree w/o affecting final answer
•Red cuts: prune tree and affect final answer

•*if-then-else* can be emulated via a cut:

```
if T is true, do Q, else R
P :- (T -> Q ; R).
P :- T, !, Q.
P :- R.
```

•Write all matches without hitting next (;)

```
likes(boy,girl). likes(programmer,code).
write_likes:- likes(X,Y), write(X),
  write(' likes '), writeln(Y), fail.
write_likes. % finish as true
```