ICS1015 - Assignment

Daniel Magro
484497M

January 2017

Contents

Question 1

```
%Question1

% 1a
% leftPush(Item,OldDeque,NewDeque).
/* The 'Item' to be pushed to the left of the 'NewDeque' is attached to the 'OldDeque'
 * using the Bar Notation. The first two arguments are 'inputs' and the last one is an 'output'.
 */

leftPush(Item,OldDeque,[Item|OldDeque]).

%leftPush(6,[1,2,3],L). => L = [6,1,2,3]



% 1b
%rightPop(OldDeque,Item,NewDeque)

% Base Case, Deque with one(/last) element
rightPop([Item],Item,[]).
/* General Case, OldDeque still has multiple elements. The Head and Tail of the 'OldDeque' are
 * split in the first argument, [H|T], and the same rule is called recusrively using the Tail.
 * Recurive calls carry on until the Base Case is reached, when only one 'Item' remains,
 * that which is to be popped, which is set as the second argument. Then the recursive unwinding
 * starts happening, where each Head is attached to the start of the List using the Bar Notation,
 * [H|T2], which will eventually become 'NewDeque'. The first recursive call back after the base
 * case attaches the empty list to the head (which is now the new last(rightmost) element),
```

```
 * which is how it is popped.
 */
rightPop([H|T],Item,[H|T2]) :- rightPop(T,Item,T2).

%rightPop([6,1,2,3],I,L). => I = 3, L = [6,1,2]



% 1c
%leftPop(OldDeque,Item,NewDeque)

/* The 'Item' to be popped from the left of the 'OldDeque' is
    removed from the 'OldDeque'
 * using the Bar Notation. The first two arguments are '
   inputs' and the last one is an 'output'.
 * The 'Item' is set to the Head of the OldDeque, and the
   NewDeque is set to the Tail of the OldDeque.
 */
leftPop([H|T],H,T).

%leftPop([6,1,2,3],I,L). => I = 6, L = [1,2,3]


%rightPush(Item,OldDeque,NewDeque).

% Base Case, pushing an item to an empty List
rightPush(Item,[],[Item]).
/* General Case, OldDeque still has multiple elements. The
   Head and Tail of the 'OldDeque' are
 * split in the second argument, [H|T], and the same rule is
   called recursively using the Tail.
 * Recurive calls carry on until the Base Case is reached,
   when the empty List is reached,
 * and the item to be pushed, which is set as the first
   argument, can be inserted easily
 * Then the recursive unwinding starts happening, where each
   Head is attached to the start of the
 * List using the Bar Notation, [H|T2], which will eventually
    become 'NewDeque'.
 */
rightPush(Item,[H|T],[H|T2]) :- rightPush(Item,T,T2).

%rightPush(6,[1,2,3],L). => L = [1,2,3,6]



% 1d
% checkEmpty(Deque)

```

```
% Predicate, The fact that a Deque is empty if it is an Empty
    List.
checkEmpty([]).

/*
 * checkEmpty([]). => yes
 * checkEmpty([1]). => no
 * checkEmpty([6,1,2,3]). => no
 */



% 1e
%dequeSize([6,1,2,3],S).

% Base Case, an empty Deque has size 0
dequeSize([],0).
/* General Case, in the first argument the Deque is Split
   using the Bar Notation, [_|T], discarding
 * the head as it has no relevance, and the same rule is
   called recursively until the Base Case is
 * reached. The Base case sets the Size to 0 and starts the
   recursive unwinding. Each time another
 * rule is unwound, the size of the deque is incremented.
 */
dequeSize([_|T],S) :- dequeSize(T,S1), S is S1+1.

%dequeSize([6,1,2,3],S). => S = 4
```

Question 2

The following mathematical steps are the reasoning and method used for question 2d, solving two simultaneous equations. These are explained from a mathematical standpoint, how each step was completed in Prolog is explained in the line comments for question 2d.

$$
\begin{aligned}
Eq1: &\quad X1x + Y1y = C1 \\
Eq2: &\quad X2x + Y2y = C2
\end{aligned}
$$

$$
\begin{gathered}
\Rightarrow Eq1\_2 : X2(X1x + Y1y = C1) \\
\Rightarrow Eq2\_2 : X1(X2x + Y2y = C2)
\end{gathered}
$$

$$
\begin{gathered}
\Rightarrow Eq1\_2 : \quad X1\_2x + Y1\_2y = C1\_2 \\
\Rightarrow Eq2\_2 : \quad X2\_2x + Y2\_2y = C2\_2 \\
Eq1\_2 \quad - \quad Eq2\_2 \\
\Rightarrow (Y1\_2 - Y2\_2)y = (C1\_2 - C2\_2) \\
\therefore y = \frac{C1\_2 - C2\_2}{Y1\_2 - Y2\_2} \\
Eq1 : \quad X1x + Y1y = C1 \\
\therefore x = \frac{C1 - (Y1 * y)}{X1}
\end{gathered}
$$

```
% Question 2

% 2a
%extraCoef(Coef,Equation,Value).

/* The following 3 rules each have the first argument set to the coefficient that may be specified in the
 * first argument, being x,y or c(constant). Depending on whichever query is entered into the console,
 * pattern matching occurs on the first argument, and the relevant rule is selected. If the query were
 * to be, say extraCoef(y,eq(1,2,3),V)., then y would be matched to the second rule, 1 and 3 would be
 * discarded, and Y would be set to 2, and V set to Y.
 */
extraCoef(x,eq(X,_,_),X).
extraCoef(y,eq(_,Y,_),Y).
extraCoef(c,eq(_,_,C),C).

%extraCoef(x,eq(2,1,4),V). => V = 2.



% 2b
%sameCoefSign(Coef1,Coef2).

/* The following 3 rules define all possible combinations of Coef1 and Coef2 that have same signs.
 * These being both positive (rule 1),both negative (rule 2) or both 0 (rule 3).
 */
sameCoefSign(X,Y) :- X>0,Y>0.
sameCoefSign(X,Y) :- X<0,Y<0.
sameCoefSign(X,Y) :- X=:=0,Y=:=0.

/*
 * sameCoefSign(-2,-1). => yes
 * sameCoefSign(-2,1). => no
 */



% 2c
%raiseEq(Multiple,OrigEq,FinalEq).

/* This rule works by first creating 3 new variables, one for each coefficient of the FinalEquation.
 * The first argument, T, receives the Multiple by which the first equation will be multiplied.
```

```
 * Then, Xt, Yt and Ct are set to their original value
   multiplied by T, (Xt = X * T, Yt = ...).
 * Finally, the Third argument is eq(Xt,Yt,Ct), which will
   return the FinalEquation.
 */
raiseEq(T,eq(X,Y,C),eq(Xt,Yt,Ct)) :- Xt is X * T, Yt is Y * T
    , Ct is C * T.

%raiseEq(2,eq(1,-1,-1),F). => F = eq(2,-2,-2)



% 2d
%solveSim(Eq1,Eq2,Xvalue,Yvalue).

/* First, Eq1 was multiplied throughout by the coefficient of
    X in Eq2, using the raiseEq relation.
 * The resulting equation was stored in Eq1_2.
 * Similary, the second equation was multiplied throughout by
    the coefficient of X in the first equation.
 * The resulting equation was stored in Eq2_2.
 *
 * Next, the coefficient of Y in the newly created first
   equation, Eq1_2, is stored inside the Variable Vy1 using
   the extraCoef relation ,
 * Similarly, the coefficient of Y in Eq2_2 was stored inside
    Vy2.
 *
 * Similarly, the constant in Eq1_2 is stored inside Vc1 and
   the constant inside Eq2_2 is stored inside Vc2.
 *
 * DiffY is set to Vy1 - Vy2 and DiffC is set to Vc1 - Vc2.
 *
 * The final value of Y, Yv, is set to DiffC / DiffY.
 *
 * The final value of X, Xv, is set to (C1-(Y1*Yv))/X1.
 */
solveSim(eq(X1,Y1,C1),eq(X2,Y2,C2),Xv,Yv) :-
  raiseEq(X2,eq(X1,Y1,C1),Eq1_2), raiseEq(X1,eq(X2,Y2,C2),
   Eq2_2),
  extraCoef(y,Eq1_2,Vy1), extraCoef(y,Eq2_2,Vy2),
  extraCoef(c,Eq1_2,Vc1), extraCoef(c,Eq2_2,Vc2),
  DiffY is Vy1 - Vy2, DiffC is Vc1 - Vc2,
  Yv is DiffC / DiffY,
  Xv is (C1-(Y1*Yv))/X1.


%solveSim(eq(2,1,4),eq(1,-1,-1),X,Y). => X = 1, Y = 2.

```

```
% 2e
%checkSol(Eq1,Eq2,Xvalue,Yvalue).

/* The coefficients of Eq1 (x,y and the costant) were stored in X1,Y1 and C1 respectively using the extraCoef relation.
 * Next, the coefficients and the values of x and y (Xv and Yv) are substituted into the equation (X1*Xv + Y1*Yv) and (C1)
 * and both sides are checked for equality.
 * The same process was repeated for the second equation (Eq2).
 * If and only if both equations hold with the substitued values of x and y will the relation return yes, otherwise it will
 * return no
 */
checkSol(Eq1,Eq2,Xv,Yv) :-
  extraCoef(x,Eq1,X1), extraCoef(y,Eq1,Y1), extraCoef(c,Eq1,C1),
  X1*Xv + Y1*Yv =:= C1,
  extraCoef(x,Eq2,X2), extraCoef(y,Eq2,Y2), extraCoef(c,Eq2,C2),
  X2*Xv + Y2*Yv =:= C2.

%checkSol(eq(2,1,4),eq(1,-1,-1),1,2). => yes
```

Question 3

```
% Question 3

% 3a
%sumOfRatios(Ratio,Sum).

% Base Case, The sum of a ratio with only one term is that term.
sumOfRatios(ratio([X]),X).
/* General Case, In the first argument, the ratio, represented as a list, is split using the Bar Notation, ratio([H|T]).
 * The same relation is called recursively using the Tail, T, until there is only one term left, and thus the base case is reached.
 * The base case sets the sum of the ratio to the single term that is left. Then the recursive call back starts happening
 * With every step of the call back, the Sum 'S' is incremented by the value of the Head 'H'.
 */
sumOfRatios(ratio([H|T]),S) :- sumOfRatios(ratio(T),S1), S is S1 + H.

%sumOfRatios(ratio([3,1,2]),S). => S = 6



% 3b
%reduceRatio(OriginalRatio,FinalRatio).

/* gcd_eff is a generic relation which calculates the Greatest Common Divisor (GCD) of any two integers.
 * gcd is another relation which calculates the GCD of a list of integers
 */
```

```prolog
% Base Case
gcd_eff(X,0,L) :- L is X.
gcd_eff(0,Y,L) :- L is Y.
% General Case, Calculates the modulo of the two integers and recursively calls the same rule with the Remainder and the smaller integer.
gcd_eff(X,Y,L) :- (X>Y,(M is X mod Y,gcd_eff(M,Y,L));(M is Y mod X,gcd_eff(X,M,L))).
% Base Case, Calculates the GCD of the last two integers in the list using the gcd_eff relation.
gcd([X,Y],GCD) :- gcd_eff(X,Y,GCD).
/* General Case, The Head and Tail are split in the first argument, [H|T],
 * Then, the same rule is called recursively using the Tail, T, until the Base Case is reached.
 * The base case returns the GCD of the last 2 elements in the list. After that the recursive call back starts happening.
 * With every step of the call back, the value of the is updated with the GCD of the Head and the previous GCD.
 */
gcd([H|T],GCD) :- gcd(T,GCD1), gcd_eff(H,GCD1,GCD).


% Base Case, when only one element is left, the reduced ratio is the element divided by the GCD.
divR([X],GCD,[FR]) :- FR is X/GCD.
/* General Case, The Head and Tail of the ratio list are split in the first argument, [H|T],
 * The Head is divided by the GCD, and stored inside X, the same relation is called again, this time with the Tail.
 * The recursive call keeps happening until there is only one element left, and thus the base case is reached.
 * With every recursive call back after the base case, the reduced ratios are attached to their preceeding term,
 * one by one, using the bar notation, [X|FR].
 * Finally, X is attached to the start of the FinalRatio in the thrid argument, [X|FR].
 */
divR([H|T],GCD,[X|FR]) :- X is H/GCD,divR(T,GCD,FR).

/* This relation first calculates the GCD of all the ratios in the list.
 * then it divides every term/dividend of the ratio by the GCD using the divR relation.
 */
reduceRatio(ratio(OR),FR) :- gcd(OR,GCD),divR(OR,GCD,FR).

%reduceRatio(ratio([30,10,20]),R). => R = ratio([3,1,2])
```

```
% 3c
%divideRatio(Amount,Ratio,Parts).

% The mulR relation multiplies every term in the Ratio by 'M'.
% This relation works exactly like divR, only every '/' is replaced by a '*'.
mulR([X],M,[FR]) :- FR is X*M.
mulR([H|T],M,[X|FR]) :- X is H*M,mulR(T,M,FR).

/* This relation first calculates the sum of all the terms in the ratio 'R', using the sumOfRatios relation, and stores the sum in 'S'.
 * Next, it divides the Amount 'A' to be divided among the ratio, by the Sum 'S', and stores the quotient in M.
 * Finally, the relation mulR is called with the Ratio 'R' and 'M', with the output being stored in 'P'.
 */
divideRatio(A,ratio(R),P) :-
  sumOfRatios(ratio(R),S),
  M is A / S,
  mulR(R,M,P).

%divideRatio(54,ratio([30,10,20]),P). => P = [27,9,18]
```

Question 4

```
% Question 4

% 4a
%mem_of(M,S).

% Base Case, If the Head of the list is the same as the element being checked for membership, M, then 'M' is a member of 'S'.
mem_of(M,[M|_]).
/* General Case, If the base case is not satisfied, The Head and the Tail of the set (list) are seperated in the second argument,
 * using the bar notation, [H|T]. However H is discarded using '_' since it is not needed. The same relation is then called
 * recursively using the tail 'T'.
 */
mem_of(M,[_|T]) :- mem_of(M,T).

/* Another possible way of implementing mem_of
 * mem_of(M,[H|T]) :- H=:=M; mem_of(M,T). */

/* An easier (and possibly more efficient) way of doing this would be to use the inbuilt Prolog method.
 * mem_of(M,S) :- member(M,S).
 */

%mem_of(6,[2,6,3,4]). => yes
%mem_of(6,[2,5,3,4]). => no



% 4b
%subset_of(S1,S2).

```

```
% Base Case, The empty set is a subset of any other set.
subset_of([],_).
/* General Case, First, the Head and the Tail of the first set 'S1' are split in the first argument, [H|T].
 * Next, it is checked whether the Head 'H' is an element of the set 'S2' and the same relation is called recursively, using the Tail 'T'.
 * The recursive calls keep occuring until all the heads have been checked
 * until the tail, T, is the empty list, which is the base case.
 * If all the elements of S1 are also present in S2, this must mean that S1 is a subset of S2.
 */
subset_of([H|T],S2) :- mem_of(H,S2),subset_of(T,S2).

%subset_of([2,4],[2,6,3,4]). => yes



% 4c
%intersect(S1,S2,S3).

% The intersection of the empty set with any set is the empty set
intersect([],_,[]).
/* The first argument, [X|S1], splits the head, X, and the tail, now named S1, of the set 'S1',
 * Then, the relation checks if 'X' is a member of S2. If so, the same relation is called using the tail of S1.
 * Also, because of the third argument, [X|S3], X is attached to the resulting set, S3.
 * If not, The next rule is applicable.
 */
intersect([X|S1],S2,[X|S3]) :- mem_of(X,S2), intersect(S1,S2,S3).
/* This rule is applicable when X is not a member of S2. This relation simply calls the intersect procedure using
 * the tail of S1, discarding the head (since it does not occur in S2 it is not part of the intersect).
 */
intersect([_|S1],S2,S3) :- intersect(S1,S2,S3).

%intersect([20,4,13,11,24],[33,2,4,11,20,68],S). => S = [20,4,11].



% 4d
```

```
%unite(S1,S2,S3).

% The union of an empty set and any other set is the set.
unite([],S,S).
/* The first argument, [X|S1], splits the head, X, and the tail, now named S1, of the set 'S1',
 * Then, the relation checks if 'X' is NOT a member of S2. If 'X' is not a member of S2, The same relation is called recursively using the tail of S1.
 * The above is done to avoid duplicates in the resulting set union. The recursive calls keep occuring until S1 is an empty set.
 * At this point, the base case is reached, and thus the resulting set becomes 'S2'. Then, with each successive call back,
 * 'X' is attached to the head of the resultant set, 'S3'.
 * If 'X' is a member of S2, then the following rule is applicable.
 */
unite([X|S1],S2,[X|S3]) :- not(mem_of(X,S2)), unite(S1,S2,S3).
/* This rule is applicable when X is a member of S2. This relation simply calls the unite procedure using the tail
 * of S1, discarding the head (since it occurs in S2, and S3 (the resulting set) will be set to S2 in the base case,
 * there is no need for S1 to contain it as it will lead to repeated elements).
 */
unite([_|S1],S2,S3) :- unite(S1,S2,S3).

%unite([5,2,6,3,4],[1,5,7,2,4],S). => S = [6,3,1,5,7,2,4]
```