Logic Programming

ICS 1015

Prolog Assignment



Name & Surname: Deborah Vella

Course: Artificial Intelligence

**Table of contents**

**Question 1** …………………………………………………………………………………………………………………………… 3

**Question 2**

2a) …………………………………………………………………………………………………………………………… 5

2b) …………………………………………………………………………………………………………………………… 5

2c) …………………………………………………………………………………………………………………………… 7

**Question 3**

3a) …………………………………………………………………………………………………………………………… 8

3b) …………………………………………………………………………………………………………………………… 8

3c) …………………………………………………………………………………………………………………………… 9

3d) …………………………………………………………………………………………………………………………… 10

3e) …………………………………………………………………………………………………………………………… 11

**Question 4**

4a) …………………………………………………………………………………………………………………………… 12

4b) …………………………………………………………………………………………………………………………… 14

**Question 1**

In this question the user has to guess the secret word hello. The code begins with the fact secret\_word, containing the list [h,e,l,l,o]. This is then used to compare the 1st letter entered to the head of the list in this fact. A rule called start was coded and it is to be executed first. This rule calls another rule named guess. The latter contains the recursive code and a base case for termination.

The rule start, begins by initialising the variable Counter and set it equal to 1 to indicate that the first letter is to be guessed. The code continues with repeat/0, so that, the lines of code following it keep looping until all statements succeed. The first goal to be reached within the repeat is to output ‘Guess the first letter’ on screen, and read the user’s input and store it inside the attribute called Head. Secondly, an if statement is used to check if the previous input matches the head of the secret\_word list. If the condition is met, then the program outputs ‘OK!’ and the rule guess starts executing passing the Tail (containing [e,l,l,o]) and the Counter (containing 1) variables.

The rule guess commences by setting the variable Counter1 equal to, the passed Counter variable added to 1. This attribute is used to keep track of which letter is next to be guessed, and is printed out every time the user is asked to input a letter. Once again repeat/0 is used to loop through the goals following it every time fail is met. The program asks the user to enter the letter wanted according to what number Counter1 is storing. The input is read and stored inside the variable Head. This is followed by an if statement, where the condition checks if the list inside variable X matches with the user input (Head) and the rest of the tail. When the condition is false, ‘Sorry – try again’ is printed and fail/0 is used. On the other hand, when the condition is true, the program continues by outputting ‘OK.’ and calls the guess rule.

This automatically enforces recursion and it terminates until it maps onto the base case. The base case is an empty list, implying that all letters where guessed correctly, and the number 5, implying that the letter counter reached the fifth letter. When the program reaches the base case, the user is congratulated and is informed that the secret word is hello and the program stops execution.

**Commented code**

/\*fact\*/

secret\_word([h,e,l,l,o]).

/\*base case\*/

guess([],5):- write('Congratulations - The word is hello'),nl.

/\*recursive predicate. Executed when called inside start\*/

guess(X,Counter):- Counter1 is Counter+1,/\*variable Counter1 is set to. V the contents of Counter plus 1\*/

repeat,

write('Letter #'),write(Counter1),read(Head), /\*Prompts user to enter the next letter indicated by Counter1, and input is store din Head\*/

((X=[Head|Tail])->write('OK. '),guess(Tail,Counter1); /\*X contains the Tail passed in start. If it is equal to Head and Tail, then print 'OK.'\*/

write('Sorry -- try again! '),fail)./\*and call the guess predicate,

enforcing recursion else, inform the user to try again and call fail predicate.\*/

/\*predicate that should be typed in for execution\*/

start :- Counter is 1, /\*variable Counter stores the letter number the

user has to guess next\*/

repeat, /\*used when fail is met\*/

write('Guess the first letter'),read(Head), /\*User is prompted to

enter a letter and the character is then read and stored inside the variable head\*/

(secret\_word([Head|Tail])->write('OK! '),guess(Tail,Counter); /\*If the

fail).

content of the variable Head matches the content of the list in the secret\_word fact, then,OK. is printed and the guess predicate is called, passing the Tail and Counter variables, else, fail and repeat again\*/

**Question 2**

**2a)**

The aim of this task is to read a list of numbers which make up a ratio and then add the numbers up together and output the final answer. The clause sumOfRatios/2 accepts a list of 3 elements ([A,B,C]) which are numbers and returns Sum as an output. The rule adds data stored inside the variables A, B and C. It then stores the result inside the attribute Sum.

**Commented code**

sumOfRatios([A,B,C],Sum):- Sum is A+B+C. /\*The addition of the entered ratios are found and stored inside the vraiable Sum and is then printed on screen\*/

**2b)**

The goal of this program is to obtain a ratio reduced to its lowest terms from the original inputted ratio. The reduced ratios are found by calculating the highest common factor of the three numbers and then divide each number with the answer obtained. The HCF is found by keeping in mind that HCF of two numbers remains the same if the larger number is replaced by its difference with the smaller number. This procedure needs to be repeated to get smaller pairs of numbers until the two numbers are equal.

The execution starts by entering a ratio made up of 3 numbers inside reduceRtio/2 predicate. The program passes the first two inputted values and the variable R (this will later store the HCF) into gcd/3. It first enters a gcd rule where it checks if X is less than Y. if it actually is smaller, X is then deducted from Y and the answer stored inside YmX (representing Y-X). Afterwards, gcd is called again passing YmX (replacing Y which is the greater number) and X (the smaller number). This enforces recursion.

On the other hand, if X isn’t less than Y, then the program jumps onto the other gcd/3 rule. Here, it checks if X is greater than or equal to Y. if this is true, Y is deducted from X and the difference is stored in XmY (representing X-Y). As seen previously, the larger number is replaced by the difference (XmY) and is passed in gcd/3 along with Y (the smaller number). At one point, the difference should eventually become equal to 0 meaning that X is equal to Y. When this happens and the values are passed in gcd/3, the program will hit the base case: gcd(0,X,X). The stopping condition checks if X is greater than 0 and the symbol ‘!’ is used to cut the recursion. The base case returns the value inside X which is the HCF.

At this point, the program starts backtracking until it reaches depth of 1 again. The HCF of the first two numbers is now stored inside the attribute R and is passed in gcd/3 together with the third value and the recursive procedure starts again. The newly obtained HCF from the last recursion, is then stored in variable R1. The reduceRatio rule continues by dividing each initially entered value, by the HCF in R1. These are the final results and therefore, are outputted as a reduced ratio for the user.

**Commented Code**

gcd(0,X,X):- X>0,!. /\*base case: if X is greater than 0, recursion is terminated\*/

/\*recursive clauses to get GCD\*/

gcd(X,Y,D):- X<Y, YmX is Y-X, gcd(YmX,X,D). /\*is X<Y? If yes, subtract X

from Y and call gcd with new values\*/

gcd(X,Y,D):- X>=Y, XmY is X-Y, gcd(XmY,Y,D). /\*is X>=Y? If yes, subtract Y

from X and call gcd with new values\*/

reduceRatio(ratio([J,K,L]),ratio([J1,K1,L1])):-

gcd(J,K,R), /\*calculating the gcd of the first two numbers by passing

the entered values\*/

gcd(R,L,R1),/\*calculating the gcd of the first two entered numbers

with the third one\*/

J1 is J/R1, /\*dividing the original numbers with the greatest common

divisor, to obtain the reduced ratios\*/

K1 is K/R1, /\*J1, K1 and L1 store the answers and then are printed

for the user to see\*/

L1 is L/R1.

**2c)**

This question requires a program that divides an inputted quantity in a given ratio. The answer is made up of smaller amounts which when added back up together they add up to the whole quantity entered by the user.

This task needed just one rule which was named divideRatio(Amount, ratio([A,B,C]),Parts). Instead of Amount, the quantity that is to be divided in a ratio is to be typed in. The list [A,B,C] should be replaced by the ratio wanted. Variable Parts will later store the answers and output them. Firstly, the total of the ratio numbers added together is found and stored in SumOfRatio. Secondly, the first number of the ratio is divided by the ratio total and is stored in Div1.

The first answer is found by multiplying the divided answer by the quantity. These steps are repeated two more times to find the answers for the last two numbers left from the ratio. Last but not least, Parts is set equal to the answers stored inside Times1, Times2 and Times3.

**Commented Code**

divideRatio(Amount, ratio([A,B,C]), Parts ):- /\*stores quantity inside Amount, and the ratio numbers inside the list[A,B,C] respectively\*/

SumOfRatio is A+B+C, /\*Adding values of the ratio inputted\*/

Div1 is A/SumOfRatio, /\*Divide the first number of the ratio

by the sum of ratio and store answer in Div1\*/

Times1 is Div1 \* Amount,/\*Mulitply the answer in Div1 by the

entered amount and store in Times1\*/

Div2 is B/SumOfRatio, /\*Divide the second number of the ratio

by the sum of ratio and store answer in Div2\*/

Times2 is Div2 \* Amount,/\*Mulitply the answer in Div2 by the

entered amount and store in Times2\*/

Div3 is C/SumOfRatio, /\*Divide the third number of the ratio

by the sum of ratio and store answer in Div3\*/

Times3 is Div3 \* Amount,/\*Mulitply the answer in Div3 by the

entered amount and store in Times3\*/

Parts = ratio([Times1,Times2,Times3]). /\*set Parts equal to the

list [Times1, Times2, Times3]. These are the answers\*/

**Question 3**

**3a) & b)**

/\*called inside permute/2\*/

deletion(X,[X|N],N).

deletion(X,[F|N],[F|P]) :- deletion(X,N,P).

permute([],[]). /\*base case\*/

permute([X|Y],Z) :- permute(Y,M), deletion(X,Z,M).

/\*recursive clause\*/

**How the algorithm works to find the first result:**

1. permute([X|Y],Z) sets X equal to the head of the entered list, while Y is set to the tail of the entered list.

2. The clause starts by calling again the predicate permute, this time passing Y which was previously set to the tail of the list.

3. Permute([X|Y],Z) once again sets X to the head of the passed list (in this case the tail in step 2) and Y to the new Tail. The list in Y is being emptied every time permute is called, as it takes away the first element of the list each time.

4. The above steps continue to recurs until all the elements are taken out and the empty list is met which is defined by the base case: permute([],[]).

5. Every time it recurs, the depth increases.

6. Permute([],[]) returns an empty list and stores it in M.

7. Once X is set to the last element of the list initially entered, and Y and M are empty lists, the program goes on the predicate deletion(X,Y,Z). This is mapped onto the predicate deletion(X,[X|N],N), coded above the permute/2 rule. Note: only deletion(X,[X|N],N). is used for the first permutation.

The X in deletion/3 matches the X in permute/2

N in deletion/3 matches the M in permute/2

[X|N] in deletion/3 matches Z in permute/2. This returns a list made up of the elements stored in X as a head and elements stored in N as a tail.

8. The new created list is stored inside Z as it starts to backtrack and the depth decreases.

9. The variable Z inside the permute/2 predicate is now populated as well.

10. The permute/2 rule is worked out by the values found and by the list returned ([X|N]) in deletion/3, in respect with each depth.

11. By the concatenation of elements happening inside the deletion/3 clause, a new list is being created consisting of the same elements entered by the user

but in a different order.

12. The program keeps on backtracking until it reaches depth 1 and at last, the data inside Z is printed on screen.

**how the algorithm works to find the rest of the permutations:**

If a semicolon is entered by the user after first output, it indicates that more permutations are needed by the user. The procedure starts again, but this time the last values found are used, therefore Z and M are already populated with previous results. Therefore, when deletion/3 is called it is now mapped also onto deletion(X,[F|N],[F|S]) instead of deletion(X,[X|N],N) only, as the program is now passing lists.

**3c)**

The query findall/3 generates a list (stored in the third argument), that satisfy the second argument. The first argument is the variable, of which, instantiations satisfying the second argument are found. These instantiations are returned as a list of lists.

In this task, the structure described above was used to find a list of all the permutations. The first argument was set to the variable F, and appears in the second argument so that the instances of F that satisfy the second argument can be found. The second argument is in fact the goal permute(ListIn,F). When a permutation is found it is returned in F as an instance. When at one point, the contents of variable F stop satisfying the second argument, the program starts collecting all contents stored in F at each depth. The final result is a list of all the permutations, which is returned by the variable G which was coded to be the third argument.

**Commented Code**

anagram(ListIn,G):- findall(F,permute(ListIn,F),G).

/\*Here the goal findall/3 generates a list G of all the permutations found by permute(ListIn,F). Each permutation found is stored in F and then passed again in permute/2. This continues until the second argument fails. When this happens, the computer starts backtracking, collecting all F's found forming a list of lists. The final resulted list of lists is returned as G.\*/

**3d)**

In this question, the user can input a list that has any length s/he wants. This list is scanned element by element comparing each number to the one following it. If the condition of the first number being less than or equal to the next one is always found to be true, then the program returns a yes. This indicates that the list is found to be in ascending order. On the other hand, if the condition is at one point found to be false, the program returns a no, this time indicating that the list is not in ascending order. The algorithm is explained in the commented code.

**Commented Code**

sorted([]). /\*base case\*/

sorted([\_]).

sorted([H,I|T]):- H=<I, /\*compare first element with the second one\*/

sorted([I|T]). /\*recursive step\*/

/\*The predicate sorted/1 takes a list as an input. It takes the first two elements of the list and check if the first element (H) is less than or equal to the second one (I). If this is true, the goal sorted is called again, this time passing I as the head of the list together with a tail. This continues to recurs until an empty list is met. The program returns yes if every H was found to be less than or equal to I, else, it returns no.\*/

**3e)**

A naive sort is a way of sorting elements of a list into ascending order. This is done by first developing a permutation of the elements and then test if the resulting list is in ascending order. If this order is not achieved yet, another permutation of the list is generated.

The coded function nsort/2 is following the naïve sort algorithm. In fact, the first goal is calling the predicate permute/2 which returns a permutation as a result. The entered list is passed inside the latter goal, and the result is then passed in sorted/1. The predicate sorted/1 analyses the list passed and checks if it is in ascending order. If this order is not reached yet, it starts backtracking to generate another permutation. Every generated list is checked inside the sorted/1 rule. Once the rule sorted/1 is satisfied, the program outputs the final list generated and a cut is used to terminate the program.

**Commented Code**

nsort(List1,List2):- permute(List1,List2),sorted(List2),!.

/\*The goal permute/2 is called, passing List1 so that a permutation is found. The returned permutation is stored in List2 and passed in goal sorted/1. Here the list is checked if it's in ascending

order. If it is not yet in ascending order, the program backtracks to permute and gives back another permutation. This recursive procedure terminates when the sorted goal is true. The final list is printed

on screen and the program cuts. \*/

**Question 4**

**4a)**

This problem starts by the knowledge base which is made up of 6 word/8 predicates. These are facts that contain the word as a whole in the first argument, while the other 7 arguments are the letters on their own, one after the other. These facts are then used inside the body of the crossword rule by extracting information from them.

The crossword clause firstly maps the predicate word/8 onto the first fact coded beforehand. The variables V1, V2, V3, H1, H2, H3 store the words as a whole and are returned at the end as an output. The rest of the variables represent the boxes where two words intersect with each other. The other values are not needed during the comparison of letters; therefore, an underscore is used instead of a variable.

When the first word/8 predicate is populated, the next two word/8 predicates are populated. In the fourth word/8 predicate, the program starts comparing the intersections. This happens because the same variables used before such as V1H1 are used again, and so the letters in the called word facts are compared to the letters in the 2nd, 4th and 5th positions. If no match was found for the letter being compared, the program backtracks and stores another word and starts comparing again.

The program continues recursion and backtracking until each variable compared finds a match with the facts. Then it is made sure that the attributes do not contain the same words. At last, the words are outputted for the user to see in which position they are supposed to be written.

**Commented code**

/\*The below is the base knowledge made up of the facts called word

each fact contains one of the words as a whole and the letters that make up the word\*/

word(astante, a,s,t,a,n,t,e). /\*the letters s, a and t are compared inside

the rule\*/

word(astoria, a,s,t,o,r,i,a). /\*the letters s, o and i are compared inside

the rule\*/

word(baratto, b,a,r,a,t,t,o). /\*the letters a, a and t are compared inside

the rule\*/

word(cobalto, c,o,b,a,l,t,o). /\*the letters o, a and t are compared inside

the rule\*/

word(pistola, p,i,s,t,o,l,a). /\*the letters i, t and l are compared inside

the rule\*/

word(statale, s,t,a,t,a,l,e). /\*the letters t, t and l are compared inside

the rule\*/

/\*In the rule below, every V means that it is vertical, and every H means that it is horizontal\*/

/\*each word/8 predicate stores the whole word in the first argument and the letters one by one in the rest of the arguments.

The letters whih are not intersecting with another letter are listed as an underscore. This is because it doesn't matter

what the letter is at thay position. As a result, only the 2nd, 4th and 6th letters are stored in a variable.\*/

crossword(V1, V2, V3, H1, H2, H3) :-

word(V1, \_,V1H1,\_,V1H2,\_,V1H3,\_), /\*Get a word and it's respective

letters\*/

word(V2, \_,V2H1,\_,V2H2,\_,V2H3,\_), /\*Get a word and it's respective

letters\*/

word(V3, \_,V3H1,\_,V3H2,\_,V3H3,\_), /\*Get a word and it's respective

letters\*/

word(H1, \_,V1H1,\_,V2H1,\_,V3H1,\_), /\*Call word and compare the 2nd,

4th & 6th letters with V1H1,V2H1,V3H1 respectively\*/

word(H2, \_,V1H2,\_,V2H2,\_,V3H2,\_), /\*Call word and compare the 2nd,

4th & 6th letters with V1H2,V2H2,V3H2 respectively\*/

word(H3, \_,V1H3,\_,V2H3,\_,V3H3,\_), /\*Call word and compare the 2nd,

4th & 6th letters with V1H3,V2H3,V3H3 respectively\*/

V1 \= V2, /\*making sure that none of the stored words are equal to

each other\*/

V2 \= V3,

V3 \= H1,

H1 \= H2,

H2 \= H3.

**4b)**

This prolog program was found on the internet. The code is split into two rules both called move. One of the rule outputs the moves being done to solve the tower of Hanoi puzzle. The other rule enforces recursion and keeps calling both of the predicates move/4. This code was chosen because the procedure done during execution can be easily understood as the code makes use of simple techniques such as unification of variables, sequence of goals inside each rule and recursion as said before. The algorithm is explained in more detail below.

The predicate move/4 consist of the variables N that stores the size of the stack, and A,B and C that store the positions of the poles(left, centre, right). The rule starts by checking if the size in N is greater than 1. N is then subtracted by 1 and the answer is stored inside the attribute M. the predicate move(M,A,C,B) is used so that the rule calls itself therefore it recurs. The elements in B & C are now exchanged with each other and M is unified with N. this procedure continues until N is equal to 1.

When the variable M is equal to 1, the predicate move/4 is mapped onto the clause move(1,A,B,\_). This rule outputs a message to update the user about what moves are being done to solve the puzzle. The disk is moved from the position stored in A and placed in the position stored in B.

When the latter rule is executed, the program backtracks into the predicate that called the other rule and then the program continues to execute the next step. This step maps once again onto move(1,A,B,\_) rule and passes the elements in A and B stored in that particular depth. The called rule once again prints the what step was taken to get closer to a solution of the puzzle. It backtracks to the previous depth and goes onto the last move/4 goal. The move rule is executed again with the values passed by the attributes C and B and then it backtracks again.

When it backtracks, it continues until depth 1 is reached again and the whole procedure is done again using the new values obtained every time. When the program backtracks to the last goal in the second rule, the values obtained in that goal are compared to the values in the head of the rule. When these values match with each other, the program stops the recursion and outputs all of the results.

**Commented code**

/\*The rule that prints the step by step solution of the puzzle\*/

move(1,A,B,\_) :- /\*Underscore is used for don't care variables\*/

write('Move top disk from '), /\*Outputs from where the disk is being

removed and to where it is placed\*/

write(A), /\*A stores from where the disk is removed\*/

write(' to '),

write(B), /\*B stores where the disk is placed\*/

nl.

/\*this clause works out the solution recursively\*/

/\*N is size of stack while A,B,C can be left, right and centre\*/

move(N,A,B,C) :-

N>1, /\*is size of stack greater than 1?\*/

M is N-1, /\*set M equal to size of stack minus one\*/

move(M,A,C,B), /\*The following goals enforce recursion\*/

move(1,A,B,\_),

move(M,C,B,A).