programming logic (prolog)

Unit Assignment report

Russell Sammut-Bonnici

ICS1015

January 15, 2018

Contents

Question 1 – Guessing Game

[Task 1 – Guess the Word 1](#_Toc503269341)

Question 2 – Ratio Predicates

[Task 2a – Sum of Ratios 4](#_Toc503269342)

[Task 2b – Reduce Ratio 5](#_Toc503269343)

[Task 2c – Divide Ratio 8](#_Toc503269344)

Question 3 – Arrangement Predicates

[Task 3a – Permutations 9](#_Toc503269345)

[Task 3b – How the Permutation Algorithm Works 11](#_Toc503269346)

[Task 3c – Anagrams 13](#_Toc503269347)

[Task 3d – Check if Sorted 14](#_Toc503269348)

[Task 3e – Naïve Sort 15](#_Toc503269349)

Question 4 – Solving Puzzles

[Task 4a – Crossword Puzzle 16](#_Toc503269350)

[Task 4b – Towers of Hanoi 18](#_Toc503269351)

Question 1 – Guessing Game

# Task 1 – Guess the Word

In this task, a program was required to firstly store a secret word “hello” in its knowledge base. Then the program had to prompt the user letter by letter until the word was guessed. If an incorrect guess is recognized then the program would tell the user to try again, and try and keep reading until the letter is correct.

Facts Declared

* secret\_word([h,e,l,l,o]).  
  This fact is used to store the secret word to be guessed by the user. It stores the word hello letter by letter in a list so that logical head and tail techniques can be used alongside recursion, to make the program functional.

Rules Declared

* start  
  This rule acts as the initiating predicate of the program, it doesn’t require any arguments when called as a query. It works by first setting counter variable C to 1 as it is on the first letter. Then it makes use of the in-built function repeat to work as a loop when the user inputs an incorrect guess.

It prompts the user to ‘Guess the first letter’ and then reads it as the variable H. It then refers to the fact secret\_word and checks if the input H matches with the head of [h,e,l,l,o]. If it does not match then it goes to fail after the OR operator, and then goes back to repeat. If it does match the program writes OK and then calls the recursive predicate letter\2. As arguments it inputs T (that is the tail of the list, [e,l,l,o]) and the counter C.

* C: Counter variable for keeping track of the current letter being guessed.
* H: Header variable, for matching inputted H with the head of the list.
* T: Tail variable, for transferring the to be guessed letters of the list.

* letter([],5)  
  This predicate is the base case of letter\2. It is accessed when all the letters in the word are guessed. It prints a celebratory message and reveals the secret word by retrieving it from the knowledge base and then writing it using the variable Y. A cut operator is then used to stop the program from going recursively backwards.
* Y: Used to store the contents of secret\_word, to be printed using write\1.
* letter(X,C)  
  This is the recursive predicate of letter\2. It accepts list X and the letter counter C. It firstly increments the counter by one by setting C1 to C+1. Using the repeat function, similarly to the initiating predicate it prompts the user until the right letter is inputted.

It outputs the letter number with C1. It reads the input letter as H and tries to match it with the head of the list X. If succeeded it recursively calls itself, inputting the tail of X as T and the new incremented counter value stored in C1. If failed it will prompt the user to try again and go back to the repeat line.

* X: Used to store the tail of the list in the initiating predicate or previous depth
* C: Used to store the counter value of the previous letter
* C1: Used to store the counter value of the current letter
* H: Header variable, used for matching to see if guess succeeded
* T: Tail variable, used to input as X in to the next depth.

Source Code (question1.pl)

/\*

Question 1.

Russell Sammut Bonnici

ICS1015

\*/

/\*fact\*/

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

/\*initiating predicate\*/

start :-

C is 1, %initialises count

repeat, %beginning of loop

write('Guess the first letter'),read(H), %prompts user

( secret\_word([H|T])->write('OK. '),letter(T,C); %succeeds, go to second letter

fail ). %fails, loops to repeat function

/\*base case\*/

letter([],5):-

write('Congratulations! The word is '),secret\_word(Y),write(Y),nl,!. %congratulate user and finish

/\*recursive predicate\*/

letter(X,C):-

C1 is C+1, %increments count

repeat, %beginning of loop

write('Letter #'),write(C1),read(H), %prompts user

((X=[H|T])->write('OK. '),letter(T,C1); %succeeds, go to next step

write('Fail. try again! '),fail ). %fails, loops to repeat function

Console Listing

| ?- start.

Guess the first letter|: e.

Guess the first letter|: h.

OK. Letter #2|: e.

OK. Letter #3|: l.

OK. Letter #4|: x.

Fail. try again! Letter #4|: l.

OK. Letter #5|: o.

OK. Congratulations! The word is [h,e,l,l,o]

yes

Question 2 – Ratio Predicates

# Task 2a – Sum of Ratios

In this task, a predicate was required to calculate the sum of ratios. In this program, the sum of three numbers in a list would be calculated and set as the variable Sum.

Rules Declared

* sumOfRatios(Ratio,Sum)  
  This rule is the requested predicate. It sets the variable Ratio equivalent to ratio([X|Y|Z]) to be able to retrieve the contents from the 3-number list. It then adds them up and binds them to variable Sum.
* Ratio: Variable for storing 3-number ratio, inputted as a list.
* X: Stores first number from list Ratio.
* Y: Stores second number from list Ratio.
* Z: Stores third number from list Ratio.
* Sum: Stores the sum of X,Y and Z.

Source Code (question2.pl)

/\* Question 2a. \*/

%predicate that binds the sum of the three numbers in Ratio to Sum

sumOfRatios(Ratio,Sum):-

Ratio = ratio([X,Y,Z]), %to access values in list

Sum is X+Y+Z. %calculate sum and append

Console Listing

| ?- sumOfRatios(ratio([1,2,3]),Sum).

Sum = 6

# Task 2b – Reduce Ratio

In this task, a predicate was required to accept a ratio as the OriginalRatio and return a simplified version of the FinalRatio. This was done using Stein’s Algorithm to find the greatest common denominator of the values in the ratio. Each value is then divided by their greatest common denominator, and stored as the FinalRatio.

Rules Declared

* reduceRatio(OriginalRatio,FinalRatio)  
  This rule is the requested predicate. It sets the variable OriginalRatio equivalent to ratio([X|Y|Z]) to be able to retrieve the contents from the 3-number list. It then sets FinalRatio equivalent to ratio([X1,Y1,Z1]) to then set its contents to the simplified version of the list after calculated and return the final ratio.

Since the gcd predicate works with only 2 arguments, the greatest common denominator is firstly found for the first two numbers X and Y and stored in D. After which, the greatest common denominator of the last number Z and D is found. X, Y and Z are then divided by D1, simplifying the ratio and returning it as the FinalRatio.

* OriginalRatio: Variable for storing 3-number ratio, inputted as a list.
* X, Y, Z: Stores the three numbers from OriginalRatio.
* FinalRatio: Variable for storing 3-number ratio, after simplification.
* X1, Y1, Z1: Stores numbers after ratio simplification for FinalRatio.
* D: Stores the greatest common denominator of the first two numbers in the ratio.
* D1: Stores the final greatest common denominator of all three numbers in the ratio.
* gcd(0,B,D) and gcd(A,0,D)  
  These predicates are the base cases used for calculating the greatest common denominator. When the first variable (A) is equal to 0 it sets D as B. When the second variable (B) is equal to 0 it sets D as A.
* A: First variable for storing greatest common denominator – or 0
* B: Second variable for storing greatest common denominator – or 0
* D: Third variable for returning greatest common denominator, copied from B or A.
* gcd(A,B,D)  
  This rule has two recursive predicates used for calculating the greatest denominator of A and B and storing it to D. The first recursive predicate is executed when A>B, if A<B, it fails the first predicate and proceeds through to the second predicate.  
    
  This works using recursion in order to continuously find the remainder of the biggest number over the smallest number. The remainder is then inputted as an argument, replacing the largest number. It then stops when the remainder is equal to 0, because when equal to 0 then the other number is the greatest common denominator of the first two initial numbers. This is known as Stein’s Algorithm.  
    
  In the first recursive predicate, when A>B, variable ArB is set to A mod B. Hence it stores the remainder of A/B. The method then recursively calls itself inputting ArB, B and D as arguments, going a depth further.  
    
  In the second recursive predicate, when A<B, variable BrA is set to B mod A. The method then recursively calls itself inputting A, BrA and D as arguments, going a depth further.
* A, B: Stores integer values of the original ratio, then remainders of previous depths throughout the algorithm.
* ArB: Stores the remainder of A/B.
* BrA: Stores the remainder of B/A.
* D: Stores the greatest common denominator of the two numbers, after one of the base cases are reached.

Source Code (question2.pl)

/\* Question 2b. \*/

%base cases (note: this is Stein's Algorithm)

gcd(0,B,D):-

D is B.

gcd(A,0,D):-

D is A.

%first recursive predicate to get gcd of 2 elements, when B>A

gcd(A,B,D):-

A>B,

ArB is A mod B,

gcd(ArB,B,D).

%second recursive predicate to get gcd of 2 elements, when A>B

gcd(A,B,D):-

A<B,

BrA is B mod A,

gcd(A,BrA,D).

%predicate that reduces ratio and binds to FinalRatio

reduceRatio(OriginalRatio,FinalRatio) :-

OriginalRatio = ratio([X,Y,Z]), %to access numbers from ratio

FinalRatio = ratio([X1,Y1,Z1]), %to return after calculated

gcd(X,Y,D), %find gcd of first two number

gcd(D,Z,D1), %find gcd of all three numbers

X1 is X/D1, Y1 is Y/D1, Z1 is Z/D1. %simplifies by dividing by gcd

Console Listing

| ?- reduceRatio(ratio([30,10,20]),R).

R = ratio([3,1,2])

# Task 2c – Divide Ratio

In this task, a predicate was required to return the ratio variable Parts that would be the amount split into three parts by the ratio variable Ratio. This predicate worked calling reduceRatio\2 and sumOfRatios\2.

Rules Declared

* divideRatio(Amount,Ratio,Parts)  
  This rule is the requested predicate. It returns the ratio variable Parts after splitting amount by ratio. It first binds Parts to variables X1,Y1 and Z1 in a list, in a ‘ratio’ predicate. The predicate reduceRatio is then called to simplify the inputted Ratio.

The simplified version, FinalRatio is returned and put as a predicate into sum of ratios. The sum of FinalRatio is returned in the variable S. FinalRatio has its values retrieved with the variables X,Y and Z. The variables X1,Y1 and Z1 are then set as X,Y and Z, divided by the ratios sum, multiplied by the inputted amount respectively.

* Amount: Stores the amount to be split, inputted through the query.
* Ratio: Stores the ratio to split the amount into, inputted through the query.
* X, Y, Z: Variables storing the three values from ratio.
* Parts: Stores the ratio to be returned.
* X1, Y1, Z1: Variables that store the values of the divided ratio to be returned.
* S: Sum of the simplified ratio.

Source Code (question2.pl)

/\* Question 2c. \*/

%predicate that splits amounts into parts by ratio and binds to Parts

divideRatio(Amount,Ratio,Parts) :- Parts = ratio([X1,Y1,Z1]),

reduceRatio(Ratio,FinalRatio),

sumOfRatios(FinalRatio,S),

FinalRatio = ratio([X,Y,Z]),

X1 is X/S\*Amount,

Y1 is Y/S\*Amount,

Z1 is Z/S\*Amount.

Console Listing

| ?- divideRatio(54,ratio([30,10,20]),P).

P = ratio([27,9,18])

Question 3 - Arrangement Predicates

# Task 3a – Permutations

In this task, a predicate was required to return the different permutations of a list.

Rules Declared

* permute([],[])  
  This predicate is the first base case for permute. When the inputted list is empty it returns an empty list.
* permute([X],[X])  
  This predicate is the second base case for permute. It copies the contents of the last value in the list to the list to be returned.
* permute([H|T],Permutation)  
  This recursive predicate is the requested predicate, it returns a permutation from the inputted list to the query. It firstly calls itself until it reaches the last element in the list and when it does, it reaches its deepest depth and returns X as T. With the cut operator, it then returns X as the last element. It then goes through the depths during appending to append an order for the list for a certain permutation. After the permutation is returned, the user can enter the OR operator ‘;’ as a query and it repeats the same process except that this time it finds for another permutation. It can keep on doing this until all the permutations are returned.
  + H: Head variable for the first element in the list
  + T: Tail variable for the rest of the list after the head
  + Permutation: List to be returned, that is a permutation
  + X: Variable used for storing the below depth’s value for X, starting from the last element of the list at the lowest depth which is reached through the base case.
  + L, P1, P2: Variables used for appending, builds up the permutation.

Source Code (question3.pl)

/\* Question 3a. \*/

%base cases

permute([],[]). %used for when the inputted list is empty

permute([X],[X]):-!. %copies last element of list to X

%recursive predicate that returns a list's permutations

permute([H|T],Permutation) :-

permute(T,X),

append(L,P2,X),

append(L,[H],P1),

append(P1,P2,Permutation).

Console Listing

| ?- permute([1,2,3],R).

R = [1,2,3] ;

R = [2,1,3] ;

R = [2,3,1] ;

R = [1,3,2] ;

R = [3,1,2] ;

R = [3,2,1] ;

no

# Task 3b – How the Permutation Algorithm Works

| ?- permute([1,2,3],R).

Consider the list [1,2,3] inputted as a list into the permute\2 predicate. H is set as [1] and T is set as [2,3] and R is the permutation to be returned, as seen in the line above. Going into the predicate, it goes to the first line shown below.

permute([H|T],Permutation) :-

permute(T,X), %uses recursion until base case

Here it makes use of recursion, calling itself and going through different layers of H and T until it gets to the base case. In this case at Depth 2 where H is 2 and T is [3].

permute([X],[X]):-!. %copies last element to list of X

In this base case, the value of the last element of the list 3 is returned as X and put in a list to be used with append. Therefore, X in the permute\2 predicate becomes [3]. The program then goes to the first call of the library function append.

append(L,P2,X), %splits X into P2 and L  
  
Here it uses the append function to splits the list X to two components. List L which would be returned as an empty list [] in this case, and to list P2 which is returned as [3]. It then goes to the next append call.

append(L,[H],P1), %appends L and [H] to get P1

Here variable L [], returned from the previous append line, is inputted alongside the head of the current list at his depth [2]. L and [H] are appended together and they produce the list P1 which becomes [2]. It then goes to the final append call.  
  
 append(P1,P2,Permutation). %appends P1 and P2 for Permutation  
  
The program appends P1 [2] to P2 [3], producing the list Permutation [2,3]. Completing this last line, the program then starts emptying. It goes from Depth 2 to Depth 1. In this depth H is 1 and T is [2,3]. It goes back to the first line.

permute(T,X), %uses recursion until base case  
  
X is returned as the Permutation from Depth 2 which is [2,3]. It then goes to the first append call.

append(L,P2,X), %splits X into P2 and L  
  
X is split into L and P2 returned as [] and [2,3] respectively. It goes to the next line.

append(L,[H],P1), %appends L and [H] to get P1

L [] and [H] [1] are inputted as arguments. P1 is returned as [1]. It goes to the final line.  
  
 append(P1,P2,Permutation). %appends P1 and P2 for Permutation  
  
P1 [1] and P2 [2,3] are inputted as arguments, Permutation is returned as [1,2,3].  
  
This process is then repeated when the OR operator ‘;’ is inserted into the console. Except this time, it finds for a different permutation because the variables used in each append function are still stored from the permutation found before so they are just switched until all possible permutations are returned.

# Task 3c – Anagrams

In this task, a predicate was required to return different anagrams for a list of characters. The different anagrams were required to be returned in one list.

Rules Declared

* anagram(L,AnagramList)  
  This rule is the requested predicate. It makes use of calling the findall\3 predicate in-built in the prolog function library. The first argument is Anagram that is returned from the second argument permute(L,Anagram). The third argument is AnagramList which is the list containing all the different anagrams returned from the permute\2 call.
  + L: List inputted in the query
  + AnagramList: List of anagrams (that are calculated using permute)
  + Anagram: Variable storing an anagram.

Source Code (question3.pl)

/\* Question 3c. \*/

%predicate that returns all the anagagrams in a list

anagram(L,AnagramList) :-

findall(Anagram,permute(L,Anagram),AnagramList). %returns AnagramList

Console Listing

| ?- anagram([a,b,c],A).

A = [[a,b,c],[b,a,c],[b,c,a],[a,c,b],[c,a,b],[c,b,a]]

# Task 3d – Check if Sorted

In this task, a predicate is required to check if an inputted list of numbers is sorted in ascending order or not. It returns yes when sorted, and no when not sorted.

Rules Declared

* sorted([]).  
  This predicate is the first base case for sorted\1. If the list is empty it stops and just returns yes because the empty list is sorted.
* sorted([\_]).  
  This predicate is the second base case for sorted\1 when the list is not empty. When at the last element (denoted with a do not care value \_) it stops going to further depths and returns yes.
* sorted([H1, H2|T])  
  This rule is the requested predicate. It stores the first two elements of the list as H1 and H2. The rest or then stored in T. Firstly, it checks if H1 is smaller or equal to H2. If it isn’t then it fails and the list is not sorted. If it is then it makes use of recursion to check if the rest of the elements are sorted. On reaching the last element it then goes to the base case sorted([\_]) and returns yes in the console.
  + H1: First element in the list
  + H2: Second element in the list
  + T: The rest of the elements in the list

Source Code (question3.pl)

/\* Question 3d. \*/

%base cases

sorted([]).

sorted([\_]).

%predicate that succeeds when numbers are in order

sorted([H1,H2|T]) :-

H1 =< H2,

sorted([H2|T]).

Console Listing

| ?- sorted([1,2,3,4,5,6,7,8,9]).

yes

# Task 3e – Naïve Sort

In this task, a predicate is required to sort a list using the permute\2 predicate and the sorted\1 predicate.

Rules Declared

* nSort(L,Permutation)  
  This rule is the requested predicate. It finds a permutation for the inputted list. It then checks if the permutation is sorted. If it is sorted it goes to the next line and returns that permutation as the sorted list, if it is not sorted it keeps on going through the different permutations until it finds the one that is.
  + L: List to be sorted
  + Permutation: Permutation to be returned that is the sorted list after the last line.

Source Code (question3.pl)

/\* Question 3e. \*/

%predicate that sorts list

nSort(L,Permutation) :-

permute(L,Permutation), %returns permutation of list that is sorted

sorted(Permutation),

!.

Console Listing

| ?- nSort([1,4,5,3,7,8,10],R).

R = [1,3,4,5,7,8,10]

Question 4 - Solving Puzzles

# Task 4a – Crossword Puzzle

In this task, a prolog program was required to solve a specific crossword puzzle with the following words: astante , astoria , baratto , cobalto , pistola , statale. The program finds all the possible solutions for solving the crossword.

Facts Declared in Knowledge Base

* word(astante, a,s,t,a,n,t,e)
* word(astoria, a,s,t,o,r,i,a)
* word(baratto, b,a,r,a,t,t,o)
* word(cobalto, c,o,b,a,l,t,o)
* word(pistola, p,i,s,t,o,l,a)
* word(statale, s,t,a,t,a,l,e)

Rules Declared

* crossword(V1,V2,V3,H1,H2,H3)  
  This rule is the requested predicate. It solves the puzzle by referring to the knowledge base with word\8. It references to the knowledge base for every row and column, therefore 3 rows and 3 columns. For each one it uses ‘\_’ as it is a do not care variable. The program only focuses on the intersections of the vertical and horizontal letters. This always occurs at position 2,4 and 6. Therefore it first sets these variables for the vertical and then checks if it matches at the intersection from the horizontal.
  + V1, V2, V3: Words stored vertically from left to right.
  + H1, H2, H3: Words stored horizontally from top to bottom.
  + V1xL2,V1xL4,V1xL6: Letters stored at intersection between V1, H1, H2 and H3.
  + V2xL2,V2xL4,V2xL6: Letters stored at intersection between V2, H1, H2 and H3.
  + V3xL2,V3xL4,V3xL6: Letters stored at intersection between V3, H1, H2 and H3.

Source Code (question3.pl)

/\* Question 4a. \*/

%knowledge base

word(astante, a,s,t,a,n,t,e).

word(astoria, a,s,t,o,r,i,a).

word(baratto, b,a,r,a,t,t,o).

word(cobalto, c,o,b,a,l,t,o).

word(pistola, p,i,s,t,o,l,a).

word(statale, s,t,a,t,a,l,e).

%predicate that gives solutions on how to fill the grid

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

word(V1, \_,V1xL2,\_,V1xL4,\_,V1xL6,\_),

word(V2, \_,V2xL2,\_,V2xL4,\_,V2xL6,\_),

word(V3, \_,V3xL2,\_,V3xL4,\_,V3xL6,\_),

word(H1, \_,V1xL2,\_,V2xL2,\_,V3xL2,\_),

word(H2, \_,V1xL4,\_,V2xL4,\_,V3xL4,\_),

word(H3, \_,V1xL6,\_,V2xL6,\_,V3xL6,\_).

Console Listing

| ?- crossword(V1,V2,V3,H1,H2,H3).

V1 = H1 = astante ,

V2 = H2 = baratto ,

V3 = H3 = statale ;

V1 = astante ,

V2 = cobalto ,

V3 = pistola ,

H1 = astoria ,

H2 = baratto ,

H3 = statale ;

V1 = astoria ,

V2 = baratto ,

V3 = statale ,

H1 = astante ,

H2 = cobalto ,

H3 = pistola ;

V1 = H1 = astoria ,

V2 = H2 = cobalto ,

V3 = H3 = pistola ;

V1 = V2 = H1 = H2 = baratto ,

V3 = H3 = statale ;

V1 = H1 = cobalto ,

V2 = H2 = baratto ,

V3 = H3 = statale ;

no

# Task 4b – Towers of Hanoi

In this task, a program is required to solve a mathematical puzzle called the towers of Hanoi problem, where there are three poles. The pole on the left has a stack of disks and one by one all of them must be transferred to the pole on the far left using the middle, with the rule that a disk can only be placed above another if the one below it is bigger.

The written code was derived from code from this site: https://www.cpp.edu/~jrfisher/www/prolog\_tutorial/2\_3.html

It stays outputting where to move the current disk at the top of the stack, specifying from which pole to which pole.

Rules Declared

* move1(1, Start, Goal, \_)  
  This predicate is the base case. Its third argument is ‘\_’ as a Do Not Care Value. This is because at this stage, the auxiliary pole is not needed. It only needs Start and Goal as it moves a disc from one place to another. Start is the pole where the disc is moved from, and Goal is the pole the disc is moved to. It writes this movement.
  + Start: The pole the disc is to be moved from.
  + Goal: The pole the disc is to be moved to.
* move1(N,Start,Goal,Temp)  
  This recursive predicate is the requested predicate. It firstly checks if N>1, then it sets variable M as N-1.   
    
  In the first move call, it makes use of recursion until N=1. It then moves this top disc from Start to Temp through the base case (where Temp unifies with Goal in the base case).  
    
  In the second move call it goes straight to the base case and moves the top disc from Start to Goal. This does this as it would move the second smallest disc from the starting pole.  
    
  In the third move call it also makes use of recursion and moves the disc when M is 1. It moves it from Temp to Goal through the base case (where Temp unifies with Start in the base case).   
    
  When finishing the last line it then start emptying the lines, repeating the algorithm until it finishes backtracking to the first depth. At this point all of the movements required to solve the Tower of Hanoi would be printed.  
  + N: This variable stores the number of disks in the stack, on different depths it splits 1 stack into smaller stacks, and focuses on the top section until it focuses on a stack when M=1, therefore 1 disc.
  + M: Permutation to be returned that is the sorted list after the last line.
  + Start: Variable for storing the starting pole as an atom (‘left’)
  + Temp: Variable for storing the auxiliary pole as an atom (‘middle’)
  + Goal: Variable for storing the goal pole as an atom (‘right’)

Source Code (question4.pl)

/\* Question 4b. \*/

%base case

move1(1,Start,Goal,\_) :-

write('Move disk from '),

write(Start),

write(' to '),

write(Goal),

nl.

%recursive predicate

move1(N,Start,Goal,Temp) :-

N>1,

M is N-1,

move1(M,Start,Temp,Goal),

move1(1,Start,Goal,\_),

move1(M,Temp,Goal,Start).

Console Listing

| ?- move1(3,left,right,middle).

Move disk from left to right

Move disk from left to middle

Move disk from right to middle

Move disk from left to right

Move disk from middle to left

Move disk from middle to right

Move disk from left to right

Yes