

Part-A(LISP)

Experiment_No : 01

Experiment Name : A LISP program for finding the area of a Circle .

Objectives :

The goals of this experiment are to:

- Implement a LISP program that computes the area of a circle.
- Explore basic arithmetic and function structures in LISP.
- Gain experience in handling user input and displaying results in LISP.

Theory :

LISP (List Processing) is a functional programming language primarily used for symbolic computation. LISP represents both code and data as lists, making it a powerful tool for recursive processing and AI-related tasks.

The area of a circle is given by the formula:

$$A=\pi\times r^2$$

Where:

- A is the area of the circle,
- r is the radius of the circle,
- π (Pi) is approximately 3.14159.

Algorithm:

1. Start.
2. Define a function circle-area that takes the radius r as input.
3. Calculate the area using the formula $\pi\times r^2$.
4. Return the calculated area.

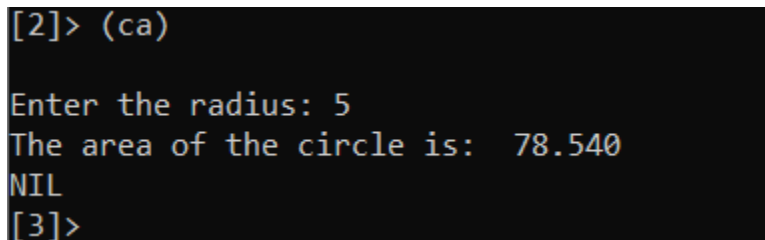
End.

Code :

```
(defun ca()
  (terpri) ;newline
  (princ "Enter the radius: ")
  (setq radius (read)) ; Read the radius as a variable
  (setq area (* 3.1416 radius radius)) ; Calculate the area
  (princ "The area of the circle is: ")
  (format t " ~,3f~%" area))
```

Input and Output:

If the user enters a radius of 10, the program calculates the area as:



```
[2]> (ca)
Enter the radius: 5
The area of the circle is: 78.540
NIL
[3]>
```

Discussion :

This experiment successfully achieved its objectives by calculating the area of a circle in LISP, demonstrating the power of functional programming for mathematical problems.

In this experiment, we developed a simple LISP program to calculate the area of a circle based on the user's input for the radius. LISP's prefix notation was used for the multiplication and exponentiation operations.

Experiment No : 02

Experiment Name : A LISP Program for Finding the Area of a Triangle

Objectives :

The main objectives of this experiment are :

- To understand the basics of LISP programming and its syntax
- To implement a function in LISP that calculates the area of a triangle.
- To perform arithmetic operations in LISP and display formatted output
- To gain practical knowledge of the principles behind geometric calculations using LISP.

Theory :

The area of a triangle can be calculated using Heron's formula or the basic formula if the base and height are known. In this experiment, we will use the simple formula for the area of a triangle:

$$\text{Area} = 1/2 \times \text{base} \times \text{height}$$

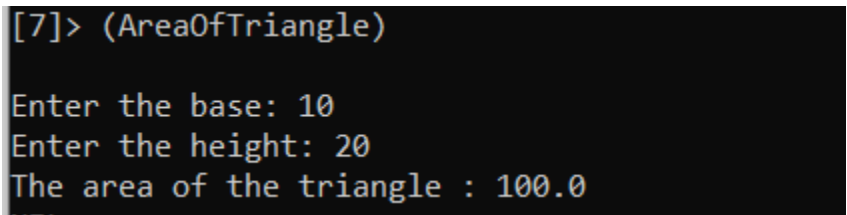
This experiment uses Common LISP, a dialect of the LISP programming language, known for its flexibility in handling functions and lists. LISP is one of the oldest high-level programming languages and is widely used in artificial intelligence.

Algorithm:

1. Start.
2. Define a function AreaOfTriangle to compute the area.
3. Prompt the user to input the base of the triangle.
4. Prompt the user to input the height of the triangle.
5. Compute the area using the formula: $1/2 \times \text{base} \times \text{height}$
6. Display the calculated area.
7. End.

Code :

```
(defun AreaOfTriangle()
  (terpri)
  (princ "Enter the base: ")
  (setq base (read))
  (princ "Enter the height: ")
  (setq height (read))
  (setq area (* 0.5 base height))
  (princ "The area of the triangle : ")
  (write area)
  (terpri))
```

Input and Output:

```
[7]> (AreaOfTriangle)
Enter the base: 10
Enter the height: 20
The area of the triangle : 100.0
```

Discussion :

This experiment successfully demonstrates the use of LISP in solving a basic geometric problem the calculation of the area of a triangle. The output shows the base and height of the triangle followed by the computed area.

Experiment No : 03

Experiment Name : A LISP Program for Finding the Average and Sum of Some Numbers .

Objectives :

The main objectives of this experiment are :

- To understand the basic syntax and working of the LISP programming language
- To develop a LISP program that calculates the sum and average of four given numbers.
- To practice defining functions and performing arithmetic operations in LISP.

Theory :

LISP (list Processing) is one of the oldest high-level programming languages, primarily used for symbolic computation and AI. It is based on recursive functions and symbolic expressions. Arithmetic operations in LISP are written in prefix notation, meaning the operator comes before the operands.

For example:

- The addition of two numbers is written as (+ 2 3) which returns 5.
- The division of two numbers is written as (/ 6 2) which returns 3.

In LISP, you can define functions using the defun keyword. A function can perform various operations, such as calculating the sum or average of numbers. The function is invoked by passing the required arguments.

In this experiment, we will define a function averagenum that takes four numbers as input and calculates the sum and average.

Algorithm:

1. Input a list of numbers.
2. Use the reduce function with the + operator to calculate the sum of the list.
3. Use the length function to find the total number of elements in the list.

4. Compute the average by dividing the sum by the number of elements.
5. Output both the sum and the average.

Code:

```
(defun sum-and-average ()  
  (terpri)  
  (princ "Enter the numbers (as a list, e.g., (1 2 3 4)): ")  
  (let ((numbers (read))) ; Read the list of numbers from user input  
    (let ((sum (reduce #' + numbers))) ; Calculate the sum of the numbers  
      (let ((average (/ sum (length numbers)))) ; Calculate the average  
        (terpri)  
        (format t "Sum: ~d~%" sum) ; Print the sum  
        (format t "Average: ~f~%" average)))) ; Print the average
```

Input and Output:

```
[9]> (sum-and-average)  
Enter the numbers (as a list, e.g., (1 2 3 4)): (10 20 30 40)  
Sum: 100  
Average: 25.0  
NIL
```

Discussion :

This experiment provided an introduction to writing simple arithmetic functions in LISP. The program successfully calculated the sum and average of four numbers, and the results were verified through sample queries.

The experiment demonstrates how to define and use basic arithmetic functions in LISP. The program effectively calculates the sum and average of four numbers using simple arithmetic operators.

Experiment No:04

Experiment Name: A LISP Program for Declaring an Array and Accessing the Elements of the Array

Objective: To write a LISP program that demonstrates how to declare an array, initialize its elements, and access individual elements of the array.

Theory: In LISP, arrays are multi-dimensional data structures used to store elements that can be accessed via indices. Arrays provide random access to elements, meaning that each element can be retrieved or modified directly using its index. LISP arrays are declared using the make-array function, and their elements can be accessed or modified using the aref function.

- **Array Declaration:** Arrays in LISP can be declared using the make-array function. For example: (make-array '(size)) declares a one-dimensional array of the specified size.
- **Accessing Array Elements:** The function aref is used to access or modify elements of the array by specifying the index. Indexing in LISP arrays starts from 0.

Program syntax:

1.Array Declaration: Use the make-array function to declare an array of a given size

2.Array Initialization: Use the setf and aref functions to initialize the elements of the array.

3.Array Access: Access the elements of the array using the aref function, specifying the index.

4.Array Modification: Modify an element by using setf with aref.

Algorithm:

Step 1 : Declare and initialize the array arr with specific values.

Step 2 : Display the message "Array:".

Step 3 : Loop through the array using dotimes to access each element.

Step 4 :Print each element of the array in sequence.

Program Code:

```
(defun array-example ()  
  (let ((my-array (make-array 5))) ;; Declare the array using LET  
    (dotimes (i 5)  
      (format t "Enter value for element ~d: " i)  
      (setf (aref my-array i) (read))) ;; Set the array elements with user input  
    (format t "~%Array Elements::~~%")  
    (dotimes (i 5) ;; Loop through the array and print elements  
      (format t "Element ~d: ~d~%" i (aref my-array i)))))
```

Input and Output:

```
[2]> (array-example)  
Enter value for element 0: 5  
Enter value for element 1: 4  
Enter value for element 2: 10  
Enter value for element 3: 15  
Enter value for element 4: 20  
  
Array Elements:  
Element 0: 5  
Element 1: 4  
Element 2: 10  
Element 3: 15  
Element 4: 20
```

Conclusion: The LISP program demonstrates how to declare an array, initialize its elements, access individual elements, and modify them. The make-array function is used to create the array, and the aref function is used for both accessing and modifying elements. The experiment successfully shows how arrays can be manipulated in LISP, which is essential for data storage and retrieval in many applications.

Experiment No:05

Experiment Name: LISP Program for Finding the Outputs of All Basic LIST Manipulation

Objective:

To develop a LISP program that demonstrates various basic list manipulation operations, including addition, deletion, searching, and traversal, and to output the results of these operations.

Theory:

In LISP, lists are one of the fundamental data structures. They can be manipulated using various built-in functions. Basic list manipulation operations include:

- **car:** Returns the first element of a list.
- **cdr:** Returns the list without its first element.
- **cons:** Adds an element to the front of a list.
- **list:** Concatenates two or more lists.

These operations are essential for working with lists in LISP, and they serve as building blocks for more complex data manipulations.

Algorithm:

Step 1 : Initialize a list with specific values and display it.

Step 2 : Create a new list by adding an element at the beginning of using the cons function, and display the new list.

Step 3 : Extract and display the first element of using the car function.

Step 4 : Extract and display the remaining elements of using the cdr function.

Step 5 : Create a nested list containing and using the list function, and display the nested list.

Code:

(write (car '(a b c d e f))) ; Access the first element of the list

(terpri) ; Newline

(write (cdr '(a b c d e f))) ; Access the rest of the list except the first element

(terpri) ; Newline

(write (cons 'a '(b c))) ; Construct a new list with 'a' as the first element and '(b c)' as the rest

(terpri) ; Newline

(write (list 'a '(b c) '(e f))) ; Create a list of three elements: 'a', '(b c)', and '(e f)'

(terpri) ; Newline

Input and Output:

```
[3]> (write (car '(a b c d e f))) ; Access the first element of the list
A
A(terpri) ; Newline

[4]>
NIL(write (cdr '(a b c d e f))) ; Access the rest of the list except the first element

[5]> (B C D E F)
(B C D E F)(terpri) ; Newline

[6]>
NIL(write (cons 'a '(b c))) ; Construct a new list with 'a' as the first element and '(b c)' as the rest

[7]> (A B C)
(A B C)(terpri) ; Newline

[8]>
NIL(write (list 'a '(b c) '(e f))) ; Create a list of three elements: 'a', '(b c)', and '(e f)'

[9]> (A (B C) (E F))
(A (B C) (E F))
```

Conclusion: The LISP program demonstrates the functionality of several fundamental list manipulation operations. By utilizing cons, the program adds an element to the front of an existing list. The car function retrieves the first element, while the cdr function returns the remaining elements of the list. The list function combines two lists into a nested list.

Experiment No:06

Experiment Name: A LISP Program for Finding the Outputs of All Additional LIST Manipulation

Objective: To develop a LISP program that demonstrates the use and output of additional list manipulation functions such as append, reverse, member, last, and others, and to understand their behavior.

Theory:

In LISP, lists are a fundamental data structure used for various types of data manipulation. LISP provides numerous built-in functions for working with lists beyond the basic operations like car, cdr, and cons. Some of the additional list manipulation functions include:

- **append:** Combines multiple lists into one. Takes multiple lists and concatenates them into a single list, keeping the order of elements.

Example: (append '(1 2) '(3 4) '(5 6))

;; Output: (1 2 3 4 5 6)

- **reverse:** Reverses the order of elements in a list. Returns a new list with the elements in reverse order. It does not modify the original list.

Example: (reverse '(1 2 3 4))

;; Output: (4 3 2 1)

- **member:** Checks if an element is a member of a list and returns the sublist starting from that element.

Example: (member 3 '(1 2 3 4))

;; Output: (3 4)

- **last:** Returns the last element(s) of a list.

Example: (last '(1 2 3 4))

;; Output: (4)

- These functions extend the basic capabilities of list manipulation and are crucial for more complex data processing tasks in LISP.

Algorithm:

Step 1 : Initialize a list with specific values and display the current list.

Step 2: Append several list using the append function and display the resulting new list.

Step 3 : Retrieve and display the last element of using the last function.

Step 4 : Check if the element is present in list using the member function; if it is, then return the sublist from this.

Step 5: Reverse the list using the reverse function and display the reversed list.

Code:

```
(write (append '(b c) '(e f) '(p q) '() '(g))) ; Concatenate several lists into one  
(terpri) ; Newline
```

```
(write (last '(a b c d (e f)))) ; Access the last element of the list, which can be a sublist  
(terpri) ; Newline
```

```
(write (reverse '(a b c d (e f)))) ; Reverse the list  
(terpri) ; Newline
```

```
(write (member 'c '(a b c d e f))) ; Check if 'c' is a member of the list and return the  
sublist starting from 'c'
```

Input and Output:

```
[11]> (write (append '(b c) '(e f) '(p q) '() '(g))) ; Concatenate several lists into one
(B C E F P Q G)
(B C E F P Q G)(terpri) ; Newline

[12]>
NIL

[13]> (write (last '(a b c d (e f)))) ; Access the last element of the list, which can be a sublist
((E F))
((E F))(terpri) ; Newline

[14]>
NIL

[15]> (write (reverse '(a b c d (e f)))) ; Reverse the list
((E F) D C B A)
((E F) D C B A)(terpri) ; Newline

[16]>
NIL

[17]> (write (member 'c '(a b c d e f))) ; Check if 'c' is a member of the list and return the sublist starting from 'c'
(C D E F)
(C D E F)
[18]>
```

Discussion :

This LISP program demonstrates key list manipulation functions such as `append`, `last`, `reverse`, and `member`, showcasing how lists can be efficiently combined, accessed, reversed, and searched. The `append` function merges multiple lists, while `last` retrieves the final element or sublist. `reverse` allows reversing the order of elements, and `member` locates an element, returning the sublist from that point onward. These functions highlight the flexibility and power of LISP's list processing capabilities, enabling concise handling of complex data structures with minimal code.

Experiment No: 07

Experiment Name: A LISP Program for Finding the Maximum and Minimum of Some Numbers.

Objective:

To develop a LISP program that computes both the maximum and minimum values from a given list of numbers.

Theory:

In LISP, lists are commonly used for storing sequences of numbers. The maximum value of a list is the largest element, and the minimum value is the smallest element. LISP provides functions like `reduce` along with the comparison functions `max` and `min` to easily find the largest and smallest elements of a list.

- **max:** Returns the largest of two or more arguments.
- **min:** Returns the smallest of two or more arguments.
- **reduce:** Applies a function recursively to the elements of a list to reduce it to a single result, such as finding the maximum or minimum.

Algorithm:

Step 1: Display a prompt asking the user to enter a list of numbers.

Step 2: Read the user input and store it as a list of numbers.

Step 3: Use the ``max`` function to find the largest number in the list.

Step 4: Use the ``min`` function to find the smallest number in the list.

Step 5: Display the maximum number.

Step 6: Display the minimum number.

Step 7: End the program.

Code:

```
(defun find-max-min ()  
  (terpri)  
  (princ "Enter a list of numbers (e.g., (1 2 3 4 5)): ")  
  (let ((numbers (read))) ; Read the list of numbers from user input  
    (let ((maximum (apply #'max numbers)) ; Find the maximum value  
          (minimum (apply #'min numbers))) ; Find the minimum value  
      (terpri)  
      (format t "Maximum: ~d~%" maximum) ; Print the maximum  
      (format t "Minimum: ~d~%" minimum)))) ; Print the minimum
```

Input and Output:

Enter a list of numbers (e.g., (1 2 3 4 5)): (5 20 44 10)

Maximum: 44

Minimum: 5

Discussion:

The LISP program successfully finds both the maximum and minimum values in a list of numbers. By using the reduce function with max and min, the program efficiently computes these values in a single traversal of the list. This demonstrates the power of functional programming in LISP to process lists with minimal code while utilizing built-in higher-order functions for list manipulation.

Experiment No:08

Experiment Name: A LISP Program for Finding the Outputs of the putprop Function

Objective:

To write a LISP program that demonstrates the use of the putprop function to assign and retrieve properties associated with symbols, and to explore its outputs when various properties are manipulated.

Theory:

In LISP, symbols can have associated properties stored in property lists (plist). The putprop function is used to associate a property with a value for a symbol. Properties are stored as key-value pairs in the symbol's property list, and each symbol can have multiple properties.

- **putprop:** This function takes three arguments: a value, a property name (key), and a symbol. It associates the property with the given value for the specified symbol.

Syntax: (putprop symbol value property)

- **get:** This function retrieves the value of a property for a given symbol.

Syntax: (get symbol property)

Property List: A list of alternating property names and values associated with a symbol. Each symbol has its own property list.

Algorithm:

1. Create a symbol to which properties will be assigned.
2. Use the putprop function to associate multiple properties (key-value pairs) with the symbol.
3. Use the get function to retrieve and display the value of each property.
4. Display the property list and the retrieved property values.

Code:

```
(defun putprop-example ()  
  ;; Define a symbol 'person  
  (setq person 'john)  
  ;; Assign properties to the symbol using putprop  
  (putprop person 'male 'gender) ; Assign gender as male  
  (putprop person 24 'age)       ; Assign age as 24  
  (putprop person 'engineer 'profession) ; Assign profession as engineer  
  ;; Retrieve and display the properties using get  
  (format t "Gender: ~a~%" (get person 'gender)) ; Retrieve gender  
  (format t "Age: ~a~%" (get person 'age))       ; Retrieve age  
  (format t "Profession: ~a~%" (get person 'profession)) ; Retrieve profession  
  ;; Display the full property list of the symbol  
  (format t "Property list of 'john: ~a~%" (symbol-plist person)))
```

Input & Output:

Gender: male

Age: 24

Profession: engineer

Property list of 'john: (profession engineer age 24 gender male)

Discussion:

The LISP program successfully demonstrates the use of the putprop function to assign properties to a symbol and the get function to retrieve these properties. The property list of a symbol is a powerful feature in LISP, allowing symbols to store additional data as key-value pairs.

Part-B(Prolog)

Experiment No : 01

Experiment Name : A Prolog program to find the GPA of a student .

Objectives :

The objective of this experiment is to implement a Prolog program to find the GPA of a student by:

- Defining facts that associate students with their GPAs.
- Using rules to retrieve and display the GPA when a student's name is entered by the user.

Theory :

Prolog is a logic programming language that uses facts, rules, and queries to solve problems. In this program, we will define facts for student names and their corresponding GPAs. A fact in Prolog represents a known piece of information, like `gpa(student, gpa_value)`.

The process works as follows:

- **Defining Facts:** We will define a set of facts representing the GPA of each student.
- **Rules:** We create a rule to read the student's name from the user input, search for their GPA using the fact database, and display the result.
- **Querying:** When the user queries the program by providing a student's name, Prolog will find the matching GPA and return the result.

Key Points in Prolog:

- `read(X)` reads input from the user.
- `write(Y)` writes or displays output to the console.
- `gpa(X, Y)` represents a fact where X is the student name and Y is the GPA.

Fact:

X = student name, Y= GPA of the student. gpa(X,Y).

Rules:

Read student name from user and write the GPA. read(X),gpa(X,Y),nl, write(Y).

Program Syntax :

- Reading User Input: Use read(X) to capture the student name from the user.
- Querying GPA: Use gpa(X, Y) to match the student's name with their GPA.
- Displaying the Output: Use write('GPA is '), write(Y) to print the result.

Algorithm:

Step 1: Prompt the user to enter the student's name.

Step 2: Read the input and match it with the corresponding student's GPA in the database.

Step 3: If the student's GPA is found, display the GPA.

Step 4: If not found, return no result (fail silently).

Program Code:

```
gpa(jim,3.66).
```

```
gpa(atel,3.80).
```

```
gpa(moni,3.25).
```

```
gpa(sadi,3.35).
```

```
gpa(abid,3.55).
```

```
result:- write('Enter student name:'),
```

```
read(X),gpa(X,Y),nl,
```

```
write('GPA is '),
```

```
write(Y).
```

Input and Output:

?- result.

Enter student name:jim.

GPA is 3.66

true.

Discussion:

The Prolog program effectively calculates the GPA of a student by utilizing lists to represent courses, their grades, and credit hours. The program uses the `grade_point` facts to retrieve numerical equivalents for letter grades, calculates total grade points, and divides by total credit hours to obtain the GPA. This experiment demonstrates the application of Prolog's logical programming capabilities in educational data processing, making it a powerful tool for academic performance evaluation.

Experiment No : 02

Experiment Name : A Prolog program to find nth number of Fibonacci series .

Objective:

To learn how to find nth number of Fibonacci series in prolog programming.

Theory:

The Fibonacci numbers, commonly denoted F_n , form a sequence, called the Fibonacci sequence, such that each number is the sum of the two preceding ones, starting from 0 and 1.

The Fibonacci sequence is defined as:

- $F(0) = 0$
- $F(1) = 1$
- $F(n) = F(n-1) + F(n-2)$ for $n > 1$

This recursive relationship can be easily translated into a Prolog program. Prolog is a logic programming language that works well with recursive structures, making it suitable for problems like the Fibonacci series. The recursion in the Fibonacci series works as follows:

- The base cases are defined for $n = 0$ and $n = 1$.
- For $n > 1$, the Fibonacci number is calculated as the sum of the two preceding Fibonacci numbers ($F(n-1)$ and $F(n-2)$).

Program Syntax:

The basic structure of the Fibonacci function in Prolog uses recursive predicates:

- **Base cases:**

fibonacci(0, 0).

fibonacci(1, 1).

• Recursive case:

fibonacci(N, Result) :-

N1 is N - 1,

N2 is N - 2,

fibonacci(N1, R1),

fibonacci(N2, R2),

Result is R1 + R2.

Algorithm:

Step 1 : Define the base cases: The Fibonacci value for the 1st and 2nd terms is 1.

Step 2 : For any number N greater than 2, calculate the two preceding Fibonacci numbers by recursively calling the function for N-1 and N-2.

Step 3 : Add the results of these recursive calls to get the Fibonacci value for N.

Step 4 : Return the computed Fibonacci value.

Program Code:

fib(0,0).

fib(1,1).

fib(N,F):- N>1,

N1 is N-1,

N2 is N-2,

fib(N1,F1),

fib(N2,F2),

F is F1+F2.

Input and Output:

?-

| fib(5,F).

F = 5 .

Discussion :

In this experiment, we implemented a Prolog program to compute the nth Fibonacci number. This Prolog program demonstrates how recursion can be used to solve problems such as finding the nth Fibonacci number. While the recursive solution works well for small values of n, more efficient approaches (like using iterative methods or dynamic programming) may be required for larger values. Nonetheless, this experiment successfully illustrates the power of recursion in Prolog.

Experiment No : 03

Experiment Name : A prolog program to calculate factorial of a number .

Objectives:

To know to calculate factorial of a number using prolog programming.

Theory:

We know, factorial of a number n is, $n! = n \times (n - 1) \times (n - 2) \times \dots \times 1$

In Prolog, recursion is used to compute the factorial. The recursive definition is as follows:

1. Base case: Factorial of 0 is 1.
2. Recursive case: Factorial of N is $N \times \text{factorial}(N-1)$.

Prolog uses pattern matching and recursion to solve this problem. The base case is the termination condition, and the recursive case reduces the problem into smaller instances until it reaches the base case.

Program Syntax:

To calculate factorial, the following facts and rules are defined:

- 1) If $N=0$, then $R=1$. i.e., $0! = 1$. `fact(0,1).`
- 2) If $N \neq 0$, $N! = R$:- `call fact(N1,R1), N=N1+1, R=R1*N.`

Algorithm:

Step 1 : Define the base case: The factorial of 0 is 1.

Step 2 : For any number N greater than 0, calculate $N-1$ and recursively call the function to find the factorial of $N-1$.

Step 3 : Multiply the result of the recursive call by N to compute the factorial of N.

Step 4 : Return the computed factorial value.

Program Code:

```
fact(0,1).  
fact(N,F):- N>0,  
N1 is N-1,  
fact(N1,F1),  
F is N*F1.
```

Input and Output:

```
?-  
| fact(5,F).  
F = 120
```

Discussion :

This experiment successfully demonstrated the use of recursion in Prolog to calculate the factorial of a number. By defining a base case and a recursive case, we built a clear and correct solution to the factorial problem. Understanding recursion is crucial for problem-solving in Prolog, and this program illustrates how recursion can be effectively utilized for mathematical computations.

The base case ($0! = 1$) ensures that the recursion terminates, while the recursive rule ($N! = N * (N-1)!$) allows for the calculation of the factorial of any positive integer. The program demonstrates how recursion can be efficiently applied in Prolog to solve mathematical problems.

Experiment No : 04

Experiment Name : A prolog program to find GCD and LCM of two numbers .

Objectives:

To learn how to compute the Greatest Common Divisor (GCD) and the Least Common Multiple (LCM) of two numbers using Prolog programming.

Theory :

GCD: The greatest common divisor (gcd) of two or more integers, which are not all zero, is the largest positive integer that divides each of the integers. For two integers x, y , the greatest common divisor of x and y is denoted $\text{gcd}(x,y)$.

LCM: The least common multiple of two integers a and b , usually denoted by $\text{lcm}(a, b)$, is the smallest positive integer that is divisible by both a and b .

Program Syntax:

Following facts and rules are defined:

- If the two numbers are X and 0 , then $\text{gcd}=X$. $\text{gcd}(X,0,X)$.
- If the two numbers are X and Y , then $R=X \bmod Y$, $\text{gcd}(Y,R,D)$ $\text{gcd}(X,Y,D):-$ R is $X \bmod Y$, $\text{gcd}(Y,R,D)$. 3) If the two numbers are X and Y , $\text{gcd}(X,Y,D)$, $M=(X*Y)/D$.

$\text{lcm}(X,Y,M):- \text{gcd}(X,Y,D),M$ is $(X*Y)/D$.

Algorithm:

GCD Calculation :

Step 1 : If $Y=0$, the GCD of X and Y is X .

Step 2 : Otherwise, compute the remainder R of X divided by Y and recursively call the GCD function with Y and R .

LCM Calculation :

Step 1 : First, calculate the GCD of X and Y.

Step 2 : Then, calculate the LCM using the formula: $LCM(X,Y)=X \times Y / GCD(X,Y)$.

Step 3 : Return the LCM value.

Program Code:

% Base case: GCD of a number and 0 is the number itself

gcd(X, 0, X).

% Recursive case: Calculate the remainder and recursively find GCD

gcd(X, Y, D) :-

 R is X mod Y,

 gcd(Y, R, D).

% LCM is calculated using GCD

lcm(X, Y, M) :-

 gcd(X, Y, D),

 M is (X * Y) // D.

Input and Output:

?- gcd(12,8,G).

G = 4 .

?- lcm(12,8,L).

L = 24

Discussion :

In this experiment, we successfully implemented a Prolog program to find the GCD and LCM of two numbers. The use of recursive definitions for GCD and the mathematical relationship between GCD and LCM makes the program both concise and efficient.

Experiment No : 05

Experiment Name: A prolog program for a cyclic graph that will find the path between two vertices in a cyclic graph.

Objectives:

- To write a Prolog program for a cyclic graph that will find a path between two vertices in the graph.
- To explore how Prolog can be used to model a graph and implement pathfinding in cyclic structures.

Theory :

- **Graph:** A graph is a data structure that consists of a set of vertices (or nodes) and edges (or arcs) that connect pairs of vertices. It is widely used in computer science to represent networks, such as social networks, transportation routes, or communication systems.
- **Cyclic Graph:** A cyclic graph contains at least one cycle, meaning that there is a path from a vertex to itself through a sequence of edges. Cyclic graphs allow multiple paths between two vertices, which makes finding paths more complex due to the risk of infinite loops.
- **Pathfinding in Cyclic Graphs:** To find a path between two vertices in a cyclic graph, we must avoid visiting the same vertex more than once. This ensures that we don't get stuck in a cycle indefinitely. We can achieve this by keeping track of the vertices we have already visited while exploring different paths.

In this experiment, we use a cyclic graph with the following structure:

Vertex	Connected with
A	B, C
B	D

C	
D	E
E	B

Algorithm:

Step 1 : Start path search from node A to node B by calling nextRoute(A, B, []).

Step 2 : Traverse the graph by following edges from node A to node X, avoiding cycles by checking visited nodes.

Step 3 : If the destination node B is found, print the path; otherwise, recursively continue the search with the next node.

Step 4 : Stop once the destination is reached and output the complete path.

Program Code :

```
% Define the edges of the graph
```

```
edge(a, b).
```

```
edge(a, c).
```

```
edge(b, d).
```

```
edge(d, e).
```

```
edge(e, b).
```

```
% Define the path predicate that finds a path from vertex A to vertex B
```

```
path(A, B) :-
```

```
    nextRoute(A, B, []),
```

```
    write(B).
```

```
% nextRoute explores the graph recursively while avoiding cycles
```

```
nextRoute(A, B, Visited) :-
```

```
edge(A, X),      % Find an edge from A to another vertex X
write(A), write(' -> '),
not(member(X, Visited)), % Ensure X has not been visited before
(
    B = X;      % If X is the target vertex, stop
    nextRoute(X, B, [A | Visited]) % Otherwise, continue from X
).
```

Input and Output:

?- path(a,d).

a -> b -> d

true

Conclusion:

This Prolog program successfully finds a path between two vertices in a cyclic graph while avoiding infinite loops through cycle detection. By using recursion and tracking visited nodes, the program efficiently traverses the graph to locate the destination.

Experiment No : 06

Experiment Name : A Prolog programs for file handling.

Objectives:

- To write Prolog programs to demonstrate file handling using tell, told, see, and seen predicates.
- To learn how to read from and write to files using Prolog predicates .

Theory :

File handling in Prolog involves reading from and writing to external files. Prolog provides specific predicates to perform these tasks:

- **tell:** This predicate is used to open a file for writing. Once a file is opened, any output sent to the console (using write) is instead redirected to the specified file.
- **told:** This predicate is used to close the file that was opened using tell. After the file is closed, the program returns to normal console operations. It is important to close files after writing to prevent data loss or corruption.
- **see:** This predicate is used to open a file for reading. Once a file is opened using see, the input from that file can be read using the read/1 predicate.
- **seen:** This predicate closes the file opened by see and returns control to the console.

In this experiment, we will explore both reading and writing to files using Prolog.

Algorithm:

Step 1 : Call writeinfile to open a file 'mfile.txt' for writing, write the text 'Start' to it, and close the file.

Step 2 : In readfromfile(Infile, Outfile), open the input file (Infile) for reading and the output file (Outfile) for writing.

Step 3 : Read three items (X1, X2, X3) from the input file using the read/1 predicate.

Step 4 : Write each of the three read items to the output file, followed by a newline after each.

Step 5 : Close both the input and output files using seen for reading and told for writing.

writeinfile :-

```
tell('G:/New folder/7th/AI/ai lab/prolog/filen.txt'), % Open file for writing
write('(this is a boy).'), nl, % Write 'radif' and add a new line
told. % Close the file
```

Program Code :

Program for Writing to a File

writeinfile :-

```
tell('G:/New folder/7th/AI/ai lab/prolog/filen.txt'), % Open file for writing
write('hello'), nl, % Write 'radif' and add a new line
told. % Close the file
```

Program for Reading from One File to another

readfromfile :-

```
readfromfile('G:/New folder/7th/AI/ai lab/prolog/filen.txt',
             'G:/New folder/7th/AI/ai lab/prolog/output.txt').
```

readfromfile(Infile, Outfile) :-

```
see(Infile), % Open input file for reading
tell(Outfile), % Open output file for writing
read(Sentence), % Read a Prolog term from input file
```

```
write(Sentence), nl, % Write the term to output file  
seen, % Close input file  
told. % Close output file
```

Input and Output:

```
?- writeinfile.  
true.  
  
?- readfromfile.  
true.  
  
?- ■
```

Conclusion:

This Prolog program demonstrates basic file handling operations, including writing data to a file and reading data from one file to write into another. By utilizing the tell, see, read, and write predicates, the program effectively performs file I/O operations. This exercise highlights Prolog's ability to manage external files, allowing for efficient data storage and retrieval within a logical programming environment.

Part-C(Searching Algorithm)

Experiment No : 01

Experiment Name : To implement an Breadth First Search(BFS) Algorithm .

Objectives:

To implement and analyze the Breadth-First Search (BFS) algorithm for traversing or searching a graph or tree data structure .

Theory :

Breadth-first search is of the most common search strategies. It generally starts from the root node and examines the neighbor nodes and then moves to the next level. It uses First-in First-out (FIFO) strategy as it gives the shortest path to achieving the solution. BFS is used where the given problem is very small and space complexity is not considered.

Algorithm :

Step-1: Form a one element queue Q consisting of the root node.

Step-2: Until the Q is empty or the goal has been reached, determine if the first element in the Q is the goal.

- a. If it is, do nothing.
- b. If it isn't, remove the first element from the Q, and add the first element's children, if any, to the BACK of the Q.

Step-3: If the goal is reached, success; else failure.

Code :

```
#include<bits/stdc++.h>

using namespace std;

vector<int> graph[100];

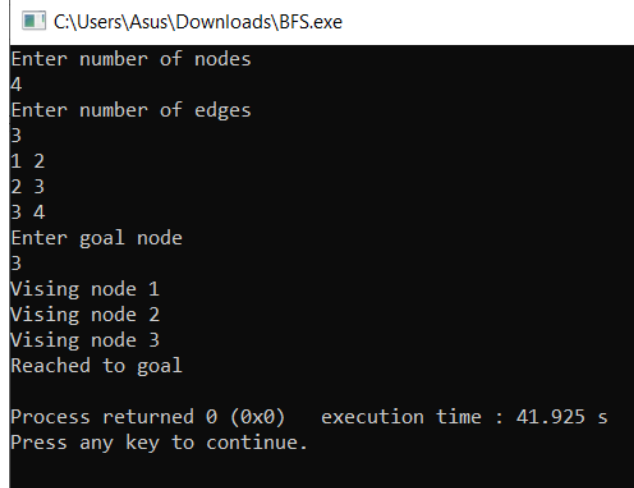
bool visited[100];
```

```
int goal;

void bfs(int src){
    queue<int> q;
    q.push(src);
    visited[src] = true;
    while (!q.empty()) {
        int current = q.front();
        q.pop();
        cout<<"Vising node " <<current << endl;
        if(current == goal){
            cout<< "Reached to goal\n";
            return;}
        for(auto &x : graph[current]){
            if (!visited[x]){
                q.push(x);
                visited[x] = true; } } } }
int main()
{
    int number_of_nodes,num_of_edges;
    cout<< "Enter number of nodes\n";
    cin>>number_of_nodes;
    cout<<"Enter number of edges\n";
    cin>>num_of_edges;
    for(int i = 0; i < num_of_edges; i++)
```

```
{  
    int u,v;  
    cin >> u >> v;  
    graph[u].push_back(v);  
    graph[v].push_back(u); }  
cout << "Enter goal node\n";  
cin>> goal;  
bfs(1);  
return 0;  
}
```

Input and Output :



```
C:\Users\Asus\Downloads\BFS.exe  
Enter number of nodes  
4  
Enter number of edges  
3  
1 2  
2 3  
3 4  
Enter goal node  
3  
Vising node 1  
Vising node 2  
Vising node 3  
Reached to goal  
  
Process returned 0 (0x0) execution time : 41.925 s  
Press any key to continue.
```

Discussion :

We successfully completed the implementation of Breadth First Search Algorithm using c++ programming language.

Experiment No : 02

Experiment Name : Implementation of Depth First Search (DFS) Algorithm .

Objectives

The main objective of this experiment is to write a program for Depth First Search(DFS) Algorithm.

Theory

The depth-first search uses Last-in, First-out (LIFO) strategy and hence it can be implemented by using stack. DFS uses backtracking. That is, it starts from the initial state and explores each path to its greatest depth before it moves to the next path.

DFS will follow : Root node —> Left node —> Right node

Algorithm :**1. Initialize:**

- Start with an initial node or root node.
- Create an empty stack to store nodes to visit.
- Push the initial node onto the stack.
- Create an empty set to keep track of visited nodes.

2. Check Goal:

- If the initial node is the goal, return the solution.

3. Loop until the stack is empty:

- Pop the top node from the stack.
- If the node is the goal, return the solution.
- If the node has not been visited, mark it as visited.

4. Expand the node:

- Get the children or successors of the current node.

- Push all unvisited children of the current node onto the stack.

5. Backtrack if necessary:

- If no more nodes to expand from the current node, backtrack by popping the next node from the stack.
- Repeat the process until the stack is empty or the goal is found.

6. Termination:

- If the stack is empty and no solution is found, return failure.

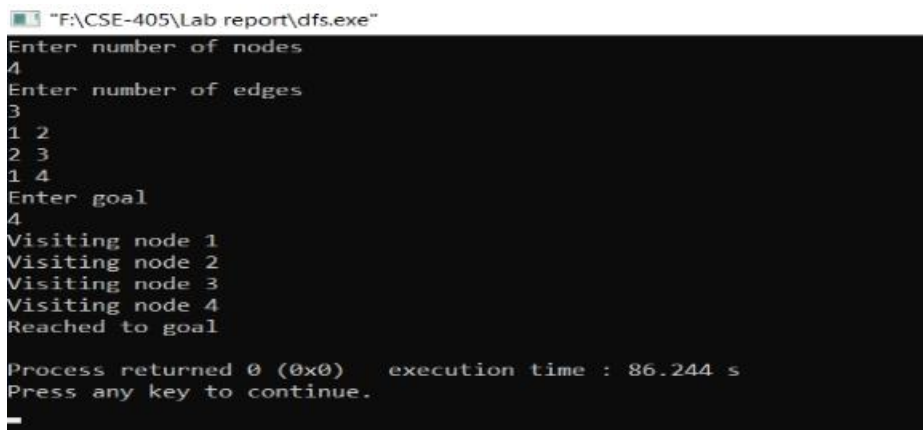
Code :

```
#include <bits/stdc++.h>
using namespace std;
vector<int> graph[100];
bool visited[100];
int goal;
void dfs(int src){
    cout << "Visiting node " << src << endl;
    visited[src] = true;
    if (src == goal){
        cout << "Reached to goal\n"; return; }
    for (auto &x : graph[src]){
        if (!visited[x]) dfs(x); }
    }
int main(){
    int number_of_nodes, num_of_edges;
```

```
cout << "Enter number of nodes\n";
cin >> number_of_nodes;
cout << "Enter number of edges\n";
cin >> num_of_edges;
for (int i = 0; i < num_of_edges; i++){
    int u, v; cin >> u >> v;
    graph[u].push_back(v);
    graph[v].push_back(u); }
cout << "Enter goal\n";
cin >> goal; dfs(1);

return 0;
}
```

Input and Output :



```
"F:\CSE-405\Lab report\dfs.exe"
Enter number of nodes
4
Enter number of edges
3
1 2
2 3
1 4
Enter goal
4
Visiting node 1
Visiting node 2
Visiting node 3
Visiting node 4
Reached to goal

Process returned 0 (0x0)   execution time : 86.244 s
Press any key to continue.
_
```

Discussion :

We successfully completed the implementation of Depth First Search Algorithm using C++ programming language.

Experiment No : 03

Experiment Name : Implementation of A* Search Algorithm .

Objectives:

The main objective of this experiment is to write a program for A* Search Algorithm.

Theory :

A* is a graph traversal and pathfinding algorithm that uses a combination of breadth-first search and heuristic optimization. It is widely used to find the shortest path in many practical applications like GPS navigation systems, AI game development, and robotics.

The A* algorithm uses two main components:

1. **g(n)**: The cost to reach the current node from the start node.
2. **h(n)**: A heuristic estimate of the cost to reach the goal from the current node.

The total estimated cost of a path is given by:

- $f(n) = g(n) + h(n)$

A* explores the node with the smallest $f(n)$ first, ensuring an efficient pathfinding process.

Algorithm :

Step-1 : Place the starting node in the OPEN list.

Step-2 : Check if the OPEN list is empty or not, if the list is empty then return failure and stops.

Step-3 : Select the node from the OPEN list which has the smallest value of evaluation function ($g+h$), if node n is goal node then return success and stop, otherwise

Step-4 : Expand node n and generate all of its successors, and put n into the closedlist. For each successor n' , check whether n' is already in the OPEN or CLOSED list, if not then compute evaluation function for n' and place into Open list.

Step-5 : Else if node n' is already in OPEN and CLOSED, then it should be attached to the back pointer which reflects the lowest $g(n')$ value.

Step-6 : Return to Step 2.

Step-7 : End .

Code :

```
#include <bits/stdc++.h>

using namespace std;

int heuristic[200], src, goal;
map<pair<int, int>, int> path_cost;
bool visited[100];
vector<int> graph[300];

struct Node {
    int cost, value;
};

struct comp {
    bool operator()(Node a, Node b) {
        return a.cost + heuristic[a.value] > b.cost + heuristic[b.value];
    }
};

queue<int> closed; // Use to store the path (just the values, not Node objects)

bool a_star() {
    priority_queue<Node, vector<Node>, comp> pq;
```

```
Node node;
node.value = src;
node.cost = 0;
pq.push(node);
while (!pq.empty()) {
    Node current = pq.top();
    pq.pop();
    if (visited[current.value]) continue; // If already visited, skip it
    visited[current.value] = true;
    closed.push(current.value);
    if (current.value == goal) return true; // Goal found
    for (int i = 0; i < graph[current.value].size(); i++) {
        Node new_node;
        new_node.value = graph[current.value][i];
        if (!visited[new_node.value]) {
            new_node.cost = path_cost[{current.value, new_node.value}] +
current.cost;
            pq.push(new_node);} } }
    return false;}
int main() {
    int nof_node, nof_edges;
    cout << "Enter number of nodes & number of edges\n";
    cin >> nof_node >> nof_edges;
    cout << "Enter edges with cost (Format: A B cost)\n";
    for (int i = 0; i < nof_edges; i++) {
```

```
char ch1, ch2;
int cost;
cin >> ch1 >> ch2 >> cost;
int u = ch1 - 'A';
int v = ch2 - 'A';
graph[u].push_back(v);
graph[v].push_back(u);
path_cost[{u, v}] = cost;
path_cost[{v, u}] = cost;
}
cout << "Enter heuristic values (Format: A heuristic)\n";
for (int i = 0; i < nof_node; i++) {
    char ch;
    cin >> ch;
    cin >> heuristic[ch - 'A'];
}
cout << "Enter source and goal (Format: A B)\n";
char s, g;
cin >> s >> g;
src = s - 'A';
goal = g - 'A';
if (a_star()) {
    cout << "Goal found\nPath: ";
    while (!closed.empty()) {
        cout << char(closed.front() + 'A') << " ";
    }
}
```

```
        closed.pop();
    }
    cout << endl;
} else {
    cout << "Not found\n";
}
return 0;
}
```

Input and Output:

```
"C:\Users\Asus\Downloads\A star.exe"
Enter number of nodes & number of edges
6 8
Enter edges with cost (Format: A B cost)
s a 1
a b 2
b d 5
a c 1
c d 3
d g 2
c g 4
a s 10
Enter heuristic values (Format: A heuristic)
s 5
a 3
b 4
c 2
d 6
g 0
Enter source and goal (Format: A B)
s g
Goal found
Path: s a c g

Process returned 0 (0x0)   execution time : 158.567 s
Press any key to continue.
```

Discussion :

We successfully completed the implementation of greedy A* search algorithm using c++ programming language. Using this program we can check in a graph for a goal node either it is reachable or not using A* search approach.