EXP 7:

// Q: Write a Java program to accept two number from the user and calculate the average.

```
import java.util.*;
public class Sample {
	public static void main(String args[])
	{
	double a,b,sum=0,avg=0;
		Scanner scan=new Scanner(System.in);
		System.out.println("Enter the First No:");
		a=scan.nextDouble();
		System.out.println("Enter the Second No:");
		b=scan.nextDouble();
		sum=a+b;
		avg=sum/2;
		System.out.println("Average="+avg);
	}
}
```

EXP 8:

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

// Define the structure for a node in the syntax tree
typedef struct TreeNode {
  char data; // For simplicity, assuming data is either an operator or operand
  struct TreeNode* left;
  struct TreeNode* right;
} TreeNode;

// Function to create a new node
TreeNode* createNode(char data) {
  TreeNode* newNode = (TreeNode*)malloc(sizeof(TreeNode));
  if (newNode == NULL) {
    printf("Memory allocation failed\n");
    exit(1);
  }
  newNode->data = data;
  newNode->left = NULL;
  newNode->right = NULL;
  return newNode;
}

// Function to traverse the syntax tree and perform arithmetic operations
```

```
int evaluate(TreeNode* root) {
  if (root == NULL) {
    return 0;
  }
  // If the node is an operand, return its value
  if (isdigit(root->data)) {
    return root->data - '0';
  }
  // Otherwise, perform the arithmetic operation based on the operator
  int leftValue = evaluate(root->left);
  int rightValue = evaluate(root->right);
  switch (root->data) {
    case '+':
      return leftValue + rightValue;
    case '-':
      return leftValue - rightValue;
    case '*':
      return leftValue * rightValue;
    case '/':
      if (rightValue == 0) {
        printf("Error: Division by zero\n");
        exit(1);
      }
      return leftValue / rightValue;
    default:
      printf("Error: Invalid operator\n");
      exit(1);
  }
}

// Function to build syntax tree from infix expression
TreeNode* buildSyntaxTree(char* expression) {
  // TODO: Implement expression parsing and syntax tree construction
  // For demonstration, let's assume a simple expression: "2 * (3 + 4)"
  TreeNode* root = createNode('*');
  root->left = createNode('2');
  root->right = createNode('+');
  root->right->left = createNode('3');
  root->right->right = createNode('4');
  return root;
}

int main() {
  char expression[100];
  printf("Enter an arithmetic expression: ");
  fgets(expression, sizeof(expression), stdin);
```

```
    // Remove trailing newline character if present
    if (expression[strlen(expression) - 1] == '\n') {
        expression[strlen(expression) - 1] = '\0';
    }

    // Build syntax tree from the input expression
    TreeNode* root = buildSyntaxTree(expression);

    // Evaluate the expression and print the result
    int result = evaluate(root);
    printf("Result: %d\n", result);

    // Free allocated memory
    // Free allocated memory
    free(root->left);
    free(root->right->left);
    free(root->right->right);
    free(root->right);
    free(root);

    return 0;
}
```

EXP 9:

```
#include <stdio.h>

int main() {
    int x = 5;
    int y = 10;

    // If statement
    if (x < y) {
        printf("x is less than y\n");
    } else {
        printf("x is greater than or equal to y\n");
    }

    // While loop
    int i = 0;
    while (i < 5) {
        printf("i: %d\n", i);
        i++;
    }

    return 0;
}
```

Intermediate Code:
1. x = 5
2. y = 10
3. if x < y goto 6
4. printf("x is greater than or equal to y\n")
5. goto 7
6. printf("x is less than y\n")
7. i = 0
8. if i < 5 goto 11
9. printf("i: %d\n", i)
10. i = i + 1
11. if i < 5 goto 9

EXP 10:

```
# MIPS Assembly code to add two numbers

.data
  num1: .word 5      # Define the first number
  num2: .word 10     # Define the second number
  result: .word 0    # Define a memory location to store the result

.text
  .globl main       # Entry point of the program

main:
  lw $t0, num1     # Load the first number into register $t0
  lw $t1, num2     # Load the second number into register $t1
  add $t2, $t0, $t1 # Add the two numbers and store the result in register $t2
  sw $t2, result    # Store the result in memory location 'result'

  # Exit the program
  li $v0, 10       # Load the system call code for exit (10)
  syscall          # Perform system call to exit
```