List of task

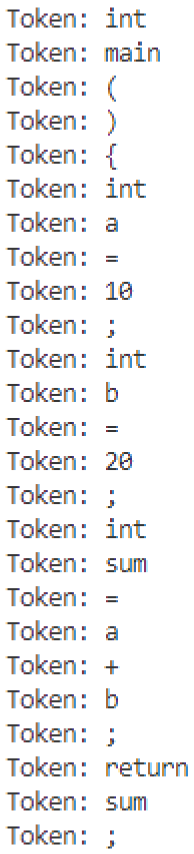
S.N	Task
1.	WAP to implement Lexical Analyzer to identify tokens.
2.	WAP to implement FIRST of grammar.
3.	WAP to implement FOLLOW of grammar.
4.	WAP to implement Shift Reduce Parser.
5.	WAP to implement LR Parser.
6.	WAP to implement Intermediate code generation
7.	WAP to implement Final code generation
8.	WAP to implement Type Conversion.
9.	WAP to check whether a given identifier is valid or not.
10.	WAP to check whether a given string is within valid comment section or not.

1. WAP to implement Lexical Analyzer to identify tokens.

```
#include <stdio.h>

#include <ctype.h>

#include <string.h>


#define MAX_TOKEN_LEN 100


typedef enum {

   KEYWORD,

   IDENTIFIER,

   NUMBER,

   OPERATOR,

   DELIMITER,

   UNKNOWN

} TokenType;


typedef struct {

   char value[MAX_TOKEN_LEN];

   TokenType type;

} Token;


const char *keywords[] = {

   "int", "float", "return", "if", "else", "while", "for", "do", "void", "char", "double",
"long",

   NULL

};


const char *operators[] = {

   "+", "-", "*", "/", "=", "==", "!=", "<", ">", "<=", ">=", "&&", "||", "!", "&", "|", "^",
"%",

   NULL

};
```

```
const char *delimiters[] = {
    "(", ")", "{", "}", "[", "]", ";", ",", NULL
};

int isKeyword(const char *str);
int isOperator(const char *str);
int isDelimiter(const char *str);
void addToken(Token tokens[], int *tokenCount, const char *value, TokenType type);

int main() {
    char input[] = "int main() { int a = 10; int b = 20; int sum = a+b ; return sum; }";
    Token tokens[100];
    int tokenCount = 0;

    int i = 0;
    while (i < strlen(input)) {
        if (isspace(input[i])) {
            i++;
            continue;
        }

        char buffer[MAX_TOKEN_LEN] = {0};
        int j = 0;

        if (isalpha(input[i])) {
            while (isalnum(input[i])) {
                buffer[j++] = input[i++];
            }
            buffer[j] = '\0';
            if (isKeyword(buffer)) {
                addToken(tokens, &tokenCount, buffer, KEYWORD);
            } else {
                addToken(tokens, &tokenCount, buffer, IDENTIFIER);
            }
        } else if (isdigit(input[i])) {
            while (isdigit(input[i]) || input[i] == '.') {
                buffer[j++] = input[i++];
            }
            buffer[j] = '\0';
            addToken(tokens, &tokenCount, buffer, NUMBER);
        } else if (isOperator((char[]){input[i], '\0'})) {
            buffer[j++] = input[i++];
            buffer[j] = '\0';
            addToken(tokens, &tokenCount, buffer, OPERATOR);
        } else if (isDelimiter((char[]){input[i], '\0'})) {
            buffer[j++] = input[i++];
            buffer[j] = '\0';
            addToken(tokens, &tokenCount, buffer, DELIMITER);
        } else {
            buffer[j++] = input[i++];
            buffer[j] = '\0';
            addToken(tokens, &tokenCount, buffer, UNKNOWN);
        }
    }
    for (i = 0; i < tokenCount; i++) {
        printf("Token: %-10s \n", tokens[i].value);
    }

    return 0;
}

int isKeyword(const char *str) {
    for (int i = 0; keywords[i] != NULL; i++) {
        if (strcmp(str, keywords[i]) == 0) {
            return 1;
        }
    }
    return 0;
}

int isOperator(const char *str) {
    for (int i = 0; operators[i] != NULL; i++) {
        if (strcmp(str, operators[i]) == 0) {
            return 1;
        }
    }
    return 0;
}

int isDelimiter(const char *str) {
    for (int i = 0; delimiters[i] != NULL; i++) {
        if (strcmp(str, delimiters[i]) == 0) {
            return 1;
        }
    }
    return 0;
}

void addToken(Token tokens[], int *tokenCount, const char *value, TokenType type) {
    strcpy(tokens[*tokenCount].value, value);
    (*tokenCount)++;
}
```

Output:

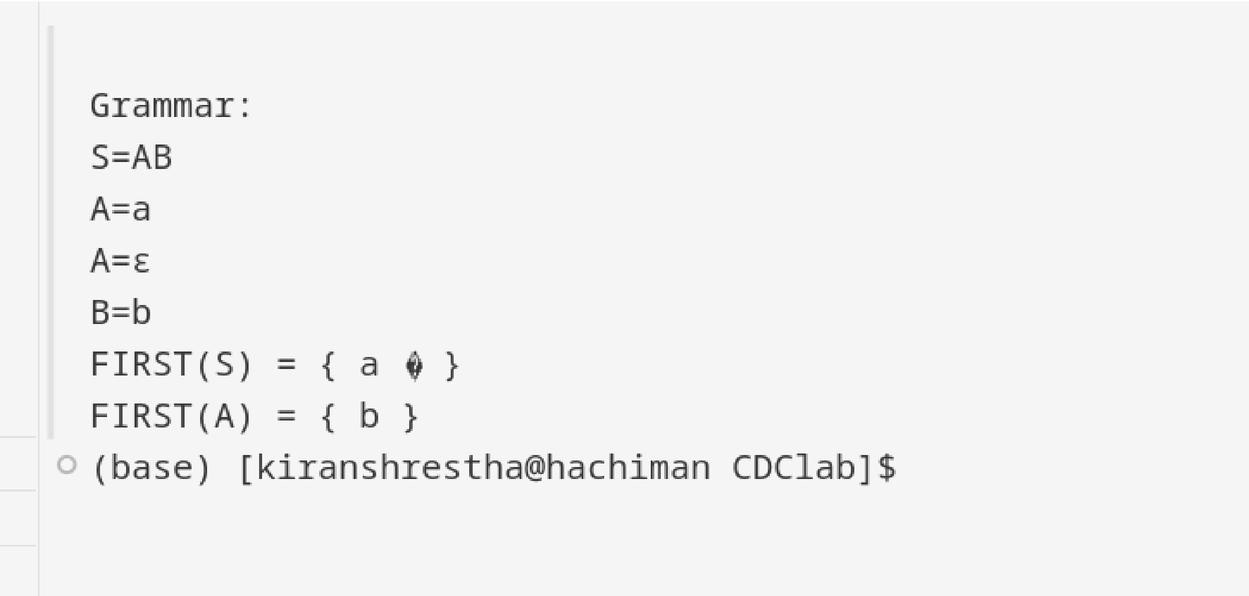


```
Token: int
Token: main
Token: (
Token: )
Token: {
Token: int
Token: a
Token: =
Token: 10
Token: ;
Token: int
Token: b
Token: =
Token: 20
Token: ;
Token: int
Token: sum
Token: =
Token: a
Token: +
Token: b
Token: ;
Token: return
Token: sum
Token: ;
```

2. WAP to implement FIRST of grammar.

```
#include <stdio.h>
#include <string.h>
#include <stdbool.h>

#define MAX 10

char production[MAX][MAX], first[MAX][MAX];
int n;

void findFirst(int, int);
void addToResultSet(char[], char);

int main() {
    int i, j;
    char result[MAX];

    strcpy(production[0], "S=AB");
    strcpy(production[1], "A=a");
    strcpy(production[2], "A=ε");
    strcpy(production[3], "B=b");
    n = 4;

    printf("Grammar:\n");
    for (i = 0; i < n; i++) {
        printf("%s\n", production[i]);
    }

    for (i = 0; i < n; i++) {
        first[i][0] = '\0';
    }
    for (i = 0; i < n; i++) {
        int nonTerminal = production[i][0] - 'A';
        findFirst(i, nonTerminal);
    }

    for (i = 0; i < n; i++) {
        if (first[i][0] != '\0') {
            printf("FIRST(%c) = { ", production[i][0]);
            for (j = 0; first[i][j] != '\0'; j++) {
                printf("%c ", first[i][j]);
            }
            printf("}\n");
        }
    }

    return 0;
}

void findFirst(int prodIndex, int nonTerminal) {
    int i, j;
    char result[MAX];

    for (i = 0; i < n; i++) {
        if (production[i][0] == (char)('A' + nonTerminal)) {
            if (production[i][2] == 'ε') {
                addToResultSet(first[nonTerminal], 'ε');
            } else {
                for (j = 2; production[i][j] != '\0'; j++) {
                    if (production[i][j] >= 'A' && production[i][j] <= 'Z') {
                        findFirst(i, production[i][j] - 'A');
                        if (!strchr(first[production[i][j] - 'A'], 'ε')) {
                            break;
                        }
                    } else {
                        addToResultSet(first[nonTerminal], production[i][j]);
                        break;
                    }
                }
            }
        }
    }
}

void addToResultSet(char result[], char c) {
    int i;
    for (i = 0; result[i] != '\0'; i++) {
        if (result[i] == c) {
            return;
        }
    }
    result[i] = c;
    result[i + 1] = '\0';
}
```

Output:



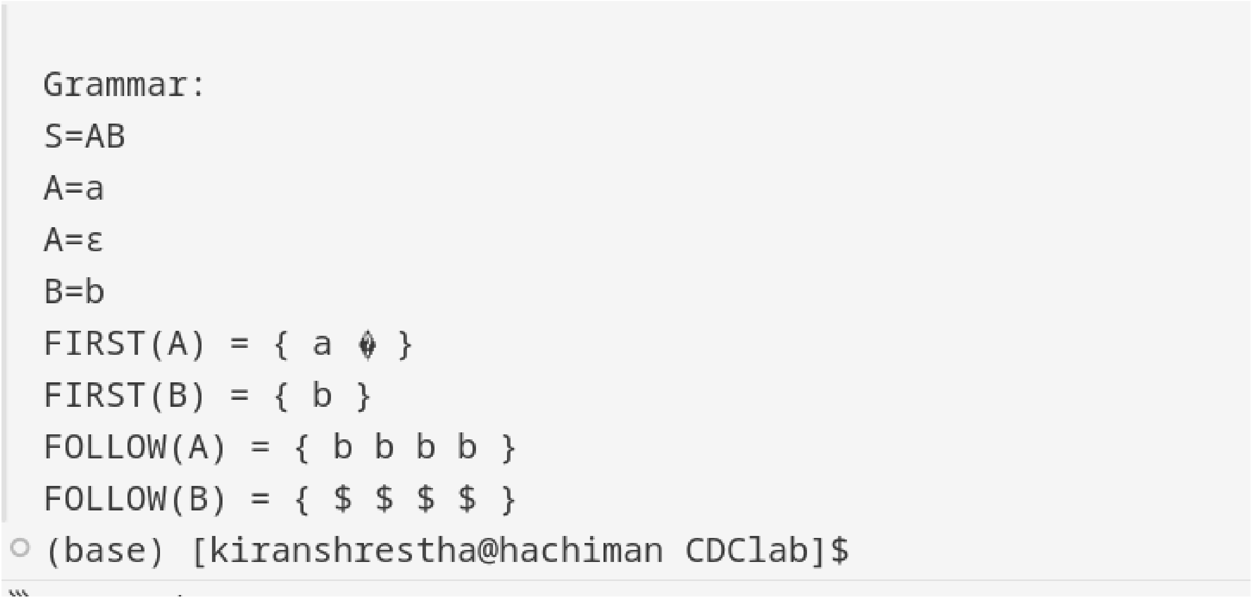
```
Grammar:
S=AB
A=a
A=ε
B=b
FIRST(S) = { a � }
FIRST(A) = { b }
○ (base) [kiranshrestha@hachiman CDClab]$
```

3. WAP to implement FOLLOW of grammar.

```
#include <stdio.h>

#include <string.h>

#include <stdbool.h>


#define MAX 10


char production[MAX][MAX], first[MAX][MAX], follow[MAX][MAX];

int n;

void findFirst(int, int);

void findFollow(int);

void addToResultSet(char[], char);

bool isNonTerminal(char);


int main() {

   int i, j;

   char result[MAX];


   strcpy(production[0], "S=AB");

   strcpy(production[1], "A=a");

   strcpy(production[2], "A=ε");

   strcpy(production[3], "B=b");

   n = 4;


   printf("Grammar:\n");

   for (i = 0; i < n; i++) {

      printf("%s\n", production[i]);

   }


   for (i = 0; i < MAX; i++) {

      first[i][0] = '\0';

      follow[i][0] = '\0';
   }

   for (i = 0; i < n; i++) {
      int nonTerminal = production[i][0] - 'A';
      findFirst(i, nonTerminal);
   }

   addToResultSet(follow['S' - 'A'], '$');

   for (i = 0; i < n; i++) {
      findFollow(i);
   }

   for (i = 0; i < MAX; i++) {
      if (first[i][0] != '\0') {
          printf("FIRST(%c) = { ", 'A' + i);
          for (j = 0; first[i][j] != '\0'; j++) {
              printf("%c ", first[i][j]);
          }
          printf("}\n");
      }
   }

   for (i = 0; i < MAX; i++) {
      if (follow[i][0] != '\0') {
          printf("FOLLOW(%c) = { ", 'A' + i);
          for (j = 0; follow[i][j] != '\0'; j++) {
              printf("%c ", follow[i][j]);
          }
          printf("}\n");
      }
   }
```

```
    return 0;
}

void findFirst(int prodIndex, int nonTerminal) {
    int i, j;
    char result[MAX];

    for (i = 0; i < n; i++) {
        if (production[i][0] == (char)('A' + nonTerminal)) {
            if (production[i][2] == 'ε') {
                addToResultSet(first[nonTerminal], 'ε');
            } else {
                for (j = 2; production[i][j] != '\0'; j++) {
                    if (isNonTerminal(production[i][j])) {
                        findFirst(i, production[i][j] - 'A');
                        if (!strchr(first[production[i][j] - 'A'], 'ε')) {
                            break;
                        }
                    } else {
                        addToResultSet(first[nonTerminal], production[i][j]);
                        break;
                    }
                }
            }
        }
    }
}

void findFollow(int prodIndex) {
    int i, j, k;
    char result[MAX];
    for (i = 0; i < n; i++) {
        for (j = 2; production[i][j] != '\0'; j++) {
            if (isNonTerminal(production[i][j])) {
                int nonTerminal = production[i][j] - 'A';

                for (k = j + 1; production[i][k] != '\0'; k++) {
                    if (isNonTerminal(production[i][k])) {
                        int nextNonTerminal = production[i][k] - 'A';
                        strcat(follow[nonTerminal], first[nextNonTerminal]);
                        if (!strchr(first[nextNonTerminal], 'ε')) {
                            break;
                        }
                    } else {
                        addToResultSet(follow[nonTerminal], production[i][k]);
                        break;
                    }
                }
                if (production[i][k] == '\0' && production[i][0] != production[i][j]) {
                    strcat(follow[nonTerminal], follow[production[i][0] - 'A']);
                }
            }
        }
    }
}

void addToResultSet(char result[], char c) {
    int i;
    for (i = 0; result[i] != '\0'; i++) {
        if (result[i] == c) {
            return;  // Avoid duplicates
        }
    }
    result[i] = c;
    result[i + 1] = '\0';
}

bool isNonTerminal(char c) {
    return c >= 'A' && c <= 'Z';
}
```

Output:

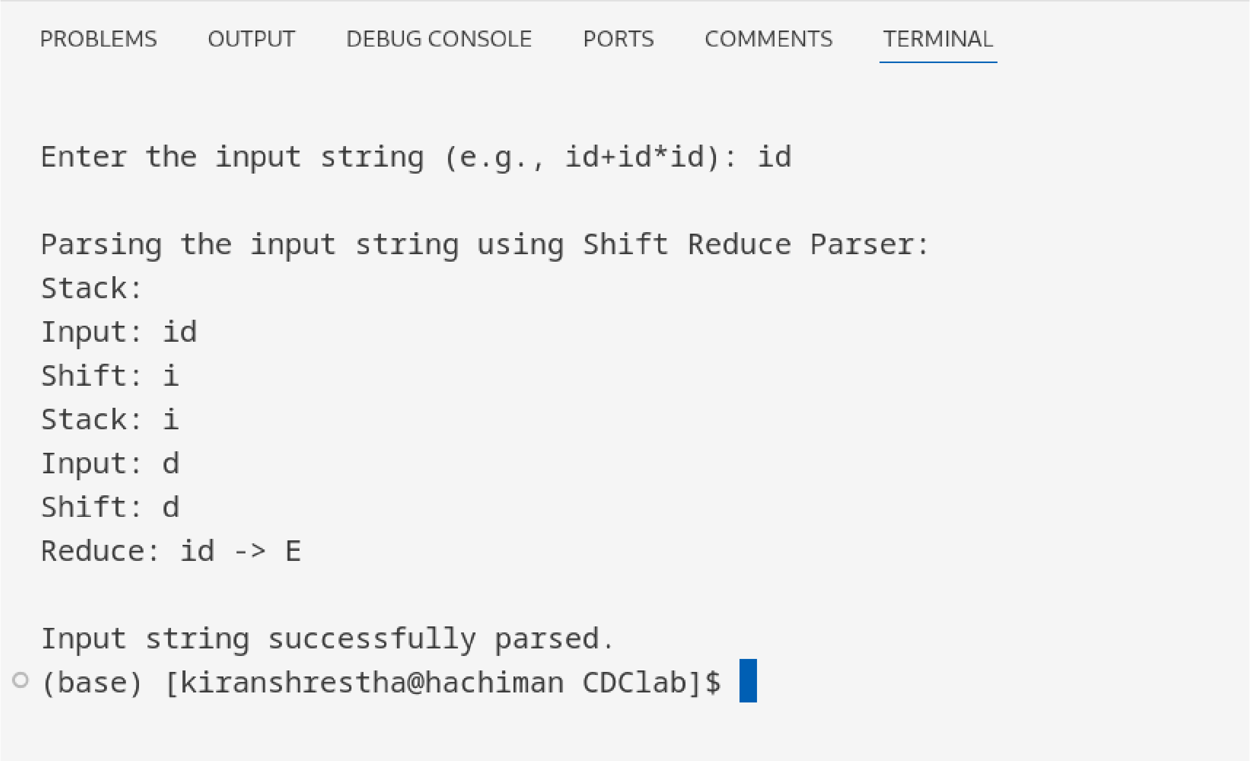


```
Grammar:
S=AB
A=a
A=ε
B=b
FIRST(A) = { a � }
FIRST(B) = { b }
FOLLOW(A) = { b b b b }
FOLLOW(B) = { $ $ $ $ }
○ (base) [kiranshrestha@hachiman CDClab]$
```

4. WAP to implement Shift Reduce Parser.

```
#include <stdio.h>

#include <string.h>

#include <stdlib.h>


#define MAX 100


char stack[MAX];

int top = -1;

char input[MAX];

int inputPointer = 0;


// Grammar rules

const char* rules[] = {

    "E+E",

    "E*E",

    "(E)",

    "id"

};


// Function prototypes

void push(char);

void pop();

void shift();

void reduce();

void displayStack();

void displayInput();


int main() {

    printf("Enter the input string (e.g., id+id*id): ");

    scanf("%s", input);
    printf("\nParsing the input string using Shift Reduce Parser:\n");
    while (1) {
        displayStack();
        displayInput();

        if (input[inputPointer] != '\0') {
            shift();
        }

        reduce();

        if (stack[0] == 'E' && top == 0 && input[inputPointer] == '\0') {
            printf("\nInput string successfully parsed.\n");
            break;
        }
    }

    return 0;
}

void push(char symbol) {
    if (top < MAX - 1) {
        stack[++top] = symbol;
    } else {
        printf("Stack overflow\n");
        exit(1);
    }
}

void pop() {
    if (top >= 0) {
        top--;
    } else {
        printf("Stack underflow\n");
        exit(1);
    }
}

void shift() {
    printf("Shift: %c\n", input[inputPointer]);
    push(input[inputPointer]);
    inputPointer++;
}

void reduce() {
    int i;
    for (i = 0; i < sizeof(rules) / sizeof(rules[0]); i++) {
        int len = strlen(rules[i]);
        if (top >= len - 1) {
            int match = 1;
            for (int j = 0; j < len; j++) {
                if (stack[top - len + 1 + j] != rules[i][j]) {
                    match = 0;
                    break;
                }
            }
            if (match) {
                printf("Reduce: %s -> E\n", rules[i]);
                for (int j = 0; j < len; j++) {
                    pop();
                }
                push('E');
                return;
            }
        }
    }
}

void displayStack() {
    printf("Stack: ");
    for (int i = 0; i <= top; i++) {
        printf("%c", stack[i]);
    }
    printf("\n");
}

void displayInput() {
    printf("Input: %s\n", input + inputPointer);
}
```

Output:



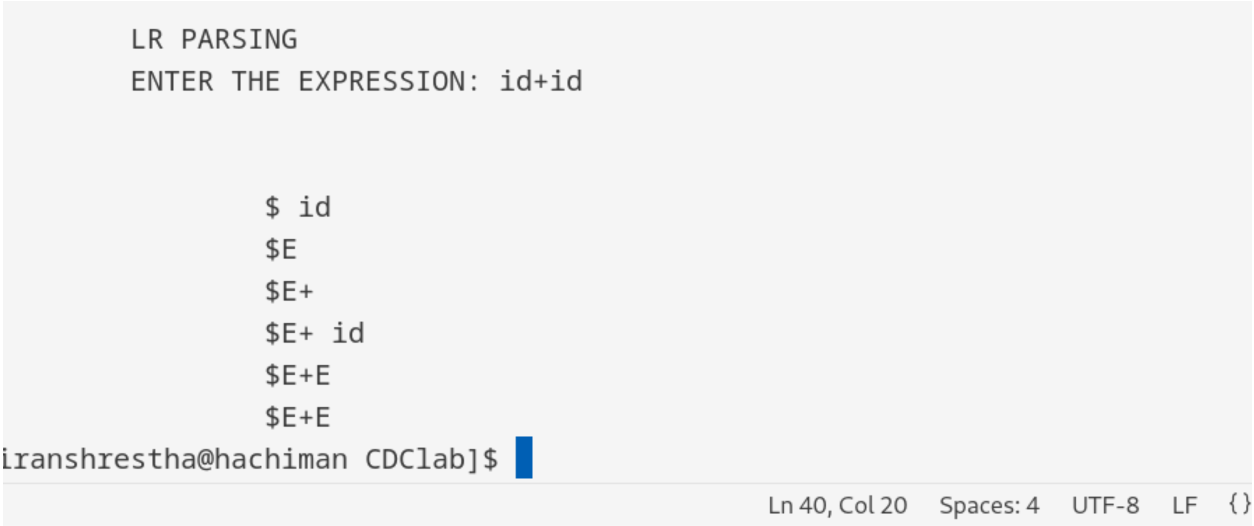
```
Enter the input string (e.g., id+id*id): id

Parsing the input string using Shift Reduce Parser:
Stack: 
Input: id
Shift: i
Stack: i
Input: d
Shift: d
Reduce: id -> E

Input string successfully parsed.
○ (base) [kiranshrestha@hachiman CDClab]$ 
```

5. WAP to implement LR Parser.

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

char stack[30];

int top = -1;

void push(char c) {

    if (top < 29) {

        top++;

        stack[top] = c;

    } else {

        printf("Stack overflow\n");

        exit(EXIT_FAILURE);

    }

}

char pop() {

    if (top != -1) {

        char c = stack[top];

        top--;

        return c;

    }

    return 'x'; // Return 'x' if stack is empty

}

void printstat() {

    printf("\n\t\t\t $");

    for (int i = 0; i <= top; i++) {

        printf("%c", stack[i]);

    }

}

int main() {

    char s1[20];
```

```
int l;
printf("\n\n\t\t LR PARSING");
printf("\n\t\t ENTER THE EXPRESSION: ");
scanf("%s", s1);
l = strlen(s1);
printf("\n\t\t $");
for (int i = 0; i < l; i++) {
    if (s1[i] == 'i' && s1[i + 1] == 'd') {
        s1[i] = ' ';
        s1[i + 1] = 'E';
        printstat();
        printf(" id");
        push('E');
        printstat();
        i++; // Skip next character
    } else if (s1[i] == '+' || s1[i] == '-' || s1[i] == '*' || s1[i] == '/' || s1[i] == 'd') {
        push(s1[i]);
        printstat();
    }
}
printstat();
while (top != -1) {
    char ch1 = pop();
    if (ch1 == 'x') {
        printf("\n\t\t\t $");
        break;
    }
    if (ch1 == '+' || ch1 == '/' || ch1 == '*' || ch1 == '-') {
        char ch3 = pop();
        if (ch3 != 'E') {
            printf("error\n");
            exit(EXIT_FAILURE);
            } else {
                push('E');
                printstat();
            }
        }
    }

    return 0;
}
```

Output:

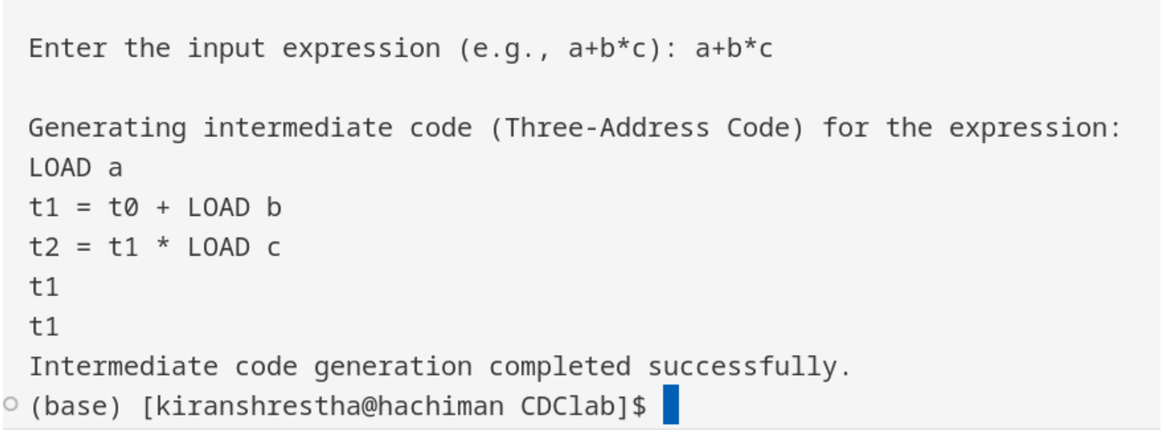


```
        LR PARSING
        ENTER THE EXPRESSION: id+id


                $ id
                $E
                $E+
                $E+ id
                $E+E
                $E+E
iranshrestha@hachiman CDClab]$ █
                                              Ln 40, Col 20   Spaces: 4   UTF-8   LF   {}
```

6. WAP to implement Intermediate code generation.

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

#define MAX 100

char input[MAX];
int inputPointer = 0;
int tempCount = 0;

// Function prototypes
void parse();
void expression();
void term();
void factor();
char* newTemp();

int main() {
    printf("Enter the input expression (e.g., a+b*c): ");
    scanf("%s", input);

    printf("\nGenerating intermediate code (Three-Address Code) for the expression:\n");
    parse();

    return 0;
}

void parse() {
    expression();
    if (input[inputPointer] == '\0') {
        printf("Intermediate code generation completed successfully.\n");
    } else {
        printf("Error: Unexpected input '%c'.\n", input[inputPointer]);
    }
}

void expression() {
    char* temp;
    term();
    while (input[inputPointer] == '+') {
        inputPointer++;
        temp = newTemp();
        printf("%s = t%d + ", temp, tempCount - 1);
        term();
        printf("t%d\n", tempCount - 1);
    }
}

void term() {
    char* temp;
    factor();
    while (input[inputPointer] == '*') {
        inputPointer++;
        temp = newTemp();
        printf("%s = t%d * ", temp, tempCount - 1);
        factor();
        printf("t%d\n", tempCount - 1);
    }
}

void factor() {
    if (input[inputPointer] >= 'a' && input[inputPointer] <= 'z') {
        printf("LOAD %c\n", input[inputPointer]);
        inputPointer++;
    } else if (input[inputPointer] == '(') {
        inputPointer++;
        expression();
        if (input[inputPointer] == ')') {
            inputPointer++;
        } else {
            printf("Error: Unmatched parenthesis\n");
            exit(1);
        }
    } else {
        printf("Error: Unexpected input '%c'\n", input[inputPointer]);
        exit(1);
    }
}

char* newTemp() {
    static char temp[10];
    snprintf(temp, sizeof(temp), "t%d", ++tempCount);
    return temp;
}
```

Output:

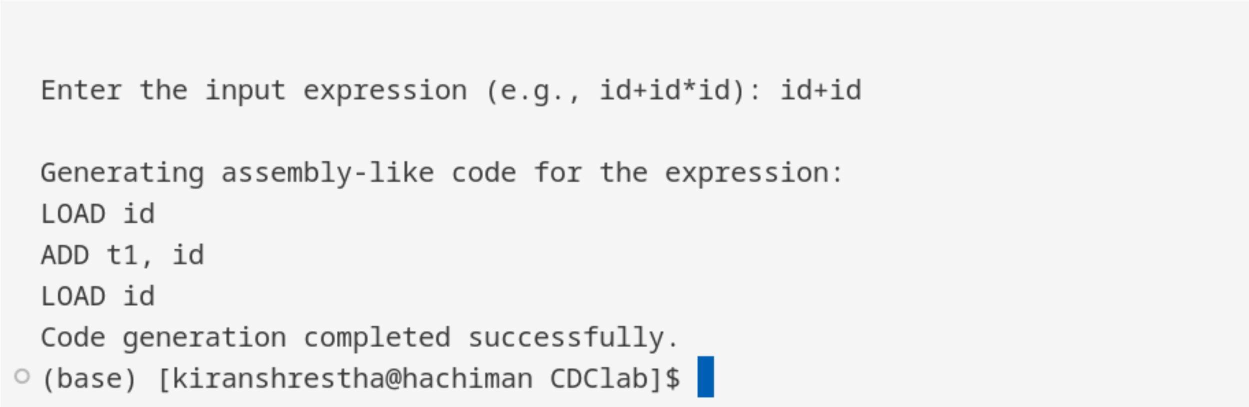


```
Enter the input expression (e.g., a+b*c): a+b*c

Generating intermediate code (Three-Address Code) for the expression:
LOAD a
t1 = t0 + LOAD b
t2 = t1 * LOAD c
t1
t1
Intermediate code generation completed successfully.
○ (base) [kiranshrestha@hachiman CDClab]$
```

7. WAP to implement Final code generation

```
#include <stdio.h>

#include <string.h>

char op[2], arg1[5], arg2[5], result[5];

int main() {

   FILE *fp1, *fp2;

   fp1 = fopen("input.txt", "r");

   if (fp1 == NULL) {

      perror("Error opening input file");

      return 1;

   }

   fp2 = fopen("output.txt", "w");

   if (fp2 == NULL) {

      perror("Error opening output file");

      fclose(fp1);

      return 1;

   }

   while (fscanf(fp1, "%s %s %s %s", op, arg1, arg2, result) == 4) {

      if (strcmp(op, "+") == 0) {

         fprintf(fp2, "\nMOV R0, %s", arg1);

         fprintf(fp2, "\nADD R0, %s", arg2);

         fprintf(fp2, "\nMOV %s, R0", result);

      } else if (strcmp(op, "*") == 0) {

         fprintf(fp2, "\nMOV R0, %s", arg1);

         fprintf(fp2, "\nMUL R0, %s", arg2);

         fprintf(fp2, "\nMOV %s, R0", result);

      } else if (strcmp(op, "-") == 0) {

         fprintf(fp2, "\nMOV R0, %s", arg1);

         fprintf(fp2, "\nSUB R0, %s", arg2);

         fprintf(fp2, "\nMOV %s, R0", result);

      } else if (strcmp(op, "/") == 0) {
            fprintf(fp2, "\nMOV R0, %s", arg1);
            fprintf(fp2, "\nDIV R0, %s", arg2);
            fprintf(fp2, "\nMOV %s, R0", result);
        } else if (strcmp(op, "=") == 0) {
            fprintf(fp2, "\nMOV R0, %s", arg1);
            fprintf(fp2, "\nMOV %s, R0", result);
        } else {
            fprintf(fp2, "\nUnknown operation: %s", op);
        }
    }
    fclose(fp1);
    fclose(fp2);
    return 0;
}
```

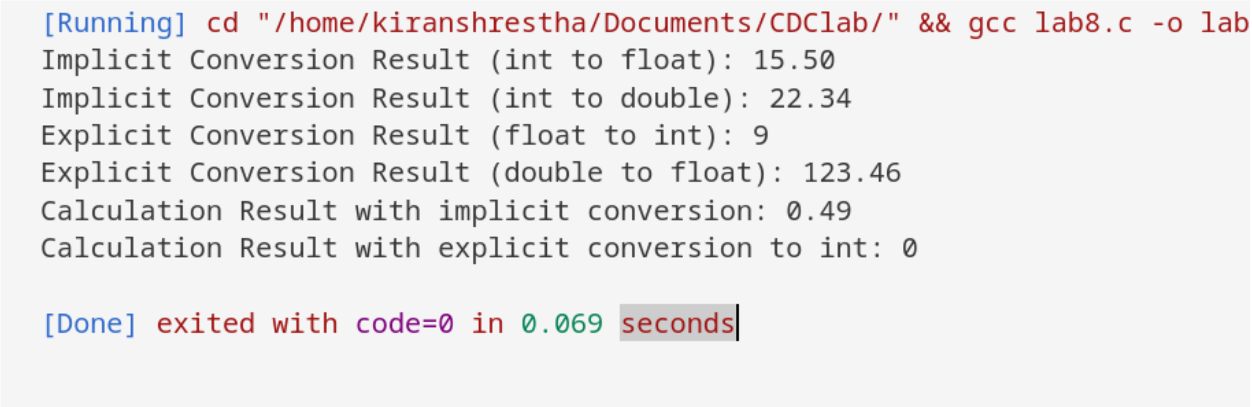
Output:



```
Enter the input expression (e.g., id+id*id): id+id

Generating assembly-like code for the expression:
LOAD id
ADD t1, id
LOAD id
Code generation completed successfully.
○ (base) [kiranshrestha@hachiman CDClab]$
```

8. WAP to implement Type Conversion.

```
#include <stdio.h>

int main() {
    // Implicit Conversion
    int intVar = 10;
    float floatVar = 5.5;
    double doubleVar = 12.34;
    // Implicit conversion: int to float
    float result1 = intVar + floatVar; // intVar is implicitly converted to float
    printf("Implicit Conversion Result (int to float): %.2f\n", result1);
    // Implicit conversion: int to double
    double result2 = intVar + doubleVar; // intVar is implicitly converted to double
    printf("Implicit Conversion Result (int to double): %.2f\n", result2);
    // Explicit Conversion (Type Casting)
    float floatVar2 = 9.99;
    int intVar2;
    // Explicit conversion: float to int
    intVar2 = (int)floatVar2; // floatVar2 is explicitly cast to int
    printf("Explicit Conversion Result (float to int): %d\n", intVar2);
    // Explicit conversion: double to float
    float floatVar3;
    double doubleVar2 = 123.456;
    floatVar3 = (float)doubleVar2; // doubleVar2 is explicitly cast to float
    printf("Explicit Conversion Result (double to float): %.2f\n", floatVar3);
    // Example of conversion in calculations
    int num1 = 10;
    double num2 = 20.5;
    // Result is of type double due to implicit conversion of num1 to double
    double result3 = num1 / num2;
    printf("Calculation Result with implicit conversion: %.2f\n", result3);
    // Result is explicitly cast to int
    int result4 = (int)(num1 / num2);
    printf("Calculation Result with explicit conversion to int: %d\n", result4);
    return 0;
}
```

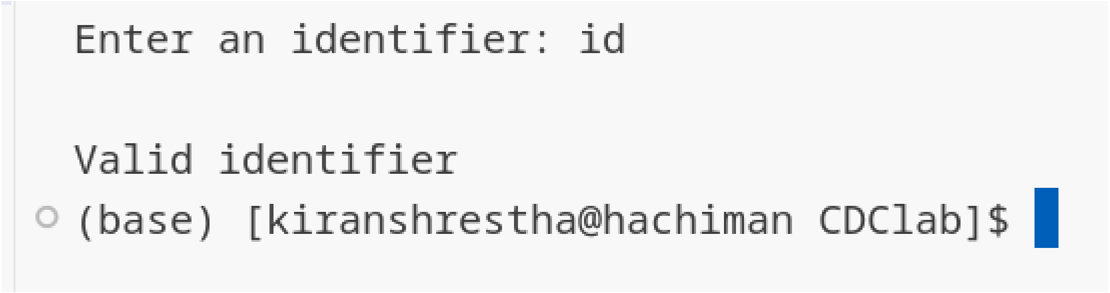


```
[Running] cd "/home/kiranshrestha/Documents/CDClab/" && gcc lab8.c -o lab
Implicit Conversion Result (int to float): 15.50
Implicit Conversion Result (int to double): 22.34
Explicit Conversion Result (float to int): 9
Explicit Conversion Result (double to float): 123.46
Calculation Result with implicit conversion: 0.49
Calculation Result with explicit conversion to int: 0

[Done] exited with code=0 in 0.069 seconds
```

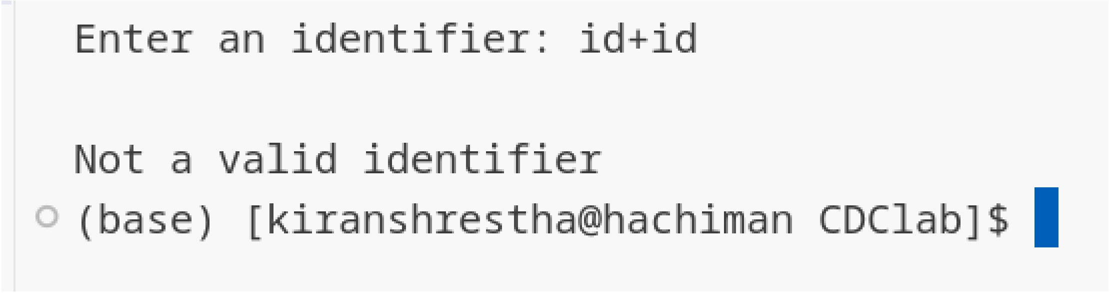
9. WAP to check whether a given identifier is valid or not.

```
#include <stdio.h>
#include <ctype.h>
int main() {
    char a[10];
    int flag = 0, i = 0;
    printf("\nEnter an identifier: ");
    fgets(a, sizeof(a), stdin);
    size_t len = strlen(a);
    if (len > 0 && a[len - 1] == '\n') {
        a[len - 1] = '\0';
    }
    // Check if the first character is valid
    if (isalpha(a[0]) || a[0] == '_') {
        flag = 1;
        while (a[i] != '\0') {
            if (!isdigit(a[i]) && !isalpha(a[i]) && a[i] != '_') {
                flag = 0;  // Invalid character found
                break;
            }
            i++;
        }
    } else {
        printf("\nNot a valid identifier");
        return 0; // Exit the program early if the first character is invalid
    }
    // Output result based on the flag
    if (flag == 1) {
        printf("\nValid identifier");
    } else {
        printf("\nNot a valid identifier");
    }
    return 0;
}
```

Output:



```
Enter an identifier: id

Valid identifier
○ (base) [kiranshrestha@hachiman CDClab]$ 
```



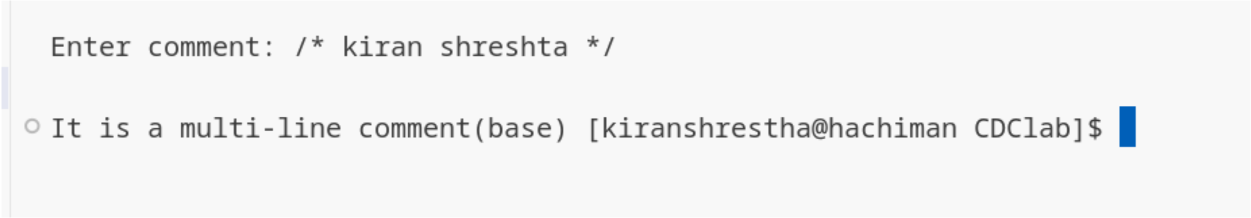
```
Enter an identifier: id+id

Not a valid identifier
○ (base) [kiranshrestha@hachiman CDClab]$ 
```

10. WAP to check whether a given string is within valid comment section or not.

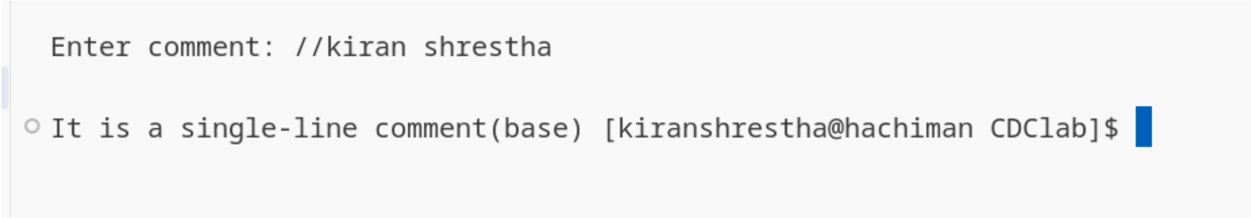
```
#include <stdio.h>
#include <string.h>
int main() {
   char com[30];
   int i;
   int isComment = 0;  // Flag to determine if the input is a comment
   printf("\nEnter comment: ");
   fgets(com, sizeof(com), stdin);
   // Remove newline character if fgets reads it
   size_t len = strlen(com);
   if (len > 0 && com[len - 1] == '\n') {
      com[len - 1] = '\0';
   }
   // Check for single-line comment
   if (com[0] == '/' && com[1] == '/') {
      printf("\nIt is a single-line comment");
      isComment = 1;
   }
   // Check for multi-line comment
   else if (com[0] == '/' && com[1] == '*') {
      for (i = 2; i < len - 1; i++) {
         if (com[i] == '*' && com[i + 1] == '/') {
            printf("\nIt is a multi-line comment");
            isComment = 1;
            break;
         }
      }
      if (isComment == 0) {
         printf("\nIt is not a multi-line comment");
      }
```

```
    } else {
        printf("\nIt is not a comment");
    }
    return 0;
}
```

Output:



```
Enter comment: /* kiran shreshta */
It is a multi-line comment(base) [kiranshrestha@hachiman CDClab]$
```



```
Enter comment: //kiran shrestha
It is a single-line comment(base) [kiranshrestha@hachiman CDClab]$
```