Experiment – 1

Title: Implementation of Deterministic Finite Automaton (DFA) from Regular Grammar using C

Aim:
To construct a DFA using a given regular grammar, and verify if a given string is accepted by the DFA by generating the transition table and computing the sequence of transitions.

Algorithm:

1. Define the set of states Q, the input alphabet T, the final states F, and the transitions Delta.
2. Read the production rules of the regular grammar.
3. From the rules, derive the transition table for the DFA.
4. Accept a string w and trace the transitions through the DFA.
5. If the final state after reading w is in F, accept the string; else, reject.

Source Code:

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

char s[10];
char st[10][10];
char t[10][10][10];
int f[10];
int ns, nc, nf, ss;

int fs(char* n) {
    for(int i = 0; i < ns; i++) {
        if(strcmp(st[i], n) == 0) {
            return i;
        }
    }
    return -1;
}

int fc(char c) {
    for(int i = 0; i < nc; i++) {
```

```
        if(s[i] == c) {
            return i;
        }
    }
    return -1;
}

int cf(int i) {
    for(int j = 0; j < nf; j++) {
        if(f[j] == i) {
            return 1;
        }
    }
    return 0;
}

void pt() {
    printf("\n   ");
    for(int j = 0; j < nc; j++) {
        printf("%c  ", s[j]);
    }
    printf("\n");
    for(int i = 0; i < ns; i++) {
        printf("%s ", st[i]);
        for(int j = 0; j < nc; j++) {
            printf("%s ", t[i][j]);
            if(strlen(t[i][j]) == 1) {
                printf(" ");
            }
        }
        printf("\n");
    }
}

int sim(char* in) {
    int cs = ss;
    int l = strlen(in);
    printf("\nString: %s\n", in);
    for(int i = 0; i < l; i++) {
        char cc = in[i];
        int si = fc(cc);
        if(si == -1) {
            printf("Symbol not found\n");
            return 0;
        }
```

```
        char* nn = t[cs][si];
        if(strcmp(nn, "-") == 0) {
            printf("δ[%s,%c] = REJECTED\n", st[cs], cc);
            return 0;
        }
        int ns = fs(nn);
        if(ns == -1) {
            printf("Invalid state\n");
            return 0;
        }
        printf("δ[%s,%c] = ", st[cs], cc);
        cs = ns;
    }
    printf("%s\n", st[cs]);
    if(cf(cs)) {
        printf("ACCEPTED\n");
        return 1;
    } else {
        printf("REJECTED\n");
        return 0;
    }
}

int main() {
    printf("Enter number of states: ");
    scanf("%d", &ns);
    printf("Enter number of symbols: ");
    scanf("%d", &nc);
    printf("Enter symbols: ");
    for(int i = 0; i < nc; i++) {
        scanf(" %c", &s[i]);
    }
    printf("Enter states: ");
    for(int i = 0; i < ns; i++) {
        scanf("%s", st[i]);
    }
    printf("Enter start state index: ");
    scanf("%d", &ss);
    printf("Enter number of final states: ");
    scanf("%d", &nf);
    printf("Enter final state indices: ");
    for(int i = 0; i < nf; i++) {
        scanf("%d", &f[i]);
    }
    for(int i = 0; i < ns; i++) {
```

```
        for(int j = 0; j < nc; j++) {
            strcpy(t[i][j], "-");
        }
    }
    printf("\nEnter transition function δ:\n");
    for(int i = 0; i < ns; i++) {
        for(int j = 0; j < nc; j++) {
            printf("δ[%s,%c] = ", st[i], s[j]);
            scanf("%s", t[i][j]);
        }
    }
    pt();
    char in[100];
    printf("\nEnter string: ");
    scanf("%s", in);
    sim(in);
    return 0;
}
```

Output:

```
student@AB1605B046:~/compiler$ gcc q1.c -o q1
student@AB1605B046:~/compiler$ ./q1
Enter number of states: 3
Enter number of symbols: 3
Enter symbols: 0 1 2
Enter states: q0 q1 q2
Enter start state index: 0
Enter number of final states: 1
Enter final state indices: 1

Enter transition function δ:
δ[q0,0] = q1
δ[q0,1] = -
δ[q0,2] = -
δ[q1,0] = -
δ[q1,1] = q2
δ[q1,2] = -
δ[q2,0] = -
δ[q2,1] = -
δ[q2,2] = q1

   0  1  2
q0 q1 -  -
q1 -  q2 -
q2 -  -  q1

Enter string: 01212

String: 01212
δ[q0,0] = δ[q1,1] = δ[q2,2] = δ[q1,1] = δ[q2,2] = q1
ACCEPTED
```

```
student@AB1605B046:~/compiler$ ./q1
Enter number of states: 3
Enter number of symbols: 3
Enter symbols: 0 1 2
Enter states: q0 q1 q2
Enter start state index: 0
Enter number of final states: 1
Enter final state indices: 1

Enter transition function δ:
δ[q0,0] = q1
δ[q0,1] = -
δ[q0,2] = -
δ[q1,0] = -
δ[q1,1] = q2
δ[q0,0] = q1
δ[q0,1] = -
δ[q0,2] = -
δ[q1,0] = -
δ[q1,1] = q2
δ[q1,2] = -
δ[q2,0] = -
δ[q2,1] = -
δ[q2,2] = q1

   0  1  2
q0 q1 -  -
q1 -  q2 -
q2 -  -  q1

Enter string: 01212

String: 01212
δ[q0,0] = δ[q1,1] = δ[q2,2] = δ[q1,1] = δ[q2,2] = q1
ACCEPTED
```

```
student@AB1605B046:~/compiler$ ./q1
Enter number of states: 3
Enter number of symbols: 3
Enter symbols: 0 1 2
Enter states: q0 q1 q2
Enter start state index: 0
Enter number of final states: 1
Enter final state indices: 1

Enter transition function δ:
δ[q0,0] = q1
δ[q0,1] = -
δ[q0,2] = -
δ[q1,0] = -
δ[q1,1] = q2
δ[q1,2] = -
δ[q2,0] = -
δ[q2,1] = -
δ[q2,2] = q1

   0  1  2
q0 q1 -  -
q1 -  q2 -
q2 -  -  q1

Enter string: 0121211

String: 0121211
δ[q0,0] = δ[q1,1] = δ[q2,2] = δ[q1,1] = δ[q2,2] = δ[q1,1] = δ[q2,1] = REJECTED
```

```
student@AB1605B046:~/compiler$ ./q1
Enter number of states: 5
Enter number of symbols: 4
Enter symbols: a b c d
Enter states: A B C D E
Enter start state index: 0
Enter number of final states: 1
Enter final state indices: 4

Enter transition function δ:
δ[A,a] = B
δ[A,b] = -
δ[A,c] = -
δ[A,d] = -
δ[B,a] = B
δ[B,b] = C
δ[B,c] = -
δ[B,d] = -
δ[C,a] = -
δ[C,b] = C
δ[C,c] = -
δ[C,d] = D
δ[D,a] = -
δ[D,b] = -
δ[D,c] = D
δ[D,d] = E
δ[E,a] = -
δ[E,b] = -
δ[E,c] = -
δ[E,d] = E
δ[D,c] = D
δ[D,d] = E
δ[E,a] = -
δ[E,b] = -
δ[E,c] = -
δ[E,d] = E

   a  b  c  d
A B  -  -  -
B B  C  -  -
C -  C  -  D
D -  -  D  E
E -  -  -  E

Enter string: aaaaaaaabbbbbbbbbbbbdccccccccccddddddddddddd

String: aaaaaaaabbbbbbbbbbbbdccccccccccddddddddddddd
δ[A,a] = δ[B,a] = δ[B,a] = δ[B,a] = δ[B,a] = δ[B,a] = δ[B,a] = δ[B,a] = δ[B,b] = δ[C,b] = δ[C,b] = δ[C,b] = δ[C,b] = δ[C,b] = δ[C,b] = δ[C,b] = δ[C,b] = δ[C,b] = δ[C,b] = δ[C,b] = δ[C,d] = δ[D,c] = δ[D,c] = δ[D
,c] = δ[D,c] = δ[D,c] = δ[D,c] = δ[D,c] = δ[D,c] = δ[D,c] = δ[D,c] = δ[D,d] = δ[E,d] = δ[E,d] = δ[E,d] = δ[E,d] = δ[E,d] = δ[E,d] = δ[E,d] = δ[E,d] = δ[E,d] = δ[E,d] = δ[E,d] = E
ACCEPTED
```

Experiment – 2

Title: Derivation of Regular Grammar from Deterministic Finite Automaton (DFA) using C

Aim:
To generate a regular grammar G = (N, T, P, S) corresponding to a given DFA.

Algorithm:

1. Define the DFA with its components: Q (states), T (input symbols), F (final states), and transition table.
2. For each transition from state A to state B on symbol x, add production A → xB.
3. If B is a final state, also add production A → x.
4. Define the start symbol S as the start state of the DFA.
5. Output all grammar rules.

Source Code:

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

char s[10];
char q[10][10];
char t[10][10][10];
int f[10];
int nq, ns, nf, start;

int findQ(char* n) {
    for(int i = 0; i < nq; i++) {
        if(strcmp(q[i], n) == 0) {
            return i;
        }
    }
    return -1;
}

int findS(char c) {
    for(int i = 0; i < ns; i++) {
        if(s[i] == c) {
            return i;
        }
    }
    return -1;
```

```c
}

int isFinal(int i) {
    for(int j = 0; j < nf; j++) {
        if(f[j] == i) {
            return 1;
        }
    }
    return 0;
}

void printTable() {
    printf("\nTransition Table:\n");
    printf("   ");
    for(int j = 0; j < ns; j++) {
        printf("%c  ", s[j]);
    }
    printf("\n");

    for(int i = 0; i < nq; i++) {
        printf("%s ", q[i]);
        for(int j = 0; j < ns; j++) {
            printf("%s ", t[i][j]);
            if(strlen(t[i][j]) == 1) {
                printf(" ");
            }
        }
        printf("\n");
    }
}

void genGrammar() {
    printf("\n=== REGULAR GRAMMAR FROM DFA ===\n");

    printf("\nNon-terminals (N): {");
    for(int i = 0; i < nq; i++) {
        printf("%s", q[i]);
        if(i < nq - 1) printf(", ");
    }
    printf("}\n");

    printf("Terminals (T): {");
    for(int i = 0; i < ns; i++) {
        printf("%c", s[i]);
        if(i < ns - 1) printf(", ");
```

```
    }
    printf("}\n");

    printf("Start Symbol (S): %s\n", q[start]);

    printf("\nProductions (P):\n");

    for(int i = 0; i < nq; i++) {
        for(int j = 0; j < ns; j++) {
            char* next = t[i][j];

            if(strcmp(next, "-") != 0) {
                int ni = findQ(next);

                printf("  %s -> %c%s\n", q[i], s[j], next);

                if(isFinal(ni)) {
                    printf("  %s -> %c\n", q[i], s[j]);
                }
            }
        }

        if(isFinal(i)) {
            printf("  %s -> ε\n", q[i]);
        }
    }

    printf("\nGrammar G = (N, T, P, S) where:\n");
    printf("  N = Non-terminals\n");
    printf("  T = Terminals\n");
    printf("  P = Productions\n");
    printf("  S = Start Symbol\n");
}

int sim(char* str) {
    int cur = start;
    int len = strlen(str);

    printf("\n=== STRING SIMULATION ===\n");
    printf("String: %s\n", str);

    for(int i = 0; i < len; i++) {
        char c = str[i];
        int si = findS(c);
```

```
        if(si == -1) {
            printf("Symbol '%c' not found in alphabet\n", c);
            return 0;
        }

        char* next = t[cur][si];

        if(strcmp(next, "-") == 0) {
            printf("δ[%s,%c] = REJECTED (no transition)\n", q[cur], c);
            return 0;
        }

        int ni = findQ(next);

        if(ni == -1) {
            printf("Invalid state: %s\n", next);
            return 0;
        }

        printf("δ[%s,%c] = %s\n", q[cur], c, next);
        cur = ni;
    }

    printf("Final state: %s\n", q[cur]);

    if(isFinal(cur)) {
        printf("Result: ACCEPTED\n");
        return 1;
    } else {
        printf("Result: REJECTED\n");
        return 0;
    }
}

int main() {
    printf("=== DFA TO REGULAR GRAMMAR CONVERTER ===\n\n");

    printf("Enter number of states: ");
    scanf("%d", &nq);

    printf("Enter number of symbols: ");
    scanf("%d", &ns);

    printf("Enter symbols: ");
    for(int i = 0; i < ns; i++) {
```

```
        scanf(" %c", &s[i]);
    }

    printf("Enter states: ");
    for(int i = 0; i < nq; i++) {
        scanf("%s", q[i]);
    }

    printf("Enter start state index (0 to %d): ", nq-1);
    scanf("%d", &start);

    printf("Enter number of final states: ");
    scanf("%d", &nf);

    printf("Enter final state indices: ");
    for(int i = 0; i < nf; i++) {
        scanf("%d", &f[i]);
    }

    for(int i = 0; i < nq; i++) {
        for(int j = 0; j < ns; j++) {
            strcpy(t[i][j], "-");
        }
    }

    printf("\nEnter transition function δ:\n");
    printf("(Enter '-' for no transition)\n");
    for(int i = 0; i < nq; i++) {
        for(int j = 0; j < ns; j++) {
            printf("δ[%s,%c] = ", q[i], s[j]);
            scanf("%s", t[i][j]);
        }
    }

    printTable();
    genGrammar();

    char ch;
    char input[100];
    printf("\nEnter string to test: ");
    scanf("%s", input);
    sim(input);

    return 0;
}
```

Output:

```
student@AB1605B046:~/compiler$ gcc q2.c -o q2
student@AB1605B046:~/compiler$ ./q2
=== DFA TO REGULAR GRAMMAR CONVERTER ===

Enter number of states: 3
Enter number of symbols: 2
Enter symbols: 0 1
Enter states: A B C
Enter start state index (0 to 2): 0
Enter number of final states: 1
Enter final state indices: 2

Enter transition function δ:
(Enter '-' for no transition)
δ[A,0] = B
δ[A,1] = -
δ[B,0] = B
δ[B,1] = C
δ[C,0] = -
δ[C,1] = C

Transition Table:
   0  1
A B  -
B B  C
C -  C

=== REGULAR GRAMMAR FROM DFA ===

Non-terminals (N): {A, B, C}
Terminals (T): {0, 1}
Start Symbol (S): A
```

```
=== REGULAR GRAMMAR FROM DFA ===

Non-terminals (N): {A, B, C}
Terminals (T): {0, 1}
Start Symbol (S): A

Productions (P):
  A -> 0B
  B -> 0B
  B -> 0B
  B -> 1C
  B -> 1
  C -> 1C
  B -> 0B
  B -> 1C
  B -> 1
  B -> 0B
  B -> 1C
  B -> 0B
  B -> 1C
  B -> 1C
  B -> 1
  B -> 1
  C -> 1C
  C -> 1C
  C -> 1
  C -> ε
  C -> ε


Grammar G = (N, T, P, S) where:
Grammar G = (N, T, P, S) where:
  N = Non-terminals
  T = Terminals
  P = Productions
  S = Start Symbol

Enter string to test: 00000111

=== STRING SIMULATION ===
String: 00000111
δ[A,0] = B
δ[B,0] = B
δ[B,0] = B
δ[B,0] = B
δ[B,0] = B
δ[B,1] = C
δ[C,1] = C
δ[C,1] = C
Final state: C
Result: ACCEPTED
```

```
=== DFA TO REGULAR GRAMMAR CONVERTER ===

Enter number of states: 4
Enter number of symbols: 3
Enter symbols: 0 1 2
Enter states: q0 q1 q2 q3
Enter start state index (0 to 3): 0
Enter number of final states: 1
Enter final state indices: 1

Enter transition function δ:
(Enter '-' for no transition)
δ[q0,0] = q1
δ[q0,1] = -
δ[q0,2] = -
δ[q1,0] = -
δ[q1,1] = q2
δ[q1,2] = q2
δ[q2,0] = -
δ[q2,1] = q3
δ[q2,2] = q3
δ[q3,0] = -
δ[q3,1] = q1
δ[q3,2] = q1

Transition Table:
   0  1  2
q0 q1 -  -
q1 -  q2 q2
q2 -  q3 q3
q3 -  q1 q1
```

```
=== REGULAR GRAMMAR FROM DFA ===

Non-terminals (N): {q0, q1, q2, q3}
Terminals (T): {0, 1, 2}
Start Symbol (S): q0

Productions (P):
  q0 -> 0q1
  q0 -> 0
  q1 -> 1q2
  q1 -> 2q2
  q1 -> ε
  q2 -> 1q3
  q2 -> 2q3
  q3 -> 1q1
  q3 -> 1
  q3 -> 2q1
  q3 -> 2

Grammar G = (N, T, P, S) where:
  N = Non-terminals
  T = Terminals
  P = Productions
  S = Start Symbol

Enter string to test: 0111

=== STRING SIMULATION ===
String: 0111
δ[q0,0] = q1
δ[q1,1] = q2
δ[q2,1] = q3
δ[q3,1] = q1
Final state: q1
Result: ACCEPTED
```

Experiment – 3

Title: Conversion of Non-Deterministic Finite Automaton (NFA) to Deterministic Finite Automaton (DFA) without ε-edges using C

Aim:
To convert a given NFA (without epsilon transitions) into an equivalent DFA using subset construction.

Algorithm:

1. Read the NFA components: Q, T, Delta, and F.
2. Start with the epsilon closure of the initial state.
3. For each set of NFA states and each input symbol, compute the resulting set of NFA states.
4. Each such resulting set represents a DFA state.
5. Mark those sets that contain any final state of the NFA as final states of the DFA.
6. Continue the process until no new DFA states are created.
7. Construct and print the DFA transition table.

Source Code:

```c
#include <stdio.h>
#include <string.h>
#include <stdbool.h>
#define N 20

int n, a, s, nf;
char st[N][10], sym[N][10];
int f[N], t[N][N][N], tc[N][N];
int dt[N][N];
char dset[N][N];
bool df[N];
int ds = 0;

void sort(char *x) {
    int l = strlen(x);
    for (int i = 0; i < l - 1; i++)
        for (int j = i + 1; j < l; j++)
            if (x[i] > x[j]) {
                char tmp = x[i];
                x[i] = x[j];
```

```
                x[j] = tmp;
            }
}

int find(char *x) {
    for (int i = 0; i < ds; i++)
        if (strcmp(dset[i], x) == 0)
            return i;
    return -1;
}

void add(char *x) {
    strcpy(dset[ds], x);
    df[ds] = false;
    for (int i = 0; i < strlen(x); i++)
        if (f[x[i] - '0']) {
            df[ds] = true;
            break;
        }
    ds++;
}

void convert() {
    char q[N][N], tmp[2];
    int fr = 0, re = 0;
    tmp[0] = s + '0';
    tmp[1] = '\0';
    sort(tmp);
    add(tmp);
    strcpy(q[re++], tmp);
    while (fr < re) {
        char cur[N];
        strcpy(cur, q[fr++]);
        int ci = find(cur);
        for (int i = 0; i < a; i++) {
            char nx[N] = "";
            for (int j = 0; j < strlen(cur); j++) {
                int idx = cur[j] - '0';
                for (int k = 0; k < tc[idx][i]; k++) {
                    char c = t[idx][i][k] + '0';
                    if (!strchr(nx, c)) {
                        int l = strlen(nx);
                        nx[l] = c;
                        nx[l + 1] = '\0';
                    }
```

```
                }
            }
            sort(nx);
            if (strlen(nx) == 0) {
                dt[ci][i] = -1;
            } else {
                int ex = find(nx);
                if (ex == -1) {
                    add(nx);
                    strcpy(q[re++], nx);
                    dt[ci][i] = ds - 1;
                } else {
                    dt[ci][i] = ex;
                }
            }
        }
    }
}

int getidx(char *x) {
    for (int i = 0; i < n; i++)
        if (strcmp(x, st[i]) == 0) return i;
    return -1;
}

int main() {
    printf("Enter number of states: ");
    scanf("%d", &n);
    printf("Enter number of symbols: ");
    scanf("%d", &a);
    printf("Enter symbols: ");
    for (int i = 0; i < a; i++) scanf("%s", sym[i]);
    printf("Enter states: ");
    for (int i = 0; i < n; i++) scanf("%s", st[i]);
    printf("Enter start state index: ");
    scanf("%d", &s);
    printf("Enter number of final states: ");
    scanf("%d", &nf);
    printf("Enter final state indices: ");
    for (int i = 0; i < nf; i++) {
        int x;
        scanf("%d", &x);
        f[x] = 1;
    }
```

```
    printf("\nEnter transition function δ:\n");
    for (int i = 0; i < n; i++)
        for (int j = 0; j < a; j++) {
            printf("δ[%s,%s] = ", st[i], sym[j]);
            char v[100];
            scanf("%s", v);
            if (strcmp(v, "-") != 0) {
                char *tok = strtok(v, ",");
                while (tok != NULL) {
                    int k = getidx(tok);
                    if (k != -1) t[i][j][tc[i][j]++] = k;
                    tok = strtok(NULL, ",");
                }
            }
        }

    convert();

    printf("\nDFA Transition Table:\n    ");
    for (int i = 0; i < a; i++) printf("%s ", sym[i]);
    printf("\n");
    for (int i = 0; i < ds; i++) {
        printf("q%d (%s)%s ", i, dset[i], df[i] ? "*" : " ");
        for (int j = 0; j < a; j++) {
            if (dt[i][j] == -1) printf("- ");
            else printf("q%d ", dt[i][j]);
        }
        printf("\n");
    }

    char in[N];
    printf("\nEnter string: ");
    scanf("%s", in);

    printf("\nString: %s\n", in);
    int cur = 0;
    for (int i = 0; i < strlen(in); i++) {
        int si = -1;
        for (int j = 0; j < a; j++) {
            if (in[i] == sym[j][0]) {
                si = j;
                break;
            }
        }
        if (si == -1 || dt[cur][si] == -1) {
```

```
            printf("δ[q%d,%c] = undefined\nREJECTED\n", cur, in[i]);
            return 0;
        } else {
            printf("δ[q%d,%c] = ", cur, in[i]);
            cur = dt[cur][si];
        }
    }
    printf("q%d\n", cur);
    if (df[cur]) printf("ACCEPTED\n");
    else printf("REJECTED\n");

    return 0;
}
```

Output:

```
student@AB1605B046:~/compiler$ gcc q3.c -o q3f
student@AB1605B046:~/compiler$ ./q3f
Enter number of states: 3
Enter number of symbols: 2
Enter symbols: a b
Enter states: q0 q1 q2
Enter start state index: 0
Enter number of final states: 1
Enter final state indices: 2

Enter transition function δ:
δ[q0,a] = q0,q1
δ[q0,b] = -
δ[q1,a] = -
δ[q1,b] = q1,q2
δ[q2,a] = -
δ[q2,b] = -

DFA Transition Table:
    a b
q0 (0)  q1 -
q1 (01)  q1 q2
q2 (12)* - q2

Enter string: aaaabb

String: aaaabb
δ[q0,a] = δ[q1,a] = δ[q1,a] = δ[q1,a] = δ[q1,b] = δ[q2,b] = q2
ACCEPTED
```

```
student@AB1605B046:~/compiler$ ./q3f
Enter number of states: 6
Enter number of symbols: 1
Enter symbols: a
Enter states: q0 q1 q2 q3 q4 q5
Enter start state index: 0
Enter number of final states: 2
Enter final state indices: 3 5

Enter transition function δ:
δ[q0,a] = q1,q4
δ[q1,a] = q2
δ[q2,a] = q3
δ[q3,a] = -
δ[q4,a] = q5
δ[q5,a] = q4

DFA Transition Table:
    a
q0 (0)  q1
q1 (14)  q2
q2 (25)* q3
q3 (34)* q4
q4 (5)* q5
q5 (4)  q4

Enter string: aaa

String: aaa
δ[q0,a] = δ[q1,a] = δ[q2,a] = q3
ACCEPTED
```

Experiment – 4(a)

Title: LEX implementation of DFA accepting odd number of a's and even number of b's

Aim:
To write a LEX program that accepts strings containing an odd number of as and an even number of bs using a DFA logic.

Algorithm:

1. Define DFA states for tracking the parity of a's and b's.
2. Track transitions using a 2-bit state (one for a, one for b).
3. Accept only if the final state indicates odd as and even bs.

Source Code:

```
%{
#include <stdio.h>
int state = 0;
%}

%%
a       {
    if(state == 0) state = 1;
    else if(state == 1) state = 0;
    else if(state == 2) state = 3;
    else if(state == 3) state = 2;
}

b       {
    if(state == 0) state = 2;
    else if(state == 1) state = 3;
    else if(state == 2) state = 0;
    else if(state == 3) state = 1;
}

\n      {
    if(state == 1) {
        printf("ACCEPTED\n");
    } else {
        printf("REJECTED\n");
    }
    state = 0;
}
```

```
%%

int main() {
    printf("Enter strings:\n");
    yylex();
    return 0;
}

int yywrap() {
    return 1;
}
```

Output:

```
student@AB1605B046:~/compiler$ lex q1.l
gcc lex.yy.c
./a.out
Enter strings:
aaabb
ACCEPTED
aabb
REJECTED
aaab
REJECTED
```

Experiment – 4(b)

Title: LEX implementation of DFA accepting strings over {a, b, c} having "bca" as a substring

Aim:
To write a LEX program that accepts strings containing "bca" as a substring using DFA logic.

Algorithm:

1. Define states tracking the sequence b, then bc, then bca.
2. Use transitions based on characters a, b, c.
3. Once the bca substring is found, move to an accepting state.
4. Stay in the accepting state regardless of further input.

Source Code:

```
b)
%{
#include <stdio.h>
int state = 0;
%}

%%
a       {
    if(state == 2) state = 3;
    else state = 0;
}

b       {
    if(state == 0 || state == 1 || state == 2) state = 1;
    else if(state == 3) state = 1;
}

c       {
    if(state == 1) state = 2;
    else if(state == 3) state = 0;
    else state = 0;
}

\n      {
    if(state == 3) {
        printf("ACCEPTED\n");
    } else {
        printf("REJECTED\n");
```

```
    }
    state = 0;
}

.       {
    state = 0;
}

%%

int main() {
    yylex();
    return 0;
}

int yywrap() {
    return 1;
}
```

Output:

```
student@AB1605B046:~/compiler$ lex q2.l
gcc lex.yy.c
./a.out
aabca
ACCEPTED
aabb
REJECTED
```

Experiment -5(a):

Title:

Identify tokens from a simple statement stored in a linear array

Aim of the experiment:

To develop a lexical analyser that identifies and categorizes tokens (keywords, identifiers, constants, strings, operators, and punctuators) from a simple C statement provided as user input.

Algorithm:

1. Read input string from user
2. Initialize index pointer to start of string 3. While not at end of string:
   - Skip whitespace characters
   - If character is double quote, process as string literal
   - If character is alphabetic or underscore, process as identifier/keyword
   - If character is digit or decimal point, process as numeric constant
   - If character is operator symbol, process as operator o If character is punctuation, process as punctuator o Otherwise, move to next character
4. For each token type, implement specific extraction logic
5. Print categorized tokens with appropriate labels

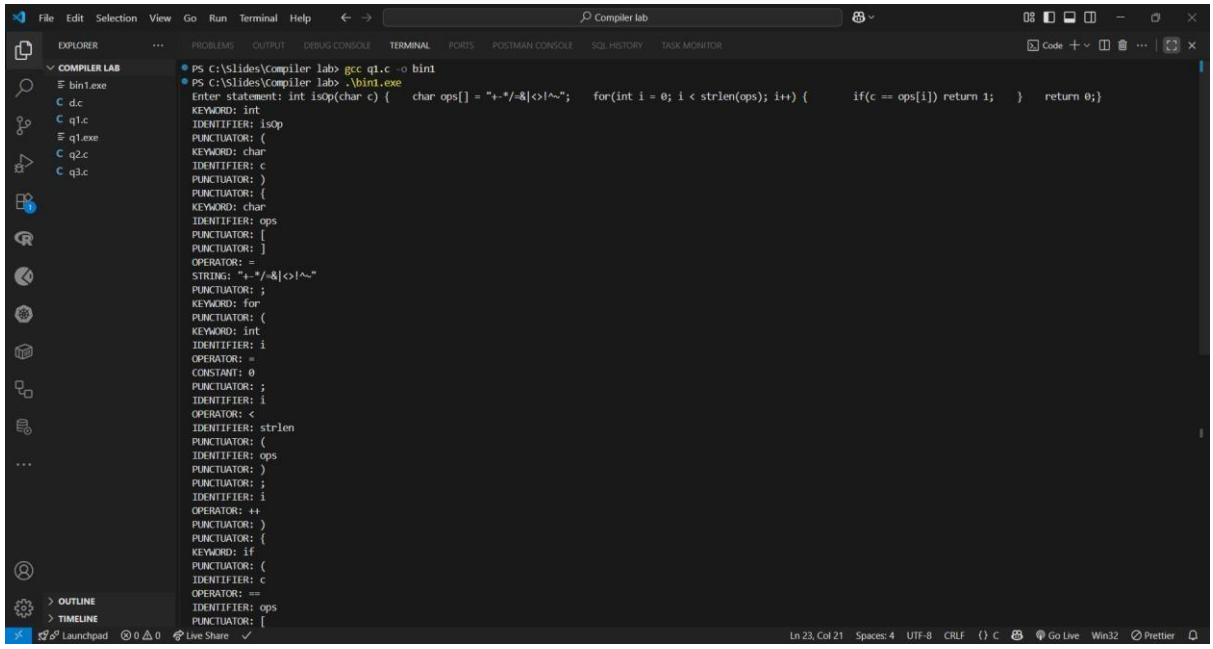
Source Code:

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


int isKw(char *w) {     char *kws[] = {"auto", "break", "case",
"char", "const", "continue", "default","do", "double", "else",
"enum", "extern",
"float", "for", "goto","if", "int", "long", "register", "return",
"short", "signed","sizeof", "static", "struct", "switch", "typedef",
"union","unsigned", "void", "volatile", "while", "bool"};
int n = sizeof(kws)/sizeof(kws[0]);     for(int i = 0; i
< n; i++) {
        if(strcmp(w, kws[i]) == 0) return 1;
    }
    return 0;

}
```

```
int isOp(char c) {
    char ops[] = "+-*/=&|<>!^~";     for(int i =
0; i < strlen(ops); i++) {         if(c ==
ops[i]) return 1;
    }
    return 0;
}

int isPunc(char c) {
    char puncs[] = ",;:(){}[]";
    for(int i = 0; i < strlen(puncs); i++) {
if(c == puncs[i]) return 1;
    }
    return 0;
}

void idOrKw(char s[], int *i) {
char t[100];     int j = 0;

    while (isalnum(s[*i]) || s[*i] == '_') {
t[j++] = s[(*i)++];
    }
    t[j] = '\0';
       printf(isKw(t) ? "KEYWORD: %s\n" : "IDENTIFIER: %s\n", t);
}

void num(char s[], int *i) {
char t[100];     int j = 0, d = 0;
       while (isdigit(s[*i]) || (s[*i] == '.' && !d && isdigit(s[*i+1])))
{
        if (s[*i] == '.') d = 1;
t[j++] = s[(*i)++];
    }
    t[j] = '\0';
    printf("CONSTANT: %s\n", t);
}

void str(char s[], int *i) {
char t[100];     int j = 0;
       t[j++] = s[(*i)++];

    while(s[*i] != '"' && s[*i] != '\0') {
t[j++] = s[(*i)++];
    }        if(s[*i] == '"') t[j++] =
s[(*i)++];

    t[j] = '\0';
    printf("STRING: %s\n", t);
}
```

```
void op(char s[], int *i) {
char t[3] = {0};     int j = 0;
        t[j++] = s[(*i)++];
        if ((t[0] == '+' && s[*i] == '+') ||(t[0] == '-' && s[*i] == '-')
||(t[0] == '=' && s[*i] == '=') || (t[0] == '!' && s[*i] == '=')
||(t[0] == '<' && s[*i] == '=') ||(t[0] == '>' && s[*i] == '=') ||(t[0]
== '&' && s[*i] == '&') ||(t[0] == '|' && s[*i] == '|'))
    {
        t[j++] = s[(*i)++];
    }         printf("OPERATOR:
%s\n", t);
}

void punc(char s[], int *i) {
    printf("PUNCTUATOR: %c\n", s[(*i)++]);
}

int main() {     char
s[1000];
    int i = 0;

    printf("Enter statement: ");     fgets(s,
sizeof(s), stdin);

    while (s[i] != '\0') {
        if (isspace(s[i])) { i++; continue; }

        if (s[i] == '"') { str(s, &i); continue; }         if (isalpha(s[i])
|| s[i] == '_') { idOrKw(s, &i); continue; }         if (isdigit(s[i]) ||
(s[i] == '.' && isdigit(s[i+1]))) { num(s,
&i); continue; }         if (isOp(s[i])) { op(s, &i);
continue; }         if (isPunc(s[i])) { punc(s, &i);
continue; }

i++;     }
    return 0;
}
```

Output:


File Edit Selection View Go Run Terminal Help
Compiler lab
EXPLORER
COMPILER LAB
bin1.exe
d.c
q1.c
q1.exe
q2.c
q3.c
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS POSTMAN CONSOLE SQL HISTORY TASK MONITOR
Code
PS C:\Slides\Compiler lab> gcc q1.c -o bin1
PS C:\Slides\Compiler lab> .\bin1.exe
Enter statement: int isOp(char c) { char ops[] = "+-*/=&|<>!^~"; for(int i = 0; i < strlen(ops); i++) { if(c == ops[i]) return 1; } return 0;}
KEYWORD: int
IDENTIFIER: isOp
PUNCTUATOR: (
KEYWORD: char
IDENTIFIER: c
PUNCTUATOR:)
PUNCTUATOR: {
KEYWORD: char
IDENTIFIER: ops
PUNCTUATOR: [
PUNCTUATOR:]
OPERATOR: =
STRING: "+-*/=&|<>!^~"
PUNCTUATOR: ;
KEYWORD: for
PUNCTUATOR: (
KEYWORD: int
IDENTIFIER: i
OPERATOR: =
CONSTANT: 0
PUNCTUATOR: ;
IDENTIFIER: i
OPERATOR: <
IDENTIFIER: strlen
PUNCTUATOR: (
IDENTIFIER: ops
PUNCTUATOR:)
PUNCTUATOR: ;
IDENTIFIER: i
OPERATOR: ++
PUNCTUATOR:)
PUNCTUATOR: {
KEYWORD: if
PUNCTUATOR: (
IDENTIFIER: c
OPERATOR: ==
IDENTIFIER: ops
PUNCTUATOR: [
OUTLINE
TIMELINE
Launchpad
Live Share
Ln 23, Col 21 Spaces: 4 UTF-8 CRLF C Go Live Win32 Prettier

```
PS C:\Slides\Compiler lab> gcc q1.c -o bin1

PS C:\Slides\Compiler lab> .\bin1.exe
Enter statement: int isOp(char c) {    char ops[] = "+-*/=&|<>!^~";
for(int i = 0; i < strlen(ops); i++) {        if(c == ops[i]) return 1;    }
return 0;}
KEYWORD: int
IDENTIFIER: isOp
PUNCTUATOR: (
KEYWORD: char
IDENTIFIER: c
PUNCTUATOR: )
PUNCTUATOR: {
KEYWORD: char
IDENTIFIER: ops
PUNCTUATOR: [
PUNCTUATOR: ]
OPERATOR: =
STRING: "+-*/=&|<>!^~"
PUNCTUATOR: ;
KEYWORD: for
PUNCTUATOR: (
KEYWORD: int
IDENTIFIER: i
OPERATOR: =
CONSTANT: 0
PUNCTUATOR: ;
IDENTIFIER: i
```

```
OPERATOR: <
IDENTIFIER: strlen
PUNCTUATOR: (
IDENTIFIER: ops
PUNCTUATOR: )
PUNCTUATOR: ;
IDENTIFIER: i OPERATOR:
++
PUNCTUATOR: )
PUNCTUATOR: { KEYWORD:
if
PUNCTUATOR: (
IDENTIFIER: c
OPERATOR: ==
IDENTIFIER: ops
PUNCTUATOR: [
IDENTIFIER: i
PUNCTUATOR: ]
PUNCTUATOR: )
KEYWORD: return
CONSTANT: 1
PUNCTUATOR: ;
PUNCTUATOR: }
KEYWORD: return
CONSTANT: 0
PUNCTUATOR: ;
PUNCTUATOR: }
PS C:\Slides\Compiler lab>
```

Experiment-5(b):

Title:
Identify tokens from a small program stored in a text file

Aim of the experiment:
To develop a lexical analyser that reads a small C program from a text file and identifies/categorizes all tokens present in the code.

Algorithm:

1. Accept filename as command line argument
2. Open and read the file content into a buffer 3. Initialize index pointer to start of string 4. While not at end of string:
    - Skip whitespace characters
    - If character is double quote, process as string literal
    - If character is alphabetic or underscore, process as identifier/keyword
    - If character is digit or decimal point, process as numeric constant
    - If character is operator symbol, process as operator o If character is punctuation, process as punctuator o Otherwise, move to next character
5. For each token type, implement specific extraction logic
6. Print categorized tokens with appropriate labels

Source Code:

```c
#include <stdio.h>
#include <ctype.h>
#include <string.h>

int isKw(char *w) {
    char *kws[] = {
        "auto", "break", "case", "char", "const", "continue", "default", "do",
        "double", "else", "enum", "extern", "float", "for", "goto", "if",
        "int", "long", "register", "return", "short", "signed", "sizeof",
        "static", "struct", "switch", "typedef", "union", "unsigned", "void",
        "volatile", "while", "bool"
    };
    int n = sizeof(kws) / sizeof(kws[0]);
    for (int i = 0; i < n; i++) {
        if (strcmp(w, kws[i]) == 0)
            return 1;
    }
```

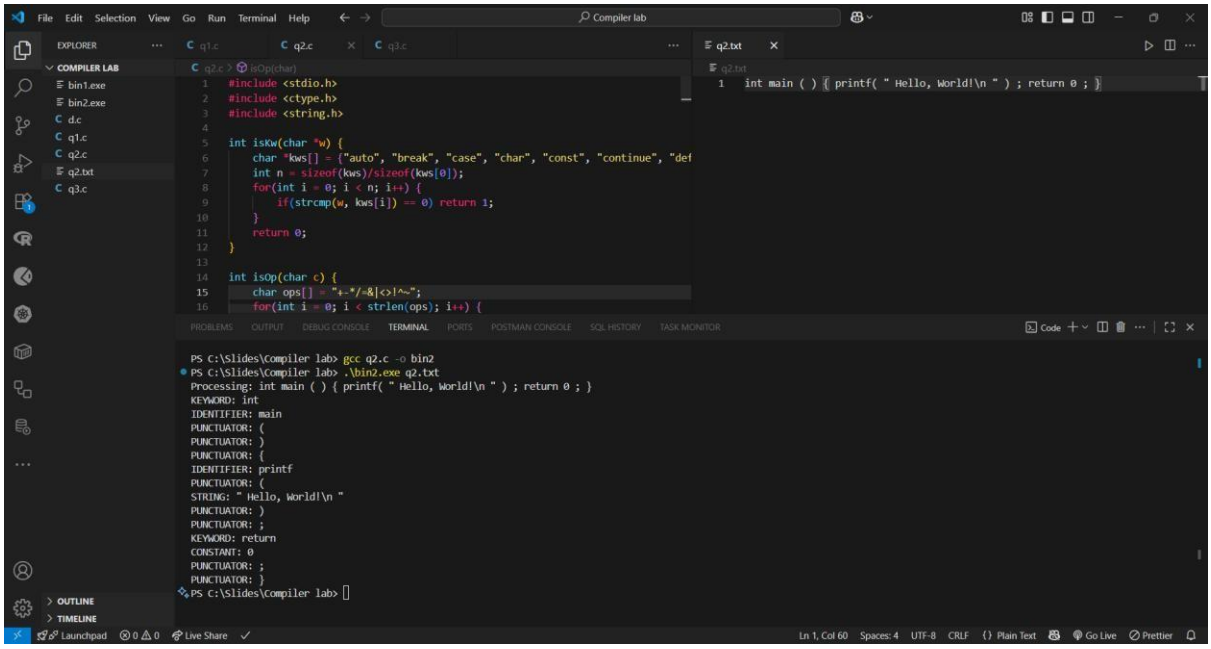
```
    return 0;
}

int isOp(char c) {
    char ops[] = "+-*/=&|<>!^~";
    for (int i = 0; i < (int)strlen(ops); i++) {
        if (c == ops[i])
            return 1;
    }
    return 0;
}

int isPunc(char c) {
    char puncs[] = ",;:(){}[]";
    for (int i = 0; i < (int)strlen(puncs); i++) {
        if (c == puncs[i])
            return 1;
    }
    return 0;
}

void idOrKw(char s[], int *i) {
    char t[100];
    int j = 0;

    while (isalnum((unsigned char)s[*i]) || s[*i] == '_') {
        t[j++] = s[(*i)++];
    }
    t[j] = '\0';
    printf(isKw(t) ? "KEYWORD: %s\n" : "IDENTIFIER: %s\n", t);
}

void num(char s[], int *i) {
    char t[100];
    int j = 0, d = 0;

    while (isdigit((unsigned char)s[*i]) ||
           (s[*i] == '.' && !d && isdigit((unsigned char)s[*i + 1]))) {
        if (s[*i] == '.')
            d = 1;
        t[j++] = s[(*i)++];
    }
    t[j] = '\0';
    printf("CONSTANT: %s\n", t);
}
```

```c
void str(char s[], int *i) {
    char t[100];
    int j = 0;

    t[j++] = s[(*i)++]; /* starting quote */

    while (s[*i] != '"' && s[*i] != '\0') {
        t[j++] = s[(*i)++];
    }
    if (s[*i] == '"')
        t[j++] = s[(*i)++]; /* closing quote */

    t[j] = '\0';
    printf("STRING: %s\n", t);
}

void op(char s[], int *i) {
    char t[3] = {0};
    int j = 0;

    t[j++] = s[(*i)++];

    if ((t[0] == '+' && s[*i] == '+') ||
        (t[0] == '-' && s[*i] == '-') ||
        (t[0] == '=' && s[*i] == '=') ||
        (t[0] == '!' && s[*i] == '=') ||
        (t[0] == '<' && s[*i] == '=') ||
        (t[0] == '>' && s[*i] == '=') ||
        (t[0] == '&' && s[*i] == '&') ||
        (t[0] == '|' && s[*i] == '|')) {
        t[j++] = s[(*i)++];
    }
    printf("OPERATOR: %s\n", t);
}

void punc(char s[], int *i) {
    printf("PUNCTUATOR: %c\n", s[(*i)++]);
}

int main(int argc, char *argv[]) {
    char s[1000];
    int i = 0;
    FILE *file;
```

```
    if (argc != 2) {
        printf("Usage: %s <input_file>\n", argv[0]);
        return 1;
    }

    file = fopen(argv[1], "r");
    if (file == NULL) {
        printf("Error opening file!\n");
        return 1;
    }

    if (fgets(s, sizeof(s), file) == NULL) {
        printf("Error reading file or file is empty!\n");
        fclose(file);
        return 1;
    }
    fclose(file);

    printf("Processing: %s\n", s);

    while (s[i] != '\0') {
        if (isspace((unsigned char)s[i])) {
            i++;
            continue;
        }

        if (s[i] == '"') {
            str(s, &i);
            continue;
        }
        if (isalpha((unsigned char)s[i]) || s[i] == '_') {
            idOrKw(s, &i);
            continue;
        }
        if (isdigit((unsigned char)s[i]) || (s[i] == '.' && isdigit((unsigned
char)s[i + 1]))) {
            num(s, &i);
            continue;
        }
        if (isOp(s[i])) {
            op(s, &i);
            continue;
        }
        if (isPunc(s[i])) {
            punc(s, &i);
```

```
            continue;
        }
        i++;
    }

    return 0;
}
```

Output:



Experiment-5(c):

Tite:

Identify tokens from user input and store in a text file

Aim of the experiment:

To develop a lexical analyser that accepts a small C program as user input, stores it in a text file, and then identifies/categorizes all tokens present in the code.

Algorithm:

1. Read input string from user

2. Store the input in a text file

3. Initialize index pointer to start of string

4. While not at end of string:

    - Skip whitespace characters
    - If character is double quote, process as string literal
    - If character is alphabetic or underscore, process as identifier/keyword
    - If character is digit or decimal point, process as numeric constant o If character is operator symbol, process as operator o If character is punctuation, process as punctuator o Otherwise, move to next character

5. For each token type, implement specific extraction logic

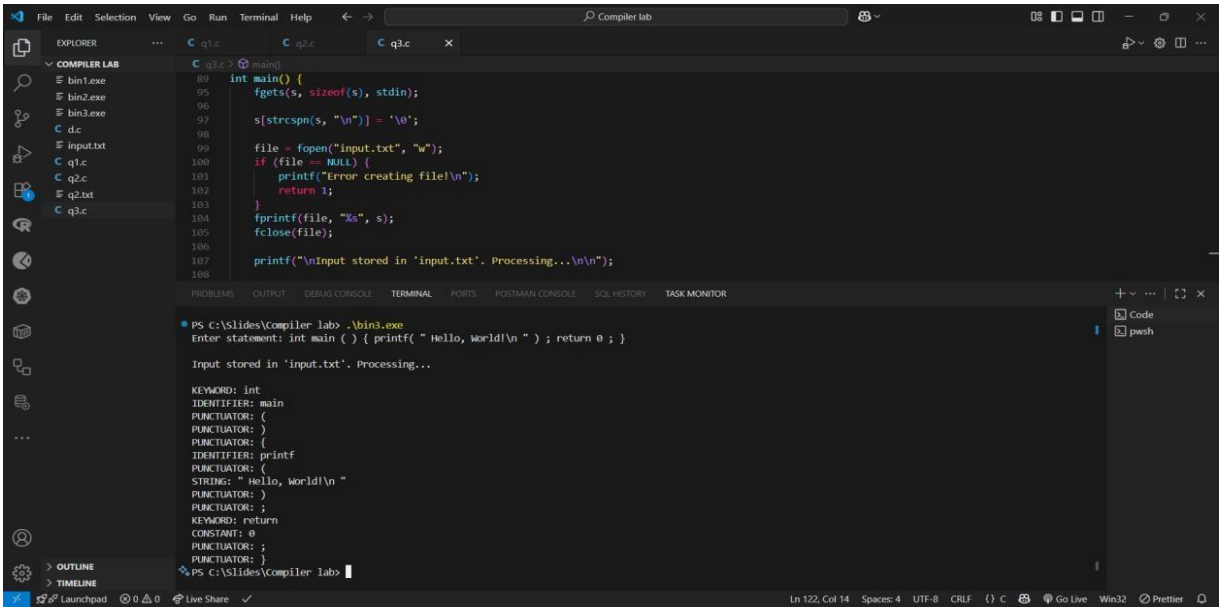
6. Print categorized tokens with appropriate labels

Source Code:

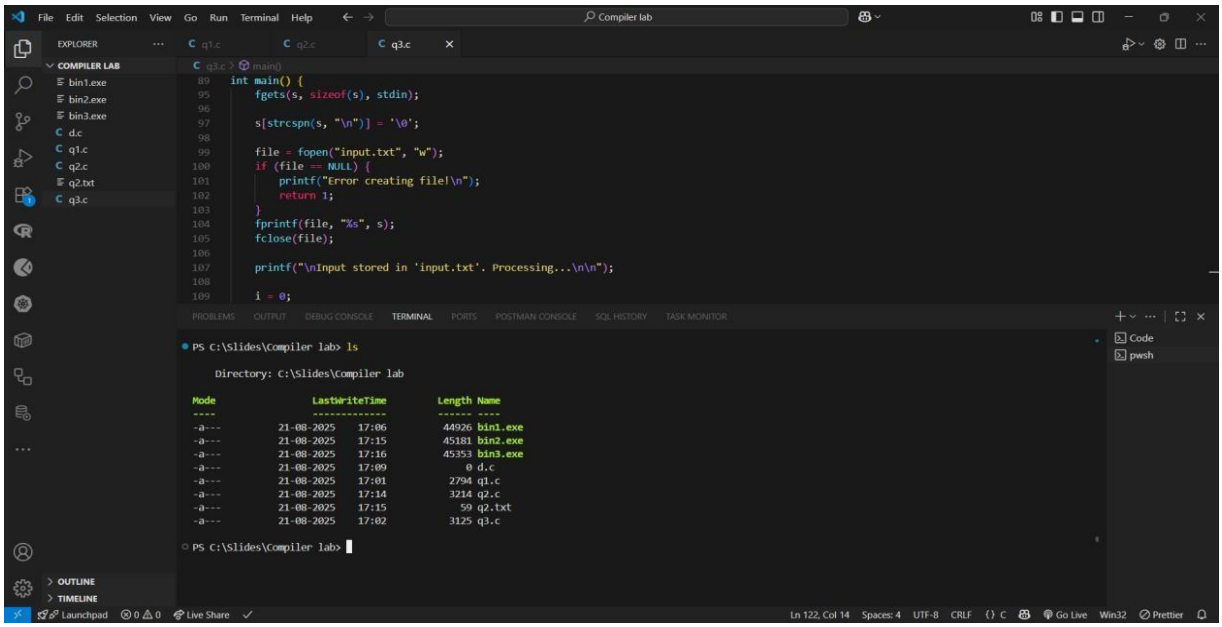
```c
#include <stdio.h>
#include <ctype.h>
#include <string.h>

int isKw(char *w) {
    char *kws[] = {
        "auto", "break", "case", "char", "const", "continue", "default", "do",
        "double", "else", "enum", "extern", "float", "for", "goto", "if",
        "int", "long", "register", "return", "short", "signed", "sizeof",
        "static", "struct", "switch", "typedef", "union", "unsigned", "void",
        "volatile", "while", "bool"
    };
    int n = sizeof(kws) / sizeof(kws[0]);

    for (int i = 0; i < n; i++) {
        if (strcmp(w, kws[i]) == 0)
            return 1;
    }
    return 0;
}

int isOp(char c) {
    char ops[] = "+-*/=&|<>!^~";
```

```
    for (int i = 0; i < strlen(ops); i++) {
        if (c == ops[i])
            return 1;
    }
    return 0;
}

int isPunc(char c) {
    char puncs[] = ",;:(){}[]";
    for (int i = 0; i < strlen(puncs); i++) {
        if (c == puncs[i])
            return 1;
    }
    return 0;
}

void idOrKw(char s[], int *i) {
    char t[100];
    int j = 0;

    while (isalnum(s[*i]) || s[*i] == '_') {
        t[j++] = s[(*i)++];
    }
    t[j] = '\0';
    printf(isKw(t) ? "KEYWORD: %s\n" : "IDENTIFIER: %s\n", t);
}

void num(char s[], int *i) {
    char t[100];
    int j = 0, d = 0;

    while (isdigit(s[*i]) || (s[*i] == '.' && !d && isdigit(s[*i + 1]))) {
        if (s[*i] == '.')
            d = 1;
        t[j++] = s[(*i)++];
    }
    t[j] = '\0';
    printf("CONSTANT: %s\n", t);
}

void str(char s[], int *i) {
    char t[100];
    int j = 0;

    t[j++] = s[(*i)++];
```

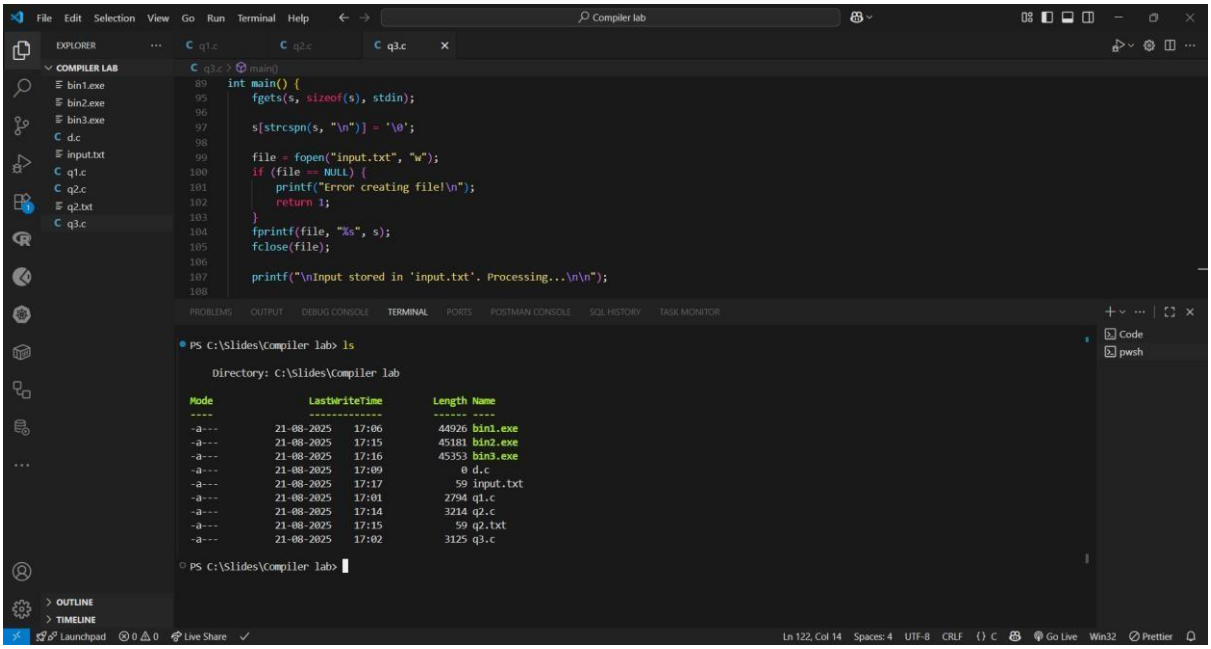
```
while (s[*i] != '"' && s[*i] != '\0') {
    t[j++] = s[(*i)++];
}
```



Before running program no input.txt



After running program input.txt is created


File Edit Selection View Go Run Terminal Help
Compiler lab
EXPLORER
COMPILER LAB
bin1.exe
bin2.exe
bin3.exe
d.c
input.txt
q1.c
q2.c
q2.txt
q3.c
q1.c
q2.c
q3.c
q3.c > main()
int main() {
fgets(s, sizeof(s), stdin);
s[strcspn(s, "\n")] = '\0';
file = fopen("input.txt", "w");
if (file == NULL) {
printf("Error creating file!\n");
return 1;
}
fprintf(file, "%s", s);
fclose(file);
printf("\nInput stored in 'input.txt'. Processing...\n\n");
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS POSTMAN CONSOLE SQL HISTORY TASK MONITOR
Code
pwsh
PS C:\Slides\Compiler lab> ls
Directory: C:\Slides\Compiler lab
Mode LastWriteTime Length Name
-a--- 21-08-2025 17:06 44926 bin1.exe
-a--- 21-08-2025 17:15 45181 bin2.exe
-a--- 21-08-2025 17:16 45353 bin3.exe
-a--- 21-08-2025 17:09 0 d.c
-a--- 21-08-2025 17:17 59 input.txt
-a--- 21-08-2025 17:01 2794 q1.c
-a--- 21-08-2025 17:14 3214 q2.c
-a--- 21-08-2025 17:15 59 q2.txt
-a--- 21-08-2025 17:02 3125 q3.c
PS C:\Slides\Compiler lab>
OUTLINE
TIMELINE
Launchpad
Live Share
Ln 122, Col 14 Spaces: 4 UTF-8 CRLF C Go Live Win32 Prettier

Experiment 6(a):

Title:

Count the frequency of a given word in a file

Aim of the experiment:

To implement a LEX program that counts the frequency of a specific word in a text file.

Algorithm:

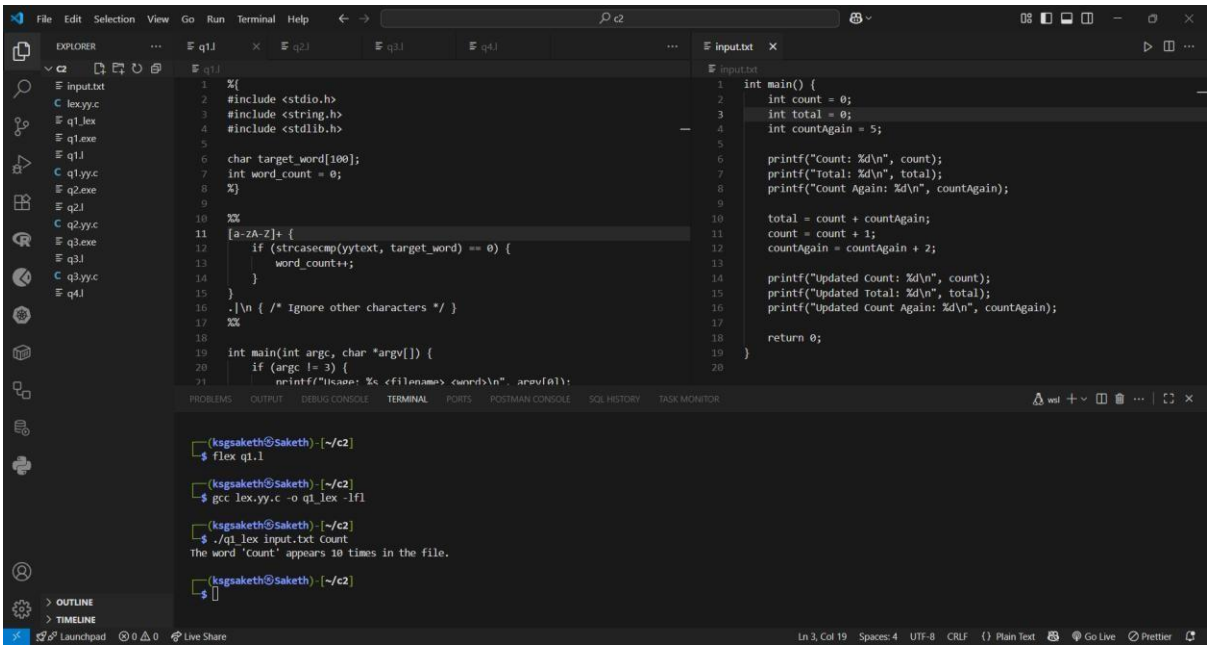
1. Define the word to search for as a pattern
2. Initialize a counter variable to zero
3. For each token matched in the input:

o If the token matches the target word, increment the counter

4. After processing all input, print the frequency count

Source Code:

Output



Experiment-6(b):

Title:

Replace a word with another taking input from file

Aim of the experiment:

To implement a LEX program that replaces all occurrences of one word with another word in a text file.

Algorithm:

1. Define patterns for the word to be replaced
2. For each token matched:
   - If it matches the target word, print the replacement word
   - Otherwise, print the original token
3. Handle all other characters normally

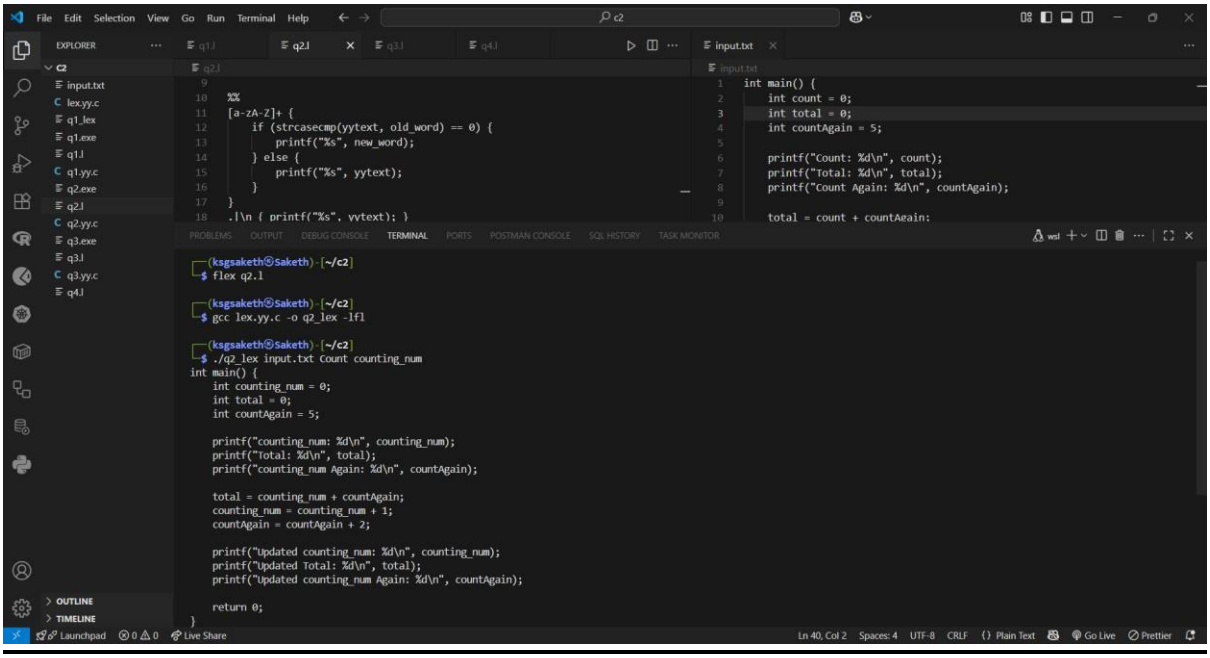
Source Code:

```
 %{
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
 char target_word[100]; int word_count = 0;
%}

%%
[a-zA-Z]+ {
    if (strcasecmp(yytext, target_word) == 0) {         word_count++;
    }
}
.|\n { /* Ignore other characters */ }
%%
int main(int argc, char *argv[]) {     if (argc != 3) {
        printf("Usage: %s <filename> <word>\n", argv[0]);         return 1;
    }
    strcpy(target_word, argv[2]);

    yyin = fopen(argv[1], "r");     if (!yyin) {
        perror("Error opening file");         return 1;
    }
    yylex();
```

```
    printf("The word '%s' appears %d times in the file.\n", target_word,
word_count);

    fclose(yyin);     return 0;
}
 (old_word, argv[2]);
    strcpy(new_word, argv[3]);

    yyin = fopen(argv[1], "r");     if (!yyin) {
        perror("Error opening file");         return 1;
    }
    yylex();

    fclose(yyin);     return 0;
}
```

```
%{
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
 char
old_word[100]; char
new_word[100];
%}

%%
[a-zA-Z]+ {
    if (strcasecmp(yytext, old_word) == 0) {
printf("%s", new_word);
    } else {
        printf("%s", yytext);
    }
}
.|\n { printf("%s", yytext); }
%%
int main(int argc, char *argv[]) {
if (argc != 4) {
        printf("Usage: %s <filename> <old_word> <new_word>\n",
argv[0]);
        return 1;
    }
    strcpy(old_word, argv[2]);
strcpy(new_word, argv[3]);

    yyin = fopen(argv[1], "r");
if (!yyin) {
        perror("Error opening file");
return 1;
    }
    yylex();

    fclose(yyin);
return 0;
}
```

Output:


File Edit Selection View Go Run Terminal Help
c2
EXPLORER
q2.l
input.txt
c2
input.txt
lex.yy.c
q1_lex
q1.exe
q1.l
q1.yy.c
q2.exe
q2.l
q2.yy.c
q3.exe
q3.l
q3.yy.c
q4.l
%%
[a-zA-Z]+ {
if (strcasecmp(yytext, old_word) == 0) {
printf("%s", new_word);
} else {
printf("%s", yytext);
}
}
.|\n { printf("%s", yytext); }
int main() {
int count = 0;
int total = 0;
int countAgain = 5;
printf("Count: %d\n", count);
printf("Total: %d\n", total);
printf("Count Again: %d\n", countAgain);
total = count + countAgain;
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS POSTMAN CONSOLE SQL HISTORY TASK MONITOR
wsl
(ksgsaketh㉿Saketh)-[~/c2]
$ flex q2.l
(ksgsaketh㉿Saketh)-[~/c2]
$ gcc lex.yy.c -o q2_lex -lfl
(ksgsaketh㉿Saketh)-[~/c2]
$ ./q2_lex input.txt Count counting_num
int main() {
int counting_num = 0;
int total = 0;
int countAgain = 5;
printf("counting_num: %d\n", counting_num);
printf("Total: %d\n", total);
printf("counting_num Again: %d\n", countAgain);
total = counting_num + countAgain;
counting_num = counting_num + 1;
countAgain = countAgain + 2;
printf("Updated counting_num: %d\n", counting_num);
printf("Updated Total: %d\n", total);
printf("Updated counting_num Again: %d\n", countAgain);
return 0;
}
OUTLINE
TIMELINE
Launchpad 0 0 Live Share
Ln 40, Col 2 Spaces: 4 UTF-8 CRLF Plain Text Go Live Prettier

Experiment - 6(c):

Title:

Find the length of the longest word

Aim of the experiment:

To implement a LEX program that finds the length of the longest word in a text file.

Algorithm:

1. Initialize a variable to track the maximum length

2. For each word token matched:

o Calculate its length o If longer than current maximum, update the maximum

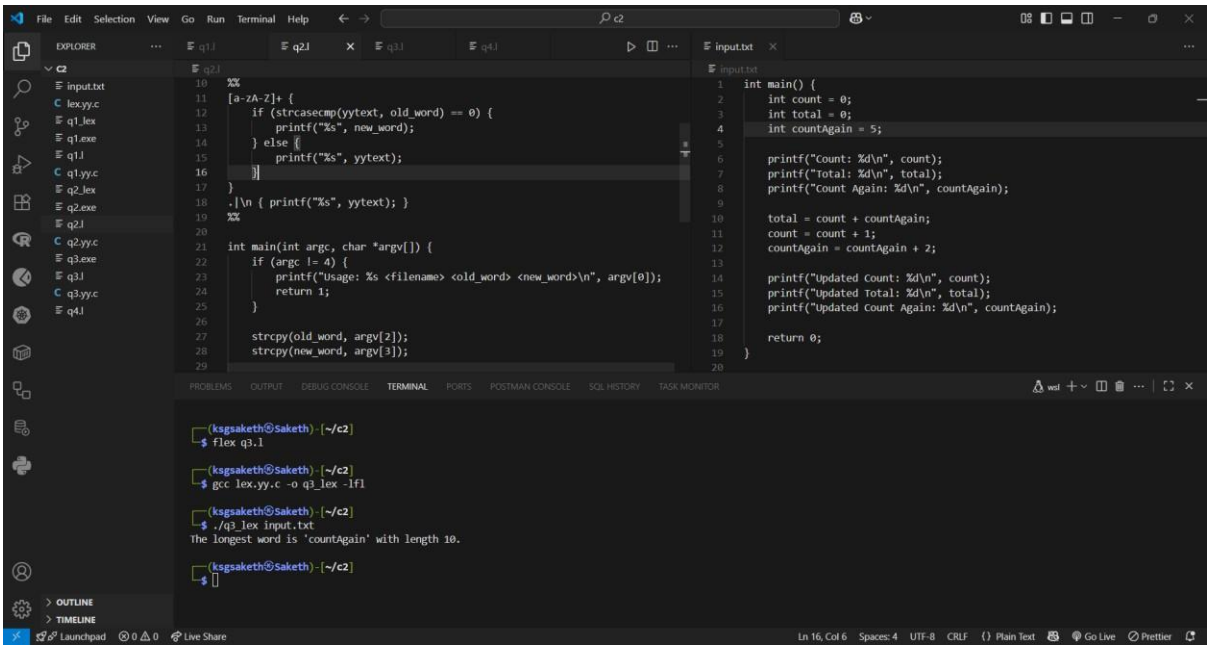
3. After processing all input, print the maximum length found

Source Code:

```
%{
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>
 int max_length = 0; char
longest_word[100];
%}

%%
```

```
[a-zA-Z]+ {
    int len = strlen(yytext);
if (len > max_length) {
max_length = len;
        strcpy(longest_word, yytext);
    }
}
.|\n { /* Ignore other characters */ }
%%
int main(int argc, char *argv[]) {
if (argc != 2) {
        printf("Usage: %s <filename>\n", argv[0]);
return 1;
    }
    yyin = fopen(argv[1], "r");
if (!yyin) {
        perror("Error opening file");
return 1;
    }
    yylex();

    printf("The longest word is '%s' with length %d.\n", longest_word,
max_length);

    fclose(yyin);
return 0;
}
```

Output:


File Edit Selection View Go Run Terminal Help
c2
EXPLORER
input.txt
lex.yy.c
q1_lex
q1.exe
q1.l
q1.yy.c
q2_lex
q2.exe
q2.l
q2.yy.c
q3.exe
q3.l
q3.yy.c
q4.l
%%
[a-zA-Z]+ {
if (strcasecmp(yytext, old_word) == 0) {
printf("%s", new_word);
} else {
printf("%s", yytext);
}
}
.|\n { printf("%s", yytext); }
%%
int main(int argc, char *argv[]) {
if (argc != 4) {
printf("Usage: %s <filename> <old_word> <new_word>\n", argv[0]);
return 1;
}
strcpy(old_word, argv[2]);
strcpy(new_word, argv[3]);
int main() {
int count = 0;
int total = 0;
int countAgain = 5;
printf("Count: %d\n", count);
printf("Total: %d\n", total);
printf("Count Again: %d\n", countAgain);
total = count + countAgain;
count = count + 1;
countAgain = countAgain + 2;
printf("Updated Count: %d\n", count);
printf("Updated Total: %d\n", total);
printf("Updated Count Again: %d\n", countAgain);
return 0;
}
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS POSTMAN CONSOLE SQL HISTORY TASK MONITOR
(ksgsaketh㉿Saketh)-[~/c2]
$ flex q3.l
(ksgsaketh㉿Saketh)-[~/c2]
$ gcc lex.yy.c -o q3_lex -lfl
(ksgsaketh㉿Saketh)-[~/c2]
$ ./q3_lex input.txt
The longest word is 'countAgain' with length 10.
(ksgsaketh㉿Saketh)-[~/c2]
$
OUTLINE
TIMELINE
Launchpad
Live Share
Ln 16, Col 6 Spaces: 4 UTF-8 CRLF Plain Text Go Live Prettier

Experiment-6(d):

Title:

Construct a lexical analyzer using LEX tool

Aim of the experiment:

To implement a complete lexical analyzer using LEX that identifies different types of tokens in a programming language.

Algorithm:

1. Define patterns for different token types:

o Keywords o Identifiers o Constants (integers, floats) o

Operators o Punctuators o Strings

2. For each matched pattern, print the token type and value

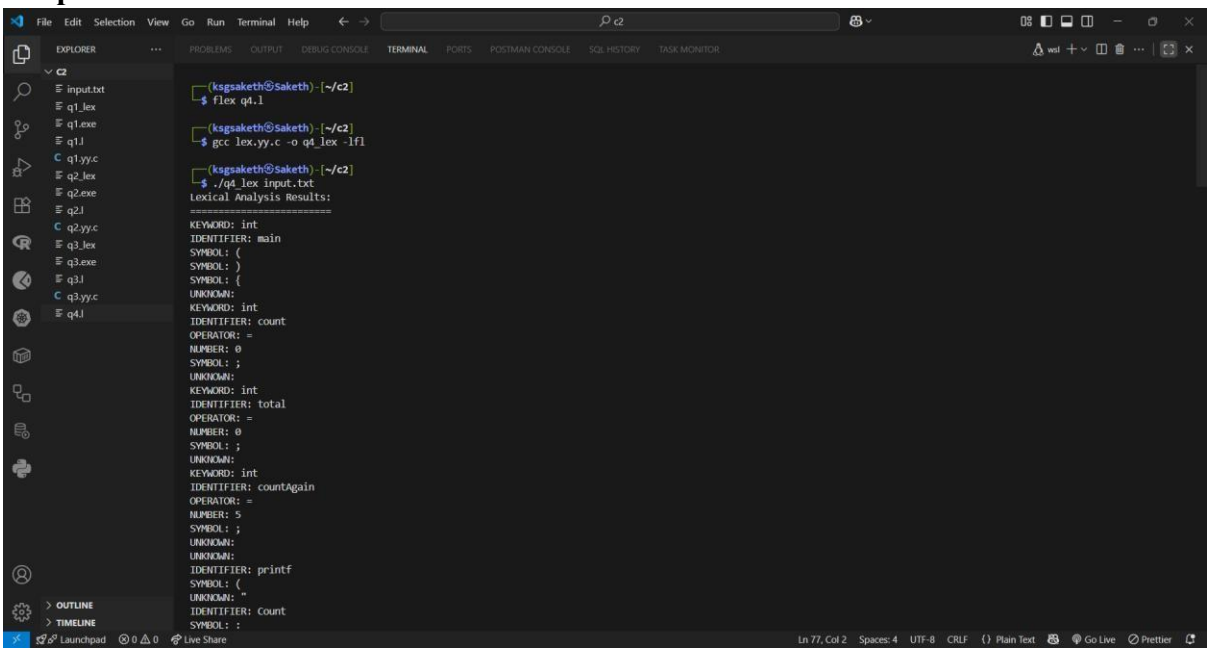
3. Skip whitespace characters

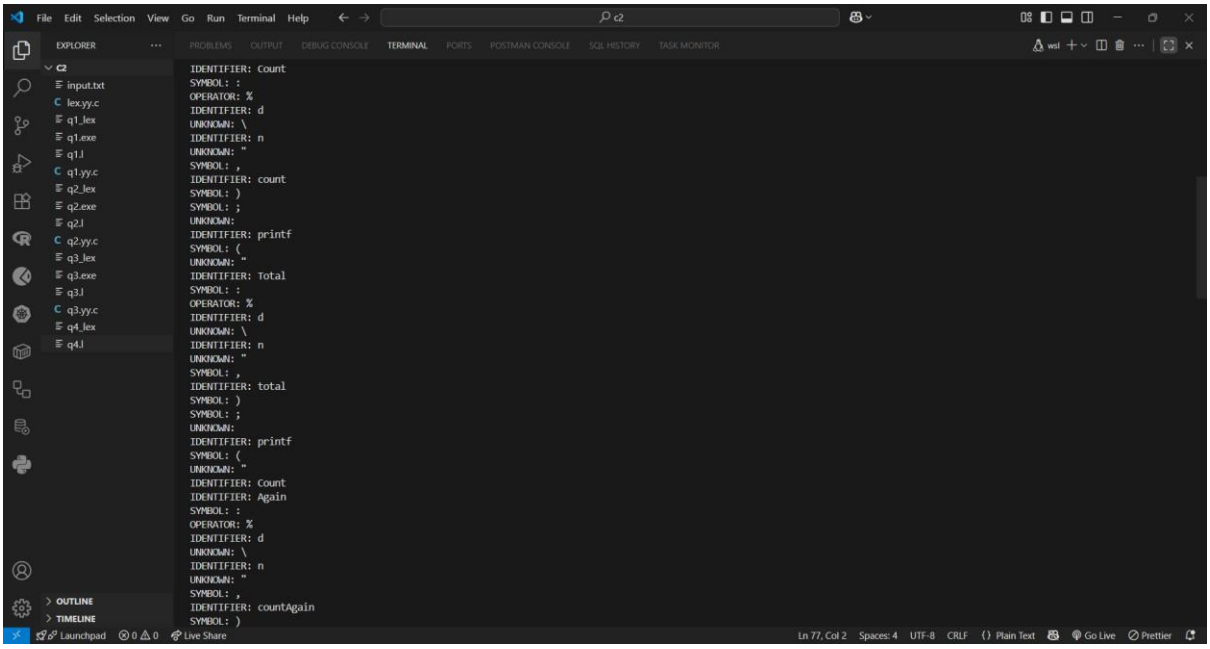
Source Code:

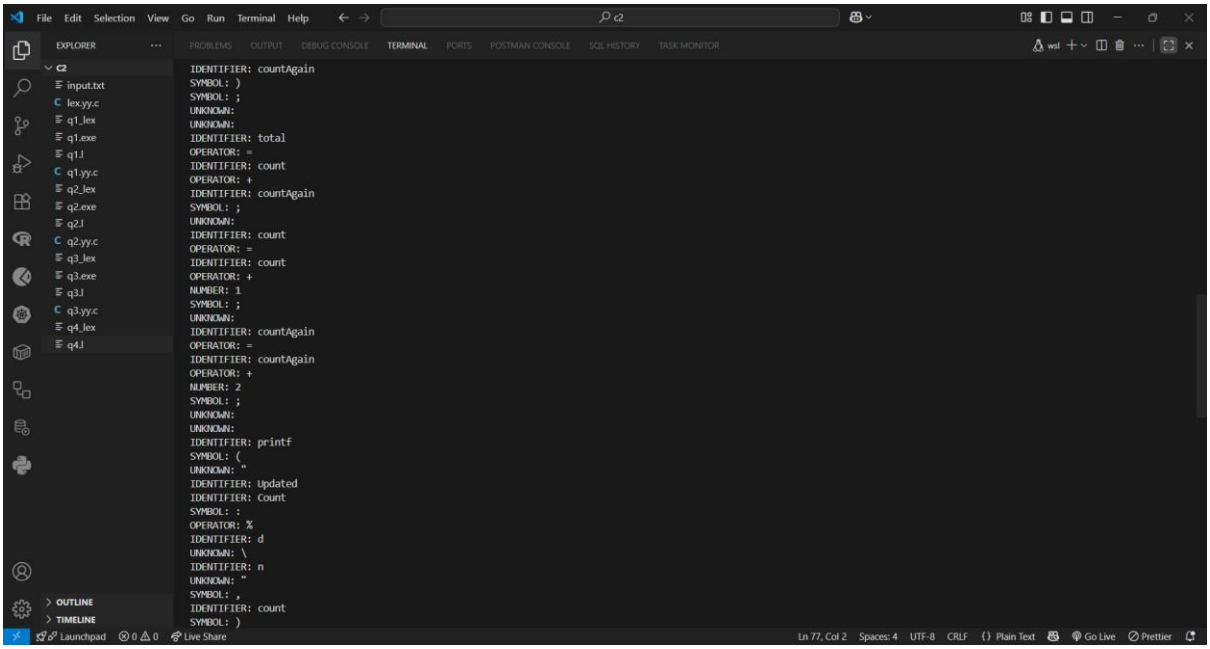
```
%{
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>

// List of keywords char
*keywords[] = {
    "auto", "break", "case", "char", "const", "continue", "default",
"do",
    "double", "else", "enum", "extern", "float", "for", "goto", "if",
"int", "long", "register", "return", "short", "signed", "sizeof",
"static",
    "struct", "switch", "typedef", "union", "unsigned", "void",
"volatile", "while"
};
int is_keyword(char *word) {
    int total_keywords = sizeof(keywords) / sizeof(keywords[0]);
for (int i = 0; i < total_keywords; i++) {         if
(strcmp(word, keywords[i]) == 0) {             return 1;
        }
}
    return 0;
}
%}
```

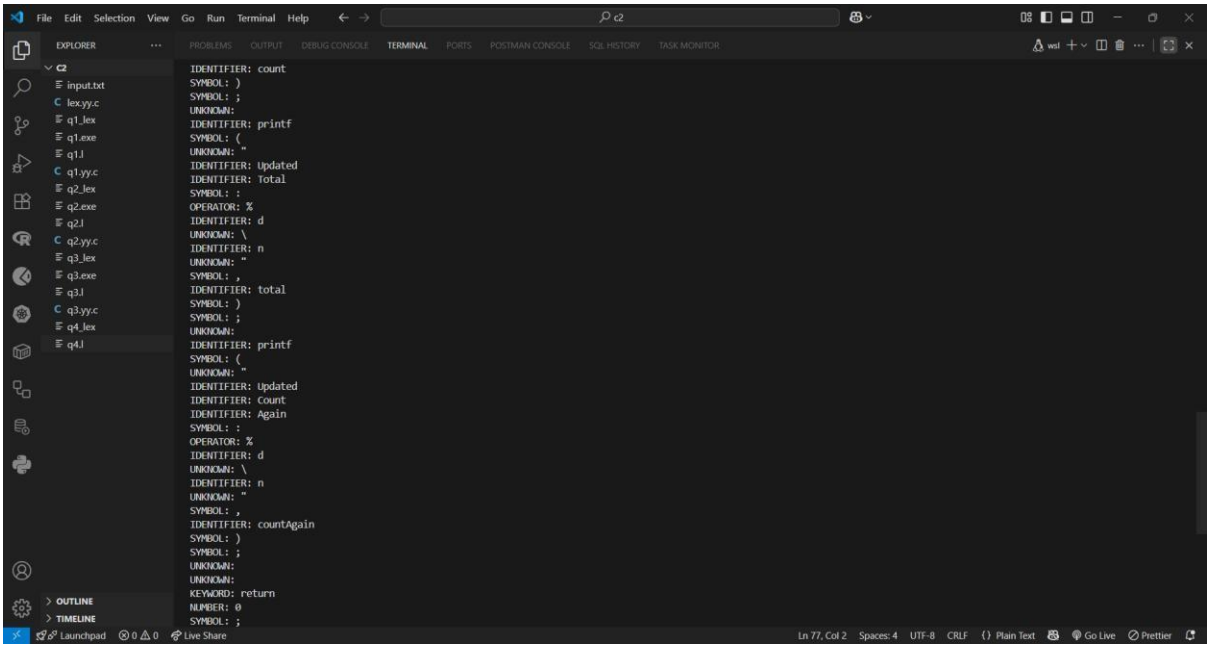
```
DIGIT    [0-9]
LETTER   [a-zA-Z]
ID       {LETTER}({LETTER}|{DIGIT})*
NUMBER   {DIGIT}+(\.{DIGIT}+)?([eE][+-]?{DIGIT}+)?
OPERATOR [+-*/%=<>!&|^]
SYMBOL   [;,.:(){}[\]]

%%

"#include"    { printf("PREPROCESSOR: %s\n", yytext); }
"#define"     { printf("PREPROCESSOR: %s\n", yytext); }
{ID} {
    if (is_keyword(yytext)) {
        printf("KEYWORD: %s\n", yytext);
    } else {
        printf("IDENTIFIER: %s\n", yytext);
    }
}

{NUMBER}      { printf("NUMBER: %s\n", yytext); }
{OPERATOR}    { printf("OPERATOR: %s\n", yytext); }
{SYMBOL}      { printf("SYMBOL: %s\n", yytext); }

[ \t\n]       { /* Ignore whitespace */ }
.             { printf("UNKNOWN: %s\n", yytext); }

%%
int main(int argc, char *argv[]) {
if (argc != 2) {
        printf("Usage: %s <filename>\n", argv[0]);
return 1;
    }
    yyin = fopen(argv[1], "r");
if (!yyin) {
        perror("Error opening file");
return 1;
    }
    printf("Lexical Analysis Results:\n");
printf("=========================\n");

    yylex();

    fclose(yyin);
return 0;
}
```

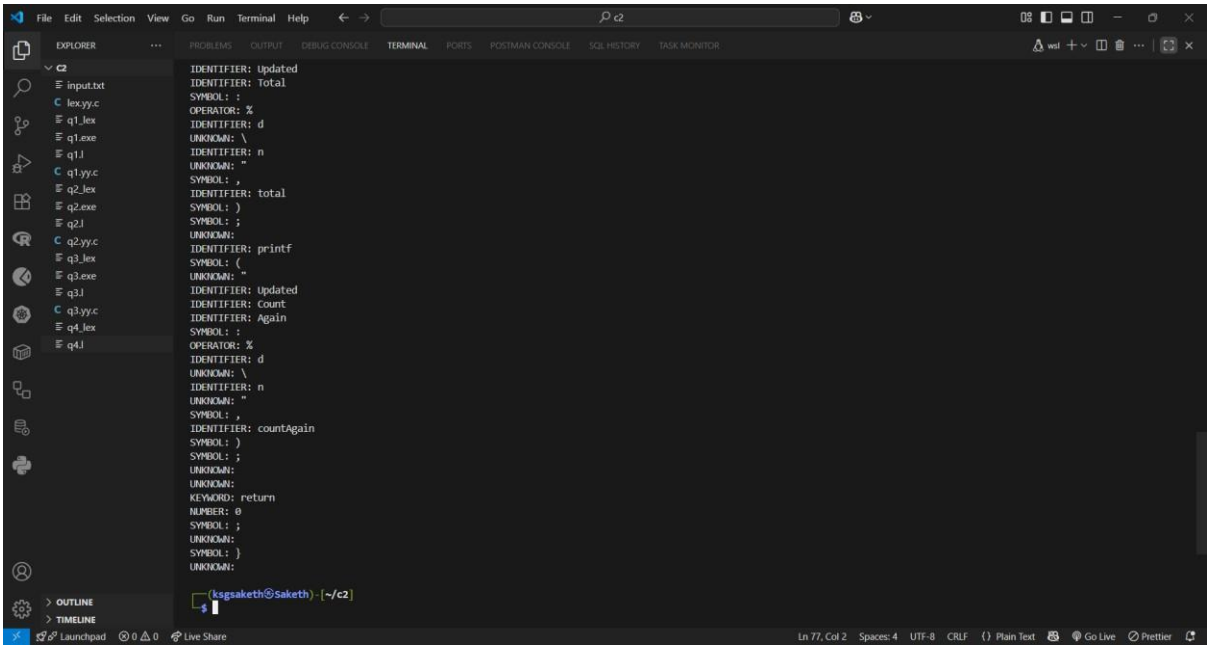
Output :


(ksgsaketh㉿Saketh)-[~/c2]
$ flex q4.l
(ksgsaketh㉿Saketh)-[~/c2]
$ gcc lex.yy.c -o q4_lex -lfl
(ksgsaketh㉿Saketh)-[~/c2]
$ ./q4_lex input.txt
Lexical Analysis Results:
KEYWORD: int
IDENTIFIER: main
SYMBOL: (
SYMBOL:)
SYMBOL: {
UNKNOWN:
KEYWORD: int
IDENTIFIER: count
OPERATOR: =
NUMBER: 0
SYMBOL: ;
UNKNOWN:
KEYWORD: int
IDENTIFIER: total
OPERATOR: =
NUMBER: 0
SYMBOL: ;
UNKNOWN:
KEYWORD: int
IDENTIFIER: countAgain
OPERATOR: =
NUMBER: 5
SYMBOL: ;
UNKNOWN:
UNKNOWN:
IDENTIFIER: printf
SYMBOL: (
UNKNOWN: "
IDENTIFIER: Count
SYMBOL: :


IDENTIFIER: Count
SYMBOL: :
OPERATOR: %
IDENTIFIER: d
UNKNOWN: \
IDENTIFIER: n
UNKNOWN: "
SYMBOL: ,
IDENTIFIER: count
SYMBOL:)
SYMBOL: ;
UNKNOWN:
IDENTIFIER: printf
SYMBOL: (
UNKNOWN: "
IDENTIFIER: Total
SYMBOL: :
OPERATOR: %
IDENTIFIER: d
UNKNOWN: \
IDENTIFIER: n
UNKNOWN: "
SYMBOL: ,
IDENTIFIER: total
SYMBOL:)
SYMBOL: ;
UNKNOWN:
IDENTIFIER: printf
SYMBOL: (
UNKNOWN: "
IDENTIFIER: Count
IDENTIFIER: Again
SYMBOL: :
OPERATOR: %
IDENTIFIER: d
UNKNOWN: \
IDENTIFIER: n
UNKNOWN: "
SYMBOL: ,
IDENTIFIER: countAgain
SYMBOL:)


File Edit Selection View Go Run Terminal Help
c2
EXPLORER
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS POSTMAN CONSOLE SQL HISTORY TASK MONITOR
c2
input.txt
lex.yy.c
q1_lex
q1.exe
q1.l
q1.yy.c
q2_lex
q2.exe
q2.l
q2.yy.c
q3_lex
q3.exe
q3.l
q3.yy.c
q4_lex
q4.l
IDENTIFIER: countAgain
SYMBOL:)
SYMBOL: ;
UNKNOWN:
UNKNOWN:
IDENTIFIER: total
OPERATOR: =
IDENTIFIER: count
OPERATOR: +
IDENTIFIER: countAgain
SYMBOL: ;
UNKNOWN:
IDENTIFIER: count
OPERATOR: =
IDENTIFIER: count
OPERATOR: +
NUMBER: 1
SYMBOL: ;
UNKNOWN:
IDENTIFIER: countAgain
OPERATOR: =
IDENTIFIER: countAgain
OPERATOR: +
NUMBER: 2
SYMBOL: ;
UNKNOWN:
UNKNOWN:
IDENTIFIER: printf
SYMBOL: (
UNKNOWN: "
IDENTIFIER: Updated
IDENTIFIER: Count
SYMBOL: :
OPERATOR: %
IDENTIFIER: d
UNKNOWN: \
IDENTIFIER: n
UNKNOWN: "
SYMBOL: ,
IDENTIFIER: count
SYMBOL:)
OUTLINE
TIMELINE
Launchpad
Live Share
Ln 77, Col 2 Spaces: 4 UTF-8 CRLF Plain Text Go Live Prettier


File Edit Selection View Go Run Terminal Help
c2
EXPLORER
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS POSTMAN CONSOLE SQL HISTORY TASK MONITOR
c2
input.txt
lex.yy.c
q1_lex
q1.exe
q1.l
q1.yy.c
q2_lex
q2.exe
q2.l
q2.yy.c
q3_lex
q3.exe
q3.l
q3.yy.c
q4_lex
q4.l
IDENTIFIER: count
SYMBOL:)
SYMBOL: ;
UNKNOWN:
IDENTIFIER: printf
SYMBOL: (
UNKNOWN: "
IDENTIFIER: Updated
IDENTIFIER: Total
SYMBOL: :
OPERATOR: %
IDENTIFIER: d
UNKNOWN: \
IDENTIFIER: n
UNKNOWN: "
SYMBOL: ,
IDENTIFIER: total
SYMBOL:)
SYMBOL: ;
UNKNOWN:
IDENTIFIER: printf
SYMBOL: (
UNKNOWN: "
IDENTIFIER: Updated
IDENTIFIER: Count
IDENTIFIER: Again
SYMBOL: :
OPERATOR: %
IDENTIFIER: d
UNKNOWN: \
IDENTIFIER: n
UNKNOWN: "
SYMBOL: ,
IDENTIFIER: countAgain
SYMBOL:)
SYMBOL: ;
UNKNOWN:
UNKNOWN:
KEYWORD: return
NUMBER: 0
SYMBOL: ;
OUTLINE
TIMELINE
Launchpad
Live Share
Ln 77, Col 2 Spaces: 4 UTF-8 CRLF Plain Text Go Live Prettier


File Edit Selection View Go Run Terminal Help
c2
EXPLORER
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS POSTMAN CONSOLE SQL HISTORY TASK MONITOR
c2
input.txt
lex.yy.c
q1_lex
q1.exe
q1.l
q1.yy.c
q2_lex
q2.exe
q2.l
q2.yy.c
q3_lex
q3.exe
q3.l
q3.yy.c
q4_lex
q4.l
IDENTIFIER: Updated
IDENTIFIER: Total
SYMBOL: :
OPERATOR: %
IDENTIFIER: d
UNKNOWN: \
IDENTIFIER: n
UNKNOWN: "
SYMBOL: ,
IDENTIFIER: total
SYMBOL:)
SYMBOL: ;
UNKNOWN:
IDENTIFIER: printf
SYMBOL: (
UNKNOWN: "
IDENTIFIER: Updated
IDENTIFIER: Count
IDENTIFIER: Again
SYMBOL: :
OPERATOR: %
IDENTIFIER: d
UNKNOWN: \
IDENTIFIER: n
UNKNOWN: "
SYMBOL: ,
IDENTIFIER: countAgain
SYMBOL:)
SYMBOL: ;
UNKNOWN:
UNKNOWN:
KEYWORD: return
NUMBER: 0
SYMBOL: ;
UNKNOWN:
SYMBOL: }
UNKNOWN:
(ksgsaketh㉿Saketh)-[~/c2]
$
OUTLINE
TIMELINE
Launchpad
Live Share
Ln 77, Col 2 Spaces: 4 UTF-8 CRLF Plain Text Go Live Prettier

Experiment-7:

Predictive Parsing

Aim

1. To construct the **Predictive Parse Table** for a given grammar (without left recursion).

2. To implement the **Predictive Parsing algorithm** in C, where the parse table and an input string are used to check if the string can be parsed successfully.

Algorithm

Step 1: Compute FIRST sets

1. If X is a terminal, then FIRST(X) = {X}.
2. If X → ε, then ε ∈ FIRST(X).

3. If X → Y1 Y2 … Yk, then

○ Add FIRST(Y1) to FIRST(X).

○ If ε ∈ FIRST(Y1), then check Y2, and so on.

○ If ε ∈ FIRST(Yi) for all i = 1…k, then add ε to FIRST(X).

Step 2: Compute FOLLOW sets

1. If S is the start symbol, add $ to FOLLOW(S).
2. If A → αBβ, then everything in FIRST(β) except ε goes into FOLLOW(B).

3. If A → αB or A → αBβ where ε ∈ FIRST(β), then everything in FOLLOW(A) goes into FOLLOW(B). **Step 3: Construct Predictive Parse Table** For each production A → α:

1. For each terminal a ∈ FIRST(α), add A → α to M[A, a].

2. If ε ∈ FIRST(α), then for each b ∈ FOLLOW(A), add A → α to M[A, b].

3. If $ ∈ FOLLOW(A) and ε ∈ FIRST(α), then also add A → α to M[A, $].

Step 4: Predictive Parsing Algorithm

Input: Parse table M, input string w$, start symbol. Process:

1. Initialize stack with [$, S] (start symbol at top, $ at bottom).

2. Repeat until stack is empty:

○ Let X be top of stack, and a be current input symbol.
○ If X is a terminal or $:

■ If X == a, pop X and advance input.

■ Else, **error**.

○ If X is a non-terminal:
■ If M[X, a] = X → α, then pop X and push α in reverse order.

■ Else, **error**.

3. If both stack and input become $, accept.

Code

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

#define MAX_PROD 50
#define MAX_SYMBOLS 50
#define MAX_STR 100
#define MAX_STACK 100
#define EPSILON 'e'


typedef struct {
```

```
    char non_terminal[10];
    char production[MAX_STR];
} Production; typedef
struct {
    char symbols[MAX_SYMBOLS];
    int count;
} Set;
typedef struct {
    char stack[MAX_STACK][10];
    int top;
} Stack;


Production grammar[MAX_PROD];
int prod_count = 0;
Set first_sets[MAX_SYMBOLS];
Set follow_sets[MAX_SYMBOLS];
char parse_table[MAX_SYMBOLS][MAX_SYMBOLS][MAX_STR];
char terminals[MAX_SYMBOLS];
char non_terminals[MAX_SYMBOLS][10];
int term_count = 0, non_term_count = 0;
char start_symbol[10];

int is_non_terminal_char(char c);
void add_to_set(Set *set, char c); int
set_contains(Set set, char c);
void compute_first_of_string(char *str, Set *result_set);
void compute_all_first();
void compute_follow();
void construct_parse_table();
void print_sets(const char *name, Set sets[]);
void print_parse_table();
void push(Stack *s, const char *symbol);
char* pop(Stack *s);
char* peek(Stack *s);
void predictive_parse(char *input);
int find_non_terminal_index(const char *symbol); int
find_terminal_index(char c);
```

```
void process_grammar_terminals();

int main() {
    int i;
    printf("=== Predictive Parser ===\n\n");

    printf("Enter number of productions: ");
    scanf("%d", &prod_count);
    getchar();

    printf("Enter productions (format: A->aB|c, use 'e' for epsilon):\n");
    for (i = 0; i < prod_count; i++) {
        char line[MAX_STR];
fgets(line, MAX_STR, stdin);
        line[strcspn(line, "\n")] = 0;

        char *arrow = strstr(line, "->");
        if (!arrow) {
            printf("Invalid production format (missing '->'): %s\n",
line);
            i--;
            continue;
        }
        char nt_buffer[MAX_STR];
        strncpy(nt_buffer, line, arrow - line);
        nt_buffer[arrow - line] = '\0';
        sscanf(nt_buffer, "%s", grammar[i].non_terminal);
char *prod_start = arrow + 2;
        while (*prod_start && isspace((unsigned char)*prod_start))
prod_start++;
        strcpy(grammar[i].production, prod_start);

        int found = 0;
        for (int j = 0; j < non_term_count; j++) {
            if (strcmp(non_terminals[j], grammar[i].non_terminal) == 0) {
                found = 1;
                break;
```

```
            }
        }
        if (!found) {
            strcpy(non_terminals[non_term_count++],
grammar[i].non_terminal);
        }
    }

    strcpy(start_symbol, grammar[0].non_terminal);

    for(i = 0; i < MAX_SYMBOLS; i++) {
        first_sets[i].count = 0;
        follow_sets[i].count = 0;
    }
    process_grammar_terminals();
    terminals[term_count++] = '$';
compute_all_first();
    int start_symbol_index = find_non_terminal_index(start_symbol);
    if(start_symbol_index != -1) {
        add_to_set(&follow_sets[start_symbol_index], '$');
    }

    compute_follow();
construct_parse_table();
    printf("\n--- FIRST Sets ---\n");
    print_sets("FIRST", first_sets);

    printf("\n--- FOLLOW Sets ---\n");
    print_sets("FOLLOW", follow_sets);

    printf("\n--- Predictive Parse Table ---\n");
    print_parse_table();
    char input[MAX_STR];
    printf("\nEnter input string to parse (end with $): ");
    scanf(" %[^\n]", input);

    printf("\n--- Parsing Steps ---\n");
```

```
    printf("%-25s %-20s %s\n", "Stack", "Input", "Action");

printf("-------------------------------------------------------------------
----\n");
    predictive_parse(input);

    return 0;
}
int is_non_terminal_char(char c) {
    return isupper(c);
}

void process_grammar_terminals() {
    for (int i = 0; i < prod_count; i++) {
        for (int j = 0; j < strlen(grammar[i].production); j++) {
            char c = grammar[i].production[j];

            if (j + 1 < strlen(grammar[i].production) && c == 'i' &&
grammar[i].production[j+1] == 'd') {
int found = 0;
                for (int k = 0; k < term_count; k++) {
                    if (terminals[k] == 'i') {
                        found = 1;
                        break;
                    }
                }
                if (!found) {
                    terminals[term_count++] = 'i';
                }
                j++;
            } else if (c != EPSILON && c != '|' &&
!is_non_terminal_char(c) && c != '\'' && !isspace(c)) {
                int found = 0;
                for (int k = 0; k < term_count; k++) {
                    if (terminals[k] == c) {
                        found = 1;
                        break;
```

```
                    }
                }
                if (!found) {
                    terminals[term_count++] = c;
                }
            }
        }
    }
}

void add_to_set(Set *set, char c) {
    if (!set_contains(*set, c)) {
        set->symbols[set->count++] = c;
    }
}

int set_contains(Set set, char c) {
    for (int i = 0; i < set.count; i++) {
        if (set.symbols[i] == c) return 1;
    }
    return 0;
}

int find_non_terminal_index(const char *symbol) {
    for (int i = 0; i < non_term_count; i++) {
        if (strcmp(non_terminals[i], symbol) == 0) return i;
    }
    return -1;
}

int find_terminal_index(char c) {
    for (int i = 0; i < term_count; i++) {
        if (terminals[i] == c) return i;
    }
    return -1;
}
```

```
void compute_first_of_string(char *str, Set *result_set) {
    result_set->count = 0;
    int all_derive_epsilon = 1;

    for (int i = 0; i < strlen(str); ) {
        if (i + 1 < strlen(str) && str[i] == 'i' && str[i+1] == 'd') {
            add_to_set(result_set, 'i');
            all_derive_epsilon = 0;
            break;
        }

        if (!is_non_terminal_char(str[i])) {
            add_to_set(result_set, str[i]);
            all_derive_epsilon = 0;
            break;
        }

        char nt_symbol[10] = {str[i], '\0'};
        if (i + 1 < strlen(str) && str[i + 1] == '\'') {
            nt_symbol[1] = '\'';
            nt_symbol[2] = '\0';
            i += 2;
        } else {
            i++;
        }

        int nt_index = find_non_terminal_index(nt_symbol);
        if (nt_index == -1) continue;

        int has_epsilon = 0;
        for (int j = 0; j < first_sets[nt_index].count; j++) {
            char symbol = first_sets[nt_index].symbols[j];
if (symbol == EPSILON) {
                has_epsilon = 1;
            } else {
                add_to_set(result_set, symbol);
```

```
            }
        }

        if (!has_epsilon) {
            all_derive_epsilon = 0;
            break;
        }
    }
    if (all_derive_epsilon) {
        add_to_set(result_set, EPSILON);
    }
}

void compute_all_first() {
    int changed;
do {
        changed = 0;
        for (int i = 0; i < prod_count; i++) {
            char *nt = grammar[i].non_terminal;
            int nt_index = find_non_terminal_index(nt);
            if(nt_index == -1) continue;

            int old_count = first_sets[nt_index].count;

            char prod_copy[MAX_STR];
            strcpy(prod_copy, grammar[i].production);

            char *alt = strtok(prod_copy, "|");
            while (alt != NULL) {
                while(*alt && isspace((unsigned char)*alt)) alt++;
                char* end = alt + strlen(alt) - 1;
                while(end >= alt && isspace((unsigned char)*end)) {
                    *end = '\0';
                    end--;
                }

                if (strcmp(alt, "e") == 0) {
```

```
                    add_to_set(&first_sets[nt_index], EPSILON);
                } else if (*alt != '\0') {
                    Set first_of_alt;
                    compute_first_of_string(alt, &first_of_alt);
                    for (int j = 0; j < first_of_alt.count; j++) {
                        add_to_set(&first_sets[nt_index],
first_of_alt.symbols[j]);
                    }
                }
                alt = strtok(NULL, "|");
            }

            if (first_sets[nt_index].count != old_count) {
                changed = 1;
            }
        }
    } while (changed);
}

void compute_follow() {
    int changed;
    do {
        changed = 0;
        for (int i = 0; i < prod_count; i++) {
            char *lhs_nt = grammar[i].non_terminal;
            int lhs_nt_index = find_non_terminal_index(lhs_nt);
            if(lhs_nt_index == -1) continue;

            char prod_copy[MAX_STR];
            strcpy(prod_copy, grammar[i].production);

            char* alt = strtok(prod_copy, "|");
while(alt != NULL) {
                for (int j = 0; j < strlen(alt); ) {
                    if (is_non_terminal_char(alt[j])) {
                        char current_nt[10] = {alt[j], '\0'};
                        int sym_len = 1;
```

```
                        if (j + 1 < strlen(alt) && alt[j+1] == '\'') {
                                current_nt[1] = '\'';
                current_nt[2] = '\0';
                            sym_len = 2;
                        }

                        int current_nt_index =
find_non_terminal_index(current_nt);
                        if(current_nt_index == -1) { j += sym_len;
continue; }


                        int old_count =
follow_sets[current_nt_index].count;

                        char *beta = alt + j + sym_len;
                        Set first_of_beta;
                        compute_first_of_string(beta, &first_of_beta);

                        int beta_has_epsilon = 0;
                        for (int k = 0; k < first_of_beta.count; k++) {
                            if (first_of_beta.symbols[k] == EPSILON) {
beta_has_epsilon = 1;
                            } else {
                                add_to_set(&follow_sets[current_nt_index],
first_of_beta.symbols[k]);
                            }
                        }

                        if (beta_has_epsilon || strlen(beta) == 0) {
                            for (int k = 0; k <
follow_sets[lhs_nt_index].count; k++) {
                                add_to_set(&follow_sets[current_nt_index],
follow_sets[lhs_nt_index].symbols[k]);
                            }
                        }

                        if(follow_sets[current_nt_index].count !=
```

```
old_count) {
                            changed = 1;
                        }
                        j += sym_len;
                    } else {
j++;
                    }
                }
                alt = strtok(NULL, "|");
            }
        }
    } while (changed);
}

void construct_parse_table() {
    for (int i = 0; i < non_term_count; i++) {
        for (int j = 0; j < term_count; j++) {
strcpy(parse_table[i][j], "-");
        }
    }

    for (int i = 0; i < prod_count; i++) {
        char *nt = grammar[i].non_terminal;
        int row = find_non_terminal_index(nt);
if (row == -1) continue;

        char prod_copy[MAX_STR];
        strcpy(prod_copy, grammar[i].production);

        char *alt = strtok(prod_copy, "|");
        while (alt != NULL) {
            while(*alt && isspace((unsigned char)*alt)) alt++;
            char* end = alt + strlen(alt) - 1;
            while(end >= alt && isspace((unsigned char)*end)) {
                *end = '\0';
                end--;
            }
```

```
            Set first_of_alt;
            compute_first_of_string(alt, &first_of_alt);

            for (int j = 0; j < first_of_alt.count; j++) {
                if (first_of_alt.symbols[j] != EPSILON) {
                    int col =
find_terminal_index(first_of_alt.symbols[j]);
if (col != -1) {
                        sprintf(parse_table[row][col], "%s->%s", nt, alt);
                    }
                }
            }

            if (set_contains(first_of_alt, EPSILON)) {
int nt_index = find_non_terminal_index(nt);
                for (int j = 0; j < follow_sets[nt_index].count; j++) {
                    char terminal = follow_sets[nt_index].symbols[j];
                    int col = find_terminal_index(terminal);
                    if (col != -1) {
                        sprintf(parse_table[row][col], "%s->%s", nt, alt);
                    }
                }
            }
            alt = strtok(NULL, "|");
        }
    }
}

void print_sets(const char *name, Set sets[]) {
    for (int i = 0; i < non_term_count; i++) {
        printf("%s(%s): { ", name, non_terminals[i]);
        int nt_index = find_non_terminal_index(non_terminals[i]);
        if (nt_index == -1) continue;

        for (int j = 0; j < sets[nt_index].count; j++) {
            printf("%c ", sets[nt_index].symbols[j]);
```

```
        }
        printf("}\n");
    }
}
void print_parse_table() {
    printf("\t");
    for (int i = 0; i < term_count; i++) {
        printf("%-8c", terminals[i]);
    }
    printf("\n");

    for (int i = 0; i < non_term_count; i++) {
        printf("%-8s", non_terminals[i]);
        for (int j = 0; j < term_count; j++) {
            printf("%-8s", parse_table[i][j]);
        }
        printf("\n");
    }
}

void push(Stack *s, const char *symbol) {
    if (s->top < MAX_STACK - 1) {
        strcpy(s->stack[++s->top], symbol);
}
}
char* pop(Stack *s) {
    if (s->top >= 0) return s->stack[s->top--];
    return NULL;
}
char* peek(Stack *s) {
    if (s->top >= 0) return s->stack[s->top];
    return NULL;
}

void predictive_parse(char *input) {
    Stack s;
```

```
    s.top = -1;
push(&s, "$");
    push(&s, start_symbol);

    char processed_input[MAX_STR] = "";
    int p_idx = 0;
    for(int i = 0; i < strlen(input); i++) {
        if(isspace(input[i])) continue;
        if(i + 1 < strlen(input) && input[i] == 'i' && input[i+1] == 'd')
{
            processed_input[p_idx++] = 'i';
            i++;
        } else {
            processed_input[p_idx++] = input[i];
        }
    }
    processed_input[p_idx] = '\0';

    int input_idx = 0;
    while (s.top >= 0) {
        char stack_str[MAX_STR * 2] = "";
        for(int i = s.top; i >= 0; i--) {
strcat(stack_str, s.stack[i]);
            strcat(stack_str, " ");
        }

        printf("%-25s %-20s ", stack_str, processed_input + input_idx);

        char *stack_top = peek(&s);
        char current_input = processed_input[input_idx];

        if (strcmp(stack_top, "$") == 0 && current_input == '$') {
            printf("Accept");
            break;
}

        if (strcmp(stack_top, "id") == 0 && current_input == 'i') {
```

```
            printf("Match 'id'");
            pop(&s);
            input_idx++;
        } else if (strlen(stack_top) == 1 && stack_top[0] ==
current_input) {
            printf("Match '%c'", current_input);
            pop(&s);
            input_idx++;
        } else if (is_non_terminal_char(stack_top[0])) {
            int row = find_non_terminal_index(stack_top);
            int col = find_terminal_index(current_input);
            if (row == -1 || col == -1 || strcmp(parse_table[row][col],
"-") == 0) {
                printf("Error: No rule for M[%s, %c]", stack_top,
current_input);
                break;
            }

            char* production_rule = parse_table[row][col];
            printf("Apply %s", production_rule);
            pop(&s);

            char *rhs = strchr(production_rule, '>') + 1;
            if (strcmp(rhs, "e") != 0) {
                char symbols[20][10];
int sym_count = 0;
                for(int k=0; k < strlen(rhs); ) {
                    if (k + 1 < strlen(rhs) && rhs[k] == 'i' && rhs[k+1]
== 'd') {
                        strcpy(symbols[sym_count++], "id");
k += 2;
                    } else if(is_non_terminal_char(rhs[k])) {
                        symbols[sym_count][0] = rhs[k];
                        if(k+1 < strlen(rhs) && rhs[k+1] == '\'') {
                            symbols[sym_count][1] = '\'';
                            symbols[sym_count][2] = '\0';
```

```
                            k += 2;
} else {
                            symbols[sym_count][1] = '\0';
                            k++;
                        }
                        sym_count++;
                    } else {
                        symbols[sym_count][0] = rhs[k];
symbols[sym_count][1] = '\0';
                        sym_count++;
                        k++;
                    }
                }
                for(int k = sym_count - 1; k >= 0; k--) {
                    push(&s, symbols[k]);
}
            }
        } else {
            printf("Error: Mismatch. Stack: %s, Input: %c", stack_top,
current_input);
            break;
        }
        printf("\n");
    }

    printf("\n");
    if (strcmp(peek(&s), "$") == 0 && processed_input[input_idx] == '$') {
        printf("\nInput string is ACCEPTED.\n");
    } else {
        printf("\nInput string is REJECTED.\n");
    }
}
```

```
Enter number of productions: 5
Enter productions (format: A->aB|c, use 'e' for epsilon):
E -> TE'
E' -> +TE' | e
T -> FT'
T' -> *FT' | e
F -> id | (E)

--- FIRST Sets ---
FIRST(E): { i ( }
FIRST(E'): { + e }
FIRST(T): { i ( }
FIRST(T'): { * e }
FIRST(F): { i ( }

--- FOLLOW Sets ---
FOLLOW(E): { $ ) }
FOLLOW(E'): { $   ) }
FOLLOW(T): { + $   ) }
FOLLOW(T'): { + $   ) }
FOLLOW(F): { * + $   ) }

--- Predictive Parse Table ---
        +       *       i       (       )       $
E       -       -       E->TE'  E->TE'  -       -
E'      E'->+TE'-       -       -       E'->e   E'->e
T       -       -       T->FT'  T->FT'  -       -
T'      T'->e   T'->*FT'-       -       T'->e   T'->e
F       -       -       F->id   F->(E)  -       -

Enter input string to parse (end with $): id+id*id$

--- Parsing Steps ---
Stack                   Input               Action
------------------------------------------------------------------------
E $                     i+i*i$              Apply E->TE'
T E' $                  i+i*i$              Apply T->FT'
F T' E' $               i+i*i$              Apply F->id
id T' E' $              i+i*i$              Match 'id'
T' E' $                 +i*i$               Apply T'->e
E' $                    +i*i$               Apply E'->+TE'
+ T E' $                +i*i$               Match '+'
T E' $                  i*i$                Apply T->FT'
F T' E' $               i*i$                Apply F->id
id T' E' $              i*i$                Match 'id'
T' E' $                 *i$                 Apply T'->*FT'
* F T' E' $             *i$                 Match '*'
F T' E' $               i$                  Apply F->id
id T' E' $              i$                  Match 'id'
T' E' $                 $                   Apply T'->e
E' $                    $                   Apply E'->e
$                       $                   Accept

Input string is ACCEPTED.
student@AB1605B049:~/compiler$
```

Experiment-8:

Operator Precedence Parsing

Aim

1. To construct the **operator precedence table** for a given operator grammar.

2. To use the operator precedence table to **parse a given input string** using operator precedence parsing.

Algorithm

Step 1: Construct Operator Precedence Table

1. Start with the given **operator grammar** (must not have ε-productions or ambiguous rules).

2. Identify **terminals (operators and operands)** and non-terminals.
3. Compute **operator precedence relations**:

○ **Equal precedence (=):**

If there exists a production $A \rightarrow \alpha\ a\ b\ \beta$, then $a = b$

(where a and b are terminals that appear adjacent).

○ **Less precedence (<·):**

If there exists $A \rightarrow \alpha\ a\ B\ \beta$, then for every terminal $b \in FIRST(B)$, we define $a <\cdot\ b$.

○ **Greater precedence (·>):**

If there exists $A \rightarrow \alpha\ B\ a\ \beta$, then for every terminal $b \in LAST(B)$, we define $b \cdot> a$.

4. Construct the **operator precedence table** using <·, =, and ·> relations among terminals.

Step 2: Operator Precedence Parsing Algorithm

Input: Operator precedence table, input string w$.

Process:

1. Initialize stack with $.

2. Read next input symbol a.

3. Repeat until both stack and input contain only $:

○ Let s be the topmost terminal on stack.

○ Compare precedence relation between s and a using the table.

○ If s <· a or s = a, then **shift** a onto stack and read next input symbol.

○ If s ·> a, then **reduce**:

■ Find the handle (rightmost substring on stack matching a production body).

■ Replace handle by its corresponding non-terminal.

○ If no valid relation exists, report **error**.

4. If stack reduces to S, then accept. Otherwise, reject.

Code

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

#define MAX_PROD 50
#define MAX_SYMBOLS 50
#define MAX_STR 100
#define MAX_STACK 100

typedef struct {
```

```
    char non_terminal[10];
    char production[MAX_STR];
} Production;

typedef struct {
    char symbols[MAX_SYMBOLS];
    int count;
} Set;
```

```
typedef struct {
    char stack[MAX_STACK];
int top;
} Stack;

Production grammar[MAX_PROD];
int prod_count = 0;
Set leading_sets[MAX_SYMBOLS];
Set trailing_sets[MAX_SYMBOLS];
char precedence_table[MAX_SYMBOLS][MAX_SYMBOLS];
char terminals[MAX_SYMBOLS];
char non_terminals[MAX_SYMBOLS][10];
int term_count = 0, non_term_count = 0;
char start_symbol[10];

void add_to_set(Set *set, char c); int
set_contains(Set set, char c);
void compute_leading();
void compute_trailing();
void construct_precedence_table();
void print_sets(const char *name, Set sets[]);
void print_precedence_table();
void push(Stack *s, char symbol); char
pop(Stack *s);
char peek_terminal(Stack *s);
void operator_precedence_parse(char *input);
int find_non_terminal_index(const char *symbol);
int find_terminal_index(char c);
void process_grammar_symbols();

int main() {
    int i;
    printf("=== Operator Precedence Parser ===\n");
    printf("NOTE: Assumes a valid operator grammar (no adjacent
non-terminals, no epsilon productions).\n\n");

    int num_rules;
    printf("Enter number of production rules (e.g., E->E+T|T is one rule):
```

```
");
    scanf("%d", &num_rules);
    getchar();

    printf("Enter productions (e.g., E->E+T|T or F->id):\n");
    for (i = 0; i < num_rules; i++) {
        char line[MAX_STR];
        fgets(line, MAX_STR, stdin);
        line[strcspn(line, "\n")] = 0;

        char *arrow = strstr(line, "->");
if (!arrow) {
            printf("Invalid production format (missing '->'): %s\n",
line);
            i--;
            continue;
        }

        char non_terminal[10];
        strncpy(non_terminal, line, arrow - line);
        non_terminal[arrow - line] = '\0';
        sscanf(non_terminal, " %s ", non_terminal);

        char *productions = arrow + 2;
        char *prod_token = strtok(productions, "|");
while (prod_token != NULL) {
            if (prod_count >= MAX_PROD) {
                printf("Exceeded maximum production limit.\n");
                break;
            }
            strcpy(grammar[prod_count].non_terminal, non_terminal);

            char *start = prod_token;
            while (*start && isspace((unsigned char)*start)) start++;
            char *end = start + strlen(start) - 1;
            while (end > start && isspace((unsigned char)*end)) {
                *end = '\0';
end--;

            }
```

```
            strcpy(grammar[prod_count].production, start);
            prod_count++;
            prod_token = strtok(NULL, "|");
        }
    }

    strcpy(start_symbol, grammar[0].non_terminal);

    process_grammar_symbols();

    int is_defined[MAX_SYMBOLS] = {0};
for(i = 0; i < prod_count; i++) {
        int nt_index = find_non_terminal_index(grammar[i].non_terminal);
        if (nt_index != -1) {
            is_defined[nt_index] = 1;
        }
    }

    int grammar_ok = 1;
    for(i = 0; i < non_term_count; i++) {
        if (!is_defined[i]) {
            if (grammar_ok) {
                 printf("\n--- Grammar Warnings ---\n");
            }
            printf("Warning: Non-terminal '%s' is used but has no
productions.\n", non_terminals[i]);
            grammar_ok = 0;
        }
    }
    if(!grammar_ok) {
        printf("Please check your grammar. The results below may be
incorrect.\n");
}

    terminals[term_count++] = '$';

    compute_leading();
compute_trailing();
```

```
    construct_precedence_table();

    printf("\n--- LEADING Sets ---\n");
    print_sets("LEADING", leading_sets);

    printf("\n--- TRAILING Sets ---\n");
print_sets("TRAILING", trailing_sets);

    printf("\n--- Operator Precedence Table ---\n");
    print_precedence_table();

    char input[MAX_STR];
    printf("\nEnter input string to parse (end with $): ");
scanf(" %[^\n]", input);

    printf("\n--- Parsing Steps ---\n");
    printf("%-25s %-20s %s\n", "Stack", "Input", "Action");

printf("-----------------------------------------------------------------
----\n");
    operator_precedence_parse(input);

    return 0;
}

void process_grammar_symbols() {
    for (int i = 0; i < prod_count; i++) {
int found_nt = 0;
        for (int j = 0; j < non_term_count; j++) {
            if (strcmp(non_terminals[j], grammar[i].non_terminal) == 0) {
                found_nt = 1;
                break;
            }
        }
        if (!found_nt) {
            strcpy(non_terminals[non_term_count++],
grammar[i].non_terminal);
        }
```

```
        char *prod = grammar[i].production;
        for (int j = 0; j < strlen(prod);) {
            char c = prod[j];
            if (isupper(c)) {
                char current_nt_str[2] = {c, '\0'};
                int found = 0;
                for (int k = 0; k < non_term_count; k++) {
                    if (strcmp(non_terminals[k], current_nt_str) == 0) {
                        found = 1;
                        break;
                    }
                }
                if (!found) {
                    strcpy(non_terminals[non_term_count++],
current_nt_str);
                }
                j++;
            } else if (j + 1 < strlen(prod) && ( (c == 'i' && prod[j+1] ==
'd') || (c == 'I' && prod[j+1] == 'D') )) {
                 int found_t = 0;
                 for (int k = 0; k < term_count; k++) { if (terminals[k]
== 'i') { found_t = 1; break; } }
                 if (!found_t) { terminals[term_count++] = 'i'; }
                 j += 2;
            }
            else {
                int found_t = 0;
                for (int k = 0; k < term_count; k++) {
if (terminals[k] == c) {
                        found_t = 1;
                        break;
                    }
                }
                if (!found_t) {
                    terminals[term_count++] = c;
}
                j++;
            }
```

```
        }
    }
}

void add_to_set(Set *set, char c) {
    if (!set_contains(*set, c)) {
        set->symbols[set->count++] = c;
    }
}

int set_contains(Set set, char c) {
    for (int i = 0; i < set.count; i++) {
        if (set.symbols[i] == c) return 1;
    }
    return 0;
}

int find_non_terminal_index(const char *symbol) {
    for (int i = 0; i < non_term_count; i++) {
        if (strcmp(non_terminals[i], symbol) == 0) return i;
    }
    return -1;
}

int find_terminal_index(char c) {
    for (int i = 0; i < term_count; i++) {
        if (terminals[i] == c) return i;
    }
    return -1;
}

void compute_leading() {
    int changed;
    for (int i = 0; i < prod_count; i++) {
        int nt_index = find_non_terminal_index(grammar[i].non_terminal);
        if (nt_index == -1) continue;
        char *prod = grammar[i].production;

        if (strlen(prod) >= 2 && ((prod[0] == 'i' && prod[1] == 'd') ||
```

```
(prod[0] == 'I' && prod[1] == 'D'))) {
            add_to_set(&leading_sets[nt_index], 'i');
        }
        else if (!isupper(prod[0])) {
            add_to_set(&leading_sets[nt_index], prod[0]);
        }
        else if (strlen(prod) > 1 && !isupper(prod[1])) {
            add_to_set(&leading_sets[nt_index], prod[1]);
        }
    }

    do {
        changed = 0;
        for (int i = 0; i < prod_count; i++) {
            int lhs_index =
find_non_terminal_index(grammar[i].non_terminal);
            char *prod = grammar[i].production;
if (strlen(prod) > 0 && isupper(prod[0])) {
                char rhs_nt_str[2] = {prod[0], '\0'};
                int rhs_index = find_non_terminal_index(rhs_nt_str);
                if (rhs_index == -1) continue;

                int old_count = leading_sets[lhs_index].count;
                for (int j = 0; j < leading_sets[rhs_index].count; j++) {
add_to_set(&leading_sets[lhs_index],
leading_sets[rhs_index].symbols[j]);
                }
                if (leading_sets[lhs_index].count != old_count) {
                    changed = 1;
                }
            }
        }
    } while (changed);
}

void compute_trailing() {
    int changed;
    for (int i = 0; i < prod_count; i++) {
        int nt_index = find_non_terminal_index(grammar[i].non_terminal);
```

```
        if (nt_index == -1) continue;
        char *prod = grammar[i].production;
        int len = strlen(prod);
        if(len == 0) continue;

        if (len >= 2 && ((prod[len-2] == 'i' && prod[len-1] == 'd') ||
(prod[len-2] == 'I' && prod[len-1] == 'D'))) {
            add_to_set(&trailing_sets[nt_index], 'i');
        }
        else if (!isupper(prod[len - 1])) {
            add_to_set(&trailing_sets[nt_index],  prod[len  -  1]);
}
        else if (len > 1 && !isupper(prod[len - 2])) {
            add_to_set(&trailing_sets[nt_index], prod[len - 2]);
        }
    }

    do {
        changed = 0;
        for (int i = 0; i < prod_count; i++) {
            int lhs_index =
find_non_terminal_index(grammar[i].non_terminal);
            char *prod = grammar[i].production;
            int len = strlen(prod);
            if (len > 0 && isupper(prod[len - 1])) {
char rhs_nt_str[2] = {prod[len - 1], '\0'};
                int rhs_index = find_non_terminal_index(rhs_nt_str);
                if (rhs_index == -1) continue;

                int old_count = trailing_sets[lhs_index].count;
                for (int j = 0; j < trailing_sets[rhs_index].count; j++) {
                    add_to_set(&trailing_sets[lhs_index],
trailing_sets[rhs_index].symbols[j]);
                }
                if (trailing_sets[lhs_index].count != old_count) {
                    changed = 1;
                }
            }
```

```
        }
    } while (changed);
}

void construct_precedence_table() {
    for (int i = 0; i < term_count; i++) {
for (int j = 0; j < term_count; j++) {
            precedence_table[i][j] = ' ';
        }
    }

    for (int i = 0; i < prod_count; i++) {
        char *prod = grammar[i].production;
char symbols[MAX_STR][10];
        int sym_count = 0;

        for (int j = 0; j < strlen(prod);) {
            if (j + 1 < strlen(prod) && ((prod[j] == 'i' && prod[j + 1] ==
'd') || (prod[j] == 'I' && prod[j + 1] == 'D'))) {
                strcpy(symbols[sym_count++], "i");
j += 2;
            } else if (isupper(prod[j])) {
                symbols[sym_count][0] = prod[j];
                symbols[sym_count][1] = '\0';
                sym_count++;
                j++;
            } else if (!isspace(prod[j])) {
symbols[sym_count][0] = prod[j];
                symbols[sym_count][1] = '\0';
                sym_count++;
                j++;
            } else {
                j++;
            }
        }

        for (int k = 0; k < sym_count - 1; k++) {
            char *s1 = symbols[k];
            char *s2 = symbols[k + 1];
```

```
            int s1_is_nt = find_non_terminal_index(s1) != -1;
int s2_is_nt = find_non_terminal_index(s2) != -1;
```

```
            if (!s1_is_nt && !s2_is_nt) {

precedence_table[find_terminal_index(s1[0])][find_terminal_index(s2[0])] =
'=';
            }
            if (k < sym_count - 2) {
                char *s3 = symbols[k + 2];
                if (!s1_is_nt && find_non_terminal_index(s3) == -1 &&
s2_is_nt) {

precedence_table[find_terminal_index(s1[0])][find_terminal_index(s3[0])] =
'=';
                }
            }
            if (!s1_is_nt && s2_is_nt) {
                int nt_index = find_non_terminal_index(s2);
                if (nt_index == -1) continue;
                for (int l = 0; l < leading_sets[nt_index].count; l++) {

precedence_table[find_terminal_index(s1[0])][find_terminal_index(leading_s
ets[nt_index].symbols[l])] = '<';
                }
            }
            if (s1_is_nt && !s2_is_nt) {
                int nt_index = find_non_terminal_index(s1);
                if (nt_index == -1) continue;
                for (int l = 0; l < trailing_sets[nt_index].count; l++) {

precedence_table[find_terminal_index(trailing_sets[nt_index].symbols[l])][
find_terminal_index(s2[0])] = '>';
                }
            }
        }
    }
```

```
    int dollar_idx = find_terminal_index('$');
    if (dollar_idx == -1) return;

    int start_idx = find_non_terminal_index(start_symbol);
    if(start_idx == -1) return;

    for (int j = 0; j < leading_sets[start_idx].count; j++) {

precedence_table[dollar_idx][find_terminal_index(leading_sets[start_idx].s
ymbols[j])] = '<';
    }

    for (int j = 0; j < trailing_sets[start_idx].count; j++) {

precedence_table[find_terminal_index(trailing_sets[start_idx].symbols[j])]
[dollar_idx] = '>';
    }
}



void print_sets(const char *name, Set sets[]) {
    for (int i = 0; i < non_term_count; i++) {
printf("%s(%s): { ", name, non_terminals[i]);
        int nt_index = find_non_terminal_index(non_terminals[i]);
        if (nt_index == -1) continue;

        for (int j = 0; j < sets[nt_index].count; j++) {
            printf("%c ", sets[nt_index].symbols[j]);
        }
        printf("}\n");
    }
}

void print_precedence_table() {
    printf("%-4c", ' ');
    for (int i = 0; i < term_count; i++) {
printf("%-4c", terminals[i]);
    }
```

```
    printf("\n");

    for (int i = 0; i < term_count; i++) {
        printf("%-4c", terminals[i]);
        for (int j = 0; j < term_count; j++) {
printf("%-4c", precedence_table[i][j]);
        }
        printf("\n");
    }
}

void push(Stack *s, char symbol) {
    if (s->top < MAX_STACK - 1) {
        s->stack[++s->top] = symbol;
    }
}
char pop(Stack *s) {
    if (s->top >= 0) return s->stack[s->top--];
    return '\0';
}
char peek_terminal(Stack *s) {
    for (int i = s->top; i >= 0; i--) {
        if (find_terminal_index(s->stack[i]) != -1) {
            return s->stack[i];
}
    }
    return '\0';
}


void operator_precedence_parse(char *input) {
Stack s;
    s.top = -1;
    push(&s, '$');

    char processed_input[MAX_STR] = "";
    int p_idx = 0;
    for(int i = 0; i < strlen(input); ) {
if(isspace(input[i])) { i++; continue; }
```

```
        if(i + 1 < strlen(input) && ( (input[i] == 'I' && input[i+1] ==
'D') || (input[i] == 'i' && input[i+1] == 'd') )) {
            processed_input[p_idx++] = 'i';
            i += 2;
        } else {
            processed_input[p_idx++] = input[i];
i++;
        }
    }
    processed_input[p_idx] = '\0';

    int input_idx = 0;
    int accepted = 0;

    while (1) {
        char stack_str[MAX_STR] = "";
        for (int i = 0; i <= s.top; i++) {
            stack_str[i] = s.stack[i];
}
        stack_str[s.top + 1] = '\0';

        char current_input_char = processed_input[input_idx];
        if (current_input_char == '\0') {
             current_input_char = '$';
        }

        printf("%-25s %-20s ", stack_str, processed_input + input_idx);

        char stack_top_term = peek_terminal(&s);

        if (stack_top_term == '$' && current_input_char == '$') {
             printf("Accept");
accepted = 1;
             break;
        }

        int row = find_terminal_index(stack_top_term);
        int col = find_terminal_index(current_input_char);
```

```
        if (row == -1 || col == -1 || stack_top_term == '\0') {
            printf("Error: Invalid parsing state (Stack: '%c', Input:
'%c').", stack_top_term, current_input_char);
            break;
        }

        char relation = precedence_table[row][col];

        if (relation == '<' || relation == '=') {
            printf("Shift %c\n", current_input_char);
            push(&s, current_input_char);
            if(input_idx < strlen(processed_input)) {
input_idx++;
            }
        } else if (relation == '>') {
            printf("Reduce\n");
            char last_popped;
            do {
                last_popped = pop(&s);
if(last_popped == '\0') {
                    break;
                }
            } while (peek_terminal(&s) != '\0' &&
(find_terminal_index(last_popped) == -1 ||
precedence_table[find_terminal_index(peek_terminal(&s))][find_terminal_ind
ex(last_popped)] != '<') );
} else {
            printf("Error: No relation between %c and %c", stack_top_term,
current_input_char);
            break;
        }
    }

    printf("\n\n");
    if (accepted) {
        printf("Input string is ACCEPTED.\n");
    } else {
        printf("Input string is REJECTED.\n");
```

```
    }
}
```

Output

```
student@AB1605B049:~/compiler$ gcc lq3_2.c -o bin4
student@AB1605B049:~/compiler$ ./bin4
=== Operator Precedence Parser ===
NOTE: Assumes a valid operator grammar (no adjacent non-terminals, no epsilon productions).

Enter number of production rules (e.g., E->E+T|T is one rule): 3
Enter productions (e.g., E->E+T|T or F->id):
E->E+T|T
T->T*F|F
F->id

--- LEADING Sets ---
LEADING(E): { + * i }
LEADING(T): { * i }
LEADING(F): { i }

--- TRAILING Sets ---
TRAILING(E): { + * i }
TRAILING(T): { * i }
TRAILING(F): { i }

--- Operator Precedence Table ---
    +   *   i   $
+   >   <   <   >
*   >   >   <   >
i   >   >       >
$   <   <   <

Enter input string to parse (end with $): id+id*id$

--- Parsing Steps ---
Stack                   Input               Action
------------------------------------------------------------------------
$                       i+i*i$              Shift i
$i                      +i*i$               Reduce
$                       +i*i$               Shift +
$+                      i*i$                Shift i
$+i                     *i$                 Reduce
$+                      *i$                 Shift *
$+*                     i$                  Shift i
$+*i                    $                   Reduce
$+*                     $                   Reduce
$+                      $                   Reduce
$                       $                   Accept

Input string is ACCEPTED.
student@AB1605B049:~/compiler$
```

Experiment-9(a)

Construct a Predictive Parse Table Aim

The objective is to write a C program that automatically generates a predictive (or LL(1)) parsing table for a given context-free grammar. The program must first compute the FIRST and FOLLOW sets for all non-terminals, which are then used to systematically fill the table entries.

Prerequisite: The input grammar must be free of left recursion and must be left-factored to be suitable for LL(1) parsing.

Algorithm

The process can be broken down into three main stages:

Stage 1: Compute FIRST Sets

For each symbol X (terminal or non-terminal) in the grammar, FIRST(X) is the set of terminals that can begin a string derived from X. If X can derive the empty string (ε), then ε is also in FIRST(X).

1. If X is a terminal, then FIRST(X) = {X}.
2. If X is a non-terminal and there is a production X -> ε, then add ε to FIRST(X).
3. If X is a non-terminal and there is a production X -> $Y_1Y_2...Y_k$:
    - Add FIRST(Y_1) (excluding ε) to FIRST(X).
    - If FIRST(Y_1) contains ε, then add FIRST(Y_2) (excluding ε) to FIRST(X).
    - Continue this process: if FIRST(Y_1)...FIRST(Y_{i-1}) all contain ε, add FIRST(Y_i) (excluding ε) to FIRST(X).
    - If FIRST(Y_1)...FIRST(Y_k) all contain ε, then add ε to FIRST(X).
4. Repeat step 3 until no more changes can be made to any FIRST set.

Stage 2: Compute FOLLOW Sets

For each non-terminal A, FOLLOW(A) is the set of terminals that can appear immediately to the right of A in some sentential form. The end-of-input marker, $, can also be in FOLLOW(A).

1. Place $ in FOLLOW(S), where S is the start symbol.
2. For every production A -> αBβ:
    - Add everything in FIRST(β), except for ε, to FOLLOW(B).

3. For every production A -> αB, or A -> αBβ where FIRST(β) contains ε:
    - Add everything in FOLLOW(A) to FOLLOW(B).
4. Repeat steps 2 and 3 until no more changes can be made to any FOLLOW set.

Stage 3: Construct the Predictive Parse Table M

Create a table M[A, a], where A is a non-terminal and a is a terminal or $.

1. For each production A -> α in the grammar, perform the following two steps: a. For each terminal a in FIRST(α), add the production A -> α to the table entry M[A, a]. b. If ε is in FIRST(α), then for each terminal b in FOLLOW(A), add the production A -> α to M[A, b]. (If $ is in FOLLOW(A), add A -> α to M[A, $] as well).
2. All table entries that are not filled by the above rules are considered error entries.

Experiment-9(b)

Implement the Predictive Parsing Algorithm Aim

The objective is to implement a C program that functions as a table-driven predictive parser. This program will take a pre-computed predictive parse table and an input string as its inputs. It will then use a stack-based algorithm to determine whether the input string can be generated by the grammar, effectively parsing the string.

Algorithm

The parser uses an input buffer, a stack, and the parsing table M.

1. **Initialization:**
    - Initialize a stack and push the end-of-input marker $ onto it, followed by the grammar's start symbol S. The stack top is now S.
    - Let ip be a pointer to the first symbol of the input string w. (Assume $ is appended to w).
2. **Main Parsing Loop:**
    - Let X be the symbol at the top of the stack and a be the symbol pointed to by ip.
    - Repeat the following steps until the stack contains only $ (X = $):

a. **If X is a terminal:** * If X matches a, pop X from the stack and advance the input pointer ip to the next symbol. * If X does not match a, declare a syntax error and terminate.

b. **If X is a non-terminal:** * Consult the parsing table entry M[X, a]. * **If M[X, a] contains a production X -> $Y_1Y_2...Y_k$:** * Pop X from the stack. * Push the symbols Y_k, Y_{k-1}, ..., Y_1 onto the stack. (Note the reverse order: Y_1 will be at the top). * If the production is X -> ε (an empty RHS), simply pop X and do not push anything. * **If M[X, a] is an error entry:** * Declare a syntax error and terminate.

3. **Termination:**

- If the loop finishes (the stack top is $) and the input pointer also points to $, the input string is successfully parsed.
- If the loop terminates for any other reason, the string contains a syntax error.

Code

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

#define MAX_PRODUCTIONS 64
#define MAX_RHS_LENGTH 64
#define MAX_SYMBOLS 64
#define MAX_ITEMS 256
#define MAX_STATES 256
#define MAX_SET_SIZE 64

typedef struct { char
lhs;
char rhs[MAX_RHS_LENGTH];
} Production;

typedef struct { int
prodIndex; int dot;
} Item;

typedef struct {
Item items[MAX_ITEMS]; int
numItems;
} ItemSet;

typedef enum {
ACTION_ERROR,
ACTION_SHIFT,
ACTION_REDUCE,
ACTION_ACCEPT
} ActionType;

typedef struct {
ActionType type; int
target;
} ActionEntry;
Production productions[MAX_PRODUCTIONS];
```

```
int numProductions = 0; char
originalStartSymbol;
char augmentedStartSymbol = '\'';

char nonTerminals[MAX_SYMBOLS];
int numNonTerminals = 0;
char terminals[MAX_SYMBOLS];
int numTerminals = 0;
char alphabet[MAX_SYMBOLS];
int numAlphabet = 0;

char FIRST[256][MAX_SET_SIZE];
int FIRST_len[256] = {0};
char FOLLOW[256][MAX_SET_SIZE];
int FOLLOW_len[256] = {0};
int hasEpsilon[256] = {0};

ItemSet states[MAX_STATES];
int numStates = 0;
int transitions[MAX_STATES][MAX_SYMBOLS];

ActionEntry ACTION_TABLE[MAX_STATES][MAX_SYMBOLS];
int GOTO_TABLE[MAX_STATES][MAX_SYMBOLS];
int conflict_found = 0;

int contains_char(const char *s, char c) { return
strchr(s, c) != NULL;
}

void add_char_unique(char *s, int *len, char c) {
if (!contains_char(s, c)) {
s[*len] = c;
s[*len + 1] = '\0'; (*len)++;
}
}

int is_nonterminal(char c) { return
(c >= 'A' && c <= 'Z');
}

int is_terminal(char c) {
return c != '\0' && !is_nonterminal(c);
```

```
}

int get_symbol_index(char c) {
for (int i = 0; i < numAlphabet; ++i) { if
(alphabet[i] == c) return i;
}
return -1;
}

void read_grammar() {
printf("Enter the number of productions: "); scanf("%d",
&numProductions);
getchar();

if (numProductions <= 0 || numProductions >= MAX_PRODUCTIONS) {
fprintf(stderr, "Error: Invalid number of productions.\n"); exit(1);
}

printf("Enter productions (use empty for epsilon). Format: A-
>alpha\n");
for (int i = 0; i < numProductions; ++i) { char
line[256];
printf("P%d: ", i + 1); fgets(line,
sizeof(line), stdin); line[strcspn(line,
"\r\n")] = 0;

        if (strlen(line) < 3 || line[1] != '-' || line[2] != '>') {
        fprintf(stderr, "Error: Invalid production format '%s'.\n",
line);
        exit(1);
}
productions[i].lhs = line[0];
strcpy(productions[i].rhs, line + 3);
}
originalStartSymbol = productions[0].lhs;
}

void augment_grammar() {
for (int i = numProductions; i >= 1; --i) {
productions[i] = productions[i - 1];
}
numProductions++;
```

```
productions[0].lhs = augmentedStartSymbol;
productions[0].rhs[0] = originalStartSymbol;
productions[0].rhs[1] = '\0';
}

void collect_symbols() {
for (int i = 0; i < numProductions; ++i) { add_char_unique(nonTerminals,
&numNonTerminals,
productions[i].lhs);
}
add_char_unique(nonTerminals, &numNonTerminals, augmentedStartSymbol);

for (int i = 0; i < numProductions; ++i) {
for (int j = 0; productions[i].rhs[j]; ++j) { char c =
productions[i].rhs[j];
if (is_terminal(c)) {
add_char_unique(terminals, &numTerminals, c);
}
}
}
add_char_unique(terminals, &numTerminals, '$');

for (int i = 0; i < numTerminals; ++i) alphabet[numAlphabet++] =
terminals[i];
for (int i = 0; i < numNonTerminals; ++i) alphabet[numAlphabet++] =
nonTerminals[i];
alphabet[numAlphabet] = '\0';
}

int add_set_to_set(char *dest, int *dest_len, const char *src, int
src_len) {
int changed = 0;
for (int i = 0; i < src_len; ++i) {
if (!contains_char(dest, src[i])) {
dest[*dest_len] = src[i]; (*dest_len)++;
dest[*dest_len] = '\0'; changed = 1;
}
}
return changed;
```

```
}

void compute_FIRST() {
int changed = 1; while
(changed) {
changed = 0;
for (int i = 0; i < numProductions; ++i) { char A =
productions[i].lhs;
char *rhs = productions[i].rhs;

int before_len = FIRST_len[(int)A];

if (strlen(rhs) == 0) { hasEpsilon[(int)A] = 1;
} else {
int k = 0;
int all_have_epsilon = 1; while (k <
strlen(rhs)) {
char B = rhs[k];
if (is_terminal(B)) { add_char_unique(FIRST[(int)A],
&FIRST_len[(int)A], B);
                        all_have_epsilon = 0;
                        break;
} else {
add_set_to_set(FIRST[(int)A], &FIRST_len[(int)A], FIRST[(int)B],
FIRST_len[(int)B]);
if (!hasEpsilon[(int)B]) { all_have_epsilon = 0;
break;
                }
                     }
k++;
}
if (all_have_epsilon) { hasEpsilon[(int)A] = 1;
}
}
if (FIRST_len[(int)A] != before_len) changed = 1;
}
}
}
```

```
void compute_FOLLOW() {
add_char_unique(FOLLOW[(int)augmentedStartSymbol],
&FOLLOW_len[(int)augmentedStartSymbol], '$');

int changed = 1; while
(changed) {
changed = 0;
for (int p = 0; p < numProductions; ++p) { char A =
productions[p].lhs;
char *rhs = productions[p].rhs;

for (int i = 0; i < strlen(rhs); ++i) { char B = rhs[i];
if (!is_nonterminal(B)) continue;

int before_len = FOLLOW_len[(int)B]; int k = i + 1;
int all_beta_has_epsilon = 1;

while (k < strlen(rhs)) { char C = rhs[k];
if (is_terminal(C)) { add_char_unique(FOLLOW[(int)B],
&FOLLOW_len[(int)B], C);
all_beta_has_epsilon = 0; break;
} else {
add_set_to_set(FOLLOW[(int)B], &FOLLOW_len[(int)B], FIRST[(int)C],
FIRST_len[(int)C]);
if (!hasEpsilon[(int)C]) { all_beta_has_epsilon = 0;
break;
                }
                     }
k++;
}

if (all_beta_has_epsilon) { add_set_to_set(FOLLOW[(int)B],
&FOLLOW_len[(int)B],
FOLLOW[(int)A], FOLLOW_len[(int)A]);
}
```

```
if (FOLLOW_len[(int)B] != before_len) changed = 1;
}
}
}
}

int item_equal(Item a, Item b) {
return a.prodIndex == b.prodIndex && a.dot == b.dot;
}

int is_item_in_set(const ItemSet *S, Item it) {
for (int i = 0; i < S->numItems; ++i) {
if (item_equal(S->items[i], it)) return 1;
}
return 0;
}

void add_item_to_set(ItemSet *S, Item it) { if
(!is_item_in_set(S, it)) {
S->items[S->numItems++] = it;
}
}

void closure(ItemSet *S) {
int changed = 1;
while (changed) { changed =
0;
for (int i = 0; i < S->numItems; ++i) { Item it =
S->items[i];
Production *p = &productions[it.prodIndex];

if (it.dot < strlen(p->rhs)) { char X = p-
>rhs[it.dot]; if (is_nonterminal(X)) {
for (int q = 0; q < numProductions; ++q) { if (productions[q].lhs
== X) {
Item newItem = { q, 0 };
if (!is_item_in_set(S, newItem)) { add_item_to_set(S, newItem);
changed = 1;
                                }
                }
                        }
```

```
    }
    }
    }
    }
    }

    ItemSet compute_GOTO_set(const ItemSet *S, char X) {
    ItemSet R = { .numItems = 0 };
    for (int i = 0; i < S->numItems; ++i) { Item
    it = S->items[i];
    Production *p = &productions[it.prodIndex];
    if (it.dot < strlen(p->rhs) && p->rhs[it.dot] == X) { Item
    advancedItem = { it.prodIndex, it.dot + 1 }; add_item_to_set(&R,
    advancedItem);
    }
    }
    if (R.numItems > 0) {
    closure(&R);
    }
    return R;
    }

    int are_itemsets_equal(const ItemSet *A, const ItemSet *B) { if
    (A->numItems != B->numItems) return 0;
    for (int i = 0; i < A->numItems; ++i) {
    if (!is_item_in_set(B, A->items[i])) return 0;
    }
    return 1;
    }

    int find_state_index(const ItemSet *X) {
    for (int s = 0; s < numStates; ++s) {
    if (are_itemsets_equal(&states[s], X)) return s;
    }
    return -1;
    }

    void build_canonical_collection() {
    for (int i = 0; i < MAX_STATES; ++i) {
    for (int j = 0; j < MAX_SYMBOLS; ++j) {
    transitions[i][j] = -1;
    }
    }
```

```
ItemSet I0 = { .numItems = 0 }; Item
startItem = { 0, 0 };
add_item_to_set(&I0, startItem);
closure(&I0); states[numStates++] =
I0;

for (int s = 0; s < numStates; ++s) {
for (int ai = 0; ai < numAlphabet; ++ai) { char X =
alphabet[ai];
ItemSet next_set = compute_GOTO_set(&states[s], X); if
(next_set.numItems == 0) continue;

            int existing_idx = find_state_index(&next_set);
            if (existing_idx == -1) {
            if (numStates >= MAX_STATES) {
            fprintf(stderr, "Error: Exceeded maximum number of
states.\n");
            exit(1);
}
states[numStates] = next_set; transitions[s][ai]
= numStates; numStates++;
} else {
transitions[s][ai] = existing_idx;
}
}
}
}

void initialize_tables() {
for (int s = 0; s < MAX_STATES; ++s) {
for (int a = 0; a < MAX_SYMBOLS; ++a) {
ACTION_TABLE[s][a].type = ACTION_ERROR;
ACTION_TABLE[s][a].target = -1;
GOTO_TABLE[s][a] = -1;
}
}
}

void set_action(int state, int symbol_idx, ActionType type, int target)
{
if (ACTION_TABLE[state][symbol_idx].type != ACTION_ERROR) {
```

```
conflict_found = 1;
fprintf(stderr, "\n--- CONFLICT DETECTED in state %d on symbol '%c' ---
\n", state, alphabet[symbol_idx]);
fprintf(stderr, "  Existing action: ");
switch(ACTION_TABLE[state][symbol_idx].type) {
case ACTION_SHIFT: fprintf(stderr, "SHIFT to %d\n",
ACTION_TABLE[state][symbol_idx].target); break;
case ACTION_REDUCE: fprintf(stderr, "REDUCE by P%d\n",
ACTION_TABLE[state][symbol_idx].target); break;
default: break;
}
fprintf(stderr, "  New action:                ");
switch(type) {
case ACTION_SHIFT: fprintf(stderr, "SHIFT to %d\n", target);
break;
case ACTION_REDUCE: fprintf(stderr, "REDUCE by P%d\n", target); break;
default: break;
}
fprintf(stderr, "This grammar is NOT SLR(1).\n");
}
ACTION_TABLE[state][symbol_idx].type = type;
ACTION_TABLE[state][symbol_idx].target = target;
}


void build_SLR_table() {
initialize_tables();

for (int s = 0; s < numStates; ++s) {
for (int ai = 0; ai < numAlphabet; ++ai) { int
target_state = transitions[s][ai]; if (target_state
== -1) continue;

char X = alphabet[ai]; if
(is_terminal(X)) {
set_action(s, ai, ACTION_SHIFT, target_state);
} else {
GOTO_TABLE[s][ai] = target_state;
}
}
}
```

```
for (int s = 0; s < numStates; ++s) {
for (int i = 0; i < states[s].numItems; ++i) { Item it =
states[s].items[i];
Production *p = &productions[it.prodIndex];

if (it.dot == strlen(p->rhs)) { if
(it.prodIndex == 0) {
int dollar_idx = get_symbol_index('$'); set_action(s, dollar_idx,
ACTION_ACCEPT, 0);
} else {
char A = p->lhs;
for (int k = 0; k < FOLLOW_len[(int)A]; ++k) { char a =
FOLLOW[(int)A][k];
int col = get_symbol_index(a); if (col >= 0) {
set_action(s, col, ACTION_REDUCE,
it.prodIndex);
}
}
}
}
}
}
}

void print_productions_list() {
printf("\n=== Grammar Productions ===\n"); for
(int i = 0; i < numProductions; ++i) {
printf("  P%-2d: %c -> %s\n", i, productions[i].lhs,
(strlen(productions[i].rhs) == 0 ? "ε" : productions[i].rhs));
}
}

void print_item(Item it) {
Production *p = &productions[it.prodIndex];
printf("%c ->", p->lhs);
for (int j = 0; j < strlen(p->rhs); ++j) { if (j
== it.dot) printf(" ·");
printf(" %c", p->rhs[j]);
}
if (it.dot == strlen(p->rhs)) printf(" ·");
}
```

```
void print_states() {
printf("\n=== Canonical LR(0) Item Sets (%d states) ===\n", numStates);
for (int s = 0; s < numStates; ++s) {
printf("I%d:\n", s);
for (int i = 0; i < states[s].numItems; ++i) { printf("  [%2d]
", states[s].items[i].prodIndex);
print_item(states[s].items[i]);
printf("\n");
}
}
}

void print_first_follow() {
printf("\n=== FIRST and FOLLOW Sets ===\n"); for
(int i = 0; i < numNonTerminals; ++i) {
char A = nonTerminals[i]; char
A_str[2] = {A, '\0'};

printf("  %-10s: FIRST = { ", (A == augmentedStartSymbol) ? "S'" :
A_str);
for(int k=0; k<FIRST_len[(int)A]; ++k) printf("%c ", FIRST[(int)A][k]);
if(hasEpsilon[(int)A]) printf("ε ");
printf("}\n");

printf("  %-10s: FOLLOW = { ", "");
for(int k=0; k<FOLLOW_len[(int)A]; ++k) printf("%c ", FOLLOW[(int)A][k]);
printf("}\n");
}
}

void print_parsing_table() {
printf("\n=== SLR Parsing Table ===\n");
printf("State |");
// Print ACTION table header
for (int i = 0; i < numTerminals; ++i) printf("  %c          ",
terminals[i]);
printf(" | ");
// Print GOTO table header, excluding the augmented start symbol for
(int i = 0; i < numNonTerminals; ++i) {
```

```
if (nonTerminals[i] != augmentedStartSymbol) printf("  %c                ",
nonTerminals[i]);
}
printf("\n"); printf("
 ------------------|");
for (int i = 0; i < numTerminals; ++i) printf(" ---------- ");
printf("-|-");
for (int i = 0; i < numNonTerminals; ++i) {
if (nonTerminals[i] != augmentedStartSymbol) printf("--------------- ");
}
printf("\n");

for (int s = 0; s < numStates; ++s) { printf("
%4d |", s);
// Print ACTION entries
for (int i = 0; i < numTerminals; ++i) {
int col = get_symbol_index(terminals[i]); ActionEntry e =
ACTION_TABLE[s][col]; switch(e.type) {
case ACTION_SHIFT: printf(" s%-4d", e.target); break; case ACTION_REDUCE:
printf(" r%-4d", e.target); break; case ACTION_ACCEPT: printf("  ACC ");
break;
case ACTION_ERROR:  printf("      "); break;
}
}
        printf(" | ");
        // Print GOTO entries, excluding the augmented start symbol's
column
        for (int i = 0; i < numNonTerminals; ++i) {
        if (nonTerminals[i] == augmentedStartSymbol) continue; int col
        = get_symbol_index(nonTerminals[i]);
        if (GOTO_TABLE[s][col] >= 0) printf(" %-5d",
GOTO_TABLE[s][col]);
else printf("                    ");
}
printf("\n");
}
}

typedef struct { int
st[1024]; int top;
} IntStack;
```

```
void push(IntStack *S, int x) { S->st[++S->top] = x; }
int pop(IntStack *S) { return S->st[S->top--]; }
int peek(IntStack *S) { return S->st[S->top]; }

void parse_input(const char *input) {
printf("\n=== LR Parsing Trace for Input: %s ===\n", input);
printf("%-20s %-20s %s\n", "Stack", "Input", "Action"); printf("
--------\n");

IntStack stack;
stack.top = -1;
push(&stack, 0);

int ip = 0;
char a = input[ip];

while (1) {
char stack_str[256] = "";
for(int i = 0; i <= stack.top; ++i) sprintf(stack_str +
strlen(stack_str), "%d ", stack.st[i]);
printf("%-20s %-20s ", stack_str, &input[ip]);

int s = peek(&stack);
int col = get_symbol_index(a); if (col <
0) {
printf("Error: Invalid input symbol '%c'\n", a); return;
}

ActionEntry entry = ACTION_TABLE[s][col];

if (entry.type == ACTION_SHIFT) { printf("Shift to
%d\n", entry.target); push(&stack, entry.target);
a = input[++ip];
} else if (entry.type == ACTION_REDUCE) { Production *p =
&productions[entry.target]; printf("Reduce by %c -> %s\n",
p->lhs, p->rhs); int len = strlen(p->rhs);
for (int k = 0; k < len; ++k) pop(&stack);
```

```
int t = peek(&stack);
int A_col = get_symbol_index(p->lhs); int
goto_state = GOTO_TABLE[t][A_col]; if (goto_state <
0) {
printf("Error: No GOTO entry from state %d on symbol
%c\n", t, p->lhs);
return;
}
push(&stack, goto_state);
} else if (entry.type == ACTION_ACCEPT) {
printf("ACCEPT\n");
return;
} else {
printf("ERROR: No action defined\n"); return;
}
}
}

int main() {
read_grammar();
augment_grammar();
collect_symbols();

compute_FIRST();
compute_FOLLOW();

build_canonical_collection(); build_SLR_table();

print_productions_list();
print_first_follow();
print_states();
print_parsing_table();

if (conflict_found) {
fprintf(stderr, "\nParsing table construction failed due to
conflicts.\n");
return 1;
}

char input[256];
printf("\nEnter input string to parse (must end with $): ");
```

```
fgets(input, sizeof(input), stdin); input[strcspn(input,
"\r\n")] = 0;

if (strlen(input) == 0 || input[strlen(input) - 1] != '$') {
fprintf(stderr, "Error: Input must be non-empty and end with
$.\n");
return 1;
}

parse_input(input); return

0;
}
```

Output

```
student@AB1665B049:~/compiler_1643$./bin6 Enter the
number of productions: 6

Enter productions (use empty for epsilon). Format: A->alpha P1: E->E+T

P2: E->T P3:T-
>T*F P4: T->F
P5:F->(E)

P6: F->a

=== Grammar Productions === P0 : ' ->
E

P1 : E -> E+T P2 : E -> T
P3 : T -> T*F P4 : T -> F
P5 : F -> (E)

P6 : F -> a


=== FIRST and FOLLOW Sets ===

     : FOLLOW = { $ }
     E     : FIRST = { ( a )
     : FOLLOW = { $ + } }
     T     : FIRST = { ( a )
     : FOLLOW = { $ + * } }
     F     : FIRST = { ( a )
     : FOLLOW = { $ + * }}
```

```
== Canonical LR(0) Item Sets (12 states) === I0:
    [ 0] ' -> E
    [ 1] E -> E + T
    [ 2] E -> T
    [ 3] T -> T * F
    [ 4] T -> F
    [ 5] F -> ( E )
    [ 6] F -> a
I1:
    [ 5] F -> ( E )
    [ 1] E -> E + T
    [ 2] E -> T
    [ 3] T -> T * F
    [ 4] T -> F
    [ 5] F -> ( E )
    [ 6] F -> a
I2:
    [ 6] F -> a
I3:
    [ 0] ' -> E
    [ 1] E -> E + T
I4:
    [ 2] E -> T
    [ 3] T -> T * F
I5:
    [ 3] T -> T . * F
I5:
    [ 4] T -> F .
I6:
    [ 5] F -> ( E . )
    [ 1] E -> E . + T
I7:
    [ 1] E -> E + . T
    [ 3] T -> . T * F
    [ 4] T -> . F
    [ 5] F -> . ( E )
    [ 6] F -> . a
I8:
    [ 3] T -> T * . F
    [ 5] F -> . ( E )
    [ 6] F -> . a
```

```
I9:
    [ 5] F -> ( E ) .
I10:
    [ 1] E -> E + T .
    [ 3] T -> T . * F
I11:
    [ 3] T -> T * F .
```

```
=== SLR  Parsing Table ===
State |     +      *     (     )      a     $   |  S'       E     T     F
S0    |              s4           s5        |            1     2     3
S1    |    s6                           acc  |
S2    |    r2    s7         r2           r2   |
S3    |    r4    r4         r4           r4   |
S4    |              s4           s5        |            8     2     3
S5    |    r6    r6         r6           r6   |
S6    |              s4           s5        |                  9     3
S7    |              s4           s5        |                        10
S8    |    s6               s11             |
S9    |    r1    s7         r1           r1   |
S10   |    r3    r3         r3           r3   |
S11   |    r5    r5         r5           r5   |
```

```
Enter input string to parse (must end with $): a*a+a$

Enter input string to parse (must end with $): a*a+a$

=== LR Parsing Trace for Input: a*a+a$ ===
Stack     Input    Action
0         a*a+a$   Shift to 5
0 5       *a+a$   Reduce by    F -> a
0 3       *a+a$   Reduce by    T -> F
0 2       *a+a$   Shift to 7
0 2 7     a+a$    Shift to 5
0 2 7 5   +a$     Reduce by    F -> a
0 2 7 10  +a$     Reduce by    T -> T*F
0 2       +a$     Reduce by    E -> T
0 1       +a$     Shift to 6
0 1 6     a$      Shift to 5
0 1 6 5   $       Reduce by    F -> a
0 1 6 3   $       Reduce by    T -> F
0 1 6 9   $       Reduce by    E -> E+T
0 1       $       Accept
```

```
Enter the number of productions: 6
Enter productions (use empty for epsilon). Format: A->alpha
P1: E->E+T
P2: E->T
P3: T->T*F
P4: T->F
P5: F->(E)
P6: F->a

=== Grammar Productions ===
    P0 : ' -> E
    P1 : E -> E+T
    P2 : E -> T
    P3 : T -> T*F
    P4 : T -> F
    P5 : F -> (E)
    P6 : F -> a

=== FIRST and FOLLOW Sets ===
    S'    : FIRST = { ( a }
    : FOLLOW = { $ }
    E     : FIRST = { ( a }
    : FOLLOW = { $ + } }
    T     : FIRST = { ( a }
    : FOLLOW = { $ + * } }
    F     : FIRST = { ( a }
    : FOLLOW = { $ + * } }

=== Canonical LR(0) Item Sets (12 states) ===
I0:
    [ 0] ' -> E
    [ 1] E -> E + T
    [ 2] E -> T
    [ 3] T -> T * F
    [ 4] T -> F
    [ 5] F -> ( E )
    [ 6] F -> a
I1:
    [ 5] F -> ( E )
```

```
    [ 5] F -> ( E )
    [ 1] E -> E + T
    [ 2] E -> T
    [ 3] T -> T * F
    [ 4] T -> F
    [ 5] F -> ( E )
    [ 6] F -> a
I2:
    [ 6] F -> a
I3:
    [ 0] ' -> E
    [ 1] E -> E + T
I4:
    [ 2] E -> T
    [ 3] T -> T * F
I5:
    [ 3] T -> T . * F
I5:
    [ 4] T -> F .
I6:
    [ 5] F -> ( E . )
    [ 1] E -> E . + T
I7:
    [ 1] E -> E + . T
    [ 3] T -> . T * F
    [ 4] T -> . F
    [ 5] F -> . ( E )
    [ 6] F -> . a
I8:
    [ 3] T -> T * . F
    [ 5] F -> . ( E )
    [ 6] F -> . a
I9:
    [ 5] F -> ( E ) .
I10:
    [ 1] E -> E + T .
    [ 3] T -> T . * F
I11:
    [ 3] T -> T * F .
```

```
I5:
    [ 4] T -> F .
I6:
    [ 5] F -> ( E . )
    [ 1] E -> E . + T
I7:
    [ 1] E -> E + . T
    [ 3] T -> . T * F
    [ 4] T -> . F
    [ 5] F -> . ( E )
    [ 6] F -> . a
I8:
    [ 3] T -> T * . F
    [ 5] F -> . ( E )
    [ 6] F -> . a
I9:
    [ 5] F -> ( E ) .
I10:
    [ 1] E -> E + T .
    [ 3] T -> T . * F
I11:
    [ 3] T -> T * F .

=== SLR Parsing Table ===
State |     +     *     (     )     a     $   |    S'     E     T     F
S0    |                 s4          s5        |           1     2     3
S1    |    s6                             acc |
S2    |    r2    s7         r2          r2   |
S3    |    r4    r4         r4          r4   |
S4    |                 s4          s5        |           8     2     3
S5    |    r6    r6         r6          r6   |
S6    |                 s4          s5        |                 9     3
S7    |                 s4          s5        |                       10
S8    |    s6                s11            |
S9    |    r1    s7         r1          r1   |
S10   |    r3    r3         r3          r3   |
S11   |    r5    r5         r5          r5   |

Enter input string to parse (must end with $): a*a+a$
```

```
=== SLR Parsing Table ===
State |     +     *     (     )     a     $   |    S'     E     T     F
S0    |                 s4          s5        |           1     2     3
S1    |    s6                             acc |
S2    |    r2    s7         r2          r2    |
S3    |    r4    r4         r4          r4    |
S4    |                 s4          s5        |           8     2     3
S5    |    r6    r6         r6          r6    |
S6    |                 s4          s5        |                 9     3
S7    |                 s4          s5        |                       10
S8    |    s6                s11              |
S9    |    r1    s7         r1          r1    |
S10   |    r3    r3         r3          r3    |
S11   |    r5    r5         r5          r5    |

Enter input string to parse (must end with $): a*a+a$

Enter input string to parse (must end with $): a*a+a$

=== LR Parsing Trace for Input: a*a+a$ ===
Stack    Input    Action
0        a*a+a$   Shift to 5
0 5      *a+a$    Reduce by F -> a
0 3      *a+a$    Reduce by T -> F
0 2      *a+a$    Shift to 7
0 2 7    a+a$     Shift to 5
0 2 7 5  +a$      Reduce by F -> a
0 2 7 10 +a$      Reduce by T -> T*F
0 2      +a$      Reduce by E -> T
0 1      +a$      Shift to 6
0 1 6    a$       Shift to 5
0 1 6 5  $        Reduce by F -> a
0 1 6 3  $        Reduce by T -> F
0 1 6 9  $        Reduce by E -> E+T
0 1      $        Accept
PS C:\Notes\compiler\parsing>
```

Experiment-10(a)

Construct Canonical LR (CLR) parse table using C language

Aim

The objective is to write a C program to construct a Canonical LR (CLR) parse table for a given context-free grammar. The program should generate LR(1) items, compute closures and transitions, and finally build the ACTION and GOTO tables.

Algorithm

1. **Start**.
2. **Input grammar**: Take a context-free grammar with a single start symbol.
3. **Augment grammar**: Add a new start production S' → S.
4. **Construct initial item set**: Create the closure of [S' → • S, $].
5. **Generate canonical collection**:
    - For each item set I in the collection:
        - For each grammar symbol X (terminal or nonterminal):
            - Compute goto(I, X) → a new item set.
            - If it is new, add it to the collection.
6. **Build parse table**:
    - For each item set:
        - If the item has form [A → α • a β, t] and goto(I, a) = J, add ACTION[I, a] = shift J.
        - If the item has form [A → α •, t], add ACTION[I, t] = reduce A → α.
        - If the item has [S' → S •, $], then ACTION[I, $] = accept.
        - For each nonterminal A with goto(I, A) = J, set GOTO[I, A] = J.
7. **Resolve conflicts**: If both shift and reduce (or multiple reduces) are generated, report grammar is not CLR(1).
8. **Output the parse table** (ACTION and GOTO).
9. **Stop**.

Experiment-10(b)

Implement the LR parsing algorithm using C language Aim

The objective is to write a C program that implements the Canonical LR (CLR) parsing algorithm. The program should use the constructed parse table along with a given input string to determine whether the string belongs to the grammar.

Algorithm

1. **Start**.
2. **Input parse table** (ACTION and GOTO) and **input string** to be parsed.
3. **Initialize stack** with state 0.
4. **Repeat until ACCEPT or ERROR**:
    - Let s = top of stack, and a = current input symbol.
    - If ACTION[s, a] = shift t:
        - Push a and state t onto the stack.
        - Advance input pointer.
    - Else if ACTION[s, a] = reduce A → β:
        - Pop 2 * |β| symbols from the stack.

        - Let s' = top of stack.
        - Push A.
        - Push GOTO[s', A].
        - Output reduction A → β.
    - Else if ACTION[s, a] = accept:
        - Report success, input string is valid.
        - Stop.
    - Else:
        - Report error, input string is invalid.
        - Stop.
5. **Stop**.

Code

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

#define MAX_PRODUCTIONS 64
#define MAX_RHS_LENGTH 64
#define MAX_SYMBOLS 64
#define MAX_ITEMS 512
#define MAX_STATES 512
#define MAX_SET_SIZE 64

typedef struct {
int prodIndex; int
dot;
char lookahead;
} Item;

typedef struct {
Item items[MAX_ITEMS]; int
numItems;
} ItemSet;

typedef struct {
char lhs;
char rhs[MAX_RHS_LENGTH];
} Production;

typedef enum {
ACTION_ERROR,
ACTION_SHIFT,
ACTION_REDUCE,
```

```
ACTION_ACCEPT
} ActionType;

typedef struct {
ActionType type; int
target;
} ActionEntry;

Production productions[MAX_PRODUCTIONS];
int numProductions = 0;
char originalStartSymbol;
char augmentedStartSymbol = '\'';

char nonTerminals[MAX_SYMBOLS];
int numNonTerminals = 0;
char terminals[MAX_SYMBOLS];
int numTerminals = 0;
char alphabet[MAX_SYMBOLS];
int numAlphabet = 0;

char FIRST[256][MAX_SET_SIZE];
int FIRST_len[256] = {0};
int hasEpsilon[256] = {0};

ItemSet states[MAX_STATES];
int numStates = 0;
int transitions[MAX_STATES][MAX_SYMBOLS];

ActionEntry ACTION_TABLE[MAX_STATES][MAX_SYMBOLS];
int GOTO_TABLE[MAX_STATES][MAX_SYMBOLS];
int conflict_found = 0;

int contains_char(const char *s, char c) { return
strchr(s, c) != NULL;
}

void add_char_unique(char *s, int *len, char c) {
if (!contains_char(s, c)) {
s[*len] = c;
s[*len + 1] = '\0'; (*len)++;
}
}
```

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

int get_symbol_index(char c) {
for (int i = 0; i < numAlphabet; ++i) { if
(alphabet[i] == c) return i;
}
return -1;
}

void read_grammar() {
printf("Enter the number of productions: "); scanf("%d",
&numProductions);
getchar();

if (numProductions <= 0 || numProductions >= MAX_PRODUCTIONS) {
fprintf(stderr, "Error: Invalid number of productions.\n"); exit(1);
}

printf("Enter productions (use empty for epsilon). Format: A-
>alpha\n");
for (int i = 0; i < numProductions; ++i) { char
line[256];
printf("P%d: ", i + 1); fgets(line,
sizeof(line), stdin); line[strcspn(line,
"\r\n")] = 0;

        if (strlen(line) < 3 || line[1] != '-' || line[2] != '>') {
        fprintf(stderr, "Error: Invalid production format '%s'.\n",
line);
        exit(1);
}
productions[i].lhs = line[0];
strcpy(productions[i].rhs, line + 3);
}
originalStartSymbol = productions[0].lhs;
}

void augment_grammar() {
for (int i = numProductions; i >= 1; --i) {
```

```
productions[i] = productions[i - 1];
}
numProductions++;

productions[0].lhs = augmentedStartSymbol;
productions[0].rhs[0] = originalStartSymbol;
productions[0].rhs[1] = '\0';
}

void collect_symbols() {
for (int i = 0; i < numProductions; ++i) { add_char_unique(nonTerminals,
&numNonTerminals,
productions[i].lhs);
}

for (int i = 0; i < numProductions; ++i) {
for (int j = 0; productions[i].rhs[j]; ++j) { char c =
productions[i].rhs[j];
if (c != '\0' && !is_nonterminal(c)) { add_char_unique(terminals,
&numTerminals, c);
}
}
}
add_char_unique(terminals, &numTerminals, '$');

for (int i = 0; i < numTerminals; ++i) alphabet[numAlphabet++] =
terminals[i];
for (int i = 0; i < numNonTerminals; ++i) alphabet[numAlphabet++] =
nonTerminals[i];
alphabet[numAlphabet] = '\0';
}

void compute_FIRST() {
int changed = 1; while
(changed) {
changed = 0;
for (int i = 0; i < numProductions; ++i) { char A =
productions[i].lhs;
const char *rhs = productions[i].rhs; int
before_len = FIRST_len[(int)A];

if (strlen(rhs) == 0) {
if (!hasEpsilon[(int)A]) {
```

```
hasEpsilon[(int)A] = 1;
changed = 1;
}
} else {
int k = 0;
int all_have_epsilon = 1; while (k <
strlen(rhs)) {
char B = rhs[k];
if (!is_nonterminal(B)) { add_char_unique(FIRST[(int)A],
&FIRST_len[(int)A], B);
all_have_epsilon = 0; break;
}
for (int fi = 0; fi < FIRST_len[(int)B]; ++fi) {
add_char_unique(FIRST[(int)A],
&FIRST_len[(int)A], FIRST[(int)B][fi]);
}
if (!hasEpsilon[(int)B]) { all_have_epsilon = 0;
break;
} k++;
}
if (all_have_epsilon && !hasEpsilon[(int)A]) { hasEpsilon[(int)A]
= 1;
changed = 1;
}
}
if (FIRST_len[(int)A] != before_len) changed = 1;
}
}
}

void compute_first_of_string(const char *s, char *first_set, int
*first_len) {
*first_len = 0; first_set[0]
= '\0';

for (int i = 0; i < strlen(s); ++i) { char
symbol = s[i];
if (!is_nonterminal(symbol)) { add_char_unique(first_set,
first_len, symbol);
```

```
return;
}
for (int j = 0; j < FIRST_len[(int)symbol]; ++j) {
add_char_unique(first_set, first_len,
FIRST[(int)symbol][j]);
}
if (!hasEpsilon[(int)symbol]) { return;
}
}
}

int item_equal(Item a, Item b) {
return a.prodIndex == b.prodIndex && a.dot == b.dot && a.lookahead
== b.lookahead;
}

int is_item_in_set(const ItemSet *S, Item it) {
for (int i = 0; i < S->numItems; ++i) {
if (item_equal(S->items[i], it)) return 1;
}
return 0;
}

void add_item_to_set(ItemSet *S, Item it) { if
(!is_item_in_set(S, it)) {
S->items[S->numItems++] = it;
}
}

void closure(ItemSet *S) {
int changed = 1;
while (changed) { changed =
0;
for (int i = 0; i < S->numItems; ++i) { Item it =
S->items[i];
Production *p = &productions[it.prodIndex];

if (it.dot < strlen(p->rhs)) { char B = p-
>rhs[it.dot]; if (is_nonterminal(B)) {
char beta_a[MAX_RHS_LENGTH + 2]; strcpy(beta_a, p->rhs + it.dot +
1);
```

```
beta_a[strlen(beta_a) + 1] = '\0'; beta_a[strlen(beta_a)] =
it.lookahead;

                    char first_of_beta_a[MAX_SET_SIZE]; int
                    first_len;
                    compute_first_of_string(beta_a, first_of_beta_a,
&first_len);



for (int q = 0; q < numProductions; ++q) { if (productions[q].lhs
== B) {
for (int k = 0; k < first_len; ++k) { Item newItem = { q, 0,
first_of_beta_a[k] };
                                if (!is_item_in_set(S, newItem)) {
                                add_item_to_set(S, newItem); changed =
                                1;
}
}
}
}
}
}
}
}

ItemSet compute_GOTO_set(const ItemSet *S, char X) {
ItemSet R = { .numItems = 0 };
for (int i = 0; i < S->numItems; ++i) { Item
it = S->items[i];
Production *p = &productions[it.prodIndex];
if (it.dot < strlen(p->rhs) && p->rhs[it.dot] == X) { Item
advancedItem = { it.prodIndex, it.dot + 1,
it.lookahead };
add_item_to_set(&R, advancedItem);
}
}
if (R.numItems > 0) {
closure(&R);
}
return R;
}
```

```
int are_itemsets_equal(const ItemSet *A, const ItemSet *B) { if
(A->numItems != B->numItems) return 0;
for (int i = 0; i < A->numItems; ++i) {
if (!is_item_in_set(B, A->items[i])) return 0;
}
return 1;
}

int find_state_index(const ItemSet *X) {
for (int s = 0; s < numStates; ++s) {
if (are_itemsets_equal(&states[s], X)) return s;
}
return -1;
}

void build_canonical_collection() {
memset(transitions, -1, sizeof(transitions));

ItemSet I0 = { .numItems = 0 }; Item
startItem = { 0, 0, '$' };
add_item_to_set(&I0, startItem);
closure(&I0); states[numStates++] =
I0;

for (int s = 0; s < numStates; ++s) {
for (int ai = 0; ai < numAlphabet; ++ai) { char X =
alphabet[ai];
ItemSet next_set = compute_GOTO_set(&states[s], X); if
(next_set.numItems == 0) continue;

int existing_idx = find_state_index(&next_set); if
(existing_idx == -1) {
if (numStates >= MAX_STATES) {
fprintf(stderr, "Error: Exceeded maximum number of states.\n"); exit(1);
}
states[numStates] = next_set; transitions[s][ai]
= numStates; numStates++;
} else {
transitions[s][ai] = existing_idx;
}
}
```

```
}
}

void set_action(int state, int symbol_idx, ActionType type, int target,
char symbol) {
if (ACTION_TABLE[state][symbol_idx].type != ACTION_ERROR) {
conflict_found = 1;
fprintf(stderr, "\n--- CLR(1) CONFLICT DETECTED in state %d on symbol
'%c' ---\n", state, symbol);
fprintf(stderr, "  Existing action: ");
if (ACTION_TABLE[state][symbol_idx].type == ACTION_SHIFT)
fprintf(stderr, "SHIFT to %d\n",
ACTION_TABLE[state][symbol_idx].target); else
fprintf(stderr, "REDUCE by P%d (%c -> %s)\n",
ACTION_TABLE[state][symbol_idx].target,
productions[ACTION_TABLE[state][symbol_idx].target].lhs,
productions[ACTION_TABLE[state][symbol_idx].target].rhs);

fprintf(stderr, "  New action:                ");
if (type == ACTION_SHIFT)
fprintf(stderr, "SHIFT to %d\n", target);
else
fprintf(stderr, "REDUCE by P%d (%c -> %s)\n", target,
productions[target].lhs, productions[target].rhs); fprintf(stderr,
"This grammar is NOT CLR(1).\n");
}
ACTION_TABLE[state][symbol_idx].type = type;
ACTION_TABLE[state][symbol_idx].target = target;
}

void build_CLR_table() {
for (int s = 0; s < MAX_STATES; ++s) {
for (int a = 0; a < MAX_SYMBOLS; ++a) {
ACTION_TABLE[s][a].type = ACTION_ERROR; GOTO_TABLE[s][a]
= -1;
}
}

for (int s = 0; s < numStates; ++s) {
for (int ai = 0; ai < numAlphabet; ++ai) { int
target_state = transitions[s][ai]; if (target_state
== -1) continue;
```

```
char X = alphabet[ai];
if (!is_nonterminal(X)) {
set_action(s, ai, ACTION_SHIFT, target_state, X);
} else {
GOTO_TABLE[s][ai] = target_state;
}
}
}

for (int s = 0; s < numStates; ++s) {
for (int i = 0; i < states[s].numItems; ++i) { Item it =
states[s].items[i];
Production *p = &productions[it.prodIndex];

if (it.dot == strlen(p->rhs)) {
int col = get_symbol_index(it.lookahead); if (col < 0)
continue;

                if (it.prodIndex == 0) {
                set_action(s, col, ACTION_ACCEPT, 0, it.lookahead);
                } else {
                set_action(s, col, ACTION_REDUCE, it.prodIndex,
it.lookahead);

}
}
}
}
}

void print_productions_list() {
printf("\n=== Grammar Productions ===\n"); for
(int i = 0; i < numProductions; ++i) {
printf("  P%-2d: %c -> %s\n", i, productions[i].lhs,
(strlen(productions[i].rhs) == 0 ? "ε" : productions[i].rhs));
}
}

void print_item(Item it) {
Production *p = &productions[it.prodIndex];
printf("[%c ->", p->lhs);
for (int j = 0; j < strlen(p->rhs); ++j) { if (j
== it.dot) printf(" .");
```

```
printf(" %c", p->rhs[j]);
}
if (it.dot == strlen(p->rhs)) printf(" ·");
printf(", %c]", it.lookahead);
}

void print_states() {
printf("\n=== Canonical LR(1) Item Sets (%d states) ===\n", numStates);
for (int s = 0; s < numStates; ++s) {
printf("I%d:\n", s);
for (int i = 0; i < states[s].numItems; ++i) { printf("
"); print_item(states[s].items[i]); printf("\n");
}
}
}

void print_parsing_table() {
printf("\n=== CLR(1) Parsing Table ===\n");
printf("State |");
for (int i = 0; i < numTerminals; ++i) printf("  %-4c", terminals[i]);
printf(" | ");
for (int i = 0; i < numNonTerminals; ++i) {
if (nonTerminals[i] != augmentedStartSymbol) printf("  %-4c",
nonTerminals[i]);
}
printf("\n"); printf("
 ------------------|");
for (int i = 0; i < numTerminals; ++i) printf(" ---------- ");
printf("-|-");
for (int i = 0; i < numNonTerminals - 1; ++i) printf(" ---------- ");
printf("\n");

for (int s = 0; s < numStates; ++s) { printf("
%4d |", s);
for (int i = 0; i < numTerminals; ++i) {
int col = get_symbol_index(terminals[i]); ActionEntry e =
ACTION_TABLE[s][col];
if (e.type == ACTION_SHIFT) printf(" s%-4d", e.target);
```

```
            else if (e.type == ACTION_REDUCE) printf(" r%-4d",
e.target);
            else if (e.type == ACTION_ACCEPT) printf("  ACC ");
            else printf("       ");
          }

printf(" | ");
for (int i = 0; i < numNonTerminals; ++i) {
if (nonTerminals[i] == augmentedStartSymbol) continue; int col =
get_symbol_index(nonTerminals[i]);
if (GOTO_TABLE[s][col] >= 0) printf(" %-5d", GOTO_TABLE[s][col]);
else printf("                    ");
}
printf("\n");
}
}

typedef struct { int st[1024]; int top; } IntStack; void
push(IntStack *S, int x) { S->st[++(S->top)] = x; } int
pop(IntStack *S) { return S->st[(S->top)--]; }
int peek(IntStack *S) { return S->st[S->top]; }

void parse_input(const char *input) {
printf("\n=== LR Parsing Trace for Input: %s ===\n", input);
printf("%-30s %-20s %s\n", "Stack", "Input", "Action"); printf("
\n");       ----------------------------------------------------------------------------------------------------------------------------------
---------------
IntStack stack;
stack.top = -1;
push(&stack, 0);

int ip = 0; while (1)
{
char stack_str[256] = "";
for(int i = 0; i <= stack.top; ++i) sprintf(stack_str +
strlen(stack_str), "%d ", stack.st[i]);
printf("%-30s %-20s ", stack_str, &input[ip]);

int s = peek(&stack); char a =
input[ip];
int col = get_symbol_index(a); if (col <
0) {
```

```
printf("Error: Invalid input symbol '%c'\n", a); return;
}

ActionEntry entry = ACTION_TABLE[s][col]; if
(entry.type == ACTION_SHIFT) {
printf("Shift to %d\n", entry.target); push(&stack,
entry.target);
ip++;
} else if (entry.type == ACTION_REDUCE) { Production *p =
&productions[entry.target]; printf("Reduce by %c -> %s\n",
p->lhs, p->rhs);
for (int k = 0; k < strlen(p->rhs); ++k) pop(&stack);

int t = peek(&stack);
int A_col = get_symbol_index(p->lhs); push(&stack,
GOTO_TABLE[t][A_col]);
} else if (entry.type == ACTION_ACCEPT) {
printf("ACCEPT\n"); return;
} else {
printf("ERROR: No action defined\n"); return;
}
}
}

int main() {
read_grammar();
augment_grammar();
collect_symbols();

compute_FIRST();

build_canonical_collection(); build_CLR_table();

print_productions_list();
print_states();
print_parsing_table();

if (conflict_found) {
fprintf(stderr, "\nParsing table construction failed due to
conflicts.\n");
return 1;
}
```

```
char input[256];
printf("\nEnter input string to parse (must end with $): ");
fgets(input, sizeof(input), stdin);
input[strcspn(input, "\r\n")] = 0;

if (strlen(input) == 0 || input[strlen(input) - 1] != '$') {
fprintf(stderr, "Error: Input must be non-empty and end with
$.\n");
return 1;
}

parse_input(input); return

0;
```

Output

```
student@AB1605B049:~/compiler_1643$ gcc CLR.c -o bind
student@AB1605B049:~/compiler_1643$ ./bind
Enter the number of productions: 3
Enter productions (use empty for epsilon). Format: A->alpha
P1: S->CC
P2: C->cC
P3: C->d

=== Grammar Productions ===
P0 : ' -> S
P1 : S -> CC P2
: C -> cC P3 : C
-> d

=== Canonical LR(1) Item Sets (10 states) ===
I0:
[' -> .S, $]
[S -> .C C, $]
[C -> .c C, c]
[C -> .c C, d]
[C -> .d, c]
[C -> .d, d]

I1:
[C -> c .C, c]
```

```
[C -> c .C, d]
[C -> .c C, c]
[C -> .c C, d]
[C -> .d, c]
[C -> .d, d]

I2:
     [C -> d ., c]
     [C -> d ., d]

I3:
[' -> S ., $]

I4:
     [S -> C .C, $]
     [C -> .c C, $]
     [C -> .d, $]

I5:
     [C -> c C ., c]
     [C -> c C ., d]

I6:
     [C -> c .C, $]
     [C -> .c C, $]
     [C -> .d, $]

I7:
[C -> d ., $]

I8:
[S -> C C ., $]

I9:
[C -> c C ., $]

=== CLR(1) Parsing Table ===
 State |      c     d     $ |   S'       S     C
 S0    |     s3    s4       |            1     2
 S1    |                acc |
 S2    |     s6    s7       |                  5
 S3    |     s3    s4       |                  8
S4     |    r3    r3        |
```

```
S5 |                   r1   |
S6 |      s6 r2  s7         |                   9
S7 |                   r3   |
S8 |             r2         |
S9 |                   r2   |

Enter input string to parse (must end with $): ccdd$
=== LR Parsing Trace for Input: ccdd$ ===
Stack     Input    Action
0        ccdd$   Shift    to 3
0 3      cdd$    Shift    to 3
0 3 3    dd$     Shift    to 4
0 3 3 4  d$      Reduce   by  C -> d
0 3 3 8  d$      Reduce   by  C -> cC
0 3 8    d$      Reduce   by  C -> cC
0 2       d$     Shift to 7
0 2 7     $      Reduce by C    -> d
0 2 5     $      Reduce by S    -> CC
0 1       $      ACCEPT
```

```
student@AB1605B049:~/compiler_1643$ gcc CLR.c -o bin1
student@AB1605B049:~/compiler_1643$ ./bin1
Enter the number of productions: 3
Enter productions (use empty for epsilon). Format: A->alpha
P1: S->CC
P2: C->cC
P3: C->d

=== Grammar Productions ===
  P0 : ' -> S
  P1 : S -> CC
  P2 : C -> cC
  P3 : C -> d

=== Canonical LR(1) Item Sets (10 states) ===
I0:
  [' -> · S, $]
  [S -> · C C, $]
  [C -> · c C, c]
  [C -> · c C, d]
  [C -> · d, c]
  [C -> · d, d]
I1:
  [C -> c · C, c]
  [C -> c · C, d]
  [C -> · c C, c]
  [C -> · d, c]
  [C -> · c C, d]
  [C -> · d, d]
I2:
  [C -> d ·, c]
  [C -> d ·, d]
I3:
  [' -> S ·, $]
I4:
  [S -> C · C, $]
  [C -> · c C, $]
  [C -> · d, $]
I5:
  [C -> c C ·, c]
  [C -> c C ·, d]
I6:
  [C -> c · C, $]
  [C -> · c C, $]
  [C -> · d, $]
I7:
  [C -> d ·, $]
I8:
  [S -> C C ·, $]
I9:
  [C -> c C ·, $]
```

```
I8:
    [S -> C C ., $]

I9:
    [C -> c C ., $]

=== CLR(1) Parsing Table ===
State |     c     d     $ |    S'     S     C
S0    |    s3    s4       |           1     2
S1    |                acc |
S2    |    s6    s7       |                 5
S3    |    s3    s4       |                 8
S4    |    r3    r3       |
S5    |                r1 |
S6    |    s6    s7       |                 9
S7    |                r3 |
S8    |    r2    r2       |
S9    |                r2 |

Enter input string to parse (must end with $): ccdd$
=== LR Parsing Trace for Input: ccdd$ ===
Stack    Input    Action
0        ccdd$    Shift to 3
0 3      cdd$     Shift to 3
0 3 3    dd$      Shift to 4
0 3 3 4  d$       Reduce by C -> d
0 3 3 8  d$       Reduce by C -> cC
0 3 8    d$       Reduce by C -> cC
0 2      d$       Shift to 7
0 2 7    $        Reduce by C -> d
0 2 5    $        Reduce by S -> CC
0 1      $        ACCEPT
PS C:\Notes\compiler\parsing>
```

Experiment-11(a)

Construct Look-Ahead LR (LALR) parse table using C language

Aim

The objective is to write a C program to construct a Look-Ahead LR (LALR) parse table for a given context-free grammar. The program should merge states with identical LR(0) items but different lookaheads, thereby reducing the number of states compared to CLR(1), while preserving parsing power for most practical grammars.

Algorithm

1. **Start**.
2. **Input grammar**: Take a context-free grammar with a single start symbol.
3. **Augment grammar**: Add a new start production S' → S.
4. **Construct LR(1) item sets**:
    - Compute the **canonical collection** of LR(1) items (as in CLR).
5. **Merge states**:
    - For all item sets that have identical LR(0) cores (same items ignoring lookaheads), merge them into a single set.
    - Combine lookahead symbols of corresponding items.
6. **Build parse table**:
    - For each merged state:
        - If an item has form [A → α • a β, t] and goto(I, a) = J, add ACTION[I, a] = shift J.
        - If an item has form [A → α •, t], add ACTION[I, t] = reduce A → α.
        - If an item has [S' → S •, $], then ACTION[I, $] = accept.
        - For each nonterminal A with goto(I, A) = J, set GOTO[I, A] = J.
7. **Check for conflicts**:
    - If both shift and reduce (or multiple reduces) occur, report grammar is not LALR(1).
8. **Output the parse table** (ACTION and GOTO).
9. **Stop**.

Experiment-11(b)

Implement the LR parsing algorithm using C language Aim

The objective is to write a C program that implements the LR parsing algorithm using the **LALR parse table**. The program should parse a given input string and determine whether it belongs to the grammar.

Algorithm

1. **Start**.
2. **Input parse table** (ACTION and GOTO) and **input string** to be parsed.
3. **Initialize stack** with state 0.
4. **Repeat until ACCEPT or ERROR**:
    - Let s = top of stack, and a = current input symbol.
    - If ACTION[s, a] = shift t:
        - Push a and state t onto the stack.
        - Advance input pointer.
    - Else if ACTION[s, a] = reduce A → β:

        - Pop 2 * |β| symbols from the stack.
        - Let s' = top of stack.
        - Push A.
        - Push GOTO[s', A].
        - Output reduction A → β.
    - Else if ACTION[s, a] = accept:
        - Report success, input string is valid.
        - Stop.
    - Else:
        - Report error, input string is invalid.
        - Stop.
5. **Stop**.

Code

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

#define MAX_PRODUCTIONS 64
#define MAX_RHS_LENGTH 64
#define MAX_SYMBOLS 64
#define MAX_ITEMS 512
#define MAX_STATES 512
#define MAX_SET_SIZE 64

typedef struct {
int prodIndex; int
dot;
char lookahead;
} Item;

typedef struct {
int prodIndex; int
dot;
} LR0_Item;

typedef struct {
Item items[MAX_ITEMS]; int
numItems;
} ItemSet;

typedef struct {
char lhs;
```

```
char rhs[MAX_RHS_LENGTH];
} Production;

typedef enum {
ACTION_ERROR,
ACTION_SHIFT,
ACTION_REDUCE,
ACTION_ACCEPT
} ActionType;

typedef struct {
ActionType type; int
target;
} ActionEntry;

Production productions[MAX_PRODUCTIONS];
int numProductions = 0;
char originalStartSymbol;
char augmentedStartSymbol = '\'';

char nonTerminals[MAX_SYMBOLS];
int numNonTerminals = 0;
char terminals[MAX_SYMBOLS];
int numTerminals = 0;
char alphabet[MAX_SYMBOLS];
int numAlphabet = 0;

char FIRST[256][MAX_SET_SIZE];
int FIRST_len[256] = {0};
int hasEpsilon[256] = {0};

ItemSet clr_states[MAX_STATES];
int num_clr_states = 0;
int transitions[MAX_STATES][MAX_SYMBOLS];

ItemSet lalr_states[MAX_STATES]; int
lalr_state_indices[MAX_STATES]; int
num_lalr_states = 0;

int lalr_merge_map[MAX_STATES][MAX_STATES];
int lalr_merge_count[MAX_STATES] = {0};
```

```
ActionEntry ACTION_TABLE[MAX_STATES][MAX_SYMBOLS];
int GOTO_TABLE[MAX_STATES][MAX_SYMBOLS];
int conflict_found = 0;

int contains_char(const char *s, char c) { return
strchr(s, c) != NULL;
}

void add_char_unique(char *s, int *len, char c) {
if (!contains_char(s, c)) {
s[*len] = c;
s[*len + 1] = '\0'; (*len)++;
}
}

int is_nonterminal(char c) { return
(c >= 'A' && c <= 'Z');
}

int get_symbol_index(char c) {
for (int i = 0; i < numAlphabet; ++i) { if
(alphabet[i] == c) return i;
}
return -1;
}

void read_grammar() {
printf("Enter the number of productions: "); scanf("%d",
&numProductions);
getchar();

if (numProductions <= 0 || numProductions >= MAX_PRODUCTIONS) {
fprintf(stderr, "Error: Invalid number of productions.\n"); exit(1);
}

printf("Enter productions (use empty for epsilon). Format: A-
>alpha\n");
for (int i = 0; i < numProductions; ++i) { char
line[256];
printf("P%d: ", i + 1); fgets(line,
sizeof(line), stdin);
```

```
line[strcspn(line, "\r\n")] = 0;

        if (strlen(line) < 3 || line[1] != '-' || line[2] != '>') {
        fprintf(stderr, "Error: Invalid production format '%s'.\n",
line);
        exit(1);
}
productions[i].lhs = line[0];
strcpy(productions[i].rhs, line + 3);
}
originalStartSymbol = productions[0].lhs;
}

void augment_grammar() {
for (int i = numProductions; i >= 1; --i) {
productions[i] = productions[i - 1];
}
numProductions++;
productions[0].lhs = augmentedStartSymbol;
productions[0].rhs[0] = originalStartSymbol;
productions[0].rhs[1] = '\0';
}

void collect_symbols() {
for (int i = 0; i < numProductions; ++i) { add_char_unique(nonTerminals,
&numNonTerminals,
productions[i].lhs);
}
for (int i = 0; i < numProductions; ++i) {
for (int j = 0; productions[i].rhs[j]; ++j) { char c =
productions[i].rhs[j];
if (c != '\0' && !is_nonterminal(c)) { add_char_unique(terminals,
&numTerminals, c);
}
}
}
add_char_unique(terminals, &numTerminals, '$');
for (int i = 0; i < numTerminals; ++i) alphabet[numAlphabet++] =
terminals[i];
for (int i = 0; i < numNonTerminals; ++i) alphabet[numAlphabet++] =
nonTerminals[i];
alphabet[numAlphabet] = '\0';
}
```

```
void compute_FIRST() {
int changed = 1; while
(changed) {
changed = 0;
for (int i = 0; i < numProductions; ++i) { char A =
productions[i].lhs;
const char *rhs = productions[i].rhs; int
before_len = FIRST_len[(int)A]; if (strlen(rhs) ==
0) {
if (!hasEpsilon[(int)A]) { hasEpsilon[(int)A] = 1;
changed = 1; }
} else {
int k = 0, all_have_epsilon = 1; while (k <
strlen(rhs)) {
char B = rhs[k];
if (!is_nonterminal(B)) { add_char_unique(FIRST[(int)A],
&FIRST_len[(int)A], B);
all_have_epsilon = 0; break;
}
for (int fi = 0; fi < FIRST_len[(int)B]; ++fi) {
add_char_unique(FIRST[(int)A],
&FIRST_len[(int)A], FIRST[(int)B][fi]);
}
if (!hasEpsilon[(int)B]) { all_have_epsilon = 0;
break; }
                k++;
                }
if (all_have_epsilon && !hasEpsilon[(int)A]) { hasEpsilon[(int)A] = 1;
changed = 1; }
}
if (FIRST_len[(int)A] != before_len) changed = 1;
}
}
}

void compute_first_of_string(const char *s, char *first_set, int
*first_len) {
*first_len = 0; first_set[0]
= '\0';
for (int i = 0; i < strlen(s); ++i) {
```

```
char symbol = s[i];
if (!is_nonterminal(symbol)) { add_char_unique(first_set,
first_len, symbol); return;
}
for (int j = 0; j < FIRST_len[(int)symbol]; ++j) {
add_char_unique(first_set, first_len,
FIRST[(int)symbol][j]);
}
if (!hasEpsilon[(int)symbol]) return;
}
}

int item_equal(Item a, Item b) {
return a.prodIndex == b.prodIndex && a.dot == b.dot && a.lookahead
== b.lookahead;
}

int is_item_in_set(const ItemSet *S, Item it) {
for (int i = 0; i < S->numItems; ++i) {
if (item_equal(S->items[i], it)) return 1;
}
return 0;
}

void add_item_to_set(ItemSet *S, Item it) { if
(!is_item_in_set(S, it)) {
S->items[S->numItems++] = it;
}
}

void closure(ItemSet *S) {
int changed = 1;
while (changed) { changed =
0;
for (int i = 0; i < S->numItems; ++i) { Item it =
S->items[i];
Production *p = &productions[it.prodIndex]; if (it.dot <
strlen(p->rhs)) {
char B = p->rhs[it.dot]; if
(is_nonterminal(B)) {
char beta_a[MAX_RHS_LENGTH + 2]; strcpy(beta_a, p->rhs + it.dot +
1);
```

```
                    beta_a[strlen(beta_a) + 1] = '\0';
                    beta_a[strlen(beta_a)] = it.lookahead;
                    char first_of_beta_a[MAX_SET_SIZE]; int
                    first_len;
                    compute_first_of_string(beta_a, first_of_beta_a,
&first_len);
                    for (int q = 0; q < numProductions; ++q) { if
                    (productions[q].lhs == B) {
                    for (int k = 0; k < first_len; ++k) { Item newItem
                    = { q, 0,
first_of_beta_a[k] };
                                if (!is_item_in_set(S, newItem)) {
                                add_item_to_set(S, newItem); changed =
                                1;
}
}
}
}
}
}
}
}
}

ItemSet compute_GOTO_set(const ItemSet *S, char X) {
ItemSet R = { .numItems = 0 };
for (int i = 0; i < S->numItems; ++i) { Item
it = S->items[i];
Production *p = &productions[it.prodIndex];
if (it.dot < strlen(p->rhs) && p->rhs[it.dot] == X) { Item
advancedItem = { it.prodIndex, it.dot + 1,
it.lookahead };
add_item_to_set(&R, advancedItem);
}
}
if (R.numItems > 0) closure(&R); return R;
}

int are_itemsets_equal(const ItemSet *A, const ItemSet *B) { if
(A->numItems != B->numItems) return 0;
for (int i = 0; i < A->numItems; ++i) {
if (!is_item_in_set(B, A->items[i])) return 0;
```

```
}
return 1;
}

int find_clr_state_index(const ItemSet *X) { for
(int s = 0; s < num_clr_states; ++s) {
if (are_itemsets_equal(&clr_states[s], X)) return s;
}
return -1;
}

void build_canonical_collection() {
memset(transitions, -1, sizeof(transitions));
ItemSet I0 = { .numItems = 0 };
add_item_to_set(&I0, (Item){ 0, 0, '$' });
closure(&I0);
clr_states[num_clr_states++] = I0;
for (int s = 0; s < num_clr_states; ++s) { for
(int ai = 0; ai < numAlphabet; ++ai) {
char X = alphabet[ai];
ItemSet next_set = compute_GOTO_set(&clr_states[s], X); if
(next_set.numItems == 0) continue;
int existing_idx = find_clr_state_index(&next_set); if
(existing_idx == -1) {
if (num_clr_states >= MAX_STATES) { fprintf(stderr, "Error:
Exceeded maximum
states.\n"); exit(1);
}
clr_states[num_clr_states] = next_set; transitions[s][ai]
= num_clr_states; num_clr_states++;
} else {
transitions[s][ai] = existing_idx;
}
}
}
}

int compare_lr0_items(const void *a, const void *b) {
LR0_Item *itemA = (LR0_Item *)a;
LR0_Item *itemB = (LR0_Item *)b;
if (itemA->prodIndex != itemB->prodIndex) return itemA->prodIndex -
itemB->prodIndex;
```

```
return itemA->dot - itemB->dot;
}

int get_unique_cores(const ItemSet *S, LR0_Item *cores) {
int numCores = 0;
for (int i = 0; i < S->numItems; ++i) { int
found = 0;
for (int j = 0; j < numCores; ++j) {
if (cores[j].prodIndex == S->items[i].prodIndex && cores[j].dot == S-
>items[i].dot) {
found = 1; break;
}
}
if (!found) {
cores[numCores].prodIndex = S->items[i].prodIndex; cores[numCores].dot =
S->items[i].dot;
numCores++;
}
}
return numCores;
}

int are_cores_equal(const ItemSet *A, const ItemSet *B) {
LR0_Item coresA[MAX_ITEMS], coresB[MAX_ITEMS];
int numCoresA = get_unique_cores(A, coresA); int
numCoresB = get_unique_cores(B, coresB); if
(numCoresA != numCoresB) return 0;
qsort(coresA, numCoresA, sizeof(LR0_Item), compare_lr0_items);
qsort(coresB, numCoresB, sizeof(LR0_Item), compare_lr0_items); return
memcmp(coresA, coresB, numCoresA * sizeof(LR0_Item)) == 0;
}

void set_action(int state, int symbol_idx, ActionType type, int target,
char symbol) {
if (ACTION_TABLE[state][symbol_idx].type != ACTION_ERROR) { if
(ACTION_TABLE[state][symbol_idx].type == type &&
ACTION_TABLE[state][symbol_idx].target == target) {
return;
}
conflict_found = 1;
fprintf(stderr, "\n--- LALR(1) CONFLICT in state %d on '%c' ---
\n", state, symbol);
```

```
fprintf(stderr, "  Existing: %s by %d\n",
ACTION_TABLE[state][symbol_idx].type == ACTION_SHIFT ? "SHIFT" :
"REDUCE", ACTION_TABLE[state][symbol_idx].target);
fprintf(stderr, "  New:                %s by %d\n", type == ACTION_SHIFT
? "SHIFT" : "REDUCE", target);
return;
}
ACTION_TABLE[state][symbol_idx].type = type;
ACTION_TABLE[state][symbol_idx].target = target;
}

void build_LALR_table() {
int clr_to_lalr_map[MAX_STATES]; memset(clr_to_lalr_map, -1,
sizeof(clr_to_lalr_map));
num_lalr_states = 0;

for (int i = 0; i < num_clr_states; ++i) { if
(clr_to_lalr_map[i] != -1) continue;

int merge_group[MAX_STATES]; int
merge_group_count = 0;
merge_group[merge_group_count++] = i;

for (int j = i + 1; j < num_clr_states; ++j) {
if (are_cores_equal(&clr_states[i], &clr_states[j])) {
merge_group[merge_group_count++] = j;
}
}

int compare_ints(const void *a, const void *b) { return (*(int*)a -
*(int*)b); }
qsort(merge_group, merge_group_count, sizeof(int), compare_ints);

int canonical_state_num;
if (merge_group_count == 1) { canonical_state_num =
merge_group[0];
} else {
char buffer[256] = ""; char
temp[32];
for (int j = 0; j < merge_group_count; j++) { sprintf(temp, "%d",
merge_group[j]); strcat(buffer, temp);
```

```
}
canonical_state_num = atoi(buffer);
}

if (canonical_state_num >= MAX_STATES) {
fprintf(stderr, "Error: Merged state number %d exceeds MAX_STATES limit
of %d.\n", canonical_state_num, MAX_STATES);
exit(1);
}

lalr_state_indices[num_lalr_states++] = canonical_state_num; ItemSet
new_lalr_state = { .numItems = 0 };

for(int k=0; k < merge_group_count; k++) { int
clr_idx = merge_group[k];
clr_to_lalr_map[clr_idx] = canonical_state_num;

lalr_merge_map[canonical_state_num][lalr_merge_count[canonical_state_nu
m]++] = clr_idx;
for (int item_idx = 0; item_idx <
clr_states[clr_idx].numItems; ++item_idx) {
add_item_to_set(&new_lalr_state, clr_states[clr_idx].items[item_idx]);
}
}
lalr_states[canonical_state_num] = new_lalr_state;
}

for (int s = 0; s < MAX_STATES; ++s) {
for (int a = 0; a < MAX_SYMBOLS; ++a) {
ACTION_TABLE[s][a].type = ACTION_ERROR; GOTO_TABLE[s][a]
= -1;
}
}

for (int i = 0; i < num_clr_states; ++i) { int
lalr_s = clr_to_lalr_map[i];
for (int ai = 0; ai < numAlphabet; ++ai) { int
clr_t = transitions[i][ai];
if (clr_t == -1) continue;
int lalr_t = clr_to_lalr_map[clr_t]; char X =
alphabet[ai];
if (!is_nonterminal(X)) {
```

```
set_action(lalr_s, ai, ACTION_SHIFT, lalr_t, X);
} else {
GOTO_TABLE[lalr_s][ai] = lalr_t;
}
}
}

for (int k = 0; k < num_lalr_states; ++k) { int s
= lalr_state_indices[k];
for (int i = 0; i < lalr_states[s].numItems; ++i) { Item it =
lalr_states[s].items[i];
if (it.dot == strlen(productions[it.prodIndex].rhs)) { int col =
get_symbol_index(it.lookahead);
if (col < 0) continue; if (it.prodIndex
== 0) {
set_action(s, col, ACTION_ACCEPT, 0, it.lookahead);
} else {
set_action(s, col, ACTION_REDUCE, it.prodIndex,
it.lookahead);
}
}
}
}
}

void print_productions_list() {
printf("\n=== Grammar Productions ===\n"); for
(int i = 0; i < numProductions; ++i) {
printf("  P%-2d: %c -> %s\n", i, productions[i].lhs,
(strlen(productions[i].rhs) == 0 ? "ε" : productions[i].rhs));
}
}

void print_state_merge_map() {
printf("\n=== LALR(1) State Merge Map ===\n"); for
(int k = 0; k < num_lalr_states; k++) {
int i = lalr_state_indices[k];
printf("LALR State %d is a merge of CLR State(s): ", i); for (int
j = 0; j < lalr_merge_count[i]; j++) {
printf("%d ", lalr_merge_map[i][j]);
}
printf("\n");
}
```

```
}


void print_states() {
printf("\n=== LALR(1) Item Sets (%d states) ===\n",
num_lalr_states);
for (int k = 0; k < num_lalr_states; ++k) { int s
= lalr_state_indices[k]; printf("I%d:\n", s);

LR0_Item printed_cores[MAX_ITEMS]; int
num_printed_cores = 0;

for (int i = 0; i < lalr_states[s].numItems; ++i) { LR0_Item
current_core = {
lalr_states[s].items[i].prodIndex, lalr_states[s].items[i].dot };

int already_printed = 0;
for (int j = 0; j < num_printed_cores; ++j) { if
(printed_cores[j].prodIndex ==
current_core.prodIndex && printed_cores[j].dot == current_core.dot) {
already_printed = 1;
break;
}
}
if (already_printed) continue;

Production *p = &productions[current_core.prodIndex]; printf("  [%c ->",
p->lhs);
for (int j = 0; j < strlen(p->rhs); ++j) { if (j ==
current_core.dot) printf(" ·"); printf(" %c", p->rhs[j]);
}
if (current_core.dot == strlen(p->rhs)) printf(" ·");

printf(", ");
int first_lookahead = 1;
for (int l = 0; l < lalr_states[s].numItems; ++l) { if
(lalr_states[s].items[l].prodIndex ==
current_core.prodIndex && lalr_states[s].items[l].dot ==
current_core.dot) {
if (!first_lookahead) { printf("/");
```

```
}
printf("%c", lalr_states[s].items[l].lookahead); first_lookahead = 0;
}
}
printf("]\n");

printed_cores[num_printed_cores++] = current_core;
}
}
}

void print_parsing_table() {
printf("\n=== LALR(1) Parsing Table ===\n");

int action_width = numTerminals * 6;
int goto_width = (numNonTerminals - 1) * 6;

printf("%-7s|", "State");

int action_padding_left = (action_width > 6) ? (action_width - 6) /
2 : 0;
int action_padding_right = (action_width > 6) ? (action_width - 6 -
action_padding_left) : 0;
for(int i = 0; i < action_padding_left; ++i) printf(" ");
printf("Action");
for(int i = 0; i < action_padding_right; ++i) printf(" "); printf("|");

int goto_padding_left = (goto_width > 4) ? (goto_width - 4) / 2 :
0;
int goto_padding_right = (goto_width > 4) ? (goto_width - 4 -
goto_padding_left) : 0;
for(int i = 0; i < goto_padding_left; ++i) printf(" "); printf("Goto");
for(int i = 0; i < goto_padding_right; ++i) printf(" "); printf("\n");

printf("%-7s|", "");
for (int i = 0; i < numTerminals; ++i) printf("  %-4c", terminals[i]);
printf("|");
for (int i = 0; i < numNonTerminals; ++i) {
```

```
if (nonTerminals[i] != augmentedStartSymbol) { printf("
%-4c", nonTerminals[i]);
}
}
printf("\n");

printf("------------|");
for (int i = 0; i < action_width; ++i) printf("-");
printf("|");
for (int i = 0; i < goto_width; ++i) printf("-");
printf("\n");

for (int k = 0; k < num_lalr_states; ++k) { int s
= lalr_state_indices[k];
printf(" %-6d|", s);
for (int i = 0; i < numTerminals; ++i) {
int col = get_symbol_index(terminals[i]); ActionEntry e =
ACTION_TABLE[s][col];
if (e.type == ACTION_SHIFT) printf(" s%-4d", e.target); else if
(e.type == ACTION_REDUCE) printf(" r%-4d",
e.target);
else if (e.type == ACTION_ACCEPT) printf("  ACC "); else
printf("                        ");
}
printf("|");
for (int i = 0; i < numNonTerminals; ++i) {
if (nonTerminals[i] == augmentedStartSymbol) continue; int col =
get_symbol_index(nonTerminals[i]);
if (GOTO_TABLE[s][col] >= 0) printf(" %-5d", GOTO_TABLE[s][col]);
else printf("                    ");
}
printf("\n");
}
}

typedef struct { int st[1024]; int top; } IntStack; void
push(IntStack *S, int x) { S->st[++(S->top)] = x; } int
pop(IntStack *S) { return S->st[(S->top)--]; }
int peek(IntStack *S) { return S->st[S->top]; }

void parse_input(const char *input) {
printf("\n=== LR Parsing Trace for Input: %s ===\n", input);
```

```
printf("%-30s %-20s %s\n", "Stack", "Input", "Action"); printf("
\n");       ----------------------------------------------------------------------------------------------------------------------------------------
----------------
IntStack stack;
stack.top = -1;
push(&stack, 0);

int ip = 0; while (1)
{
char stack_str[256] = "";
for(int i = 0; i <= stack.top; ++i) sprintf(stack_str +
strlen(stack_str), "%d ", stack.st[i]);
printf("%-30s %-20s ", stack_str, &input[ip]);

int s = peek(&stack); char a =
input[ip];
int col = get_symbol_index(a);
if (col < 0) { printf("Error: Invalid input symbol '%c'\n", a); return;
}

ActionEntry entry = ACTION_TABLE[s][col]; if
(entry.type == ACTION_SHIFT) {
printf("Shift to %d\n", entry.target); push(&stack,
entry.target);
ip++;
} else if (entry.type == ACTION_REDUCE) { Production *p =
&productions[entry.target]; printf("Reduce by %c -> %s\n",
p->lhs, p->rhs);
for (int k = 0; k < strlen(p->rhs); ++k) pop(&stack); int t =
peek(&stack);
int A_col = get_symbol_index(p->lhs); push(&stack,
GOTO_TABLE[t][A_col]);
} else if (entry.type == ACTION_ACCEPT) {
printf("ACCEPT\n"); return;
} else {
printf("ERROR: No action defined\n"); return;
}
}
}

int main() {
read_grammar();
```

```
augment_grammar();
collect_symbols();
compute_FIRST();

build_canonical_collection(); build_LALR_table();

print_productions_list();
print_state_merge_map();
print_states();
print_parsing_table();

if (conflict_found) {
fprintf(stderr, "\nParsing table construction failed due to
conflicts.\n");
return 1;
}

char input[256];
printf("\nEnter input string to parse (must end with $): "); fgets(input,
sizeof(input), stdin);
input[strcspn(input, "\r\n")] = 0;

if (strlen(input) == 0 || input[strlen(input) - 1] != '$') {
fprintf(stderr, "Error: Input must be non-empty and end with
$.\n");
return 1;
}

parse_input(input);

return 0;
}



student@AB1605B049:~/compiler_1643$ gcc CLR.c -o bind
student@AB1605B049:~/compiler_1643$ ./bind
Enter the number of productions: 3
Enter productions (use empty for epsilon). Format: A->alpha P1:
S->CC
P2: C->cC P3:
C->d
```

```
=== Grammar Productions === P0
: ' -> S
P1 : S -> CC P2 :
C -> cC P3 : C ->
d

=== Canonical LR(1) Item Sets (10 states) === I0:
[' -> .S, $]
[S -> .C C, $]
[C -> .c C, c]
[C -> .c C, d]
[C -> .d, c]
[C -> .d, d]

I1:
     [C -> c .C, c]
     [C -> c .C, d]
     [C -> .c C, c]
     [C -> .c C, d]
     [C -> .d, c]
     [C -> .d, d]

I2:
     [C -> d ., c]
     [C -> d ., d]

I3:
[' -> S ., $]

I4:
     [S -> C .C, $]
     [C -> .c C, $]
     [C -> .d, $]

I5:
     [C -> c C ., c]
     [C -> c C ., d]

I6:
     [C -> c .C, $]
     [C -> .c C, $]
     [C -> .d, $]
```

```
I7:
[C -> d ., $]

I8:
[S -> C C ., $]

I9:
[C -> c C ., $]

=== CLR(1) Parsing Table ===
 State |      c     d     $ |   S'       S   C
 S0    |   s3    s4         |            1   2
 S1    |                acc |
 S2    |   s6    s7         |                5
 S3    |   s3    s4         |                8
 S4    |   r3    r3         |
 S5    |                 r1 |
 S6    |   s6    s7         |                9
 S7    |                 r3 |
 S8    |     r2    r2       |
 S9    |                 r2 |

Enter input string to parse (must end with $): ccdd$
=== LR Parsing Trace for Input: ccdd$ ===
Stack      Input    Action
 0        ccdd$   Shift   to 3
 0 3      cdd$    Shift   to 3
 0 3 3    dd$     Shift   to 4
 0 3 3 4  d$      Reduce   by C -> d
 0 3 3 8  d$      Reduce   by C -> cC
 0 3 8    d$      Reduce   by C -> cC
 0 2   d$         Shift to 7
 0 2 7 $          Reduce by C   -> d
 0 2 5 $          Reduce by S   -> CC
0 1        $        ACCEPT
PS C:\Notes\compiler\parsing> &
C:\Users\ksgsa\AppData\Local\Programs\Python\Python313\python.exe
c:/Notes/compiler/parsing/lalr.py student@AB1605B049:~/compiler_1643$
gcc LALR.c -o bin2
/usr/bin/ld: warning: /tmp/cc4YnD4M.o: requires executable stack
(because the .note.GNU-stack section is executable)
student@AB1605B049:~/compiler_1643$ ./bin2
```

```
Enter the number of productions: 3
Enter productions (use empty for epsilon). Format: A->alpha P1:
S->CC
P2: C->cC P3:
C->d

=== Grammar Productions === P0
: ' -> S
P1 : S -> CC P2
: C -> cC P3 : C
-> d

=== LALR(1) State Merge Map ===
LALR State 0 is a merge of CLR State(s): 0 LALR
State 16 is a merge of CLR State(s): 1 6 LALR
State 27 is a merge of CLR State(s): 2 7 LALR
State 3 is a merge of CLR State(s): 3 LALR
State 4 is a merge of CLR State(s): 4 LALR
State 59 is a merge of CLR State(s): 5 9 LALR
State 8 is a merge of CLR State(s): 8

=== LALR(1) Item Sets (7 states) === I0:
[' -> · S, $]
[S -> · C C, $]
   [C ->  ·c C, c/d]
 [C   ->  ·d, c/d]
 I16:
 [C   ->  c · C, c/d/$]
   [C ->  ·c C, c/d/$]
 [C   ->  ·d, c/d/$]
 I27:
 [C   ->  d ·, c/d/$]
 I3:
 ['   ->  S ·, $]
 I4:
 16      |s16   s27        |     59
 27      |r3    r3      r3  |
 3    |                ACC |
 4       |s16   s27        |     8
 59      |r2    r2      r2  |
8       |             r1   |
```

```
Enter input string to parse (must end with $): ccdd$

=== LR Parsing Trace for Input: ccdd$ ===
Stack                           Input                Action
------------------------------------------------------------------------------------------
0                       ccdd$                        Shift to 16
0 16                    cdd$                         Shift to 16
0 16 16                 dd$                          Shift to 27
0 16 16 27              d$                           Reduce by C -> d
0 16 16 59              d$                           Reduce by C -> cC
0 16 59                 d$                           Reduce by C -> cC
0 4                     d$                           Shift to 27
0 4 27                  $                            Reduce by C -> d
0 4 8                   $                            Reduce by S -> CC
0 3                     $                            ACCEPT
```

Output

```
student@AB1605B049:~/compiler_1643$ gcc LALR.c -o bin2
/usr/bin/ld: warning: /tmp/cc4YnD4M.o: requires executable stack
(because the .note.GNU-stack section is executable)
student@AB1605B049:~/compiler_1643$ ./bin2
Enter the number of productions: 3
Enter productions (use empty for epsilon). Format: A->alpha
P1: S->CC
P2: C->cC
P3: C->d

=== Grammar Productions ===
P0 : ' -> S
P1 : S -> CC P2
: C -> cC P3 :
C -> d

=== LALR(1) State Merge Map ===
LALR State 0 is a merge of CLR State(s): 0
LALR State 16 is a merge of CLR State(s): 1 6
LALR State 27 is a merge of CLR State(s): 2 7
LALR State 3 is a merge of CLR State(s): 3
LALR State 4 is a merge of CLR State(s): 4
LALR State 59 is a merge of CLR State(s): 5 9
LALR State 8 is a merge of CLR State(s): 8
```

```
=== LALR(1) Item Sets (7 states) === I0:

    [' -> ·S, $]
    [S -> ·C C, $]
    [C -> ·c C, c/d]
    [C -> ·d, c/d]
  I16:
  [C   -> c · C, c/d/$]
    [C -> ·c C, c/d/$]
    [C -> ·d, c/d/$]
  I27:
  [C   -> d ·, c/d/$]
  I3:
  ['   -> S ·, $]
  I4:
  [S   -> C · C, $]
    [C -> ·c C, $]
    [C -> ·d, $]
  I59:
I [C   -> c C ·, c/d/$]
  [S -> C C ·, $]

  === LALR(1) Parsing Table === State
  |               Action       |    Goto
  |  c              d     $    |  S     C
  |                            |
---0------ -----|---s16------s27------------ ---|---3----------4-----
  16       | s16   s27          |        59
  27       | r3    r3    r3     |
  3        |               ACC  |
  4        | s16   s27          |        8
  59       | r2    r2    r2     |
  8        |             r1     |

  Enter input string to parse (must end with $): ccdd$

  === LR Parsing Trace for Input: ccdd$ ===
  Stack                           Input                Action

-----0                  ccdd$                          Shift to 16---------
    0 16               cdd$                           Shift to 16
    0 16 16            dd$                            Shift to 27
```

```
0 16  16 27                    d$                  Reduce by C -> d
0 16  16 59                    d$                  Reduce by C -> cC
0 16  59                       d$                  Reduce by C -> cC
0 4                            d$          Shift to 27
0 4  27                        $           Reduce by C          -> d
0 4  8                         $           Reduce by S          -> CC
0 3                            $           ACCEPT
student@AB1605B049:~/compiler_1643$
```

```
student@AB1605B049:~/compiler_1643$ gcc LALR.c -o bin2
/usr/bin/ld: warning: /tmp/cc4YnD4M.o: requires executable stack (because the .note.GNU-stack section is executabl
student@AB1605B049:~/compiler_1643$ ./bin2
Enter the number of productions: 3
Enter productions (use empty for epsilon). Format: A->alpha
P1: S->CC
P2: C->cC
P3: C->d

=== Grammar Productions ===
  P0 : ' -> S
  P1 : S -> CC
  P2 : C -> cC
  P3 : C -> d

=== LALR(1) State Merge Map ===
LALR State 0 is a merge of CLR State(s): 0
LALR State 16 is a merge of CLR State(s): 1 6
LALR State 27 is a merge of CLR State(s): 2 7
LALR State 3 is a merge of CLR State(s): 3
LALR State 4 is a merge of CLR State(s): 4
LALR State 59 is a merge of CLR State(s): 5 9
LALR State 8 is a merge of CLR State(s): 8

=== LALR(1) Item Sets (7 states) ===
I0:
  [' -> · S, $]
  [S -> · C C, $]
  [C -> · c C, c/d]
  [C -> · d, c/d]
I16:
  [C -> c · C, c/d/$]
  [C -> · c C, c/d/$]
  [C -> · d, c/d/$]
I27:
  [C -> d ·, c/d/$]
I3:
  [' -> S ·, $]
I4:
  [S -> C · C, $]
  [C -> · c C, $]
  [C -> · d, $]
I59:
  [C -> c C ·, c/d/$]
I8:
  [S -> C C ·, $]
```

```
  [S -> C · C, $]
  [C -> · c C, $]
  [C -> · d, $]
I59:
  [C -> c C ·, c/d/$]
I8:
  [S -> C C ·, $]

=== LALR(1) Parsing Table ===
State  |      Action      |    Goto
       |  c     d     $   |  S     C
-------|------------------|------------
 0     | s16   s27        | 3     4
 16    | s16   s27        |       59
 27    | r3    r3    r3   |
 3     |               ACC |
 4     | s16   s27        |       8
 59    | r2    r2    r2   |
 8     |             r1   |

Enter input string to parse (must end with $): ccdd$

=== LR Parsing Trace for Input: ccdd$ ===
Stack                          Input               Action
-------------------------------------------------------------------------
0                              ccdd$               Shift to 16
0 16                           cdd$                Shift to 16
0 16 16                        dd$                 Shift to 27
0 16 16 27                     d$                  Reduce by C -> d
0 16 16 59                     d$                  Reduce by C -> cC
0 16 59                        d$                  Reduce by C -> cC
0 4                            d$                  Shift to 27
0 4 27                         $                   Reduce by C -> d
0 4 8                          $                   Reduce by S -> CC
0 3                            $                   ACCEPT
student@AB1605B049:~/compiler_1643$
```

Experiment 12

Aim:

Implementation of a simple calculator using LEX and YACC tools.

Algorithm:

1. Create a LEX file to identify tokens such as numbers and operators (+, -, *, /, (,)).
2. Create a YACC file that defines grammar rules for valid arithmetic expressions.
3. Use YACC actions to perform calculations according to operator precedence.
4. In the main program, call yyparse() to start parsing input expressions.
5. When a valid expression is entered, the corresponding arithmetic result is displayed.

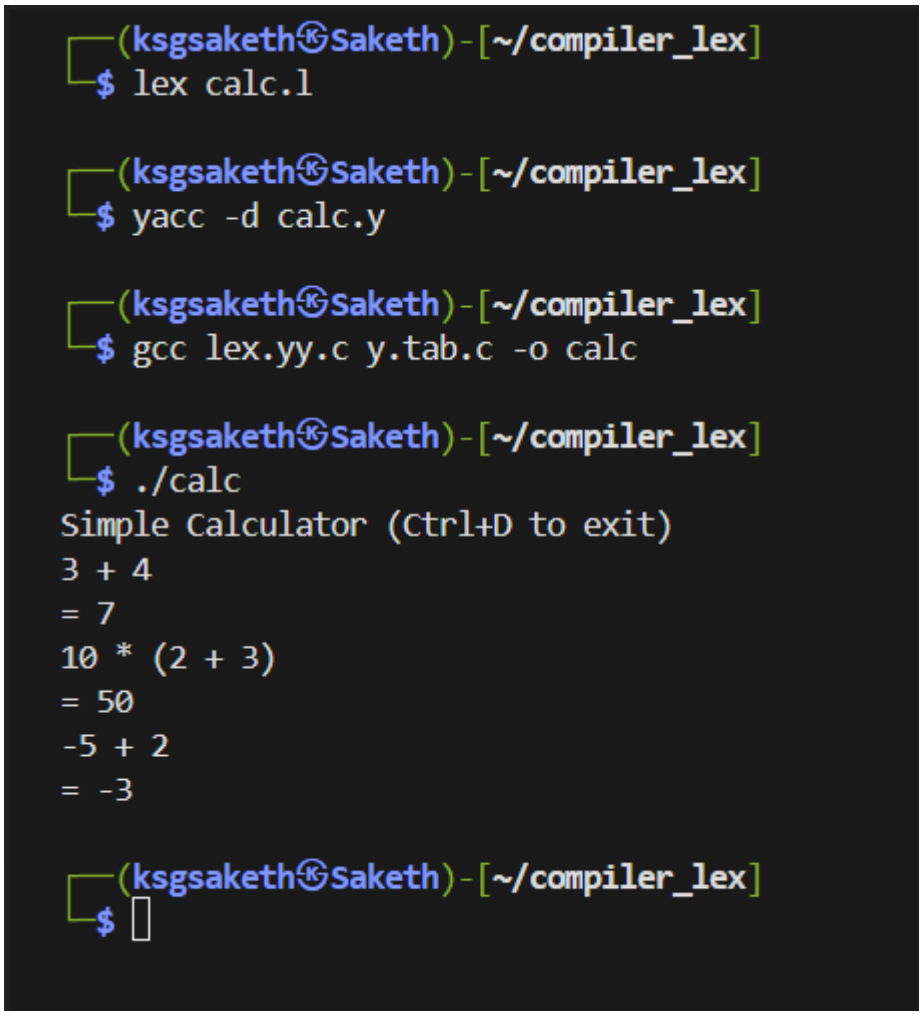
Code:

```
%{
#include "y.tab.h"
#include <stdlib.h>
%}

%%
[0-9]+      { yylval = atoi(yytext); return NUMBER; }
[ \t]       ;
\n          { return EOL; }
.           { return yytext[0]; }
%%

int yywrap() {
    return 1;
}
```

```
%{
#include <stdio.h>
#include <stdlib.h>

int yylex();
void yyerror(const char *s);
%}

%token NUMBER
%token EOL

%left '+' '-'
```

```
%left '*' '/'
%right UMINUS

%%
input: /* empty */
     | input line
     ;

line: EOL
    | expr EOL   { printf("= %d\n", $1); }
    ;

expr: NUMBER                { $$ = $1; }
    | expr '+' expr         { $$ = $1 + $3; }
    | expr '-' expr         { $$ = $1 - $3; }
    | expr '*' expr         { $$ = $1 * $3; }
    | expr '/' expr         { $$ = $1 / $3; }
    | '(' expr ')'          { $$ = $2; }
    | '-' expr %prec UMINUS { $$ = -$2; }
    ;
%%

void yyerror(const char *s) {
    fprintf(stderr, "Error: %s\n", s);
}

int main() {
    printf("Simple Calculator (Ctrl+D to exit)\n");
    yyparse();
    return 0;
}
```

Output:


(ksgsaketh㉿Saketh)-[~/compiler_lex]
$ lex calc.l
(ksgsaketh㉿Saketh)-[~/compiler_lex]
$ yacc -d calc.y
(ksgsaketh㉿Saketh)-[~/compiler_lex]
$ gcc lex.yy.c y.tab.c -o calc
(ksgsaketh㉿Saketh)-[~/compiler_lex]
$ ./calc
Simple Calculator (Ctrl+D to exit)
3 + 4
= 7
10 * (2 + 3)
= 50
-5 + 2
= -3
(ksgsaketh㉿Saketh)-[~/compiler_lex]
$

Experiment 13

Aim:

Implementation of Abstract syntax tree –Infix to postfix using the LEX and YACC tools.

Algorithm:

1. Create a LEX file to recognize operands (identifiers, numbers) and operators (+, -, *, /, (,)).
2. Create a YACC file to define grammar rules for infix expressions.
3. For each grammar rule, build nodes of an Abstract Syntax Tree (AST) representing the expression.
4. After parsing, traverse the AST in postorder to convert the infix expression into postfix form.
5. Display the generated postfix expression.

Code:

```
%{
#include "y.tab.h"
%}

%%
[0-9]+      { yylval.num = atoi(yytext); return NUMBER; }
[ \t\n]     ;
"+"         { return PLUS; }
"-"         { return MINUS; }
"*"         { return MUL; }
"/"         { return DIV; }
"("         { return LPAREN; }
")"         { return RPAREN; }
.           { return yytext[0]; }
%%

int yywrap() { return 1; }
```

```
%{
#include <stdio.h>
#include <stdlib.h>

typedef struct node {
    char op;
    int val;
    struct node *left, *right;
```

```
} Node;

Node* makeNode(char op, int val, Node* left, Node* right) {
    Node* n = (Node*)malloc(sizeof(Node));
    n->op = op;
    n->val = val;
    n->left = left;
    n->right = right;
    return n;
}

void printPostfix(Node* root) {
    if (!root) return;
    printPostfix(root->left);
    printPostfix(root->right);
    if (root->op == 'n')
        printf("%d ", root->val);
    else
        printf("%c ", root->op);
}

Node* root;

/* Forward declarations */
int yylex();
int yyerror(char* s);
%}

%union {
    int num;
    struct node* n;
}

%token <num> NUMBER
%type <n> expr
%left PLUS MINUS
%left MUL DIV
%right UMINUS
%token LPAREN RPAREN

%%

input: expr { root = $1; printPostfix(root); printf("\n"); }
     ;

expr:
      expr PLUS expr   { $$ = makeNode('+', 0, $1, $3); }
    | expr MINUS expr  { $$ = makeNode('-', 0, $1, $3); }
```

```
    | expr MUL expr    { $$ = makeNode('*', 0, $1, $3); }
    | expr DIV expr    { $$ = makeNode('/', 0, $1, $3); }
    | LPAREN expr RPAREN { $$ = $2; }
    | MINUS expr %prec UMINUS { $$ = makeNode('-', 0,
makeNode('n',0,NULL,NULL), $2); }
    | NUMBER { $$ = makeNode('n', $1, NULL, NULL); }
    ;

%%
int main() {
    return yyparse();
}

int yyerror(char* s) {
    printf("Error: %s\n", s);
    return 0;
}
```

Output:

```
┌──(ksgsaketh㉿Saketh)-[~/compiler_lex]
└─$ lex ast.l

┌──(ksgsaketh㉿Saketh)-[~/compiler_lex]
└─$ yacc -d ast.y

┌──(ksgsaketh㉿Saketh)-[~/compiler_lex]
└─$ gcc lex.yy.c y.tab.c -o ast

┌──(ksgsaketh㉿Saketh)-[~/compiler_lex]
└─$ ./ast
(3+4)*2-5
3 4 + 2 * 5 -

┌──(ksgsaketh㉿Saketh)-[~/compiler_lex]
└─$
```

Experiment 14

Aim:

Using LEX and YACC tools to recognize the strings of the following context-free languages:

1. L(G) = { anbm / m ≠ n}
2. L(G) = { ab (bbaa)n bba (ba)n / n ≥ 0}

Algorithm:

1)

1. Create a LEX file to identify symbols a and b.
2. In the YACC file, define rules to count the number of a's and b's.
3. After reading the input string, compare the two counts.
4. If the counts are not equal (m ≠ n), accept the string; otherwise, reject it.

2)

1. Create a LEX file to recognize symbols a and b.
2. In the YACC file, define grammar rules that generate the pattern ab, followed by zero or more (bbaa), then bba, and finally the same number of (ba) pairs.
3. If the input string matches this pattern, accept it; otherwise, reject it.

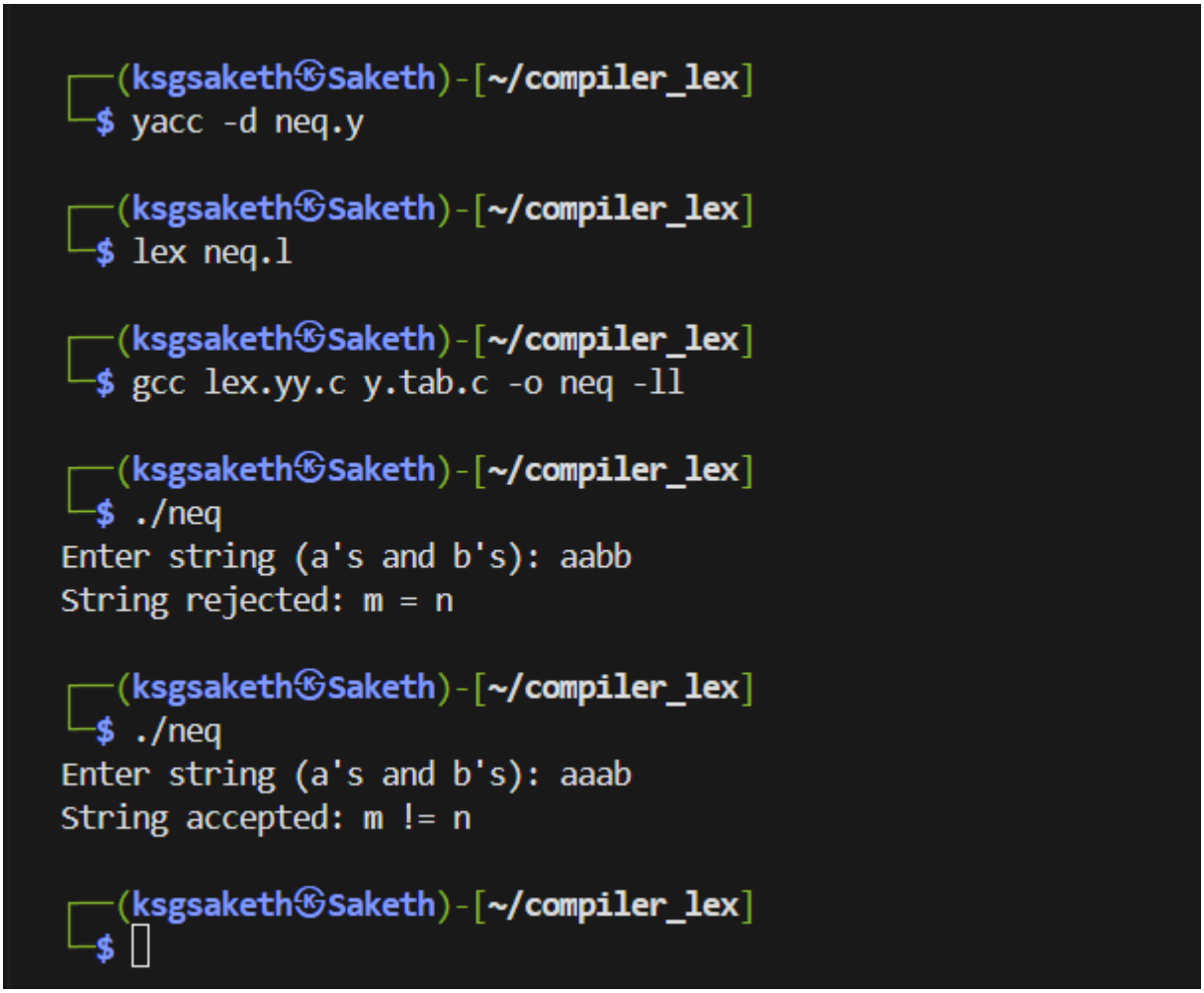
Code:

```
%{
#include "y.tab.h"
%}

%%
a   { return A; }
b   { return B; }
\n  { return 0; }
.   ;
%%
int yywrap() { return 1; }
```

```
%{
#include <stdio.h>

int yylex(void);
void yyerror(char *s);
int aCount = 0, bCount = 0;
%}

%token A B

%%
start: str {
```

```
    if (aCount != bCount)
        printf("String accepted: m != n\n");
    else
        printf("String rejected: m = n\n");
};

str: str A { aCount++; }
   | str B { bCount++; }
   | /* empty */
   ;
%%

int main() {
    printf("Enter string (a's and b's): ");
    yyparse();
    return 0;
}

void yyerror(char *s) {
    // Just suppress error message for clean output
}
```

Output:

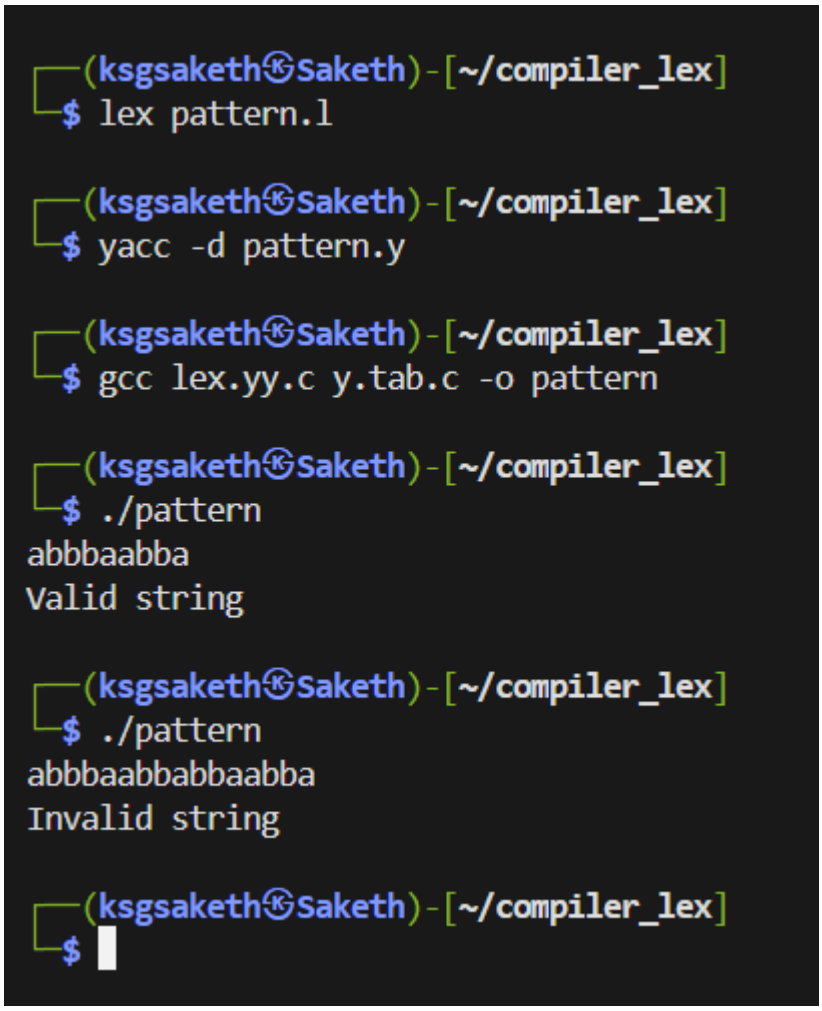


Code:

```
%{
#include "y.tab.h"
%}
%option noyywrap

%%
a   { return A; }
b   { return B; }
\n  { return 0; }
.   ;
%%
```

```
%{
#include <stdio.h>
#include <stdlib.h>

int yylex(void);          /* prototype for lexer */
int yyerror(const char *); /* prototype for error handler */
%}

%token A B

%%
S : AB X BBA Y   { printf("Valid string\n"); }
  ;

X : /* empty */
  | X BBAA
  ;

Y : /* empty */
  | Y BA
  ;

AB  : A B ;
BBA : B B A ;
BBAA: B B A A ;
BA  : B A ;
%%

int main() {
    yyparse();
    return 0;
}

int yyerror(const char *s) {
    printf("Invalid string\n");
    return 0;
}
```

Output:


(ksgsaketh㉿Saketh)-[~/compiler_lex]
$ lex pattern.l
(ksgsaketh㉿Saketh)-[~/compiler_lex]
$ yacc -d pattern.y
(ksgsaketh㉿Saketh)-[~/compiler_lex]
$ gcc lex.yy.c y.tab.c -o pattern
(ksgsaketh㉿Saketh)-[~/compiler_lex]
$ ./pattern
abbbaabba
Valid string
(ksgsaketh㉿Saketh)-[~/compiler_lex]
$ ./pattern
abbbaabbabbaabba
Invalid string
(ksgsaketh㉿Saketh)-[~/compiler_lex]
$

Experiment 15

Implementation of Three Address Codes using LEX and YACC

Aim:

To design and implement a compiler phase using LEX and YACC tools that takes a simple high-level language program (e.g., arithmetic and assignment statements) as input and generates the corresponding Three-Address Code (TAC) as output.

Algorithm:

1. **Define the Grammar:** Specify the context-free grammar for the source language. This typically includes rules for expressions, assignment statements, and possibly control flow statements.
    - Example: S -> id = E;, E -> E + T | T, T -> T * F | F, F -> (E) | id | number
2. **LEX Specification (Scanner):**
    - Write a lex.l file that defines regular expressions for tokens.
    - **Tokens:** Identify keywords (e.g., if, while), operators (e.g., +, -, *, /), identifiers ([a-zA-Z_][a-zA-Z0-9_]*), numbers ([0-9]+), and parentheses.
    - **Action:** For each token matched, return the corresponding token type to the YACC parser.
3. **YACC Specification (Parser & TAC Generator):**
    - Write a yacc.y file that uses the grammar defined in step 1.
    - **Semantic Actions:** Associate semantic actions with each grammar production.
    - **Attribute Handling:** Use attributes (e.g., $$, $1, $2) to pass values (like symbol table entries or temporary variable names) up the parse tree.
    - **TAC Generation:**
        - For an assignment A = B + C * D;, the action should generate TAC lines like:
            - t1 = C * D
            - t2 = B + t1
            - A = t2
        - Create a new temporary variable (e.g., t1, t2) for each intermediate result in an expression.

- Maintain a counter to ensure unique temporary variable names.
- Print or store the generated TAC statements.

4. **Compilation and Execution:**
    - Use lex to generate lex.yy.c from lex.l.
    - Use yacc to generate y.tab.c from yacc.y.
    - Compile both C files together (e.g., gcc y.tab.c lex.yy.c -o tac_gen).
    - Run the executable with a source program as input to see the generated TAC.

Code:

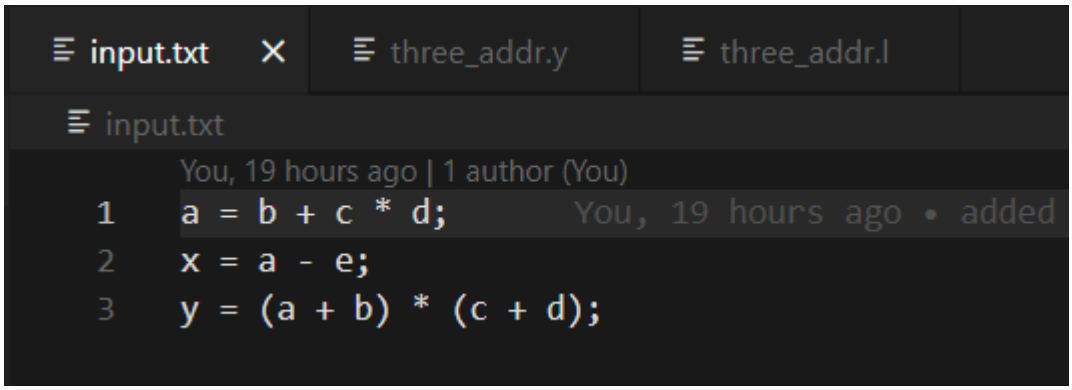
```
%{
#include "y.tab.h"
#include <stdlib.h>
#include <string.h>
void yyerror(char *);
%}

%%
[0-9]+                  { yylval.str = strdup(yytext); return NUM; }
[a-zA-Z_][a-zA-Z0-9_]* { yylval.str = strdup(yytext); return ID; }

"+"                     return '+';
"-"                     return '-';
"*"                     return '*';
"/"                     return '/';
"="                     return '=';
";"                     return ';';
"("                     return '(';
")"                     return ')';

[ \t\r]+                ;
\n                      ;

.                       {
                          fprintf(stderr, "Error: Unexpected character
'%c'\n", yytext[0]);
                        }
%%

int yywrap(void) {
    return 1;
}
```

```
%{
```

```
#include "y.tab.h"
#include <stdlib.h>
#include <string.h>
void yyerror(char *);
%}

%%
[0-9]+                  { yylval.str = strdup(yytext); return NUM; }
[a-zA-Z_][a-zA-Z0-9_]* { yylval.str = strdup(yytext); return ID; }

"+"                     return '+';
"-"                     return '-';
"*"                     return '*';
"/"                     return '/';
"="                     return '=';
";"                     return ';';
"("                     return '(';
")"                     return ')';

[ \t\r]+                ;
\n                      ;

.                       {
                          fprintf(stderr, "Error: Unexpected character
'%c'\n", yytext[0]);
                        }
%%

int yywrap(void) {
    return 1;
}
```

Output:

```
┌──(ksgsaketh㉿Saketh)-[~/compiler_lex]
└─$ yacc -d three_addr.y
lex three_addr.l
gcc y.tab.c lex.yy.c -o three_addr_generator
./three_addr_generator < input.txt

--- Three Address Code ---
t0 = c * d
t1 = b + t0
a = t1
t2 = a - e
x = t2
t3 = a + b
t4 = c + d
t5 = t3 * t4
y = t5

┌──(ksgsaketh㉿Saketh)-[~/compiler_lex]
└─$
```



Experiment 16

Implementation of Simple Code Optimization Techniques

Aim:
To implement simple source code and intermediate code optimization techniques, specifically Constant Folding, Strength Reduction, and Algebraic Transformation, to improve the efficiency of a program.

Algorithm:
The algorithm involves scanning the Three-Address Code (TAC) and applying transformations.

1. **Input:** A sequence of Three-Address Code statements.
2. **Processing:** Iterate through each TAC statement and apply the following checks and transformations:
    - **A. Constant Folding:**
        - **Check:** If both operands of an operator (e.g., +, -, *, /) are constants (literal numbers).
        - **Action:** Compute the value at compile time and replace the operation with the result.
            - *Example:* t1 = 5 + 3 * 2 can be folded into t1 = 11.
    - **B. Strength Reduction:**
        - **Check:** Identify operations that can be replaced by a less computationally expensive (weaker) operation.
        - **Action:**
            - Replace multiplication by a power of 2 with a left shift (e.g., x * 8 -> x << 3).
            - Replace division by a power of 2 with a right shift (e.g., x / 4 -> x >> 2).
            - Replace expensive operations like x * 2 with cheaper ones like x + x.
    - **C. Algebraic Transformation:**
        - **Check:** Identify expressions where algebraic identities can be applied to simplify the code.
        - **Action:**

- Apply simplification rules: x + 0 = x, x * 1 = x, x * 0 = 0.
- Apply reduction rules: x - x = 0.

3. **Iteration:** A single pass might not catch all optimizations. Perform multiple passes over the TAC until no more changes occur.
4. **Output:** Print the optimized sequence of Three-Address Code statements.

Code:

```
/* three_addr.l: (Final Version) Lexer for 3-Address Code Generation */
%{
#include "y.tab.h"
#include <stdlib.h>
#include <string.h>
void yyerror(char *);
%}

%%
[0-9]+                  { yylval.str = strdup(yytext); return NUM; }
[a-zA-Z_][a-zA-Z0-9_]* { yylval.str = strdup(yytext); return ID; }

"+"                     return '+';
"-"                     return '-';
"*"                     return '*';
"/"                     return '/';
"="                     return '=';
";"                     return ';';
"("                     return '(';
")"                     return ')';

[ \t\r]+                ;
\n                      ;

.                       {
                          fprintf(stderr, "Error: Unexpected character
'%c'\n", yytext[0]);
                        }
%%

int yywrap(void) {
    return 1;
}
```

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

```
#include <ctype.h>

char* make_str(const char *s) {
    char *buf = malloc(100);
    if (buf) strcpy(buf, s);
    return buf;
}

int is_number(const char *s) {
    for (int i = 0; s[i]; i++) {
        if (!isdigit(s[i])) return 0;
    }
    return 1;
}

int yylex(void);
int yyerror(char *s);
%}

%union {
    char* str;
}

%token <str> NUMBER ID
%type <str> expr
%left '+' '-'
%left '*' '/'
%right UMINUS

%%

program:
        program stmt
      |
      ;

stmt:   ID '=' expr '\n'    { printf("Optimized Assignment: %s = %s\n", $1,
$3); free($1); free($3); }
      | ID '=' expr ';'     { printf("Optimized Assignment: %s = %s\n", $1,
$3); free($1); free($3); }
      | expr '\n'          { printf("Optimized Expression: %s\n", $1);
free($1); }
      | expr ';'           { printf("Optimized Expression: %s\n", $1);
free($1); }
      ;

expr:   expr '+' expr     {
                            if (is_number($1) && is_number($3)) {
```

```
                                int val = atoi($1) + atoi($3);
                                char buf[50]; sprintf(buf,"%d",val);
                                $$ = make_str(buf);
                                free($1); free($3);
                            } else if (strcmp($1, "0") == 0) {
                                $$ = $3; /* Reuse $3 */
                                free($1); /* Free only $1 */
                            } else if (strcmp($3, "0") == 0) {
                                $$ = $1; /* Reuse $1 */
                                free($3); /* Free only $3 */
                            } else {
                                char buf[100]; sprintf(buf,"%s+%s",$1,$3);
                                $$ = make_str(buf);
                                free($1); free($3);
                            }
                        }
      | expr '-' expr     {
                            if (is_number($1) && is_number($3)) {
                                int val = atoi($1) - atoi($3);
                                char buf[50]; sprintf(buf,"%d",val);
                                $$ = make_str(buf);
                                free($1); free($3);
                            } else if (strcmp($3, "0") == 0) {
                                $$ = $1; /* Reuse $1 */
                                free($3); /* Free only $3 */
                            } else {
                                char buf[100]; sprintf(buf,"%s-%s",$1,$3);
                                $$ = make_str(buf);
                                free($1); free($3);
                            }
                        }
      | expr '*' expr     {
                            if (is_number($1) && is_number($3)) {
                                int val = atoi($1) * atoi($3);
                                char buf[50]; sprintf(buf,"%d",val);
                                $$ = make_str(buf);
                                free($1); free($3);
                            } else if (strcmp($1, "1") == 0) {
                                $$ = $3; /* Reuse $3 */
                                free($1); /* Free only $1 */
                            } else if (strcmp($3, "1") == 0) {
                                $$ = $1; /* Reuse $1 */
                                free($3); /* Free only $3 */
                            } else if (strcmp($1, "0") == 0 || strcmp($3, "0")
== 0) {
                                $$ = make_str("0");
                                free($1); free($3);
                            } else if (strcmp($1, "2") == 0) {
```

```
                                char buf[100]; sprintf(buf,"%s<<1",$3);
                                $$ = make_str(buf);
                                free($1); free($3);
                            } else if (strcmp($3, "2") == 0) {
                                char buf[100]; sprintf(buf,"%s<<1",$1);
                                $$ = make_str(buf);
                                free($1); free($3);
                            } else {
                                char buf[100]; sprintf(buf,"%s*%s",$1,$3);
                                $$ = make_str(buf);
                                free($1); free($3);
                            }
                        }
      | expr '/' expr     {
                            if (is_number($1) && is_number($3) && atoi($3) !=
0) {
                                int val = atoi($1) / atoi($3);
                                char buf[50]; sprintf(buf,"%d",val);
                                $$ = make_str(buf);
                                free($1); free($3);
                            } else if (strcmp($3, "1") == 0) {
                                $$ = $1; /* Reuse $1 */
                                free($3); /* Free only $3 */
                            } else if (strcmp($3, "2") == 0) {
                                char buf[100]; sprintf(buf,"%s>>1",$1);
                                $$ = make_str(buf);
                                free($1); free($3);
                            } else {
                                char buf[100]; sprintf(buf,"%s/%s",$1,$3);
                                $$ = make_str(buf);
                                free($1); free($3);
                            }
                        }
      | NUMBER            { $$ = $1; }
      | ID                { $$ = $1; }
      | '-' expr %prec UMINUS {
                            char buf[100]; sprintf(buf,"-%s",$2);
                            $$ = make_str(buf);
                            free($2);
                        }
      | '(' expr ')'      { $$ = $2; }
      ;
%%

int yyerror(char *s) {
    fprintf(stderr, "Error: %s\n", s);
    return 0;
}
```

```c
int main() {
    printf("Enter expressions (Ctrl+D to exit):\n");
    yyparse();
    return 0;
}
```

Output:

```
┌──(ksgsaketh㉿Saketh)-[~/compiler_lex]
└─$ yacc -d optimizer.y
lex optimizer.l
gcc y.tab.c lex.yy.c -o optimizer
./optimizer
Enter expressions (Ctrl+D to exit):
2 * 3 + b * 1
Optimized Expression: 6+b
a + 0
Optimized Expression: a
(b * 0) + (c + 0)
Optimized Expression: c
x+0
Optimized Expression: x
a*2
Optimized Expression: a<<1
a/2
Optimized Expression: a>>1

┌──(ksgsaketh㉿Saketh)-[~/compiler_lex]
└─$
```

Experiment 17

Implementing a Back-End for 8086 Assembly

Aim:
To design and implement the back-end of a compiler that translates a given sequence of Three-Address Code (TAC) into equivalent 8086 assembly language instructions.

Algorithm:

1. **Input:** A sequence of Three-Address Code statements.
2. **Initialization:**
    - Define the 8086 instruction set to be used (e.g., MOV, ADD, SUB, MUL, IMUL, DIV, IDIV).
    - Define a mapping from variables (including temporaries) in TAC to storage locations in 8086 (e.g., CPU registers like AX, BX, CX, DX, or memory locations addressed by [BP - offset]).
3. **Code Generation for Each TAC Statement:**
    - **A. Assignment Statement (**x = y**):**
        - MOV AX, [address_of_y] ; Load y into a register (e.g., AX)
        - MOV [address_of_x], AX ; Store the value from AX into x
    - **B. Binary Operations (**x = y op z**):**
        - **Load** y into a register (e.g., MOV AX, [address_of_y]).
        - **Perform Operation** op with z.
            - For +: ADD AX, [address_of_z]
            - For -: SUB AX, [address_of_z]
            - For * (assuming 16-bit): MOV BX, [address_of_z] followed by MUL BX (result in DX:AX).
            - For /: More complex, requires setup for DIV.
        - **Store the result** into x (e.g., MOV [address_of_x], AX).
    - **C. Handling Constants:**
        - Use immediate addressing mode. E.g., for x = 5, generate MOV [address_of_x], 5.
    - **D. Conditional Jumps (**if x relop y goto L**):**

- MOV AX, [address_of_x]
- CMP AX, [address_of_y] ; Compare x and y
- J<relop> L ; Jump to label L based on the relation (e.g., JE for ==, JNE for !=, JL for <, JGE for >=).

4. **Register Allocation (Simple Scheme):**
    - Use a simple strategy where all variables are stored in memory, and a few registers (AX, BX, etc.) are used as temporary scratchpads for computations. A more advanced algorithm would try to keep frequently used variables in registers.
5. **Function Prologue/Epilogue (if applicable):**
    - For a procedure/function, generate code to set up the stack frame at the start (PUSH BP, MOV BP, SP, SUB SP, <space_for_locals>) and restore it before returning (MOV SP, BP, POP BP, RET).
6. **Output:** Print the generated 8086 assembly code, including necessary assembler directives (e.g., .MODEL SMALL, .STACK, .DATA, .CODE).

Code:

```
/* codegen.y: The yacc parser file */
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void yyerror(char const *s);
int yylex(void);
%}

%union {
    char *var;
}
%token <var> ID
%token ASSIGN

%%
program:
       | program statement
       ;

statement: ID ASSIGN ID '+' ID '\n' {
             printf("\tMOV AX, %s\n", $3
             printf("\tADD AX, %s\n", $5);
             printf("\tMOV %s, AX\n\n", $1);
```

```
            free($1); free($3); free($5);
          }
        | ID ASSIGN ID '-' ID '\n' {
            printf("\tMOV AX, %s\n", $3);
            printf("\tSUB AX, %s\n", $5);
            printf("\tMOV %s, AX\n\n", $1);
            free($1); free($3); free($5);
          }
        | ID ASSIGN ID '*' ID '\n' {
            printf("\tMOV AX, %s\n", $3);
            printf("\tMOV BX, %s\n", $5);
            printf("\tMUL BX\n");
            printf("\tMOV %s, AX\n\n", $1);
            free($1); free($3); free($5);
          }
        | ID ASSIGN ID '/' ID '\n' {
            printf("\tMOV AX, %s\n", $3);
            printf("\tMOV BX, %s\n", $5);
            printf("\tDIV BX\n");
            printf("\tMOV %s, AX\n\n", $1);
            free($1); free($3); free($5);
          }
        | '\n' { /* Ignore blank lines */ }
        | error '\n' { yyerrok; }

%%

int main() {
    printf("Enter Three Address Code (e.g., res := op1 + op2).\n");
    printf("Press Ctrl+D (Linux/macOS) or Ctrl+Z (Windows) when done.\n\n");
    printf("--- Generated 8086 Assembly ---\n");
    yyparse();
    printf("--- End of Assembly ---\n");
    return 0;
}

void yyerror(char const *s) {
    fprintf(stderr, "Syntax Error: '%s'. Invalid statement format.\n", s);
}
```

```
/* codegen.l: The lexer file */
%{
#include "y.tab.h"
#include <string.h>
extern YYSTYPE yylval;
%}

%%
```

```
[a-zA-Z][a-zA-Z0-9]* {
    yylval.var = strdup(yytext);
    return ID;
}
":=" { return ASSIGN; }
"+"  { return '+'; }
"-"  { return '-'; }
"*"  { return '*'; }
"/"  { return '/'; }
"\n" { return '\n'; }
[ \t]+ ;
.    { fprintf(stderr, "Error: Unknown character '%s'\n", yytext); }
%%

int yywrap(void) {
    return 1;
}
```

Output:

```
┌──(ksgsaketh㉿Saketh)-[~/compiler_lex]
└─$ yacc -d codegen.y
lex codegen.l
gcc y.tab.c lex.yy.c -o codegen
./codegen
Enter Three Address Code (e.g., res := op1 + op2).
Press Ctrl+D (Linux/macOS) or Ctrl+Z (Windows) when done.

--- Generated 8086 Assembly ---
T1 := A + B
        MOV AX, A
        ADD AX, B
        MOV T1, AX

T2 := C * D
        MOV AX, C
        MOV BX, D
        MUL BX
        MOV T2, AX

Result := T1 - T2
        MOV AX, T1
        SUB AX, T2
        MOV Result, AX

--- End of Assembly ---

┌──(ksgsaketh㉿Saketh)-[~/compiler_lex]
└─$
```