LAB Assignment 3 PART 2

Experiment-9(a) Construct Simple LR (SLR) parse table using C language.

Experiment-9(b) Implement the LR parsing algorithm, get both parse table and input string are inputs. Use C language for implementation.

AIM

The objective of this experiment is to:

1. Construct a **Simple LR (SLR) Parse Table** for a given context-free grammar (CFG) using C language.
2. Implement the **LR parsing algorithm** in C such that both the parse table and an input string are taken as inputs.
3. Simulate the parsing process step-by-step, detecting whether the given input string is **accepted** or **rejected** according to the grammar.

ALGORITHM

Algorithm

Step 1: Grammar Input

- Read the grammar productions from the user.
- Augment the grammar (add a new start symbol S' → S).

Step 2: Construct Canonical Collection of LR(0) Items

1. Start with the augmented grammar's initial item (S' → •S).
2. Compute the **closure** of the item set.
3. Apply the **goto() function** for each grammar symbol (terminals and non-terminals).
4. Repeat until all states (item sets) are constructed.

Step 3: Construct the SLR Parse Table

1. For each state:
    - If an item has A → α • a β, where a is a terminal, then add a **shift** action to the table.

- If an item has A ⟶ α •, then add a **reduce** action by production A ⟶ α for all symbols in FOLLOW(A).
- If an item is S' ⟶ S•, then add an **accept** action.

2. Construct the **GOTO table** for non-terminals.
3. Detect **conflicts** (shift/reduce or reduce/reduce).

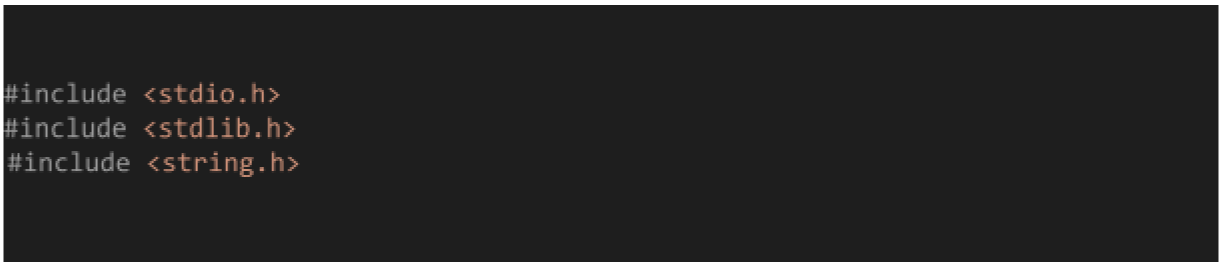
Step 4: LR Parsing Algorithm (Simulation)

1. Initialize **stack = [0]** (start state).
2. Read the **input string** from left to right with $ as end marker.
3. Repeat until ACCEPT or ERROR:
   - Let s = top of stack, a = current input symbol.
   - Look up ACTION[s, a] in the parse table.
     - If ACTION[s, a] = shift t: push a and t on the stack; move input pointer forward.
     - If ACTION[s, a] = reduce A ⟶ β: pop 2 × |β| symbols from stack, let s' = new top, push A, and then push GOTO[s', A].
     - If ACTION[s, a] = accept: report success and terminate.
     - Otherwise: report error and terminate.

Step 5: Output

- Display the constructed **SLR parse table** (ACTION and GOTO tables).
- Show the **step-by-step parsing process** (stack contents, remaining input, applied action).
- Display whether the input string is **ACCEPTED** or **REJECTED**.

SOURCE CODE



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

```c
#include <ctype.h>

#define MAXP 200

#define MAXSYM 200

#define MAXRHS 50

#define MAXITEMS 2000

#define MAXSTATES 1000

#define MAXTOKLEN 64

#define MAXFIRST 500

typedef struct {

    char lhs[64];

    char rhs[MAXRHS][64];

    int rhslen;

} Production;

Production prod[MAXP];

int P = 0;

char symbols[MAXSYM][64];

int sym_is_terminal[MAXSYM];

int sym_count = 0;

int idx_of_sym(const char *s) {

    for (int i = 0; i < sym_count; i++) if (!strcmp(symbols[i], s)) return i;

    return -1;

}

int add_symbol(const char *s) {

    int id = idx_of_sym(s);
```

LAB Assignment 3 PART 2

```
    if (id >= 0) return id;

    strcpy(symbols[sym_count], s);

    sym_is_terminal[sym_count] = 1; // default terminal, will mark
nonterminals later

    return sym_count++;

}

/* FIRST sets: firstsets[sym_index] is array of strings */

char firstsets[MAXSYM][MAXFIRST][64];

int firstcount[MAXSYM];

int nullable[MAXSYM];

/* FOLLOW sets */

char followsets[MAXSYM][MAXFIRST][64];

int followcount[MAXSYM];

/* Items and itemsets for LR(0) */

typedef struct {

    int prod_no;

    int dot;    // position 0..rhslen

} Item;

typedef struct {

    Item items[MAXITEMS];

    int n;

} ItemSet;

ItemSet states[MAXSTATES];

int state_count = 0;
```

~Aditya
23BCE1344

LAB Assignment 3 PART 2

```
/* Utility trim */
void trim(char *s) {
    int i=0; while (s[i] && isspace((unsigned char)s[i])) i++;
    int j = strlen(s)-1; while (j>=0 && isspace((unsigned char)s[j])) j--;
    int k=0;
    for (; i<=j; i++) s[k++] = s[i];
    s[k]=0;
}
/* Compare items ignoring lookaheads (LR0) */
int item_eq_lr0(const Item *a, const Item *b) {
    return a->prod_no == b->prod_no && a->dot == b->dot;
}
int itemset_contains_lr0(ItemSet *S, Item *it) {
    for (int i=0;i<S->n;i++) if (item_eq_lr0(&S->items[i], it)) return 1;
    return 0;
}
void itemset_add_lr0(ItemSet *S, Item it) {
    if (!itemset_contains_lr0(S, &it)) S->items[S->n++] = it;
}
/* FIRST computation */
void compute_FIRST() {
    for (int i=0;i<sym_count;i++) {
        firstcount[i]=0;
        nullable[i]=0;
```

~Aditya
23BCE1344

LAB Assignment 3 PART 2

```
    }
    /* terminals: FIRST(term) = {term} */
    for (int i=0;i<sym_count;i++) if (sym_is_terminal[i]) {
        strcpy(firstsets[i][firstcount[i]++], symbols[i]);
    }
    int changed = 1;
    while (changed) {
        changed = 0;
        for (int p=0;p<P;p++) {
            int A = idx_of_sym(prod[p].lhs);
            if (prod[p].rhslen == 1 && !strcmp(prod[p].rhs[0], "epsilon")) {
                if (!nullable[A]) { nullable[A]=1; changed=1; }
                continue;
            }
            int allnull = 1;
            for (int k=0;k<prod[p].rhslen;k++) {
                char *Y = prod[p].rhs[k];
                int yi = idx_of_sym(Y);
                if (yi == -1) {
                    /* Y is terminal */
                    int found = 0;
                    for (int f=0; f<firstcount[A]; f++) if
(!strcmp(firstsets[A][f], Y)) { found=1; break; }
                    if (!found) { strcpy(firstsets[A][firstcount[A]++], Y);
changed = 1; }
```

~Aditya
23BCE1344

LAB Assignment 3 PART 2

```
                    allnull = 0;

                    break;

                } else {

                    /* add FIRST(Y) \ {epsilon} to FIRST(A) */

                    for (int f=0; f<firstcount[yi]; f++) {

                        if (!strcmp(firstsets[yi][f], "epsilon")) continue;

                        int found = 0;

                        for (int g=0; g<firstcount[A]; g++) if
(!strcmp(firstsets[A][g], firstsets[yi][f])) { found=1; break; }

                        if (!found) { strcpy(firstsets[A][firstcount[A]++],
firstsets[yi][f]); changed=1; }

                    }

                    if (!nullable[yi]) { allnull = 0; break; }

                }

            }

            if (allnull && !nullable[A]) { nullable[A]=1; changed=1; }

        }

    }

}

/* FOLLOW computation */

void add_to_follow(int A, const char *sym) {

    for (int i=0;i<followcount[A];i++) if (!strcmp(followsets[A][i], sym))
return;

    strcpy(followsets[A][followcount[A]++], sym);

}

void compute_FOLLOW() {
```

~Aditya
23BCE1344

LAB Assignment 3 PART 2

```c
for (int i=0;i<sym_count;i++) followcount[i]=0;

/* start symbol is prod[0].lhs after augmentation */

int Sidx = idx_of_sym(prod[0].lhs);

add_to_follow(Sidx, "$");

int changed = 1;

while (changed) {

    changed = 0;

    for (int p=0;p<P;p++) {

        int A = idx_of_sym(prod[p].lhs);

        for (int i=0;i<prod[p].rhslen;i++) {

            char *Bsym = prod[p].rhs[i];

            int B = idx_of_sym(Bsym);

            if (B == -1) continue; /* terminal */

            int allnullable = 1;

            for (int j=i+1;j<prod[p].rhslen;j++) {

                char *Ysym = prod[p].rhs[j];

                int Y = idx_of_sym(Ysym);

                if (Y == -1) {

                    int before = followcount[B];

                    add_to_follow(B, Ysym);

                    if (followcount[B] != before) changed = 1;

                    allnullable = 0;

                    break;

                } else {
```

~Aditya
23BCE1344

```
                        for (int f=0; f<firstcount[Y]; f++) {
                            if (!strcmp(firstsets[Y][f], "epsilon")) continue;
                            int before = followcount[B];
                            add_to_follow(B, firstsets[Y][f]);
                            if (followcount[B] != before) changed = 1;
                        }
                        if (!nullable[Y]) { allnullable = 0; break; }
                    }
                }
                if (allnullable) {
                    for (int f=0; f<followcount[A]; f++) {
                        int before = followcount[B];
                        add_to_follow(B, followsets[A][f]);
                        if (followcount[B] != before) changed = 1;
                    }
                }
            }
        }
    }
}
/* closure LR(0) */
void closure_lr0(ItemSet *I) {
    int added = 1;
    while (added) {
        added = 0;
        for (int i=0;i<I->n;i++) {
            Item it = I->items[i];
            Production *p = &prod[it.prod_no];
            if (it.dot >= p->rhslen) continue;
            char *B = p->rhs[it.dot];
            int Bi = idx_of_sym(B);
            if (Bi == -1) continue; /* terminal after dot */
            for (int q=0;q<P;q++) {
                if (!strcmp(prod[q].lhs, B)) {
                    Item nit; nit.prod_no = q; nit.dot = 0;
                    if (!itemset_contains_lr0(I, &nit)) {
                        I->items[I->n++] = nit;
                        added = 1;
                    }
                }
            }
        }
    }
}

/* goto LR(0) on symbol X (index) */
ItemSet goto_lr0(ItemSet *I, int X) {
    ItemSet J; J.n = 0;
    for (int i=0;i<I->n;i++) {
        Item it = I->items[i];
        Production *p = &prod[it.prod_no];
        if (it.dot < p->rhslen) {
            char *sym = p->rhs[it.dot];
            int sidx = idx_of_sym(sym);
            if (sidx == X) {
                Item nit = it; nit.dot++;
                itemset_add_lr0(&J, nit);
            } else if (sidx == -1) {
                /* terminal, compare by name */
                if (!strcmp(sym, symbols[X])) {
                    Item nit = it; nit.dot++;
                    itemset_add_lr0(&J, nit);
                }
            }
        }
    }
    closure_lr0(&J);
    return J;
}
/* state equality by LR(0) core */
int same_lr0_core(ItemSet *A, ItemSet *B) {
    if (A->n != B->n) return 0;
```

LAB Assignment 3 PART 2

```
    for (int i=0;i<A->n;i++) {
        int found = 0;
        for (int j=0;j<B->n;j++) if (A->items[i].prod_no == B-
>items[j].prod_no && A->items[i].dot == B->items[j].dot) { found = 1; break; }
        if (!found) return 0;
    }
    return 1;
}
int find_state(ItemSet *I) {
    for (int s=0;s<state_count;s++) {
        if (same_lr0_core(&states[s], I)) return s;
    }
    return -1;
}
/* Build LR(0) canonical collection */
void build_canonical_lr0() {
    state_count = 0;
    ItemSet I0; I0.n = 0;
    Item it0; it0.prod_no = 0; it0.dot = 0;
    itemset_add_lr0(&I0, it0);
    closure_lr0(&I0);
    states[state_count++] = I0;
    for (int i=0;i<state_count;i++) {
        for (int X=0; X<sym_count; X++) {
            ItemSet J = goto_lr0(&states[i], X);
```

~Aditya
23BCE1344

LAB Assignment 3 PART 2

```c
            if (J.n == 0) continue;
            int s = find_state(&J);
            if (s == -1) {
                states[state_count++] = J;
            }
        }
    }
}
/* Parse table */
char ACTION[MAXSTATES][MAXSYM][32];
int GOTO[MAXSTATES][MAXSYM];
void init_table() {
    for (int i=0;i<MAXSTATES;i++) {
        for (int j=0;j<MAXSYM;j++) {
            ACTION[i][j][0] = 0;
            GOTO[i][j] = -1;
        }
    }
}
/* Build SLR table */
int build_SLR_table() {
    init_table();
    for (int i=0;i<state_count;i++) {
        ItemSet *I = &states[i];
```

LAB Assignment 3 PART 2

```
        for (int k=0;k<I->n;k++) {
            Item it = I->items[k];
            Production *p = &prod[it.prod_no];
            if (it.dot < p->rhslen) {
                char *a = p->rhs[it.dot];
                int aidx = idx_of_sym(a);
                /* compute goto on symbol a (using index mapping) */
                int symidx = aidx;
                if (aidx == -1) {
                    /* a is a terminal that might not be in symbols? It should
be, but find by string */
                    for (int s=0;s<sym_count;s++) if (!strcmp(symbols[s], a))
{ symidx = s; break; }
                    if (symidx == -1) symidx = add_symbol(a),
sym_is_terminal[symidx]=1;
                }
                ItemSet J = goto_lr0(I, symidx);
                if (J.n == 0) continue;
                int j = find_state(&J);
                if (j == -1) continue;
                if (sym_is_terminal[symidx]) {
                    char buf[32]; sprintf(buf, "s%d", j);
                    if (ACTION[i][symidx][0] && strcmp(ACTION[i][symidx],
buf)) {
                        printf("Shift/Other conflict at state %d symbol %s\n",
i, symbols[symidx]);
```

~Aditya
23BCE1344

```
                        return 0;
                    }
                    strcpy(ACTION[i][symidx], buf);
                } else {
                    /* goto entry */
                    GOTO[i][symidx] = j;
                }
            } else {
                /* dot at end: reduction */
                if (it.prod_no == 0) {
                    int dollar = idx_of_sym("$");
                    strcpy(ACTION[i][dollar], "acc");
                } else {
                    int A = idx_of_sym(p->lhs);
                    for (int f=0; f<followcount[A]; f++) {
                        int aidx = idx_of_sym(followsets[A][f]);
                        if (aidx == -1) aidx = add_symbol(followsets[A][f]),
sym_is_terminal[aidx]=1;
                        char buf[32]; sprintf(buf, "r%d", it.prod_no);
                        if (ACTION[i][aidx][0] && strcmp(ACTION[i][aidx],
buf)) {
                            printf("SLR Conflict at state %d, terminal %s:
existing=%s new=%s\n", i, symbols[aidx], ACTION[i][aidx], buf);
                            return 0;
                        }
                        strcpy(ACTION[i][aidx], buf);
                    }
                }
            }
        }
        /* fill goto for nonterminals where goto(i,A)=j */
        for (int A=0; A<sym_count; A++) if (!sym_is_terminal[A]) {
            ItemSet J = goto_lr0(&states[i], A);
            if (J.n == 0) continue;
            int j = find_state(&J);
            if (j>=0) GOTO[i][A] = j;
        }
    }
    return 1;
}
/* Print table */
/* Print formatted SLR Parsing Table with ACTION and GOTO columns */
void print_table() {
    printf("\n=== SLR Parsing Table ===\n");
    // Count terminals and nonterminals
    int term_count = 0, nonterm_count = 0;
    for (int i = 0; i < sym_count; i++) {
        if (sym_is_terminal[i]) term_count++;
        else nonterm_count++;
    }
```

LAB Assignment 3 PART 2

```
    // Print header
    printf("%-8s", "State");
    for (int i = 0; i < sym_count; i++) {
        if (sym_is_terminal[i]) printf("%-10s", symbols[i]); // ACTION
columns
    }
    for (int i = 0; i < sym_count; i++) {
        if (!sym_is_terminal[i]) printf("%-10s", symbols[i]); // GOTO columns
    }
    printf("\n");
    // Print separator
    int total_cols = term_count + nonterm_count;
    for (int i = 0; i < 8 + total_cols * 10; i++) printf("-");
    printf("\n");
    // Print rows for each state
    for (int s = 0; s < state_count; s++) {
        printf("%-8d", s);
        for (int i = 0; i < sym_count; i++) {
            if (sym_is_terminal[i]) {
                if (ACTION[s][i][0]) printf("%-10s", ACTION[s][i]);
                else printf("%-10s", "");
            }
        }
        for (int i = 0; i < sym_count; i++) {
```

~Aditya
23BCE1344

```
            if (!sym_is_terminal[i]) {
                if (GOTO[s][i] != -1) printf("%-10d", GOTO[s][i]);
                else printf("%-10s", "");
            }
        }
        printf("\n");
    }


}


/* Parser: uses ACTION and GOTO */
void run_parser() {
    printf("\nEnter input tokens separated by spaces (like: id + id * id ).
End with $ or omit and program will append $:\n");
    char line[1024];
    /* read full line(s) */
    if (!fgets(line, sizeof(line), stdin)) return;
    if (strlen(line)==0) if (!fgets(line, sizeof(line), stdin)) return;
    trim(line);
    if (strlen(line) == 0) return;
    char tokens[200][64]; int tcount=0;
    char *tok = strtok(line, " \t\r\n");
    while (tok) {
        strcpy(tokens[tcount++], tok);
        tok = strtok(NULL, " \t\r\n");
```

LAB Assignment 3 PART 2

```
    }

    if (tcount == 0) return;

    if (strcmp(tokens[tcount-1], "$") != 0) strcpy(tokens[tcount++], "$");

    int states_stack[1024]; int sttop = 0; states_stack[0] = 0;

    char sym_stack[1024][64]; int symtop = 0;

    int ip = 0;

    printf("\n%-30s %-30s %s\n", "States(stack)", "Symbols(stack)", "Action");

    while (1) {

        /* print stacks */

        char sstates[512] = ""; char ssym[512] = "";

        for (int i=0;i<=sttop;i++) { char b[16]; sprintf(b, "%d ",
states_stack[i]); strcat(sstates, b); }

        for (int i=0;i<symtop;i++) { strcat(ssym, sym_stack[i]); strcat(ssym,
" "); }

        printf("%-30s %-30s ", sstates, ssym);

        int state = states_stack[sttop];

        char *a = tokens[ip];

        int aidx = idx_of_sym(a);

        if (aidx == -1) { aidx = add_symbol(a); sym_is_terminal[aidx] = 1; }
/* new terminal if not present */

        if (ACTION[state][aidx][0] == 0) {

            printf("ERROR: no action for state %d and token %s\n", state, a);

            return;

        }

        char act[32]; strcpy(act, ACTION[state][aidx]);
```

~Aditya
23BCE1344

```c
        if (!strcmp(act, "acc")) {
            printf("ACCEPT\n");
            return;
        } else if (act[0] == 's') {
            int s = atoi(act+1);
            printf("shift %d\n", s);
            /* push token onto symbol stack and state s */
            strcpy(sym_stack[symtop++], a);
            states_stack[++sttop] = s;
            ip++;
        } else if (act[0] == 'r') {
            int pno = atoi(act+1);
            Production *pr = &prod[pno];
            printf("reduce by %s ->", pr->lhs);
            for (int k=0;k<pr->rhslen;k++) printf(" %s", pr->rhs[k]);
            printf("\n");
            /* pop |rhs| symbols and states (if rhs is epsilon, pop none) */
            if (!(pr->rhslen == 1 && !strcmp(pr->rhs[0], "epsilon"))) {
                for (int k=0;k<pr->rhslen;k++) {
                    if (symtop>0) symtop--;
                    if (sttop>0) sttop--;
                }
            }
            /* push LHS */
```

LAB Assignment 3 PART 2

```
            strcpy(sym_stack[symtop++], pr->lhs);

            int topstate = states_stack[sttop];

            int Aidx = idx_of_sym(pr->lhs);

            int goto_s = GOTO[topstate][Aidx];

            if (goto_s == -1) { printf("ERROR: no goto from state %d on %s\n",
topstate, pr->lhs); return; }

            states_stack[++sttop] = goto_s;

        } else {

            printf("Unknown action '%s'\n", act);

            return;

        }

    }

}

void print_grammar() {

    printf("\nAugmented Grammar (numbered productions):\n");

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

        printf("%d: %s ->", i, prod[i].lhs);

        for (int j=0;j<prod[i].rhslen;j++) printf(" %s", prod[i].rhs[j]);

        printf("\n");

    }

}

int main() {

    printf("SLR(1) Parser Generator (educational)\n");

    printf("Enter number of productions:\n");

    int N;
```

~Aditya
23BCE1344

LAB Assignment 3 PART 2

```c
    if (scanf("%d\n", &N) != 1) return 0;

    char line[1024];

    P = 0; sym_count = 0;

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

        if (!fgets(line, sizeof(line), stdin)) return 0;

        if (strlen(line)==0 || line[0]=='\n') { i--; continue; }

        trim(line);

        char *arrow = strstr(line, "->");

        if (!arrow) { printf("Production format error: %s\n", line); return 0;
}

        char lhs[64]; strncpy(lhs, line, arrow-line); lhs[arrow-line]=0;
trim(lhs);

        char rhspart[512]; strcpy(rhspart, arrow+2); trim(rhspart);

        Production p; strcpy(p.lhs, lhs); p.rhslen = 0;

        char *tok = strtok(rhspart, " \t\r\n");

        while (tok) {

            strcpy(p.rhs[p.rhslen++], tok);

            tok = strtok(NULL, " \t\r\n");

        }

        if (p.rhslen == 0) { strcpy(p.rhs[p.rhslen++], "epsilon"); }

        prod[P++] = p;

        /* register symbols */

        add_symbol(lhs); sym_is_terminal[idx_of_sym(lhs)] = 0;

        for (int k=0;k<p.rhslen;k++) if (strcmp(p.rhs[k], "epsilon"))
add_symbol(p.rhs[k]);
```

~Aditya
23BCE1344

LAB Assignment 3 PART 2

```
    }

    /* ensure $ symbol exists */

    if (idx_of_sym("$") == -1) { add_symbol("$");
sym_is_terminal[idx_of_sym("$")] = 1; }

    /* mark nonterminals by LHSs (re-mark in case) */

    for (int i=0;i<sym_count;i++) sym_is_terminal[i] = 1;

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

        int id = idx_of_sym(prod[i].lhs);

        if (id >= 0) sym_is_terminal[id] = 0;

    }

    /* Augment grammar: insert S' -> S at prod[0] */

    for (int i=P;i>0;i--) prod[i] = prod[i-1];

    Production aug; strcpy(aug.lhs, "S'"); aug.rhslen = 1; strcpy(aug.rhs[0],
prod[1].lhs);

    prod[0] = aug; P++;

    add_symbol("S'"); sym_is_terminal[idx_of_sym("S'")] = 0;

    /* compute FIRST & FOLLOW */

    compute_FIRST();

    compute_FOLLOW();

    /* display grammar, FIRST, FOLLOW */

    print_grammar();

    printf("\nFIRST sets:\n");

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

        if (sym_is_terminal[i]) continue;

        printf("FIRST(%s) = { ", symbols[i]);
        for (int f=0; f<firstcount[i]; f++) printf("%s ", firstsets[i][f]);
```

~Aditya
23BCE1344

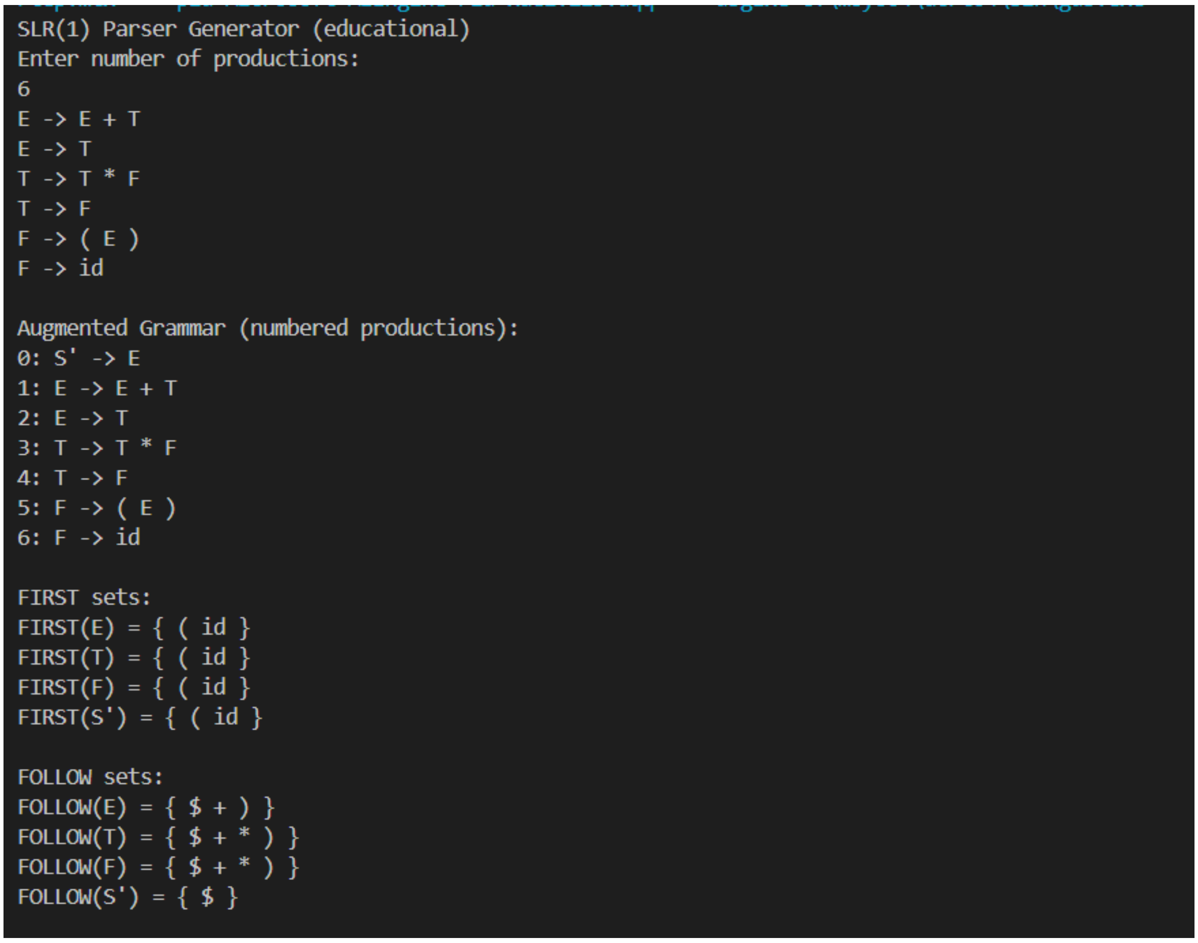
LAB Assignment 3 PART 2

```c
        printf("}\n");
    }
    printf("\nFOLLOW sets:\n");
    for (int i=0;i<sym_count;i++) {
        if (sym_is_terminal[i]) continue;
        printf("FOLLOW(%s) = { ", symbols[i]);
        for (int f=0; f<followcount[i]; f++) printf("%s ", followsets[i][f]);
        printf("}\n");
    }
    /* build canonical LR(0) */
    build_canonical_lr0();
    printf("\nBuilt %d LR(0) states.\n", state_count);
    /* build SLR table */
    if (!build_SLR_table()) { printf("Failed to build SLR table due to conflicts.\n"); return 0; }
    printf("SLR parse table built successfully.\n");
    /* print parse table */
    print_table();
    /* consume remaining newline before reading input */
    if (!fgets(line, sizeof(line), stdin)) {}
    run_parser();
    return 0;
}
```

~Aditya
23BCE1344

LAB Assignment 3 PART 2

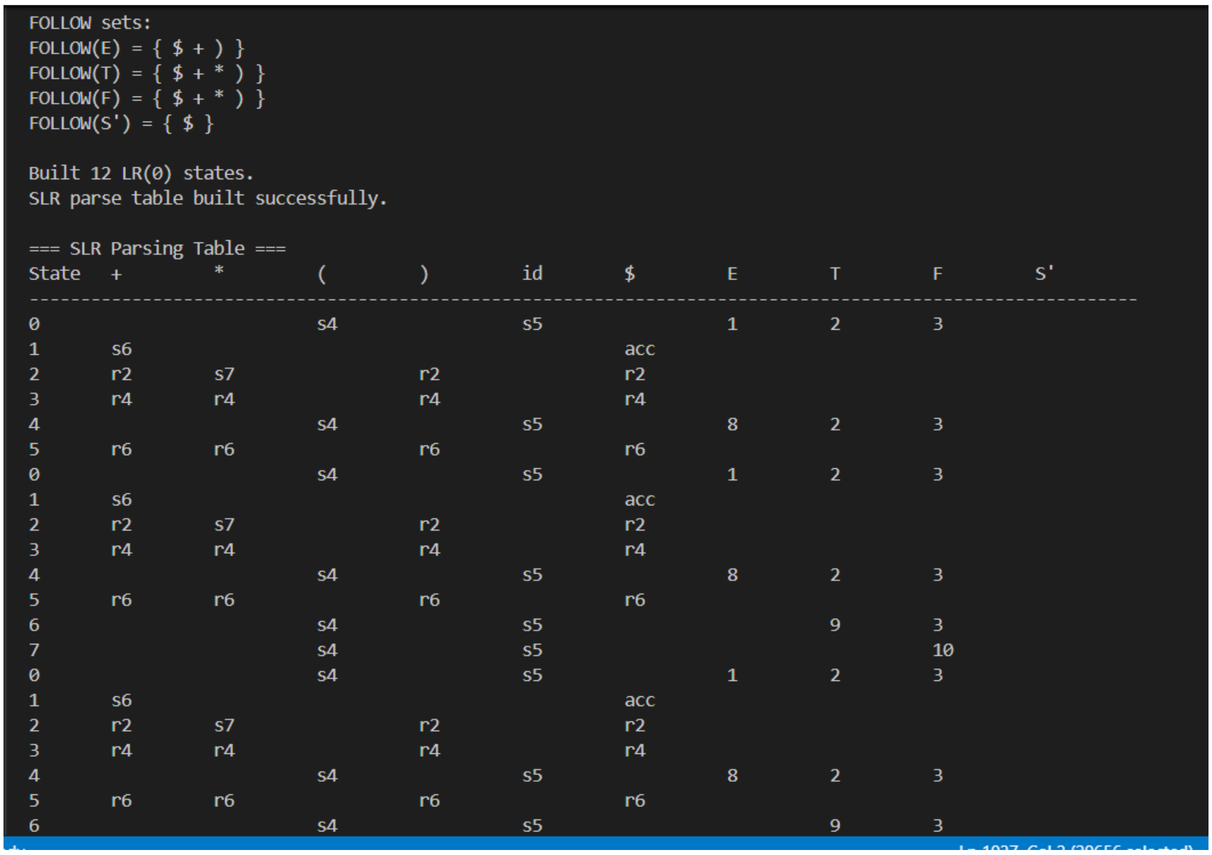
OUTPUT



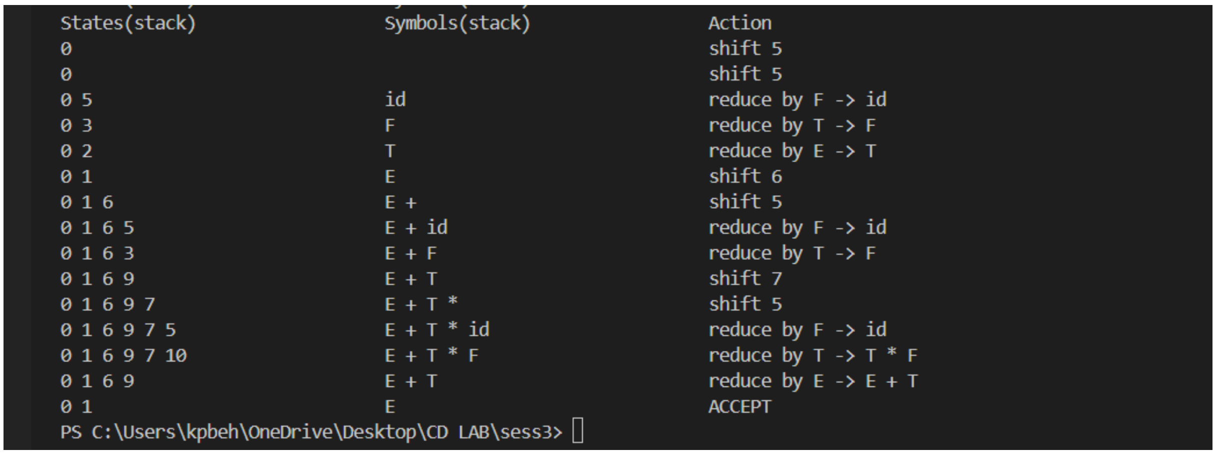
```
SLR(1) Parser Generator (educational)
Enter number of productions:
6
E -> E + T
E -> T
T -> T * F
T -> F
F -> ( E )
F -> id

Augmented Grammar (numbered productions):
0: S' -> E
1: E -> E + T
2: E -> T
3: T -> T * F
4: T -> F
5: F -> ( E )
6: F -> id

FIRST sets:
FIRST(E) = { ( id }
FIRST(T) = { ( id }
FIRST(F) = { ( id }
FIRST(S') = { ( id }

FOLLOW sets:
FOLLOW(E) = { $ + ) }
FOLLOW(T) = { $ + * ) }
FOLLOW(F) = { $ + * ) }
FOLLOW(S') = { $ }
```

~Aditya
23BCE1344

LAB Assignment 3 PART 2



```
FOLLOW sets:
FOLLOW(E) = { $ + ) }
FOLLOW(T) = { $ + * ) }
FOLLOW(F) = { $ + * ) }
FOLLOW(S') = { $ }

Built 12 LR(0) states.
SLR parse table built successfully.

=== SLR Parsing Table ===
State   +         *         (         )         id        $         E         T         F         S'
------------------------------------------------------------------------------------------------------------
0                           s4                  s5                  1         2         3
1       s6                                                acc
2       r2        s7                  r2                  r2
3       r4        r4                  r4                  r4
4                           s4                  s5                  8         2         3
5       r6        r6                  r6                  r6
0                           s4                  s5                  1         2         3
1       s6                                                acc
2       r2        s7                  r2                  r2
3       r4        r4                  r4                  r4
4                           s4                  s5                  8         2         3
5       r6        r6                  r6                  r6
6                           s4                  s5                            9         3
7                           s4                  s5                                      10
0                           s4                  s5                  1         2         3
1       s6                                                acc
2       r2        s7                  r2                  r2
3       r4        r4                  r4                  r4
4                           s4                  s5                  8         2         3
5       r6        r6                  r6                  r6
6                           s4                  s5                            9         3
```



```
States(stack)                  Symbols(stack)                  Action
0                                                              shift 5
0                                                              shift 5
0 5                            id                              reduce by F -> id
0 3                            F                               reduce by T -> F
0 2                            T                               reduce by E -> T
0 1                            E                               shift 6
0 1 6                          E +                             shift 5
0 1 6 5                        E + id                          reduce by F -> id
0 1 6 3                        E + F                           reduce by T -> F
0 1 6 9                        E + T                           shift 7
0 1 6 9 7                      E + T *                         shift 5
0 1 6 9 7 5                    E + T * id                      reduce by F -> id
0 1 6 9 7 10                   E + T * F                       reduce by T -> T * F
0 1 6 9                        E + T                           reduce by E -> E + T
0 1                            E                               ACCEPT
PS C:\Users\kpbeh\OneDrive\Desktop\CD LAB\sess3>
```

Experiment-10(a) Construct Canonical LR (CLR) parse table using C language.

Experiment-10(b) Implement the LR parsing algorithm, get both parse table and input string are inputs. Use C language for implementation.

The objective of this experiment is to:

1. Construct the **Canonical LR (CLR) Parse Table** for a given context-free grammar using C language.

LAB Assignment 3 PART 2

2. Implement the **LR parsing algorithm** in C that uses the CLR parse table along with an input string.
3. Simulate the parsing process to determine whether the given input string is **syntactically valid** according to the grammar.

Algorithm

Step 1: Grammar Input

- Read grammar productions from user.
- Augment the grammar by adding S' → S.

Step 2: Construct Canonical Collection of LR(1) Items

1. Start with the initial item: S' → •S, $.
2. Compute **closure(I)**:
    - For every item [A → α • B β, a] in I and for each production B → γ,
    - Add [B → •γ, b] for every terminal b in FIRST(βa).
    - Repeat until no more items can be added.
3. Compute **goto(I, X)** for each grammar symbol X.
4. Repeat until all item sets (states) are generated.

Step 3: Construct the CLR Parse Table

1. For each state I:
    - If [A → α • a β, b] exists, where a is terminal, set ACTION[I, a] = shift J (where J = goto(I, a)).
    - If [A → α •, a] exists (dot at end), set ACTION[I, a] = reduce A → α.
    - If [S' → S •, $] exists, set ACTION[I, $] = accept.
    - For each non-terminal A, if goto(I, A) = J, set GOTO[I, A] = J.
2. Construct **ACTION** and **GOTO** tables.

~Aditya
23BCE1344

LAB Assignment 3 PART 2

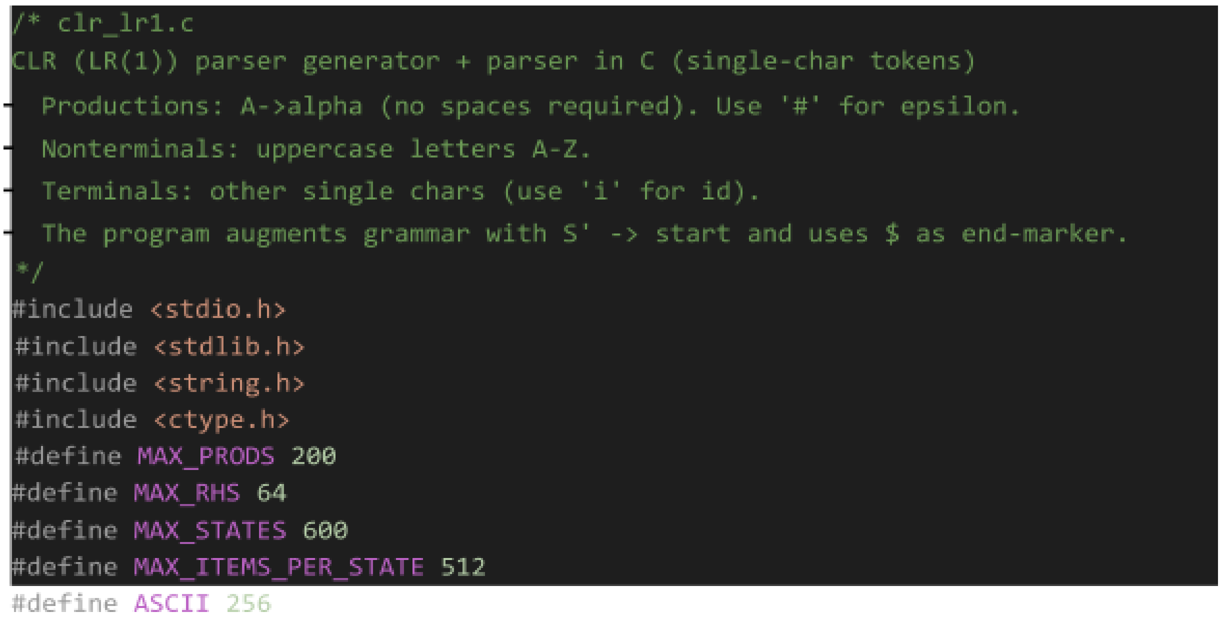
Step 4: LR Parsing Algorithm (Using CLR Table)

1. Initialize stack = [0].
2. Append $ to input string.
3. Repeat:
    - Let s = top of stack, a = current input symbol.
    - If ACTION[s, a] = shift t, push a, t and advance input pointer.
    - If ACTION[s, a] = reduce A → β, pop 2 × |β| elements, let s' = top of stack, push A and GOTO[s', A].
    - If ACTION[s, a] = accept, report success.
    - Otherwise, report error.

Step 5: Output

- Print the **Canonical LR(1) item sets**.
- Display the **CLR Parse Table (ACTION + GOTO)**.
- Show the **parsing steps** (stack, input, action at each step).
- Finally, state whether the input string is **ACCEPTED** or **REJECTED**.

SOURCE CODE



```c
/* clr_lr1.c
CLR (LR(1)) parser generator + parser in C (single-char tokens)
  Productions: A->alpha (no spaces required). Use '#' for epsilon.
  Nonterminals: uppercase letters A-Z.
  Terminals: other single chars (use 'i' for id).
  The program augments grammar with S' -> start and uses $ as end-marker.
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#define MAX_PRODS 200
#define MAX_RHS 64
#define MAX_STATES 600
#define MAX_ITEMS_PER_STATE 512
#define ASCII 256
```

~Aditya
23BCE1344

LAB Assignment 3 PART 2

```c
typedef struct {
char lhs;
char rhs[MAX_RHS];
int rhs_len;
} Prod;
/* Global grammar */
static Prod prods[MAX_PRODS];
static int prod_count = 0;
/* which characters appear in grammar */
static int used_sym[ASCII];
static int is_nonterm[ASCII];
/* FIRST sets: FIRST[X][t] == 1 if terminal t in FIRST(X) */
static unsigned char FIRST[ASCII][ASCII];
/* CLR states */
typedef struct {
int prod[MAX_ITEMS_PER_STATE];
int dot[MAX_ITEMS_PER_STATE];
unsigned char look[MAX_ITEMS_PER_STATE][ASCII]; /* lookahead bitset */
int n;
} State;
static State *CLR = NULL;
static int CLR_count = 0;
/* ACTION, GOTO tables */
static int ACTION[MAX_STATES][ASCII]; /* -999 empty; >0 shift (state+1); <= -2
reduce
-(prod+2); 100000 accept */
static int GOTO_TABLE[MAX_STATES][ASCII]; /* -999 empty, else state */
/* Utility helpers */
static void die(const char *msg) { fprintf(stderr, "%s\n", msg); exit(1); }
/* Trim spaces */
static void trim_whitespace(char *s) {
char *p = s;
while (*p && isspace((unsigned char)*p)) p++;
if (p != s) memmove(s, p, strlen(p)+1);
int L = strlen(s);
while (L>0 && isspace((unsigned char)s[L-1])) s[--L] = '\0';
}
/* Remove spaces from a copy (useful to parse A->alpha when user may write
spaces) */
static void remove_spaces(const char *in, char *out) {
int i,j = 0;
for (i=0; in[i]; ++i) {
if (!isspace((unsigned char)in[i])) out[j++] = in[i];
}
out[j] = '\0';
}
/* Read grammar lines. Accepts A->alpha with optional spaces around tokens. */
static void read_grammar() {
```

LAB Assignment 3 PART 2

```c
int n;
printf("Enter number of productions: ");
if (scanf("%d", &n) != 1) die("bad number");
getchar();
if (n <= 0) die("need at least one production");
int i;
for (i = 0; i < n; ++i) {
char line[1024];
if (!fgets(line, sizeof(line), stdin)) die("unexpected EOF");
/* allow blank lines */
if (line[0] == '\n') { i--; continue; }
line[strcspn(line, "\n")] = '\0';
trim_whitespace(line);
if (strlen(line) == 0) { i--; continue; }
char compact[1024];
remove_spaces(line, compact);
char *arrow = strstr(compact, "->");
if (!arrow || arrow == compact) {
fprintf(stderr, "Invalid production: %s\n", line);
exit(1);
}
/* LHS must be single uppercase char */
char lhs = compact[0];
if (!isupper((unsigned char)lhs)) {
fprintf(stderr, "LHS must be uppercase nonterminal (single char), got '%c' in:
%s\n", lhs,
line);
exit(1);
}
char *rhsstart = arrow + 2;
if (rhsstart[0] == '\0') {
/* epsilon */
rhsstart = "#";
}
if (prod_count >= MAX_PRODS-2) die("too many productions");
prods[prod_count].lhs = lhs;
prods[prod_count].rhs_len = 0;
/* rhsstart is compact (no spaces). Each character is a symbol. '#' stands for
epsilon. */
if (strcmp(rhsstart, "#") == 0) {
prods[prod_count].rhs_len = 0;
} else {
int k;
for (k = 0; rhsstart[k] != '\0'; ++k) {
prods[prod_count].rhs[prods[prod_count].rhs_len++] = rhsstart[k];
used_sym[(unsigned char)rhsstart[k]] = 1;
}
}
```

~Aditya
23BCE1344

LAB Assignment 3 PART 2

```c
used_sym[(unsigned char)lhs] = 1;
is_nonterm[(unsigned char)lhs] = 1;
prod_count++;
}
}
/* Build symbol info and ensure '#' and '$' present */
static void finalize_symbols() {
used_sym[(unsigned char)'#'] = 1; /* epsilon symbol */
used_sym[(unsigned char)'$'] = 1; /* end marker */
is_nonterm[(unsigned char)'#'] = 0;
is_nonterm[(unsigned char)'$'] = 0;
}
/* FIRST helpers */
/* Add a terminal t to FIRST[X]; returns 1 if changed */
static int FIRST_add(int X, int t) {
if (!FIRST[X][t]) { FIRST[X][t] = 1; return 1; }
return 0;
}
/* Compute FIRST sets (classic fixed-point). '#' used for epsilon. */
static void compute_FIRST() {
/* clear */
int i,j,ch;
for (i = 0; i < ASCII; ++i) for (j = 0; j < ASCII; ++j) FIRST[i][j] = 0;
/* terminals: FIRST[t] contains t */
for (ch = 0; ch < ASCII; ++ch) {
if (used_sym[ch] && !is_nonterm[ch]) {
FIRST[ch][ch] = 1;
}
}
/* epsilon '#' first contains '#' */
FIRST['#']['#'] = 1;
int changed = 1;
while (changed) {
changed = 0;
int p,pos,t;
for (p = 0; p < prod_count; ++p) {
int A = (unsigned char)prods[p].lhs;
/* compute FIRST(rhs) */
int all_nullable = 1;
for (pos = 0; pos < prods[p].rhs_len; ++pos)
{ int X = (unsigned char)prods[p].rhs[pos];
/* add FIRST[X] \ {#} to FIRST[A] */
for (t = 0; t < ASCII; ++t) {
if (FIRST[X][t] && t != (unsigned char)'#') {
if (FIRST_add(A, t)) changed = 1;
}
}
if (!FIRST[X][(unsigned char)'#']) { all_nullable = 0; break; }
```

~Aditya
23BCE1344

LAB Assignment 3 PART 2

```
}
if (prods[p].rhs_len == 0) all_nullable = 1;
if (all_nullable) {
if (FIRST_add(A, (unsigned char)'#')) changed = 1;
}
}
}
}
/* Compute FIRST of sequence beta followed by lookahead la (la is a char
code), result in out[]
bitset */
static void FIRST_of_sequence(char beta[], int blen, int la_char, unsigned
char out[ASCII]) {
int i,t;
for (i = 0; i < ASCII; ++i) out[i] = 0;
if (blen == 0) { out[la_char] = 1; return; }
int all_nullable = 1;
for (i = 0; i < blen; ++i) {
int X = (unsigned char)beta[i];
/* add FIRST[X] \ {#} */
for (t = 0; t < ASCII; ++t) {
if (FIRST[X][t] && t != (unsigned char)'#') out[t] = 1;
}
if (!FIRST[X][(unsigned char)'#']) { all_nullable = 0; break; }
}
if (all_nullable) out[la_char] = 1;
}
/* State item operations */
static int add_item(State *S, int prod_idx, int dotpos, unsigned char
*lookset) {
/* find core match */
int i,t;
for (i = 0; i < S->n; ++i) {
if (S->prod[i] == prod_idx && S->dot[i] == dotpos) {
/* union looksets; return 1 if changed */
int changed = 0;
for (t = 0; t < ASCII; ++t) {
if (lookset[t] && !S->look[i][t]) { S->look[i][t] = 1; changed = 1; }
}
return changed;
}
}
if (S->n >= MAX_ITEMS_PER_STATE) die("Exceeded items per state");
S->prod[S->n] = prod_idx;
S->dot[S->n] = dotpos;
memcpy(S->look[S->n], lookset, ASCII);
S->n++;
return 1;
```

~Aditya
23BCE1344

```
}
/* Closure: operate in-place on state; returns nothing */
static void closure(State *S) {
int changed = 1;
while (changed) {
changed = 0;
/* iterate items */
int i;
for (i = 0; i < S->n; ++i) {
int p = S->prod[i];
int d = S->dot[i];
if (d < prods[p].rhs_len) {
char B = prods[p].rhs[d];
if (is_nonterm[(unsigned char)B]) {
/* build beta (symbols after B) */
char beta[MAX_RHS];
int blen = 0;
int k;
for (k = d + 1; k < prods[p].rhs_len; ++k) beta[blen++] = prods[p].rhs[k];
/* for each lookahead symbol 'a' in S->look[i], compute FIRST(beta + a) and
add
items */
int la;
for (la = 0; la < ASCII; ++la) if (S->look[i][la]) {
unsigned char fseq[ASCII];
FIRST_of_sequence(beta, blen, la, fseq);
/* for every production B->gamma */
int q,b,z;
for (q = 0; q < prod_count; ++q) {
if (prods[q].lhs == B) {
/* for every terminal b in fseq (except '#') add [B->.gamma, b] */
for (b = 0; b < ASCII; ++b) {
if (fseq[b] && (unsigned char)b != (unsigned char)'#') {
unsigned char tmplook[ASCII];
for (z=0; z<ASCII; ++z) tmplook[z]=0;
tmplook[b] = 1;
if (add_item(S, q, 0, tmplook)) changed = 1;
}
}
}
}
}
}
}
}
}
}
/* goto operation: produce new state J from state I on symbol X */
```

LAB Assignment 3 PART 2

```
static State goto_on(const State *I, char X)
{ State J; J.n = 0;
int i;
for (i = 0; i < I->n; ++i) {
int p = I->prod[i];
int d = I->dot[i];
if (d < prods[p].rhs_len && prods[p].rhs[d] == X) {
/* copy lookset */
unsigned char tmp[ASCII];
memcpy(tmp, I->look[i], ASCII);
add_item(&J, p, d+1, tmp);
}
}
if (J.n == 0) return J;
closure(&J);
return J;
}
/* Compare two states for LR(1) equality (items + looksets) */
static int states_equal(const State *A, const State *B) {
if (A->n != B->n) return 0;
int i;
for (i = 0; i < A->n; ++i) {
int j,found = 0;
for (j = 0; j < B->n; ++j) {
if (A->prod[i] == B->prod[j] && A->dot[i] == B->dot[j] &&
memcmp(A->look[i], B->look[j],
ASCII) == 0) { found = 1; break; }
}
if (!found) return 0;
}
return 1;
}
/* Find state index in CLR[] that equals S; return -1 if not found */
static int find_state(const State *S) {
int i;
for (i = 0; i < CLR_count; ++i) {
if (states_equal(S, &CLR[i])) return i;
}
return -1;
}
/* Build canonical LR(1) collection */
static void build_CLR_collection() {
CLR = (State *) malloc(sizeof(State) * MAX_STATES);
if (!CLR) die("malloc CLR");
CLR_count = 0;
/* initial item [S' -> .start, $] */
State I0; I0.n = 0;
int t;
```

~Aditya
23BCE1344

LAB Assignment 3 PART 2

```c
unsigned char init_look[ASCII]; for (t=0;t<ASCII;t++) init_look[t]=0;
init_look[(unsigned char)'$'] = 1;
add_item(&I0, 0, 0, init_look);
closure(&I0);
CLR[CLR_count++] = I0;
int changed = 1;
/* list of used symbols for transitions (only symbols that appear in grammar)
*/
char syms[ASCII]; int syms_n = 0;
int c,s,si;
for (c = 0; c < ASCII; ++c) if (used_sym[c]) syms[syms_n++] = (char)c;
while (changed) {
changed = 0;
for (s = 0; s < CLR_count; ++s)
{ for (si = 0; si < syms_n;
++si) { char X = syms[si];
State J = goto_on(&CLR[s],
X); if (J.n == 0) continue;
if (find_state(&J) == -1) {
if (CLR_count >= MAX_STATES) die("too many CLR states");
CLR[CLR_count++] = J;
changed = 1;
}
}
}
}
}
/* Allocate/initialize ACTION and GOTO table */
static void init_tables() {
int i,c;
for (i = 0; i < MAX_STATES; ++i) {
for (c = 0; c < ASCII; ++c) {
ACTION[i][c] = -999;
GOTO_TABLE[i][c] = -999;
}
}
}
/* Build ACTION & GOTO from CLR states */
static void build_tables() {
init_tables();
/* compute temp goto mapping by computing goto for each CLR state and symbol,
to avoid
recomputing */
/* but simpler: for each state s and symbol X compute goto_on and find index
j. Use
find_state. */
int s;
for (s = 0; s < CLR_count; ++s) {
```

~Aditya
23BCE1344

LAB Assignment 3 PART 2

```
State *st = &CLR[s];
int i;
for (i = 0; i < st->n; ++i) {
int p = st->prod[i];
int d = st->dot[i];
if (d < prods[p].rhs_len) {
char a = prods[p].rhs[d];
if (!is_nonterm[(unsigned char)a]) {
State J = goto_on(st, a);
if (J.n == 0) continue;
int j = find_state(&J);
if (j == -1) die("goto state not found");
int tindex = (unsigned char)a;
/* shift to j: encode as j+1 */
if (ACTION[s][tindex] == -999) ACTION[s][tindex] = j + 1;
else if (ACTION[s][tindex] != j + 1) {
printf("Shift/other conflict at ACTION[%d]['%c'] existing=%d new=%d\n",
s, a, ACTION[s][tindex], j+1);
}
} else {
/* nonterminal -> GOTO entry */
State J = goto_on(st, a);
if (J.n == 0) continue;
int j = find_state(&J);
if (j == -1) die("goto state not found for nonterminal");
GOTO_TABLE[s][(unsigned char)a] = j;
}
} else {
/* dot at end -> reduce (or accept) */
if (p == 0) {
/* augmented production -> accept on $ */
int dollar = (unsigned char)'$';
if (ACTION[s][dollar] == -999) ACTION[s][dollar] = 100000;
else if (ACTION[s][dollar] != 100000)
printf("Conflict at ACTION[%d]['$']\n", s);
} else {
/* reduce by production p on each lookahead in item i */
int la;
for (la = 0; la < ASCII; ++la) if (st->look[i][la]) {
if (ACTION[s][la] == -999) ACTION[s][la] = -(p + 2);
else if (ACTION[s][la] != -(p + 2)) {
printf("Reduce/other conflict at ACTION[%d]['%c'] existing=%d new=%d\n", s,
la, ACTION[s][la], -(p + 2));
}
}
}
}
}
```

~Aditya
23BCE1344

```c
}
}
/* Print FIRST sets (nonterminals) - debugging */
static void print_FIRST() {
printf("\nFIRST sets (showing
terminals):\n"); int c,t;
for (c = 0; c < ASCII; ++c) if (used_sym[c] && is_nonterm[c]) {
printf("FIRST(%c) = { ", c);
for (t = 0; t < ASCII; ++t) if (FIRST[c][t]) printf("%c ", t);
printf("}\n");
}
}
/* Print CLR states (items) */
static void print_CLR_states() {
printf("\nCLR(1) canonical collection (%d states):\n", CLR_count);
int s;
for (s = 0; s < CLR_count; ++s) {
printf("I%d:\n", s);
State *st = &CLR[s];
int i;
for (i = 0; i < st->n; ++i) {
int p = st->prod[i], d = st->dot[i];
printf(" (%d) %c -> ", p, prods[p].lhs);
int k;
for (k = 0; k <= prods[p].rhs_len; ++k) {
if (k == d) printf(".");
if (k < prods[p].rhs_len) printf("%c", prods[p].rhs[k]);
}
printf(" , {");
int first = 1;
int la;
for (la = 0; la < ASCII; ++la) if (st->look[i][la]) {
if (!first) printf(", "); first = 0;
printf("%c", la);
}
printf("}\n");
}
}
}
/* Print ACTION and GOTO table neatly */
static void print_parsing_table() {
/* build list of terminal symbols to print (exclude '#', include '$') */
char term_list[ASCII]; int term_n = 0;
int c;
for (c = 0; c < ASCII; ++c) {
if (used_sym[c] && !is_nonterm[c]) {
if (c == (unsigned char)'#') continue;
term_list[term_n++] = (char)c;
}
}
/* build list of nonterminals */
char nonterm_list[ASCII]; int nonterm_n = 0;
for (c = 0; c < ASCII; ++c) if (used_sym[c] && is_nonterm[c])
nonterm_list[nonterm_n++] =
(char)c;
printf("\nParsing TABLE (CLR):\n ");
int t;
for (t = 0; t < term_n; ++t) printf("%6c", term_list[t]);
printf(" |");
int nt;
for (nt = 0; nt < nonterm_n; ++nt) printf("%6c", nonterm_list[nt]);
printf("\n");
int s;
for (s = 0; s < CLR_count; ++s) {
printf("%3d ", s);
for (t = 0; t < term_n; ++t) {
int col = (unsigned char)term_list[t];
int a = ACTION[s][col];
if (a == -999) printf("%6s", ".");
else if (a == 100000) printf("%6s", "acc");
else if (a > 0) { char buf[16]; snprintf(buf, sizeof(buf), "s%d", a-1);
printf("%6s", buf); }
else { int pidx = -(a) - 2; char buf[16]; snprintf(buf, sizeof(buf), "r%d",
pidx); printf("%6s",
buf); }
}
printf(" |");
int nt;
for (nt = 0; nt < nonterm_n; ++nt) {
int col = (unsigned char)nonterm_list[nt];
int g = GOTO_TABLE[s][col];
if (g == -999) printf("%6s", ".");
else { char buf[16]; snprintf(buf, sizeof(buf), "%d", g); printf("%6s", buf);
}
}
printf("\n");
}
}
/* Preprocess input: if token "id" appears, map to 'i'. If input contains
spaces, treat tokens,
otherwise char-by-char. */
static int build_input_tokens(const char *line, char tokens[], int *tcount) {
char tmp[4096];
strncpy(tmp, line, sizeof(tmp)-1); tmp[sizeof(tmp)-1]=0;
/* if spaces present, split by spaces */
int count = 0;
```

```c
char *p = strtok(tmp, " \t\r\n");
if (p == NULL) { *tcount = 0; return 0; }
/* if only one token and no spaces => also accept char-by-char */
int multi = strchr(line, ' ') != NULL || strchr(line, '\t') !=
NULL; if (!multi && strlen(p) > 1) {
/* treat string as sequence of characters */
int i;
for (i=0;i<strlen(p);++i) {
/* map "id" sequence -> 'i' (if appears), otherwise each char */
if (i+1 < strlen(p) && p[i]=='i' && p[i+1]=='d') { tokens[count++] = 'i'; ++i;
}
else tokens[count++] = p[i];
}
} else {
/* tokens separated by spaces */
while (p) {
if (strcmp(p, "id") == 0) tokens[count++] = 'i';
else if (strlen(p) == 1) tokens[count++] = p[0];
else {
/* if user typed "id" as single token or something unknown, attempt to
compress to first
char */
if (strcmp(p, "i") == 0) tokens[count++] = 'i';
else {
//fprintf(stderr, "Unknown token '%s' in input; please use single-char tokens
(or 'id' for
//identifier).\n", p);
return 0;
}
}
p = strtok(NULL, " \t\r\n");
}
}
*tcount = count;
return 1;
}
/* Parse using CLR ACTION/GOTO */
static void parse_input(const char *line) {
char tokens[4096];
int tn = 0;
if (!build_input_tokens(line, tokens, &tn)) return;
/* append $ if not present */
if (tn == 0) { printf("No tokens provided\n"); return; }
if (tokens[tn-1] != '$') tokens[tn++] = '$';
/* check tokens are terminals */
int i;
for (i = 0; i < tn; ++i) {
int ch = (unsigned char)tokens[i];
```

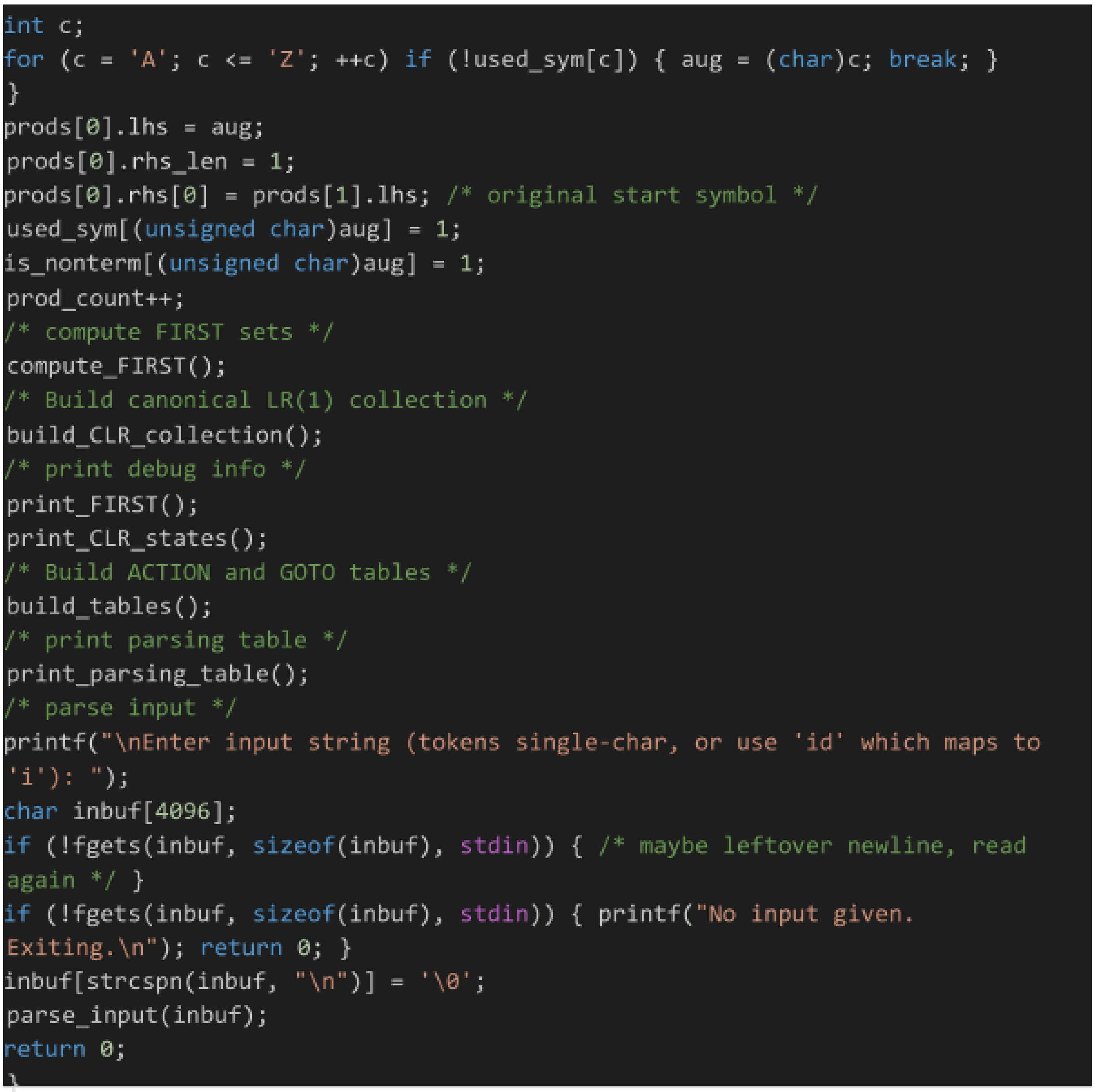
LAB Assignment 3 PART 2

```c
if (!used_sym[ch] || is_nonterm[ch]) {
fprintf(stderr, "Token '%c' not recognized as terminal in grammar. Aborting
parse.\n",
tokens[i]);
return;
}
}
int stack[16384]; int top = 0;
stack[top++] = 0;
int ip = 0;
printf("\nParsing steps:\n");
printf("%-30s %-20s %s\n", "Stack(states)", "Remaining", "Action");
printf("----------------------------------------------------------------\n");
while (1) {
/* stack display */
char sd[512] = ""; char rem[512] = "";
int j;
for (j = 0; j < top; ++j) { char buf[16]; snprintf(buf, sizeof(buf), "%d ",
stack[j]); strncat(sd,
buf, sizeof(sd)-strlen(sd)-1); }
for (j = ip; j < tn; ++j) { char buf[4]; snprintf(buf, sizeof(buf), "%c",
tokens[j]); strncat(rem,
buf, sizeof(rem)-strlen(rem)-1); if (j+1<tn) strncat(rem, " ", sizeof(rem)-
strlen(rem)-1); }
int state = stack[top-1];
int a = (unsigned char)tokens[ip];
int act = ACTION[state][a];
if (act == -999) {
printf("%-30s %-20s ERROR(no action)\n", sd, rem);
break;
}
if (act == 100000) {
printf("%-30s %-20s %s\n", sd, rem,
"Accept"); printf("Input accepted.\n");
break;
}
if (act > 0) {
int to = act -
1;
printf("%-30s %-20s Shift %d\n", sd, rem,
to); stack[top++] = to;
ip++;
continue;
}
if (act <= -2) {
int pidx = -(act) - 2;
/* perform reduction by pidx */
Prod *pr = &prods[pidx];
char rhsstr[128] = "";
```

~Aditya
23BCE1344

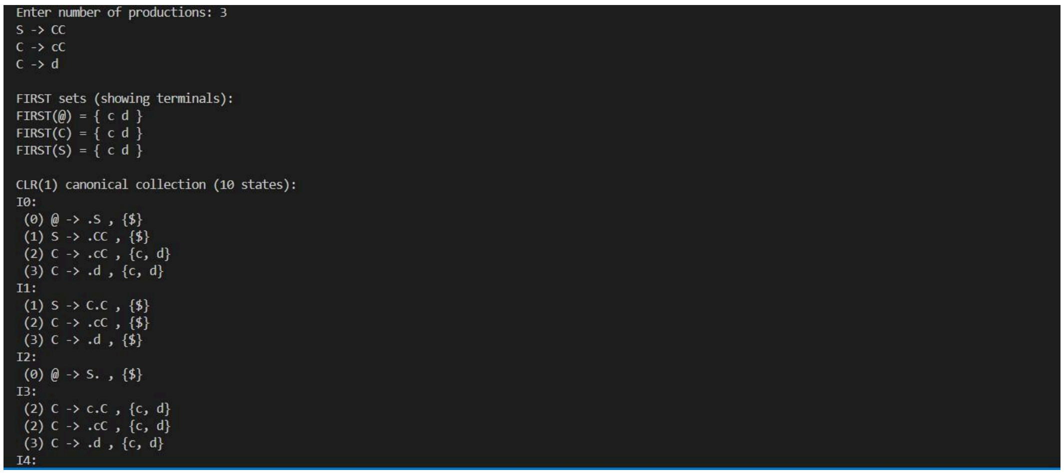
```
if (pr->rhs_len == 0) strcpy(rhsstr, "#");
else {
    int k;
for (k=0;k<pr->rhs_len;++k) { char b[2] = { pr->rhs[k], '\0' };
strncat(rhsstr, b,
sizeof(rhsstr)-strlen(rhsstr)-1); }
}
char actmsg[128];
snprintf(actmsg, sizeof(actmsg), "Reduce by %c->%s (prod %d)", pr->lhs,
rhsstr, pidx);
printf("%-30s %-20s %s\n", sd, rem, actmsg);
/* pop */
int popc = pr->rhs_len;
/* if rhs is epsilon (#) then pop 0 */
if (pr->rhs_len == 1 && pr->rhs[0] == '#') popc = 0;
top -= popc;
if (top <= 0) { printf("Parse stack underflow\n"); return; }
int stt = stack[top-1];
int g = GOTO_TABLE[stt][(unsigned char)pr->lhs];
if (g == -999) { printf("Goto missing after reduce: GOTO[%d][%c]\n", stt, pr-
>lhs); return; }
stack[top++] = g;
continue;
}
}
}
/* ------------------- Main ------------------- */
int main() {
/* read grammar */
memset(used_sym, 0, sizeof(used_sym));
memset(is_nonterm, 0, sizeof(is_nonterm));
prod_count = 0;
read_grammar();
finalize_symbols();
/* augment grammar: insert S' -> S (at prods[0]) */
if (prod_count + 1 >= MAX_PRODS) die("Too many productions");
int i;
for (i = prod_count; i >= 1; --i) prods[i] = prods[i-1];
/* choose augmented symbol S' stored as character '!' (rarely used) or "S'"?
We need
single-char; choose '`' if safe.
To keep single-char, pick uppercase not used; else use 'Z'+1 not possible.
Simpler: use character 1 (ASCII SOH) as augmented lhs, but printing would
be odd.
We'll use character '@' as augmented lhs (unlikely to be in grammar). */
char aug = '@';
/* ensure '@' not already used; if used, pick non-used ASCII */
if (used_sym[(unsigned char)aug]) {
```

LAB Assignment 3 PART 2



```
int c;
for (c = 'A'; c <= 'Z'; ++c) if (!used_sym[c]) { aug = (char)c; break; }
}
prods[0].lhs = aug;
prods[0].rhs_len = 1;
prods[0].rhs[0] = prods[1].lhs; /* original start symbol */
used_sym[(unsigned char)aug] = 1;
is_nonterm[(unsigned char)aug] = 1;
prod_count++;
/* compute FIRST sets */
compute_FIRST();
/* Build canonical LR(1) collection */
build_CLR_collection();
/* print debug info */
print_FIRST();
print_CLR_states();
/* Build ACTION and GOTO tables */
build_tables();
/* print parsing table */
print_parsing_table();
/* parse input */
printf("\nEnter input string (tokens single-char, or use 'id' which maps to
'i'): ");
char inbuf[4096];
if (!fgets(inbuf, sizeof(inbuf), stdin)) { /* maybe leftover newline, read
again */ }
if (!fgets(inbuf, sizeof(inbuf), stdin)) { printf("No input given.
Exiting.\n"); return 0; }
inbuf[strcspn(inbuf, "\n")] = '\0';
parse_input(inbuf);
return 0;
}
```

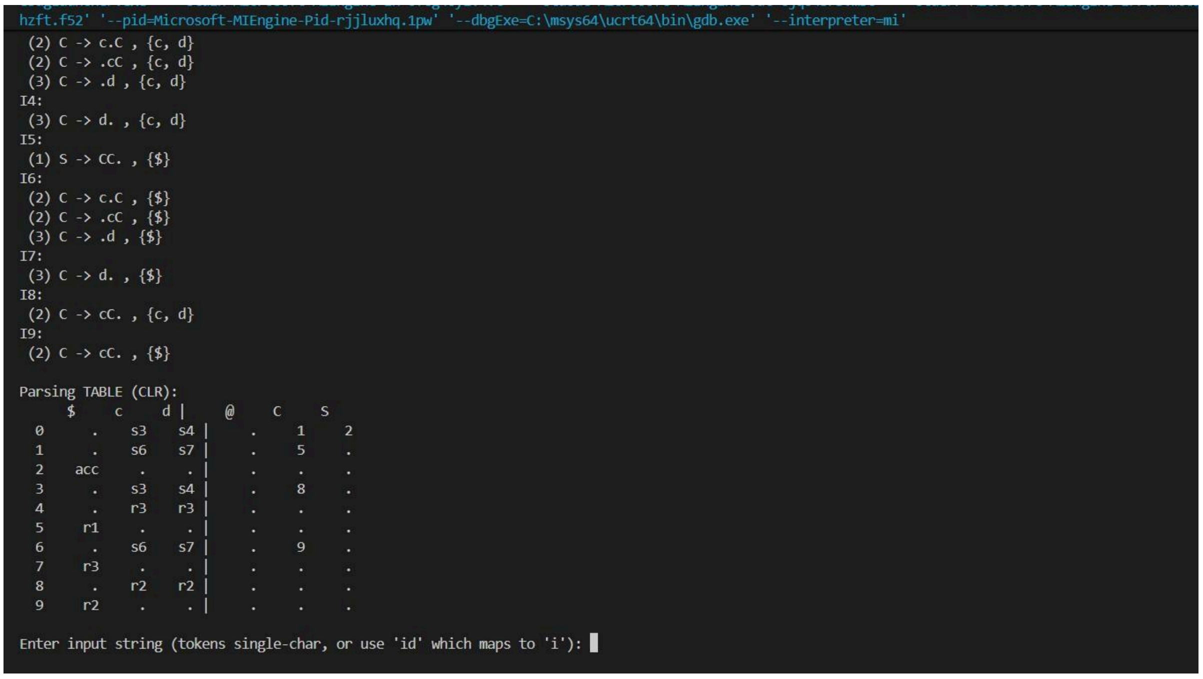
OUTPUT



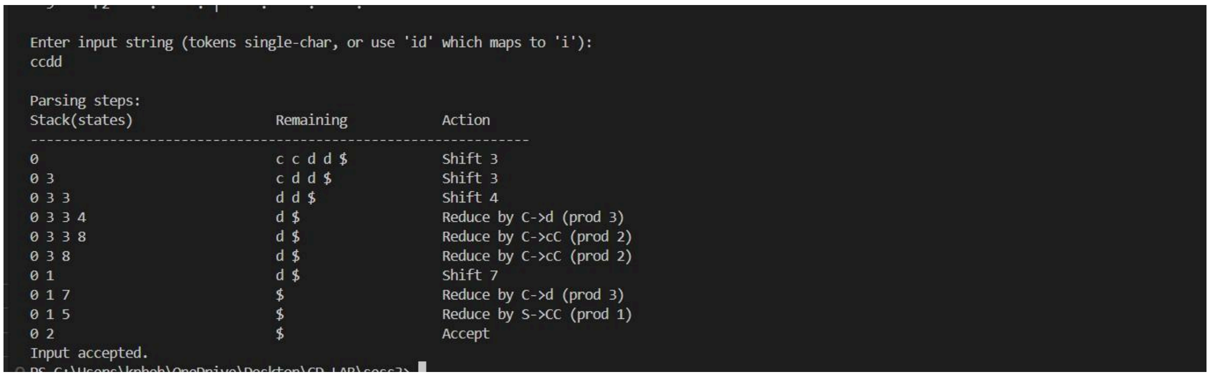
```
Enter number of productions: 3
S -> CC
C -> cC
C -> d

FIRST sets (showing terminals):
FIRST(@) = { c d }
FIRST(C) = { c d }
FIRST(S) = { c d }

CLR(1) canonical collection (10 states):
I0:
 (0) @ -> .S , {$}
 (1) S -> .CC , {$}
 (2) C -> .cC , {c, d}
 (3) C -> .d , {c, d}
I1:
 (1) S -> C.C , {$}
 (2) C -> .cC , {$}
 (3) C -> .d , {$}
I2:
 (0) @ -> S. , {$}
I3:
 (2) C -> c.C , {c, d}
 (2) C -> .cC , {c, d}
 (3) C -> .d , {c, d}
I4:
```

~Aditya
23BCE1344

LAB Assignment 3 PART 2



```
hzft.f52' '--pid=Microsoft-MIEngine-Pid-rjjluxhq.1pw' '--dbgExe=C:\msys64\ucrt64\bin\gdb.exe' '--interpreter=mi'
 (2) C -> c.C , {c, d}
 (2) C -> .cC , {c, d}
 (3) C -> .d , {c, d}
I4:
 (3) C -> d. , {c, d}
I5:
 (1) S -> CC. , {$}
I6:
 (2) C -> c.C , {$}
 (2) C -> .cC , {$}
 (3) C -> .d , {$}
I7:
 (3) C -> d. , {$}
I8:
 (2) C -> cC. , {c, d}
I9:
 (2) C -> cC. , {$}

Parsing TABLE (CLR):
      $     c     d |     @     C     S
  0     .    s3    s4 |     .     1     2
  1     .    s6    s7 |     .     5     .
  2   acc     .     . |     .     .     .
  3     .    s3    s4 |     .     8     .
  4     .    r3    r3 |     .     .     .
  5    r1     .     . |     .     .     .
  6     .    s6    s7 |     .     9     .
  7    r3     .     . |     .     .     .
  8     .    r2    r2 |     .     .     .
  9    r2     .     . |     .     .     .

Enter input string (tokens single-char, or use 'id' which maps to 'i'): 
```



```
Enter input string (tokens single-char, or use 'id' which maps to 'i'):
ccdd

Parsing steps:
Stack(states)                 Remaining           Action
---------------------------------------------------------------
0                             c c d d $           Shift 3
0 3                           c d d $             Shift 3
0 3 3                         d d $               Shift 4
0 3 3 4                       d $                 Reduce by C->d (prod 3)
0 3 3 8                       d $                 Reduce by C->cC (prod 2)
0 3 8                         d $                 Reduce by C->cC (prod 2)
0 1                           d $                 Shift 7
0 1 7                         $                   Reduce by C->d (prod 3)
0 1 5                         $                   Reduce by S->CC (prod 1)
0 2                           $                   Accept
Input accepted.
```

Experiment-11(a) Construct Look-Ahead LR (LALR) parse table using C language.

Experiment-11(b) Implement the LR parsing algorithm, get both parse table and input string are inputs. Use C language for implementation.

AIM

The objective of this experiment is to:

1. Construct the **Look-Ahead LR (LALR) Parse Table** for a given context-free grammar using C language.
2. Implement the **LR parsing algorithm** in C, taking the parse table and input string as inputs.
3. Demonstrate how LALR parsing maintains the power of CLR parsing but reduces the size of the parse table by merging equivalent states.

~Aditya
23BCE1344

LAB Assignment 3 PART 2

ALGORITHM

Step 1: Grammar Input

- Read grammar productions.
- Augment the grammar (S' → S).

Step 2: Construct Canonical LR(1) Collection

1. Start with initial item: [S' → •S, $].
2. Compute **closure(I)** and **goto(I, X)** as in CLR.
3. Generate the complete set of LR(1) item sets.

Step 3: Merge States for LALR(1)

1. Identify **states with identical LR(0) cores** (same productions ignoring lookaheads).
   - Example:
     - State I1: [A → α •, a]
     - State I2: [A → α •, b]
     - Both share LR(0) core [A → α •].
2. Merge such states by **combining their lookaheads**.
3. Repeat until no further merging is possible.

Step 4: Construct the LALR Parse Table

1. For each merged state:
   - If [A → α • a β, x] exists, add ACTION[state, a] = shift goto(state, a).
   - If [A → α •, x] exists, add ACTION[state, x] = reduce A → α.
   - If [S' → S •, $] exists, add ACTION[state, $] = accept.
   - For non-terminals, set GOTO[state, A] accordingly.
2. If conflicts (shift/reduce or reduce/reduce) occur → report them.

Step 5: LR Parsing Algorithm (Using LALR Table)

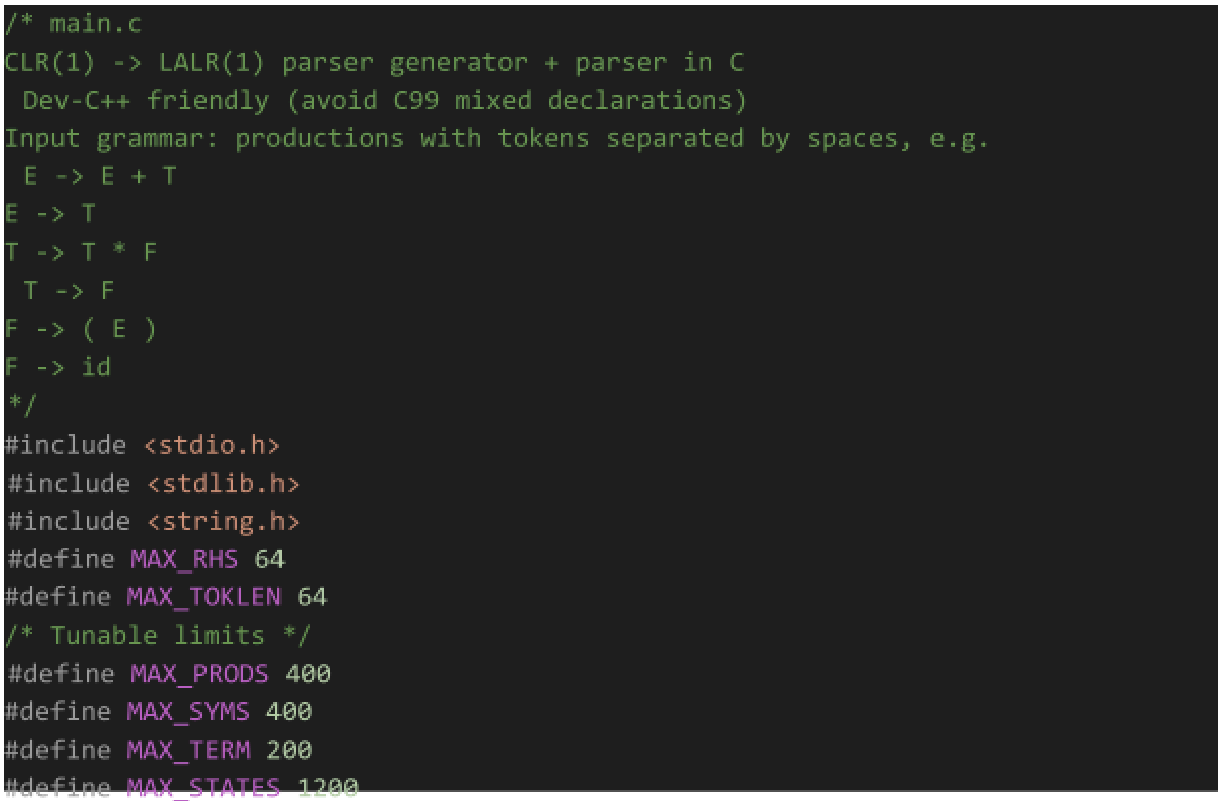
LAB Assignment 3 PART 2

1. Initialize stack with state 0.
2. Append $ at the end of input.
3. Loop until acceptance or error:
    - Let s = top of stack, a = current input symbol.
    - If ACTION[s, a] = shift t, push a, t and move input forward.
    - If ACTION[s, a] = reduce A ⟶ β, pop 2 × |β| symbols, let s' = top, push A and GOTO[s', A].
    - If ACTION[s, a] = accept, report success.
    - Else, report error.

Step 6: Output

- Print **LALR item sets** after merging.
- Display the **LALR parse table** (ACTION + GOTO).
- Show **step-by-step parsing** (stack, input, action).
- Final result: whether the string is **ACCEPTED** or **REJECTED**.

SOURCE CODE



```c
/* main.c
CLR(1) -> LALR(1) parser generator + parser in C
 Dev-C++ friendly (avoid C99 mixed declarations)
Input grammar: productions with tokens separated by spaces, e.g.
 E -> E + T
E -> T
T -> T * F
 T -> F
F -> ( E )
F -> id
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX_RHS 64
#define MAX_TOKLEN 64
/* Tunable limits */
#define MAX_PRODS 400
#define MAX_SYMS 400
#define MAX_TERM 200
#define MAX_STATES 1200
```

LAB Assignment 3 PART 2

```c
#define MAX_ITEMS_PER_STATE 1024
/* Symbol table */
static char sym_name[MAX_SYMS][MAX_TOKLEN];
static int sym_count = 0;
static int is_terminal[MAX_SYMS]; /* 1 = terminal, 0 = nonterminal */
static int term_index_of_sym[MAX_SYMS]; /* maps symbol idx -> terminal index,
-1 if not a terminal */
static int terminals[MAX_TERM];
static int term_count = 0;
/* Productions */
typedef struct {
int lhs;
 int rhs[MAX_RHS];
 int rhs_len;
} Prod;
static Prod prods[MAX_PRODS];
static int prod_count = 0;
/* FIRST sets (symbol x terminal bitset) */
static unsigned char FIRST[MAX_SYMS][MAX_TERM];
/* LR(1) Item */
typedef struct {
 int prod; /* production index */
 int dot; /* dot position */
 unsigned char look[MAX_TERM]; /* lookahead set over terminals */
} Item;
/* State: set of items */
typedef struct {
 Item items[MAX_ITEMS_PER_STATE];
 int item_count;
} State;
/* Dynamic arrays for states and tables */
static State *CLRstates = NULL;
static State *LALRstates = NULL;
static int CLRstate_count = 0;
static int LALRstate_count = 0;
/* ACTION/GOTO tables: allocated after sizes known */
static int **ACTION_CLR = NULL; /* ACTION_CLR[state][term_index] */
static int **GOTO_CLR = NULL; /* GOTO_CLR[state][symbol_index] */
static int **ACTION_LALR = NULL; /* ACTION_LALR[state][term_index] */
static int **GOTO_LALR = NULL; /* GOTO_LALR[state][symbol_index] */
/* Utility: add/get symbol id */
int get_sym(const char *s) { int
i;
 for (i = 0; i < sym_count; ++i)
 if (strcmp(sym_name[i], s) == 0) return i;
 if (sym_count >= MAX_SYMS) { fprintf(stderr, "Exceeded MAX_SYMS\n"); exit(1);
}
 strncpy(sym_name[sym_count], s, MAX_TOKLEN-1);
```

~Aditya
23BCE1344

```
 sym_name[sym_count][MAX_TOKLEN-1] = '\0';
 is_terminal[sym_count] = 1; /* default terminal; will mark LHS as nonterminal
later */
 term_index_of_sym[sym_count] = -1;
 sym_count++;
 return sym_count - 1;
}
/* register terminal symbol into terminals[] */
int add_terminal_sym(int s) {
 if (term_index_of_sym[s] != -1) return term_index_of_sym[s];
 if (term_count >= MAX_TERM) { fprintf(stderr, "Exceeded MAX_TERM\n");
exit(1); }
 term_index_of_sym[s] = term_count;
 terminals[term_count] = s;
 term_count++;
 return term_count - 1;
}
/* tokenize a line into tokens separated by whitespace */
int tokenize_line(char *line, char tokens[][MAX_TOKLEN]) {
int n = 0;
 char *p = strtok(line, " \t\r\n");
 while (p) {
 strncpy(tokens[n], p,
 MAX_TOKLEN-1);
 tokens[n][MAX_TOKLEN-1] = '\0';
 n++;
 p = strtok(NULL, " \t\r\n");
 }
 return n;
}
/* compute FIRST sets (conservative: terminals only; no epsilon support) */
void compute_FIRST() {
 int i, j;
 /* initialize */
 for (i = 0; i < sym_count; ++i)
 for (j = 0; j < term_count; ++j)
 FIRST[i][j] = 0;
 /* FIRST of terminal contains itself */
 for (i = 0; i < term_count; ++i) {
 int s = terminals[i];
 FIRST[s][i] = 1;
 }
 int changed = 1;
 while (changed) {
 changed = 0;
 for (i = 0; i < prod_count; ++i) {
 int A = prods[i].lhs;
 if (prods[i].rhs_len == 0) continue;
 int X = prods[i].rhs[0];
```

~Aditya
23BCE1344

LAB Assignment 3 PART 2

```
 for  (j = 0; j < term_count; ++j)
 {      if      (!FIRST[A][j]     &&
 FIRST[X][j]) { FIRST[A][j] = 1;
 changed = 1;
 }
 }
 }
 }
}
/* lookahead helpers */
void look_clear(Item *it)
{ int i;
 for (i = 0; i < term_count; ++i) it->look[i] = 0;
}
void look_copy(Item *dst, Item *src) {
 int i;
 for (i = 0; i < term_count; ++i) dst->look[i] = src->look[i];
}
/* compare item core (prod,dot) only */
int item_core_eq(Item *a, Item *b) { return a->prod == b->prod && a->dot == b-
>dot; }
/* find item index by core in state, returns index or -1 */
int state_find_item_core(State *st, Item *it) {
 int i;
 for (i = 0; i < st->item_count; ++i)
 if (st->items[i].prod == it->prod && st->items[i].dot == it->dot) return i;
 return -1;
}
/* add item to state; if core exists, union lookahead; return 1 if added or
lookahead changed */
int state_add_item(State *st, Item *it) {
 int idx = state_find_item_core(st, it);
 if (idx == -1) {
 if (st->item_count >= MAX_ITEMS_PER_STATE) { fprintf(stderr, "Exceeded items
per state\n"); exit(1); }
 st->items[st->item_count++] = *it;
 return 1;
 } else {
 int changed = 0;
 int t;
 for (t = 0; t < term_count; ++t) {
 if (it->look[t] && !st->items[idx].look[t])
 { st->items[idx].look[t] = 1;
 changed = 1;
 }
 }
 return changed;
 }
```

~Aditya
23BCE1344

LAB Assignment 3 PART 2

```
}
/* closure operation */
void closure(State *st) {
int added = 1;
 while (added) {
 added = 0;
 {
 int i;
 for (i = 0; i < st->item_count; ++i) {
 Item *it = &st->items[i];
 Prod *p = &prods[it->prod];
 if (it->dot >= p->rhs_len) continue;
 int B = p->rhs[it->dot];
 if (is_terminal[B]) continue; /* only expand nonterminals */
 /* For each production B -> gamma, create item (B -> . gamma) with lookahead
computed
 as FIRST(beta) U lookahead (conservative approximation: FIRST of first symbol
of beta)
 */
 {
 int pr;
 for (pr = 0; pr < prod_count; ++pr)
 { if (prods[pr].lhs != B) continue;
 Item newit;
 newit.prod = pr;
 newit.dot = 0;
 look_clear(&newit);
 /* compute lookahead: if beta empty -> use it->look, else use FIRST of first
symbol of beta */
 if (it->dot + 1 >= p->rhs_len) {
 /* beta empty */
 int la;
 for (la = 0; la < term_count; ++la) if (it->look[la]) newit.look[la] = 1;
 } else {
 int X = p->rhs[it->dot + 1];
 int tt;
 for (tt = 0; tt < term_count; ++tt) if (FIRST[X][tt]) newit.look[tt] = 1;
 }
 if (state_add_item(st, &newit)) added = 1;
 }
 }
 }
 }
 }
}
/* goto operation: compute items with dot advanced over X */
void go_to(State *I, int X, State *out) {
 int i;
```

~Aditya
23BCE1344

LAB Assignment 3 PART 2

```
 out->item_count = 0;
 for (i = 0; i < I->item_count; ++i) {
 Item *it = &I->items[i];
 Prod *p = &prods[it->prod];
 if (it->dot < p->rhs_len && p->rhs[it->dot] == X) {
 Item nit;
 nit.prod = it->prod;
 nit.dot = it->dot + 1;
 look_clear(&nit);
 look_copy(&nit, it);
 state_add_item(out, &nit);
 }
 }
 closure(out);
}
/* compare two states for exact LR(1) equality (items + lookahead) */
int state_equal(State *A, State *B) {
 int i, j, t;
 if (A->item_count != B->item_count) return
 0; for (i = 0; i < A->item_count; ++i) {
 int found = 0;
 for (j = 0; j < B->item_count; ++j) {
 if (A->items[i].prod == B->items[j].prod && A->items[i].dot == B-
>items[j].dot) {
 int same = 1;
 for (t = 0; t < term_count; ++t) if (A->items[i].look[t] != B-
>items[j].look[t]) { same = 0; break; }
 if (same) { found = 1; break; }
 }
 }
 if (!found) return 0;
 }
 return 1;
}
/* core signature for LR(0) core: textual */
void core_signature(State *st, char *buf, int buf_sz) {
 int i;
 buf[0] = '\0';
 for (i = 0; i < st->item_count; ++i) {
 char tmp[128];
 /* snprintf is usually available; using it here */
 snprintf(tmp, sizeof(tmp), "%d:%d;", st->items[i].prod, st->items[i].dot);
 /* safe strcat */
 strncat(buf, tmp, buf_sz - strlen(buf) - 1);
 }
}
/* build CLR(1) canonical collection */
void build_CLR_collection() {
```

~Aditya
23BCE1344

LAB Assignment 3 PART 2

```
CLRstates = (State *) malloc(sizeof(State) * MAX_STATES);
if (!CLRstates) { fprintf(stderr, "malloc failed\n"); exit(1); }
CLRstate_count = 0;
/* initial item: augmented production should be prods[0] */
{
State I0;
I0.item_count = 0;
Item it0; it0.prod = 0; it0.dot = 0; look_clear(&it0);
/* find terminal index of $ */
int doll_sym = get_sym("$");
is_terminal[doll_sym] = 1;
add_terminal_sym(doll_sym); /* ensure terminal index exists */
/* we need the terminal index for $ */
/* find it */
{
int dollar_term_idx = term_index_of_sym[doll_sym];
if (dollar_term_idx >= 0) it0.look[dollar_term_idx] = 1;
}
state_add_item(&I0, &it0);
closure(&I0);
CLRstates[CLRstate_count++] = I0;
}
{
int changed = 1;
while (changed) {
changed = 0;
{
int s;
for (s = 0; s < CLRstate_count; ++s) {
int X;
for (X = 0; X < sym_count; ++X) {
State J; go_to(&CLRstates[s], X,
&J); if (J.item_count == 0)
continue;
/* check if J already present */
int found = -1;
{
int k;
for (k = 0; k < CLRstate_count; ++k) if (state_equal(&CLRstates[k], &J)) {
found = k; break; }
}
if (found == -1) {
if (CLRstate_count >= MAX_STATES) { fprintf(stderr, "Exceeded MAX_STATES\n");
exit(1); }
CLRstates[CLRstate_count++] = J;
changed = 1;
}
}
}
```

~Aditya
23BCE1344

```
 }
 }
 }
}
/* Allocate ACTION/GOTO for CLR dynamically
*/ void alloc_CLR_tables() {
 int i, j;
 ACTION_CLR = (int **) malloc(sizeof(int *) * CLRstate_count);
 GOTO_CLR = (int **) malloc(sizeof(int *) * CLRstate_count);
 if (!ACTION_CLR || !GOTO_CLR) { fprintf(stderr, "malloc failed\n"); exit(1);
}
 for (i = 0; i < CLRstate_count; i++) {
 ACTION_CLR[i] = (int *) malloc(sizeof(int) * term_count);
 GOTO_CLR[i] = (int *) malloc(sizeof(int) * sym_count);
 if (!ACTION_CLR[i] || !GOTO_CLR[i]) { fprintf(stderr, "malloc failed\n");
exit(1); }
 for (j = 0; j < term_count; j++) ACTION_CLR[i][j] = -1;
 for (j = 0; j < sym_count; j++) GOTO_CLR[i][j] = -1;
 }
}
/* Build CLR ACTION/GOTO tables */
void build_CLR_tables() {
alloc_CLR_tables();
 /* We'll use a flattened temp_goto: temp_goto[s * sym_count + X] = target CLR
state or -1 */
 int *temp_goto = (int *) malloc(sizeof(int) * CLRstate_count * sym_count);
 if (!temp_goto) { fprintf(stderr, "malloc failed\n"); exit(1); }
 {
 int i;
 for (i = 0; i < CLRstate_count * sym_count; ++i) temp_goto[i] = -1;
 }
 {
 int s, X;
 for (s = 0; s < CLRstate_count; ++s)
 { for (X = 0; X < sym_count; ++X) {
 State J; go_to(&CLRstates[s], X, &J);
 if (J.item_count == 0) continue;
 int found = -1;
 {
 int k;
 for (k = 0; k < CLRstate_count; ++k) if (state_equal(&CLRstates[k], &J)) {
found = k; break; }
 }
 if (found != -1) temp_goto[s * sym_count + X] = found;
 }
 }
 }
 /* Populate ACTION_CLR and GOTO_CLR using LR(1) items */
```

LAB Assignment 3 PART 2

```
 {
 int s, i2, t;
 for (s = 0; s < CLRstate_count; ++s) {
 State *st = &CLRstates[s];
 for (i2 = 0; i2 < st->item_count; ++i2) {
 Item *it = &st->items[i2];
 Prod          *p          =
 &prods[it->prod];        if
 (it->dot  <  p->rhs_len)  {
 int a = p->rhs[it->dot];
 if (is_terminal[a]) {
 int tidx = term_index_of_sym[a];
 int to = temp_goto[s * sym_count + a];
 if (tidx >= 0 && to != -1) ACTION_CLR[s][tidx] = to + 1; /* shift */
 }
 } else {
 if (it->prod == 0) {
 for (t = 0; t < term_count; ++t) if (it->look[t]) ACTION_CLR[s][t] = 100000;
/* accept */
 } else {
 for (t = 0; t < term_count; ++t) if (it->look[t]) ACTION_CLR[s][t] = -(it-
>prod + 2); /* reduce */
 }
 }
 }
 {
 int X;
 for (X = 0; X < sym_count; ++X) {
 if (temp_goto[s * sym_count + X] != -1) GOTO_CLR[s][X] = temp_goto[s *
sym_count + X];
 }
 }
 }
 }
 free(temp_goto);
}
/* Build LALR by merging states with same LR(0) core (merge lookahead sets) */
void build_LALR_from_CLR() {
 /* compute signature for each CLR state */
 char **signatures = (char **) malloc(sizeof(char *) * CLRstate_count);
 if (!signatures) { fprintf(stderr, "malloc failed\n"); exit(1); }
 {
 int s;
 for (s = 0; s < CLRstate_count; ++s) {
 signatures[s] = (char *) malloc(8192);
 if (!signatures[s]) { fprintf(stderr, "malloc failed\n"); exit(1); }
 core_signature(&CLRstates[s], signatures[s], 8192);
 }
 }
```

~Aditya
23BCE1344

LAB Assignment 3 PART 2

```
 int *mapping = (int *) malloc(sizeof(int) * CLRstate_count);
 if (!mapping) { fprintf(stderr, "malloc failed\n"); exit(1);
 }
 {
 int i;
 for (i = 0; i < CLRstate_count; ++i) mapping[i] = -1;
 }
 {
 int groups = 0;
 int s, t;
 for (s = 0; s < CLRstate_count; ++s) {
 if (mapping[s] != -1) continue;
 mapping[s] = groups;
 for (t = s + 1; t < CLRstate_count; ++t)
 if (strcmp(signatures[s], signatures[t]) == 0) mapping[t] = mapping[s];
 groups++;
 }
 LALRstate_count = groups;
 }
 LALRstates = (State *) malloc(sizeof(State) * LALRstate_count);
 if (!LALRstates) { fprintf(stderr, "malloc failed\n"); exit(1);
 }
 {
 int g;
 for (g = 0; g < LALRstate_count; ++g) LALRstates[g].item_count = 0;
 }
 /* merge items (union lookaheads for same cores) */
 {
 int s, it;
 for (s = 0; s < CLRstate_count; ++s) {
 int g = mapping[s];
 for (it = 0; it < CLRstates[s].item_count; ++it) {
 Item *ci = &CLRstates[s].items[it];
 int found = -1;
 {
 int j;
 for (j = 0; j < LALRstates[g].item_count; ++j) {
 if (LALRstates[g].items[j].prod == ci->prod && LALRstates[g].items[j].dot
== ci->dot) { found = j; break; }
 }
 }
 if (found == -1) {
 if (LALRstates[g].item_count >= MAX_ITEMS_PER_STATE) { fprintf(stderr,
"Exceeded items per LALR state\n"); exit(1); }
LALRstates[g].items[LALRstates[g].item_count++] = *ci;
 } else {
 int la;
 for (la = 0; la < term_count; ++la) if (ci->look[la])
LALRstates[g].items[found].look[la] = 1;
 }
```

~Aditya
23BCE1344

```
 }
 }
 }
 {
 int s;
 for (s = 0; s < CLRstate_count; ++s) free(signatures[s]);
 }
 free(signatures);
 free(mapping);
}
/* Allocate ACTION/GOTO tables for LALR */
void alloc_LALR_tables() {
 int i, j;
 ACTION_LALR = (int **) malloc(sizeof(int *) * LALRstate_count);
 GOTO_LALR = (int **) malloc(sizeof(int *) * LALRstate_count);
 if (!ACTION_LALR || !GOTO_LALR) { fprintf(stderr, "malloc failed\n");
exit(1); }
 for (i = 0; i < LALRstate_count; ++i) {
 ACTION_LALR[i] = (int *) malloc(sizeof(int) * term_count);
 GOTO_LALR[i] = (int *) malloc(sizeof(int) * sym_count);
 if (!ACTION_LALR[i] || !GOTO_LALR[i]) { fprintf(stderr, "malloc failed\n");
exit(1); }
 for (j = 0; j < term_count; ++j) ACTION_LALR[i][j] = -1;
 for (j = 0; j < sym_count; ++j) GOTO_LALR[i][j] = -1;
 }
}
/* Recompute CLR goto mapping temporarily and build LALR tables using state
core mapping */
void build_LALR_tables_from_CLR() {
 /* First compute CLR goto mapping temp_goto[CLRstate][symbol] -> CLR target
(flattened) */
 int *temp_goto = (int *) malloc(sizeof(int) * CLRstate_count * sym_count);
 if (!temp_goto) { fprintf(stderr, "malloc failed\n"); exit(1); }
 {
 int i;
 for (i = 0; i < CLRstate_count * sym_count; ++i) temp_goto[i] = -1;
 }
 {
 int s, X;
 for (s = 0; s < CLRstate_count; ++s)
 { for (X = 0; X < sym_count; ++X) {
 State J; go_to(&CLRstates[s], X, &J);
 if (J.item_count == 0) continue;
 int found = -1;
 {
 int k;
 for (k = 0; k < CLRstate_count; ++k) if (state_equal(&CLRstates[k], &J)) {
found = k; break; }
```

LAB Assignment 3 PART 2

```
}
if (found != -1) temp_goto[s * sym_count + X] = found;
}
}
}
/* Build mapping CLR state -> LALR state by core signature */
char **signatures = (char **) malloc(sizeof(char *) * CLRstate_count);
if (!signatures) { fprintf(stderr, "malloc failed\n"); exit(1); }
{
int s;
for (s = 0; s < CLRstate_count; ++s) {
signatures[s] = (char *) malloc(8192);
if (!signatures[s]) { fprintf(stderr, "malloc failed\n"); exit(1); }
core_signature(&CLRstates[s], signatures[s], 8192);
}
}
int *map = (int *) malloc(sizeof(int) * CLRstate_count);
if (!map) { fprintf(stderr, "malloc failed\n"); exit(1);
}
{
int i;
for (i = 0; i < CLRstate_count; ++i) map[i] = -1;
}
{
int gcount = 0;
int s, t;
for (s = 0; s < CLRstate_count; ++s) {
if (map[s] != -1) continue;
map[s] = gcount;
for (t = s + 1; t < CLRstate_count; ++t) if (strcmp(signatures[s],
signatures[t]) == 0) map[t] = map[s];
gcount++;
}
if (gcount != LALRstate_count) {
/* adjust if needed */
LALRstate_count = gcount;
}
}
alloc_LALR_tables();
/* Build GOTO_LALR */
{
int s, X;
for (s = 0; s < CLRstate_count; ++s) {
int L = map[s];
for (X = 0; X < sym_count; ++X) {
int t = temp_goto[s * sym_count +
X]; if (t == -1) continue;
int T = map[t];
GOTO_LALR[L][X] = T;
```

~Aditya
23BCE1344

```
 }
 }
 }
 /* Build ACTION_LALR: map CLR actions into merged LALR states */
 {
 int s, i2;
 for (s = 0; s < CLRstate_count; ++s) {
 int L = map[s];
 State *st = &CLRstates[s];
 for (i2 = 0; i2 < st->item_count; ++i2) {
 Item *it = &st->items[i2];
 Prod          *p          =
 &prods[it->prod];         if
 (it->dot  <  p->rhs_len)  {
 int a = p->rhs[it->dot];
 if (is_terminal[a]) {
 int t = term_index_of_sym[a];
 int to = temp_goto[s * sym_count + a];
 if (t >= 0 && to != -1) {
 int T = map[to];
 ACTION_LALR[L][t] = T + 1; /* shift */
 }
 }
 } else {
 int la;
 if (it->prod == 0) {
 for (la = 0; la < term_count; ++la) if (it->look[la]) ACTION_LALR[L][la] =
100000;
 } else {
 for (la = 0; la < term_count; ++la) if (it->look[la]) ACTION_LALR[L][la] =
- (it->prod + 2);
 }
 }
 }
 }
 }
 {
 int s;
 for (s = 0; s < CLRstate_count; ++s) free(signatures[s]);
 }
 free(signatures);
 free(map);
 free(temp_goto);
}
/* printing helpers */
void print_prod(int idx)
{ int i;
 printf("%s ->", sym_name[prods[idx].lhs]);
```

LAB Assignment 3 PART 2

```
 for (i = 0; i < prods[idx].rhs_len; ++i) printf(" %s",
sym_name[prods[idx].rhs[i]]);
}
/* Parse input using LALR table */
void parse_input_with_LALR(char *input_line)
 { char tokens[4096][MAX_TOKLEN];
 int tn = 0;
 char tmp[16384];
 strncpy(tmp, input_line, sizeof(tmp) - 1);
 tmp[sizeof(tmp) - 1] = 0;
 {
 char *p = strtok(tmp, " \t\r\n");
 while (p && tn < 4096) { strncpy(tokens[tn++], p, MAX_TOKLEN - 1); tokens[tn
- 1][MAX_TOKLEN - 1] = 0; p = strtok(NULL, " \t\r\n"); }
 }
 if (tn == 0) { printf("No input provided\n"); return; }
 if (strcmp(tokens[tn - 1], "$") != 0) { strncpy(tokens[tn++], "$", MAX_TOKLEN
- 1); }
 {
 int input_term_idx[4096];
 int i, s;
 for (i = 0; i < tn; ++i) {
 int sym = -1;
 for (s = 0; s < sym_count; ++s) if (strcmp(sym_name[s], tokens[i]) == 0) {
sym = s; break; }
 if (sym == -1) { printf("Token '%s' not recognized in grammar symbols.
Aborting parse.\n", tokens[i]); return; }
 if (!is_terminal[sym]) { printf("Token '%s' is not a terminal according to
grammar. Aborting parse.\n", tokens[i]); return; }
 {
 int tidx = term_index_of_sym[sym];
 if (tidx == -1) { printf("Token '%s' not in terminal index. Aborting.\n",
tokens[i]); return; }
 input_term_idx[i] = tidx;
 }
 }
 int state_stack[16384];
 int top = 0;
 state_stack[top++] = 0;
 int ip = 0;
 printf("\nParsing steps:\n");
 while (1) {
 int state = state_stack[top - 1];
 int a = input_term_idx[ip];
 int act = ACTION_LALR[state][a];
 int k;
 printf("Stack:");
 for (k = 0; k < top; ++k) printf(" %d", state_stack[k]);
```

~Aditya
23BCE1344

LAB Assignment 3 PART 2

```c
 printf(" Next token: %s Action: ", sym_name[terminals[a]]);
 if (act == -1) { printf("error\n"); break; }
 if (act == 100000) { printf("accept\n"); printf("Input accepted.\n"); break;
}
 if (act > 0) {
 printf("shift to %d\n", act - 1);
 state_stack[top++] = act - 1;
 ip++;
 } else if (act <= -2) {
 int prod_idx = -(act) -
 2;
 printf("reduce by prod %d: ", prod_idx); print_prod(prod_idx); printf("\n");
 {
 int pop = prods[prod_idx].rhs_len;
 top -= pop;
 int stt = state_stack[top - 1];
 int A = prods[prod_idx].lhs;
 int gotoState = GOTO_LALR[stt][A];
 if (gotoState == -1) { printf("Goto error after reduction\n"); break; }
 state_stack[top++] = gotoState;
 }
 } else {
 printf("unknown action\n"); break;
 }
 }
 }
}
/* MAIN */
int main() {
int i;
 for (i = 0; i < MAX_SYMS; ++i) is_terminal[i] = 1;
 for (i = 0; i < MAX_SYMS; ++i) term_index_of_sym[i] = -1;
 printf("Enter number of productions (not counting augmented production): ");
 {
 int n;
 if (scanf("%d", &n) != 1) return 0;
 getchar();
 if (n <= 0) { printf("Need at least one production.\n"); return 0; }
printf("Enter productions, one per line, tokens separated by spaces. Example:
E -> E + T\n");
 char line[4096];
 char tokens[256][MAX_TOKLEN];
 for (i = 0; i < n; ++i) {
 if (!fgets(line, sizeof(line), stdin)) { fprintf(stderr, "Unexpected EOF\n");
return 1; }
 if (line[0] == '\n' || line[0] == '\0') { i--; continue; }
 line[strcspn(line, "\n")] = '\0';
 char copy[4096];
 strncpy(copy, line, sizeof(copy) - 1);
```

~Aditya
23BCE1344

LAB Assignment 3 PART 2

```
 copy[sizeof(copy) - 1] = 0;
 {
 int tn = tokenize_line(copy, tokens);
 if (tn < 3) { printf("Bad production format: %s\n", line); return 1; }
if (strcmp(tokens[1], "->") != 0 && strcmp(tokens[1], "?") != 0) {
printf("Bad production arrow in: %s\n", line); return 1; }
 int lhs_sym = get_sym(tokens[0]);
 is_terminal[lhs_sym] = 0;
 Prod p; p.lhs = lhs_sym; p.rhs_len = 0;
 {
 int k;
 for (k = 2; k < tn; ++k) {
 int s = get_sym(tokens[k]);
 p.rhs[p.rhs_len++] = s;
 }
 }
 if (prod_count >= MAX_PRODS) { fprintf(stderr, "Too many productions\n");
return 1; }
 prods[prod_count++] = p;
 }
 }
 /* Create augmented production S' -> start (put at prods[0]) */
 {
 int start_sym = prods[0].lhs;
 int aug_sym = get_sym("S'");
 is_terminal[aug_sym] = 0;
 if (prod_count + 1 >= MAX_PRODS) { fprintf(stderr, "Too many productions\n");
return 1; }
 {
 int j;
 for (j = prod_count; j > 0; --j) prods[j] = prods[j - 1];
 }
 prods[0].lhs = aug_sym;
 prods[0].rhs_len = 1;
 prods[0].rhs[0] = start_sym;
 prod_count++;
 }
 /* determine terminals set */
 {
 int s;
 for (s = 0; s < sym_count; ++s) {
 if (is_terminal[s]) add_terminal_sym(s);
 }
 }
 compute_FIRST();
 build_CLR_collection();
 printf("\nConstructed CLR(1) states: %d\n", CLRstate_count);
 build_CLR_tables();
```

~Aditya
23BCE1344

LAB Assignment 3 PART 2

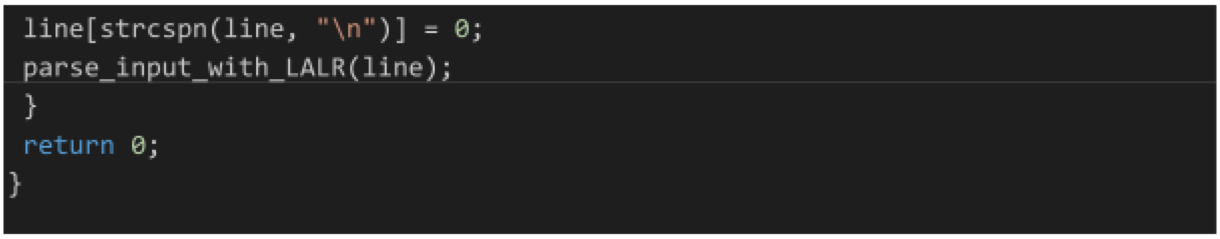
```c
/* print CLR states */
{
int s, it;
for (s = 0; s < CLRstate_count; ++s) {
printf("\nState %d:\n", s);
State *st = &CLRstates[s];
for (it = 0; it < st->item_count; ++it) {
Item *item = &st->items[it];
print_prod(item->prod);
printf(" . at %d look={", item->dot);
{
int first = 1;
int t;
for (t = 0; t < term_count; ++t) if (item->look[t]) {
if (!first) printf(", ");
printf("%s", sym_name[terminals[t]]);
first = 0;
}
}
printf("}\n");
}
}
}
build_LALR_from_CLR();
build_LALR_tables_from_CLR();
printf("\nConstructed LALR(1) states: %d\n", LALRstate_count);
/* print LALR states */
{
int s, it;
for (s = 0; s < LALRstate_count; ++s) {
printf("\nLALR State %d:\n", s);
State *st = &LALRstates[s];
for (it = 0; it < st->item_count; ++it) {
Item *item = &st->items[it];
print_prod(item->prod);
printf(" . at %d look={", item->dot);
{
int first = 1;
int t;
for (t = 0; t < term_count; ++t) if (item->look[t]) {
if (!first) printf(", ");
printf("%s", sym_name[terminals[t]]);
first = 0;
}
}
printf("}\n");
}
}
```

~Aditya
23BCE1344

LAB Assignment 3 PART 2

```c
}
/* Print LALR ACTION/GOTO tables */
printf("\nLALR ACTION table (rows: states, cols: terminals):\n ");
{
int t;
for (t = 0; t < term_count; ++t) printf("%8s", sym_name[terminals[t]]);
}
printf("\n");
{
int s, t;
for (s = 0; s < LALRstate_count; ++s) {
printf("%3d ", s);
for (t = 0; t < term_count; ++t) {
int a = ACTION_LALR[s][t];
if (a == -1) printf("%8s", ".");
else if (a == 100000) printf("%8s", "acc");
else if (a > 0) { char buf[32]; sprintf(buf, "s%d", a - 1); printf("%8s",
buf); }
else { int pidx = -(a) - 2; char buf[32]; sprintf(buf, "r%d", pidx);
printf("%8s", buf); }
}
printf("\n");
}
}
printf("\nLALR GOTO table (rows: states, cols: nonterminals):\n ");
{
int s;
for (s = 0; s < sym_count; ++s) if (!is_terminal[s]) printf("%8s",
sym_name[s]);
}
printf("\n");
{
int i2, s;
for (i2 = 0; i2 < LALRstate_count; ++i2) {
printf("%3d ", i2);
for (s = 0; s < sym_count; ++s) if (!is_terminal[s]) {
int g = GOTO_LALR[i2][s];
if (g == -1) printf("%8s", ".");
else { char buf[16]; sprintf(buf, "%d", g); printf("%8s", buf); }
}
printf("\n");
}
}
/* parse input */
printf("\nEnter input tokens separated by spaces (end with $ or it will be
appended):\n");
if (!fgets(line, sizeof(line), stdin)) { fprintf(stderr, "Unexpected EOF\n");
return 1; }
```

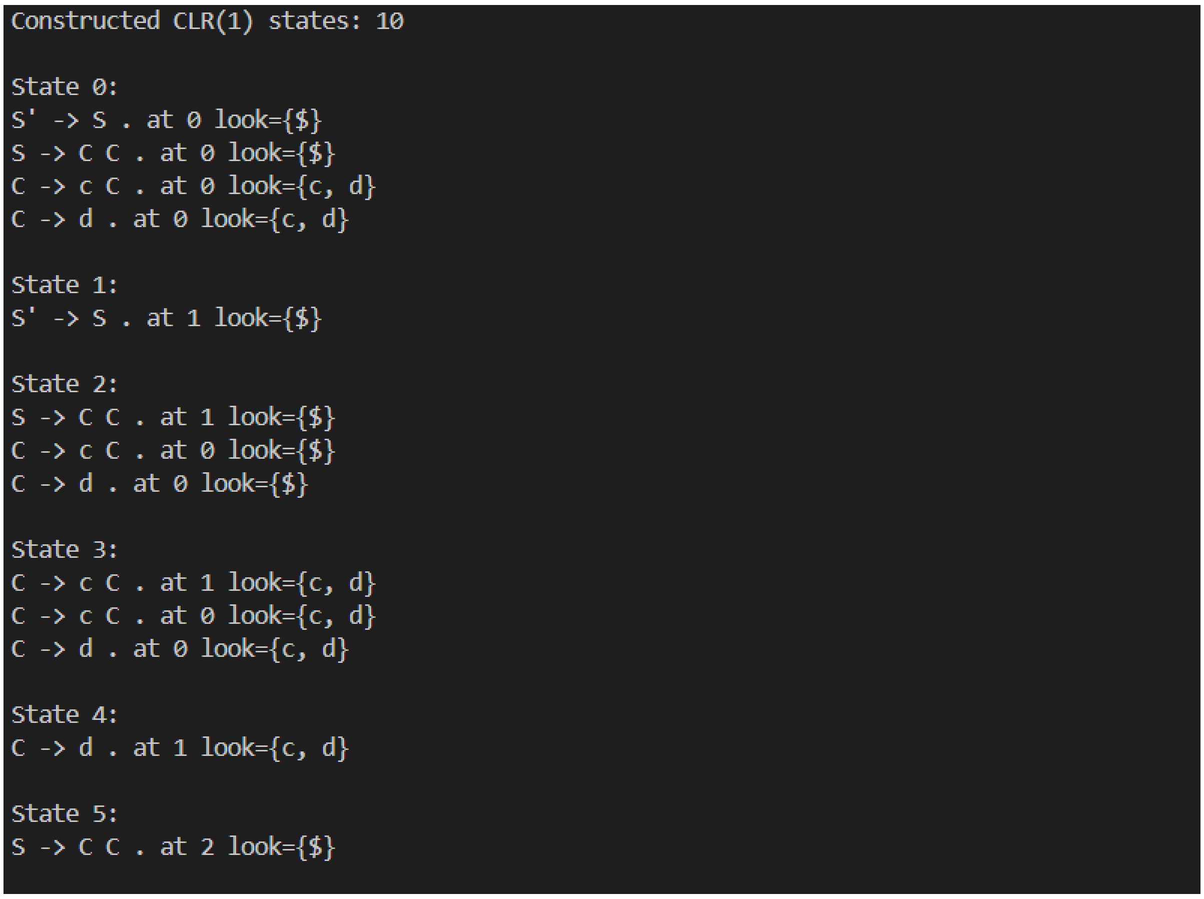
~Aditya
23BCE1344



```
 line[strcspn(line, "\n")] = 0;
 parse_input_with_LALR(line);
 }
 return 0;
}
```

LAB Assignment 3 PART 2

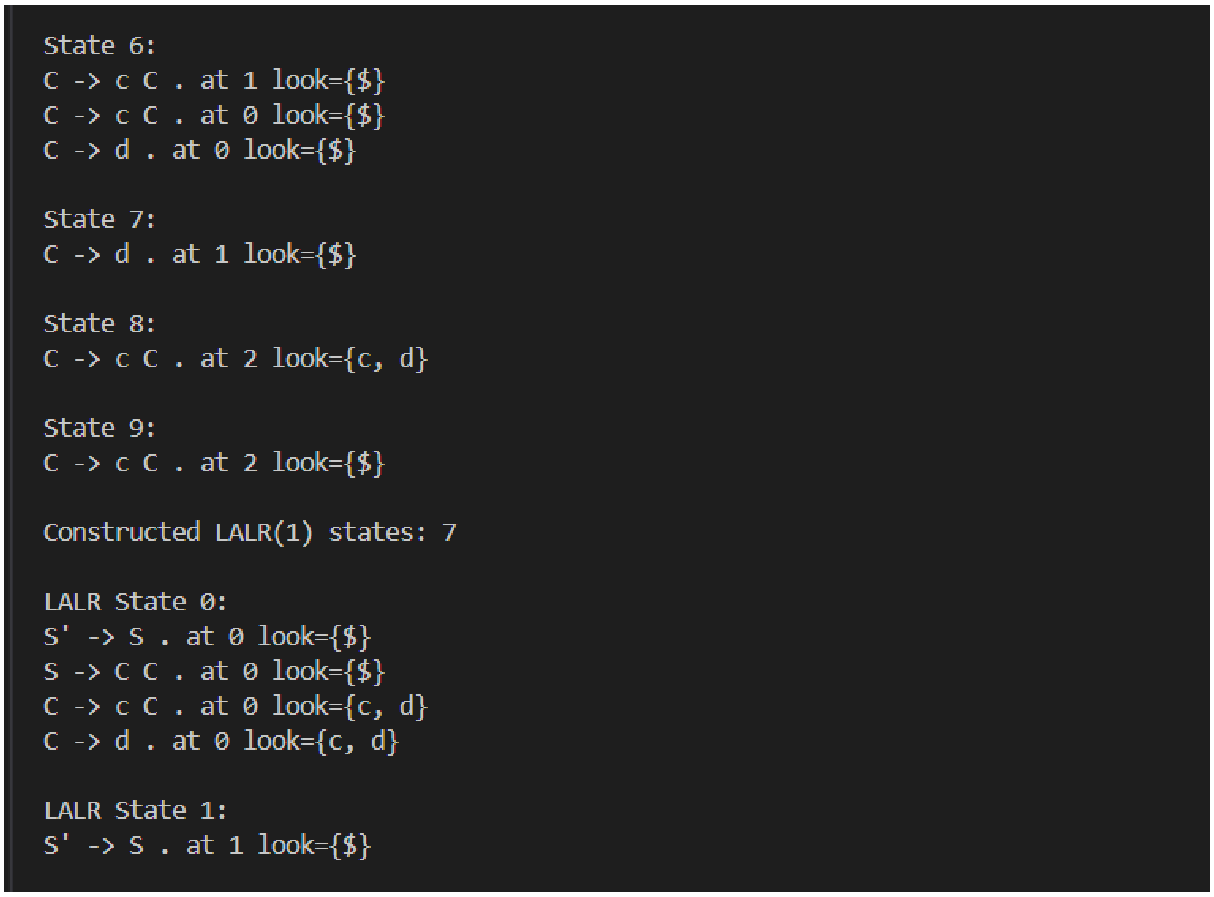
OUTPUT



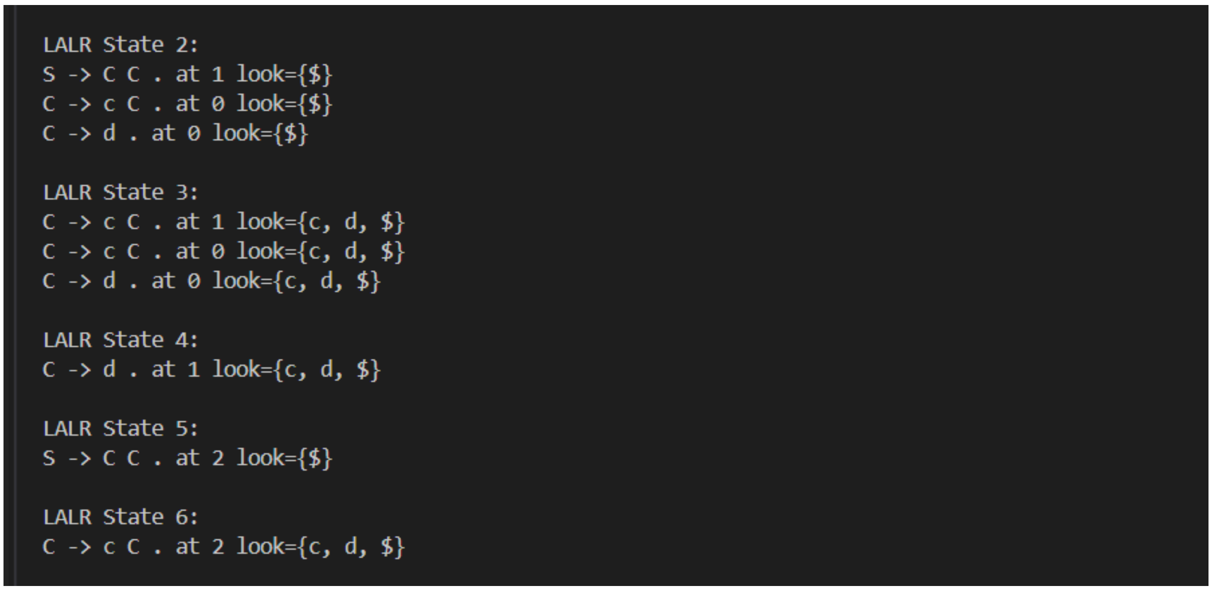
```
Constructed CLR(1) states: 10

State 0:
S' -> S . at 0 look={$}
S -> C C . at 0 look={$}
C -> c C . at 0 look={c, d}
C -> d . at 0 look={c, d}

State 1:
S' -> S . at 1 look={$}

State 2:
S -> C C . at 1 look={$}
C -> c C . at 0 look={$}
C -> d . at 0 look={$}

State 3:
C -> c C . at 1 look={c, d}
C -> c C . at 0 look={c, d}
C -> d . at 0 look={c, d}

State 4:
C -> d . at 1 look={c, d}

State 5:
S -> C C . at 2 look={$}
```

~Aditya
23BCE1344

LAB Assignment 3 PART 2



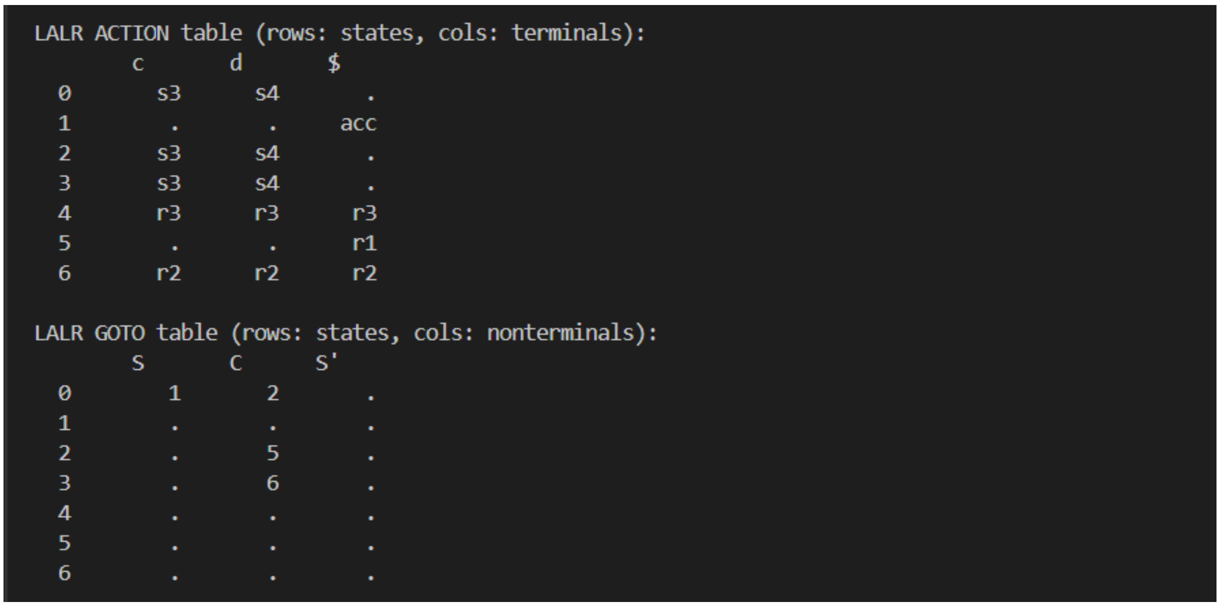
```
State 6:
C -> c C . at 1 look={$}
C -> c C . at 0 look={$}
C -> d . at 0 look={$}

State 7:
C -> d . at 1 look={$}

State 8:
C -> c C . at 2 look={c, d}

State 9:
C -> c C . at 2 look={$}

Constructed LALR(1) states: 7

LALR State 0:
S' -> S . at 0 look={$}
S -> C C . at 0 look={$}
C -> c C . at 0 look={c, d}
C -> d . at 0 look={c, d}

LALR State 1:
S' -> S . at 1 look={$}
```



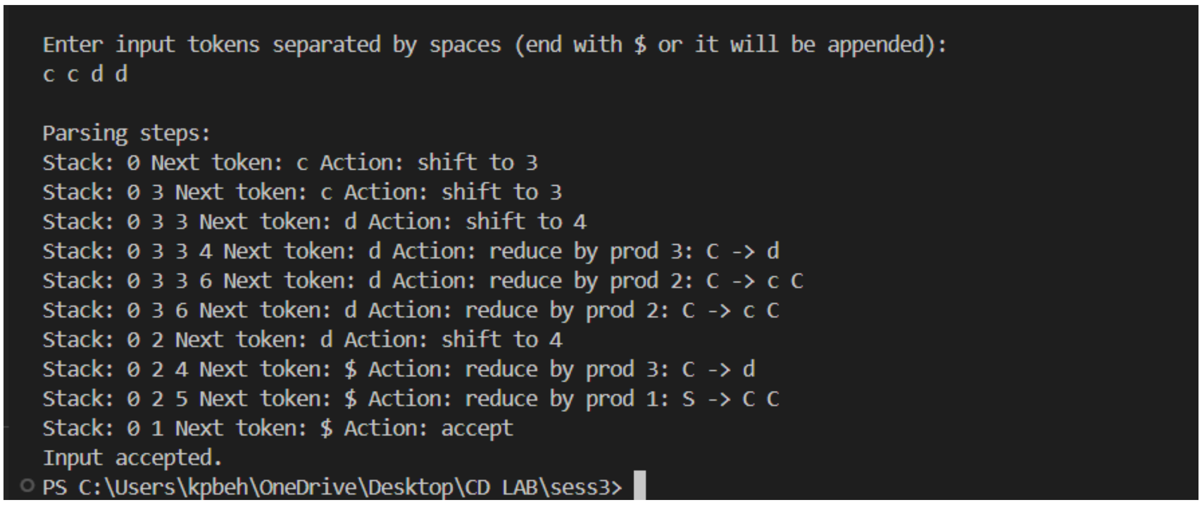
```
LALR State 2:
S -> C C . at 1 look={$}
C -> c C . at 0 look={$}
C -> d . at 0 look={$}

LALR State 3:
C -> c C . at 1 look={c, d, $}
C -> c C . at 0 look={c, d, $}
C -> d . at 0 look={c, d, $}

LALR State 4:
C -> d . at 1 look={c, d, $}

LALR State 5:
S -> C C . at 2 look={$}

LALR State 6:
C -> c C . at 2 look={c, d, $}
```

~Aditya
23BCE1344

LAB Assignment 3 PART 2



```
LALR ACTION table (rows: states, cols: terminals):
        c       d       $
  0      s3      s4       .
  1       .       .      acc
  2      s3      s4       .
  3      s3      s4       .
  4      r3      r3      r3
  5       .       .      r1
  6      r2      r2      r2

LALR GOTO table (rows: states, cols: nonterminals):
        S       C      S'
  0       1       2       .
  1       .       .       .
  2       .       5       .
  3       .       6       .
  4       .       .       .
  5       .       .       .
  6       .       .       .
```



```
Enter input tokens separated by spaces (end with $ or it will be appended):
c c d d

Parsing steps:
Stack: 0 Next token: c Action: shift to 3
Stack: 0 3 Next token: c Action: shift to 3
Stack: 0 3 3 Next token: d Action: shift to 4
Stack: 0 3 3 4 Next token: d Action: reduce by prod 3: C -> d
Stack: 0 3 3 6 Next token: d Action: reduce by prod 2: C -> c C
Stack: 0 3 6 Next token: d Action: reduce by prod 2: C -> c C
Stack: 0 2 Next token: d Action: shift to 4
Stack: 0 2 4 Next token: $ Action: reduce by prod 3: C -> d
Stack: 0 2 5 Next token: $ Action: reduce by prod 1: S -> C C
Stack: 0 1 Next token: $ Action: accept
Input accepted.
PS C:\Users\kpbeh\OneDrive\Desktop\CD LAB\sess3>
```

~Aditya
23BCE1344