**INSTITUTE OF ENGINEERING & MANAGEMENT**

**Department of Computer Science & Engineering**

|  |  |
| --- | --- |
| **Name** | **: Soaham Roy** |
| **Class Roll** | **: 107** |
| **Enrollment No.** | **: 12019002002143** |
| **Subject Name** | **: Compiler design lab** |
| **Subject code**  **Year**  **Semester**  **Section**  **Lab instructor** | **: PCCCS691**  **: 3rd**  **: 6**  **: B**  **: Prof. Bavrabi Ghosh** |

|  |  |  |
| --- | --- | --- |
| **Serial**  **No** | **Name of the experiment** | **Page No** |
| 1 | For the following grammar write a program to determine whether it is acceptable by LR (0) parser or SLR (1) parser  E --> T + E / T  T --> id | 3-36 |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

**NAME OF THE EXPERIMENT:**

For the following grammar write a program to determine whether it is acceptable by LR (0) parser or SLR (1) parser

E --> T + E / T

T --> id

**DATE OF THE EXPERIMENT:** 19-04-2022

**DATE OF SUBMISSION:** 26-04-2022

**NAME OF THE EXPERIMENT**:

For the following grammar write a program to determine whether it is acceptable by LR (0) parser or SLR (1) parser

E --> T + E / T

T --> id

**Algorithm:**

Step-1: Start.

Step-2: Include the basic header files for performing string manipulations, including stdlio.h, stdlib.h and string.h.

Step-3: Include the external header files created by us majorly being the closure\_goto.h, parsingtable.h and first\_follow.h.

Step-4: Within the main method call the start() function from closure\_goto.h header file, then call initialize\_first\_follow() function from the first\_follow.h header file.

Step-5: Then call the functions compute\_first() and compute\_follow() to calculate the first and follow of the input grammar non-terminals.

Step-6: Call the create\_parsing\_table() function present within the parsingtable.h header file to form the respective parsing table.

Step-7: Finally return 0; to end the main function.

Step-8: Within the closure\_goto.h header file, declare a function check() that returns 1 if the production strings encounter a character that is a terminal and returns 0 otherwise.

Step-9: Then declare the function generate\_terminals() that augments the productions rules to formulate the final set of rules to generate the terminals of the input productions.

Step-10: Then declare another function check2() to return 1 for non-terminals in input production rules and 0 otherwise.

Step-11: Then declare another function generate\_nonterminals() to generate the non-terminals from the input production rules similar to the ay we generated the terminals above.

Step-12: Declare function initialize\_items() to call the generate\_terminals() and generate\_nonterminals() functions and calculate the number of items in total.

Step-13: Declare function generate\_item() to generate the final production rules to iterate over to find the final results.

Step-14: Declare functions item\_found and isterminal to check for a particular item within a production rule and check whether a particular character is a terminal or not respectively.

Step-15: Declare a function closure to work with the canonical forms of the input production rules via working with the ‘.’ points added to form the corresponding item states.

Step-16: Declare functions Goto1(), state\_found and transition\_item\_found to mark the goto points during canonical analysis, checking for the states in the production rules along with the transition items to derive the final result.

Step-17: Declare the functions compute\_closure\_goto to compute the goto actions of the canonical structure for different item states along with print() to print the total number of states along with each item states derived from the input production rules.

Step-18: Finally call the start() function that takes the total number of productions as input from the user along with the respective production ules for the particular grammar. Then augmentation of rules take place and the above functions are called accordingly to derive the final parsing.

Step-19: Declare a flag variable flUse along with the struct to define a new user defined data type consisting of char type Action table and an int type Goto table with the struct being called as table.

Step-20: Initialize the action table with the function initialize\_table() and printing the actions of the action table with the print\_table() function.

Step-21: Declare functions Goto, get\_state and get\_pos to declare the pattern of the goto methodology, getting the state of the items and the positions of the terminals along with non-terminals from the production rules input by the user.

Step-22: Declare the function get\_production\_no to derive the rule number of the particular production from the table of the production rules.

Step-23: Declare the compute\_action function that fills the action table based on the production rules and the canonical item state division as input by the user in the production rules.

Step-24: Finally declare the function create\_parsing\_table to derive the final parsing table by calling the above declared functions accordingly and yielding appropriate messages to the user.

Step-25: Within the first\_follow.h header file declare the global int variable epsilon\_flag and call the function initialize\_first\_follow() to initialize the first and follows of the input grammar non-terminals.

Step-26: Declare the function add\_symbol to add new symbols to the first and follow arrays based on the particular criteria they satisfy.

Step-27: Declare the function first() that checks within the production rules for a particular character passed as input to the function via another function compute\_first() that iterates over the production rules and stores the first of the non-terminals within the array First.

Step-28: Declare the function follow() that checks within the production rules for a particular character passed as input to the function via another function compute\_follow() that iterates over the production rules and stores the follow of the non-terminals within the array Follow.

Step-29: Finally display the output to the user from the main based on the input grammar.

Step-30: STOP

**Source code:**

**Main C file:**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include "closure\_goto.h"

#include "parsingtable.h"

#include "first\_follow.h"

int main()

{

    start();

    initialize\_first\_follow();

    compute\_first();

    compute\_follow();

    create\_parsing\_table();

    return 0;

}

**closure\_goto.h header file:**

// Variables used in most of the other modules.

char items[30][100][100];

char augmented\_grammar[100][100], terminals[10], nonterminals[10];

int no\_of\_productions = 0, no\_of\_states = 0, no\_of\_items[30], no\_of\_terminals = 0, no\_of\_nonterminals = 0;

char FIRST[2][10][10];

char FOLLOW[10][10];

// Variables used only in this module.

int state\_index = 0, goto\_state\_index = 0, closure\_item\_index = 0;

int check(char c)

{

    int i;

    for (i = 0; i < no\_of\_terminals; i++)

        if (terminals[i] == c)

            return 1;

    return 0;

}

void generate\_terminals()

{

    int i, j;

    int index = 0;

    for (i = 0; i < no\_of\_productions; i++)

    {

        for (j = 0; augmented\_grammar[i][j] != '>'; j++)

            ;

        j++;

        for (; augmented\_grammar[i][j] != '\0'; j++)

        {

            if (augmented\_grammar[i][j] < 65 || augmented\_grammar[i][j] > 90)

            {

                if (!check(augmented\_grammar[i][j]))

                {

                    terminals[index] = augmented\_grammar[i][j];

                    no\_of\_terminals++;

                    index++;

                }

            }

        }

    }

    terminals[index] = '$';

    no\_of\_terminals++;

    index++;

    terminals[index] = '\0';

}

int check2(char c, int index)

{

    int i;

    for (i = 0; i < index; i++)

        if (nonterminals[i] == c)

            return 1;

    return 0;

}

void generate\_nonterminals()

{

    int i, index = 0;

    for (i = 0; i < no\_of\_productions; i++)

        if (!check2(augmented\_grammar[i][0], index))

        {

            nonterminals[index] = augmented\_grammar[i][0];

            index++;

        }

    no\_of\_nonterminals = index;

    nonterminals[index] = '\0';

}

void initialize\_items()

{

    generate\_terminals();

    generate\_nonterminals();

    int i;

    for (i = 0; i < 30; i++)

        no\_of\_items[i] = 0;

}

void generate\_item(char \*s, char \*t)

{

    int i;

    for (i = 0; i < 3; i++)

        t[i] = s[i];

    t[i] = '.';

    if (s[i] != '@')

        for (; i < strlen(s); i++)

            t[i + 1] = s[i];

    t[i + 1] = '\0';

}

int item\_found(char \*s)

{ // Check for items in a state.

    int i;

    for (i = 0; i < closure\_item\_index; i++)

    {

        if (!strcmp(s, items[state\_index][i])) // If the strings match.

            return 1;

    }

    return 0;

}

int isterminal(char s)

{

    int i;

    for (i = 0; i < no\_of\_terminals; i++)

        if (s == terminals[i])

            return 1;

    return 0;

}

void closure(char \*s)

{

    int i, j;

    for (i = 0; s[i] != '.'; i++)

        ;

    i++;

    if (!item\_found(s))

    {

        strcpy(items[state\_index][closure\_item\_index], s);

        closure\_item\_index++;

        //      printf("%s\n", items[state\_index][closure\_item\_index-1]);

    }

    if (s[i] == s[0] && s[i - 2] == '>') // To avoid infinite loop due to left recursion.

        return;

    if (isterminal(s[i]))

        return;

    else

    { // Not a terminal

        for (j = 0; j < no\_of\_productions; j++)

        {

            char temp[100];

            if (augmented\_grammar[j][0] == s[i])

            {

                generate\_item(augmented\_grammar[j], temp);

                closure(temp);

            }

        }

    }

}

int Goto1(char s, char temp[][100])

{ // Find Goto on symbol s. GOTO(goto\_state\_index, s)

    int i, j;

    int n = 0;

    char t, temp2[100];

    if (s == '\0')

    {

        return n;

    }

    for (i = 0; i < no\_of\_items[goto\_state\_index]; i++)

    {

        strcpy(temp2, items[goto\_state\_index][i]);

        for (j = 0; temp2[j] != '.'; j++)

            ;

        if (temp2[j + 1] == '\0')

            continue;

        if (temp2[j + 1] == s)

        {

            t = temp2[j];

            temp2[j] = temp2[j + 1];

            temp2[j + 1] = t;

            strcpy(temp[n], temp2);

            n++;

        }

    }

    return n;

}

int state\_found(char \*s)

{ // Checks for existance of same state.

    int i;

    for (i = 0; i < state\_index; i++)

    {

        if (!strcmp(s, items[i][0])) // Compare with the first item of each state.

            return 1;

    }

    return 0;

}

int transition\_item\_found(char \*t\_items, char s, int t\_index)

{

    int i;

    for (i = 0; i < t\_index; i++)

        if (s == t\_items[i])

            return 1;

    return 0;

}

void compute\_closure\_goto()

{

    char temp[100][100], transition\_items[100];

    int i, no\_of\_goto\_items, j, transition\_index = 0;

    generate\_item(augmented\_grammar[0], temp[0]);

    closure(temp[0]);

    no\_of\_items[state\_index] = closure\_item\_index;

    closure\_item\_index = 0;

    state\_index++;

    // state\_index is 1 now.

    while (goto\_state\_index < 30)

    {

        transition\_index = 0;

        transition\_items[transition\_index] = '\0';

        for (i = 0; i < no\_of\_items[goto\_state\_index]; i++)

        {

            for (j = 0; items[goto\_state\_index][i][j] != '.'; j++)

                ;

            j++;

            if (!transition\_item\_found(transition\_items, items[goto\_state\_index][i][j], transition\_index))

            {

                transition\_items[transition\_index] = items[goto\_state\_index][i][j];

                transition\_index++;

            }

        }

        transition\_items[transition\_index] = '\0';

        for (i = 0; i < transition\_index; i++)

        {

            int add\_flag = 0;

            no\_of\_goto\_items = Goto1(transition\_items[i], temp);

            for (j = 0; j < no\_of\_goto\_items; j++)

            {

                if (!state\_found(temp[j]))

                {

                    add\_flag = 1;

                    closure(temp[j]);

                }

                else

                    break;

            }

            if (add\_flag)

            {

                no\_of\_items[state\_index] = closure\_item\_index;

                closure\_item\_index = 0;

                state\_index++;

            }

        }

        goto\_state\_index++;

    }

    no\_of\_states = state\_index;

}

void print()

{

    int i, j;

    printf("\nNumber of states = %d.\n", no\_of\_states);

    for (i = 0; i < no\_of\_states; i++)

    {

        printf("\n\nItems in State %d...\n\n", i);

        for (j = 0; j < no\_of\_items[i]; j++)

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

    }

}

void start()

{

    char str[100];

    printf("Enter number of productions : ");

    scanf("%d", &no\_of\_productions);

    printf("Enter the productions : \n");

    int i;

    for (i = 1; i <= no\_of\_productions; i++)

        scanf("%s", augmented\_grammar[i]);

    printf("\n\nAugmented Grammar is...\n\n");

    strcpy(augmented\_grammar[0], "Z->");

    str[0] = augmented\_grammar[1][0];

    str[1] = '\0';

    strcat(augmented\_grammar[0], str);

    no\_of\_productions++;

    for (i = 0; i < no\_of\_productions; i++)

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

    initialize\_items();

    compute\_closure\_goto();

    print();

}

**parsingtable.h header file:**

// Parsing Table.

int flUse = -1;

struct Parsing\_Table

{ // Structure to represent the Parsing Table.

    char ACTION[30][100][100];

    int GOTO[30][100];

} table;

void initialize\_table()

{ // Initialize all entries to indicate Error.

    int i, j;

    for (i = 0; i < no\_of\_states; i++)

    {

        for (j = 0; j < no\_of\_terminals; j++)

            strcpy(table.ACTION[i][j], "e");

        for (j = 0; j < no\_of\_nonterminals; j++)

            table.GOTO[i][j] = -1;

    }

}

void print\_table()

{

    int i, j;

    printf("%10s   ", "");

    for (i = 0; i < no\_of\_terminals; i++)

        printf("%10c", terminals[i]);

    printf(" | ");

    for (i = 1; i < no\_of\_nonterminals; i++)

        printf("%10c", nonterminals[i]);

    printf("\n\n");

    for (i = 0; i < no\_of\_states; i++)

    {

        printf("%10d | ", i);

        for (j = 0; j < no\_of\_terminals; j++)

        {

            if (!strcmp(table.ACTION[i][j], "e"))

                printf("%10s", ".");

            else

                printf("%10s", table.ACTION[i][j]);

        }

        printf(" | ");

        for (j = 1; j < no\_of\_nonterminals; j++)

        {

            if (table.GOTO[i][j] == -1)

                printf("%10s", ".");

            else

                printf("%10d", table.GOTO[i][j]);

        }

        printf("\n");

    }

}

void Goto(int i, int item, char \*temp)

{ // Computes goto for 'item'th item of 'i'th state.

    char t;

    strcpy(temp, items[i][item]);

    for (i = 0; temp[i] != '\0'; i++)

        if (temp[i] == '.')

        {

            t = temp[i];

            temp[i] = temp[i + 1];

            temp[i + 1] = t;

            break;

        }

}

int get\_state(char \*t, int state)

{ // Returns the state of a given item.

    int i, j;

    for (i = state; i < (no\_of\_states + state); i++)

    { // Start searching from current state and then wrap around.

        for (j = 0; j < no\_of\_items[i % no\_of\_states]; j++)

        {

            if (!strcmp(t, items[i % no\_of\_states][j]))

                return i % no\_of\_states;

        }

    }

    printf("No match for string! (%s)\n", t);

}

int get\_pos(int flag, char symbol)

{ // Returns index of a terminal or a non-terminal from the corresponding arrays.

    int i;

    if (flag == 0)

        for (i = 0; i < no\_of\_terminals; i++)

        {

            if (terminals[i] == symbol)

                return i;

        }

    else

        for (i = 0; i < no\_of\_nonterminals; i++)

        {

            if (nonterminals[i] == symbol)

                return i;

        }

    if (flag == 0)

        printf("Terminal not found in get\_pos! (%c)\n", symbol);

    else

        printf("Non-terminal not found in get\_pos! (%c)\n", symbol);

}

int get\_production\_no(char \*item)

{ // Given an item, it returns the production number of the equivalent production.

    int i, j;

    char production[20];

    for (i = 0, j = 0; item[i] != '\0'; i++)

        if (item[i] != '.')

        {

            production[j] = item[i];

            j++;

        }

    if (j == 3)

    { // If it's an epsilon production, the production won't have a body.

        production[j] = '@';

        j++;

    }

    production[j] = '\0';

    for (i = 0; i < no\_of\_productions; i++)

    {

        if (!strcmp(production, augmented\_grammar[i]))

            return i;

    }

    printf("Production not found! (%s)\n", production);

}

void compute\_action()

{

    int i, item, j;

    char temp[100], symbol;

    for (i = 0; i < no\_of\_states; i++)

    {

        for (item = 0; item < no\_of\_items[i]; item++)

        {

            char \*s = strchr(items[i][item], '.'); // Returns a substring starting with '.'

            if (!s)

            { // In case of error.

                printf("Item not found! State = %d, Item = %d\n", i, item);

                exit(-1);

            }

            if (strlen(s) > 1)

            { // dot is not at end of string. SHIFT ACTION!!

                if (isterminal(s[1]))

                { // For terminals. Rule 1.

                    if (!strcmp(table.ACTION[i][get\_pos(0, s[1])], "e"))

                    { // Multiple entries conflict.

                        // printf("\n\nConflict(1): Multiple entries found for (%d, %c)\n", i, s[1]);

                        // printf("\nGrammar is not in LR(0)!\n");

                        flUse = 1;

                    }

                    char state[3];

                    Goto(i, item, temp); // Store item in temp.

                    j = get\_state(temp, i);

                    sprintf(state, "%d", j);

                    strcpy(temp, "S:");

                    strcat(temp, state);

                    strcpy(table.ACTION[i][get\_pos(0, s[1])], temp);

                }

                else

                { // For non-terminals. Rule 4.

                    Goto(i, item, temp); // Store item in temp.

                    j = get\_state(temp, i);

                    if (table.GOTO[i][get\_pos(1, s[1])] == -1) // To avoid multiple entries.

                        table.GOTO[i][get\_pos(1, s[1])] = j;

                }

            }

            else

            { // dot is at end of string. Rule 2. REDUCE ACTION!!

                char f[10], production\_no[3];

                int k, n;

                n = get\_production\_no(items[i][item]); // Get production number from Augmented Grammar.

                sprintf(production\_no, "%d", n);

                strcpy(temp, "R:");

                strcat(temp, production\_no);

                strcpy(f, FOLLOW[get\_pos(1, items[i][item][0])]); // Get follow of production head.

                for (k = 0; f[k] != '\0'; k++)

                {

                    if (!strcmp(table.ACTION[i][get\_pos(0, f[k])], "e"))

                    { // Multiple entries conflict.

                        // printf("\n\nConflict(3): Multiple entries found for (%d, %c)\n", i, f[k]);

                        // printf("\nGrammar is not in LR(0)!\n");

                        flUse = 1;

                    }

                    strcpy(table.ACTION[i][get\_pos(0, f[k])], temp);

                }

            }

        }

    }

    strcpy(table.ACTION[1][get\_pos(0, '$')], "acc"); // Accept-entry for item [S'->S.]

}

void create\_parsing\_table()

{

    initialize\_table();

    compute\_action();

    if (flUse == 1)

    {

        printf("\n\nConflict(3): Multiple entries found!!\n");

        printf("\nGrammar is not accepted by LR(0)!\n");

        printf("\n\nGrammar is now checked for SLR(1):\n");

        printf("\nThe Parsing Table for the given grammar for SLR(1) parser is...\n\n");

    }

    else

    {

        printf("\nSince Grammar is accepted by LR(0), it is also accepted by SLR(1) : \n");

        printf("\nThe Parsing Table for the given grammar is...\n\n");

        printf("\nThe Parsing Table for the given grammar for both LR(0) and SLR(1) parsers is...\n\n");

    }

    print\_table();

    printf("\n\n");

}

**first\_follow.h header file:**

int epsilon\_flag = 0;

void initialize\_first\_follow()

{ // Initialize to null strings.

    int i;

    for (i = 0; i < no\_of\_terminals; i++)

        FIRST[0][i][0] = '\0';

    for (i = 0; i < no\_of\_nonterminals; i++)

    {

        FIRST[1][i][0] = '\0';

        FOLLOW[i][0] = '\0';

    }

}

void add\_symbol(int flag, char \*f, char \*s)

{ // Adds a symbol to FIRST or FOLLOW if it doesn't already exist in it.

    int i, j;

    int found;

    if (flag == 0)

    { // For FIRST.

        for (i = 0; i < strlen(s); i++)

        {

            found = 0;

            for (j = 0; j < strlen(f); j++)

            {

                if (s[i] == f[j])

                    found = 1;

            }

            if (!found)

            {

                char temp[2];

                temp[0] = s[i];

                temp[1] = '\0';

                strcat(f, temp);

            }

        }

    }

    else

    { // For FOLLOW.

        for (i = 0; i < strlen(s); i++)

        {

            found = 0;

            if (s[i] == '@')

            {

                epsilon\_flag = 1;

                continue;

            }

            for (j = 0; j < strlen(f); j++)

            {

                if (s[i] == f[j])

                    found = 1;

            }

            if (!found)

            {

                char temp[2];

                temp[0] = s[i];

                temp[1] = '\0';

                strcat(f, temp);

            }

        }

    }

}

void first(char s)

{

    if (isterminal(s))

    { // For terminals.

        FIRST[0][get\_pos(0, s)][0] = s;

        FIRST[0][get\_pos(0, s)][1] = '\0';

    }

    else

    { // For non-terminals.

        int i, flag = 0;

        for (i = 0; i < no\_of\_productions; i++)

        {

            if (augmented\_grammar[i][0] == s)

            { // Productions with head as s.

                int j;

                for (j = 0; augmented\_grammar[i][j] != '>'; j++)

                    ;

                j++;

                char next\_sym = augmented\_grammar[i][j];

                if (next\_sym == '@')

                { // Epsilon Production.

                    add\_symbol(0, FIRST[1][get\_pos(1, s)], "@");

                    flag = 1;

                }

                else

                {

                    if (next\_sym == s)

                    { // In case of left recursion, to avoid infinite loop.

                        if (flag)

                            next\_sym = augmented\_grammar[i][++j];

                        else

                            continue;

                    }

                    first(next\_sym); // Recursive call, to find FIRST of next symbol.

                    if (isterminal(next\_sym)) // Add first of next symbol to first of current symbol.

                        add\_symbol(0, FIRST[1][get\_pos(1, s)], FIRST[0][get\_pos(0, next\_sym)]);

                    else

                        add\_symbol(0, FIRST[1][get\_pos(1, s)], FIRST[1][get\_pos(1, next\_sym)]);

                }

            }

        }

    }

}

void compute\_first()

{

    int i;

    for (i = 0; i < no\_of\_terminals; i++)

        first(terminals[i]);

    for (i = 0; i < no\_of\_nonterminals; i++)

        first(nonterminals[i]);

    // for (i = 0; i < no\_of\_nonterminals; i++)

    //  printf("%s\n", FIRST[get\_pos(1, nonterminals[i])]);

}

// FOLLOW

void follow(char s)

{

    if (s == nonterminals[0])

        add\_symbol(1, FOLLOW[0], "$");

    else if (s == nonterminals[1])

        add\_symbol(1, FOLLOW[1], "$");

    else

    {

        int i, j;

        for (i = 0; i < no\_of\_productions; i++)

        {

            for (j = 3; j < strlen(augmented\_grammar[i]); j++)

            {

                epsilon\_flag = 0;

                if (augmented\_grammar[i][j] == s)

                {

                    char next\_sym = augmented\_grammar[i][j + 1];

                    if (next\_sym != '\0')

                    {                             // If current symbol is not the last symbol of production body.

                        if (isterminal(next\_sym)) // For terminals.

                            add\_symbol(1, FOLLOW[get\_pos(1, s)], FIRST[0][get\_pos(0, next\_sym)]);

                        else

                        { // For non-terminals.

                            add\_symbol(1, FOLLOW[get\_pos(1, s)], FIRST[1][get\_pos(1, next\_sym)]);

                            if (epsilon\_flag)

                            { // If FIRST[next\_sym] has epsilon, find FOLLOW[next\_sym].

                                follow(next\_sym);

                                add\_symbol(1, FOLLOW[get\_pos(1, s)], FOLLOW[get\_pos(1, next\_sym)]);

                            }

                        }

                    }

                    else

                    {                                    // If current symbol is the last symbol of production body.

                        follow(augmented\_grammar[i][0]); // Follow of production head.

                        add\_symbol(1, FOLLOW[get\_pos(1, s)], FOLLOW[get\_pos(1, augmented\_grammar[i][0])]);

                    }

                }

            }

        }

    }

}

void compute\_follow()

{

    int i;

    for (i = 0; i < no\_of\_nonterminals; i++)

        follow(nonterminals[i]);

    // for (i = 0; i < no\_of\_nonterminals; i++)

    //  printf("%s\n", FOLLOW[get\_pos(1, nonterminals[i])]);

}

**Output:**



