Assignment 6(date: 20/3/2018)

Submitted By: LIPSA CHHOTRAY, Roll no: 115cs0203

A6.1 Removal of left recursion and application of left factoring.

Program:

#include<stdio.h>

#include<string.h>

char input[100],\*l,\*r,\*temp,tempprod[20],productions[25][50];

void left\_factoring(char \*input)

{

char a1[50],a2[50],a3[50],a4[50],a5[50];

int i,j=0,k,l,flag=0;

char \*az=strtok(input,"->");

char \*a=strtok(NULL,"->");

for(i=0;a[i]!='|';i++,j++)

a1[j]=a[i];

a1[j]='\0';

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

a2[i]=a[j];

a2[i]='\0';

k=0;

l=0;

for(i=0;i<strlen(a1)||i<strlen(a2);i++)

{

if(a1[i]==a2[i] && flag!=1)

{

a3[k]=a1[i];

k++;

}

else

{

break;

/\*flag=1;

a4[l]=a1[i];

a5[l]=a2[i];

l++;\*/

}

}

a3[k]='X';

a3[k+1]='\0';

while(i<strlen(a1))

{

a4[l]=a1[i];

i++;

l++;

}

a4[l]='|';

a4[l+1]='\0';

l=0;

while(k<strlen(a2))

{

a5[l]=a2[k];

k++;

l++;

}

a5[l]='\0';

strcat(a4,a5);

printf("\n %s->%s",az,a3);

printf("\n X->%s\n",a4);

}

void remove\_left\_recursion()

{

int i=0,j=0,flag=0;

l = strtok(input,"->");//strtok() split the string by delimiter -> for eg

r = strtok(NULL,"->");

temp = strtok(r,"|");

while(temp) {

if(temp[0] == l[0]) {

flag = 1;

sprintf(productions[i++],"%s'->%s%s'\0",l,temp+1,l);//sprintf()

}

else

sprintf(productions[i++],"%s->%s%s'\0",l,temp,l);

temp = strtok(NULL,"|");

}

sprintf(productions[i++],"%s'->$",l);

if(flag == 0)

printf("The given productions don't have Left Recursion");

else

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

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

}

printf("\n");

productions[j+1][0]='\0';

}

main()

{

printf("Enter the productions: ");

scanf("%s",input);

remove\_left\_recursion();

int i=0;

char \*c,\*d,\*l,\*r,\*temp;

int flag=0,flag1=0;

while(1)

{

if(productions[i][0]=='\0')

{

break;

}

else

{

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

if(productions[i][1]!='-')

{

// for A'

if(flag==0)

{

c=strcat(productions[i],"|");

flag=1;

}

else

{

l=strtok(productions[i],"->");

r=strtok(NULL,"->");

c=strcat(c,r);

}

}

else

{

// for A

if(flag1==0)

{

d=strcat(productions[i],"|");

flag1=1;

}

else

{

l=strtok(productions[i],"->");

r=strtok(NULL,"->");

d=strcat(d,r);

}

}

}

i++;

}

printf("\nproductions after removal of left recursion\n");

printf("%s \n %s\n \n",c,d);

printf("\nproductions after removal of left factoring\n");

left\_factoring(c);

left\_factoring(d);

/\*while(1)

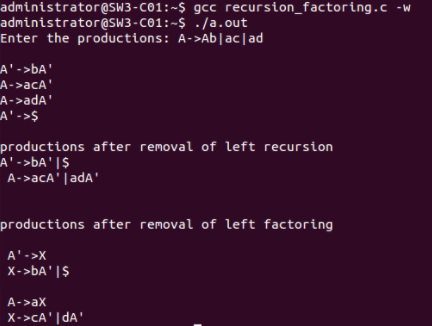
{

}

\*/

}

Output:



A6.2 Implementation of first and follow methods.

Program:

/\*\*first and follow operation\*\*/

#include<iostream>

#include<string.h>

#include<stdlib.h>

using namespace std;

int n;

int value=0;

void follow(char a[][10],char ch);

int first(char a[][10],char ch)

{

int j=3;

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

{

if(islower(ch))

{

cout<<ch<<"\t";

return 0;

}

if(a[i][0]==ch){

if(a[i][j]=='@')

{

//epsilon condition

if(value==0)

{

// cout<<"@"<<"\t";

follow(a,ch);

return j++;

}

else

{

return j++;

}

}

else if(!isupper(a[i][j]))

{

cout<<a[i][j]<<"\t";

}

else

{

value=1;

first(a,a[i][j]);

j++;

while(value!=0)

{

value=first(a,a[i][j]);

j++;

}

return 0;

}

}

}

return 0;

}

void follow(char a[][10],char ch)

{

if(ch=='S')

{

cout<<"$\t"

}

for(int j=0;j<n;j++)

{

for(int i=0;i<strlen(a[j]);i++)

{

if(a[j][i]==ch)

{

if(i==strlen(a[j])-1)

{

follow(a,a[j][0]);

return;

}

first(a,a[j][i+1]);

}

}

}

}

main()

{

char ch;

cout<<"enter the number of productions\n";

cin>>n;

char a[n][10];

cout<<"enter the productions\n";

cout<<"NOTE:enter epsilon productions of each variable if any after the non epsilon production\nfor epsilon enter :@\nthe starting variable should be S\n";

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

{

scanf("%s",&a[i]);

}

/\*first\*/

cout<<"\nenter the variable whose first is to be found out\n";

cin>>ch;

cout<<"\nfirst of "<<ch<<" is: \n"<<endl;

first(a,ch);

while(1)

{

char c;

cout<<"\nenter y to continue \n n to terminate\n";

cin>>c;

if(c=='y')

{

cout<<"\nenter the variable whose first is to be found out\n";

cin>>ch;

cout<<"\nfirst of "<<ch<<" is: \n"<<endl;

first(a,ch);

}

else

{

break;

}

}

/\*follow\*/

cout<<"\nenter the variable whose follow is to be found out\n";

cin>>ch;

cout<<"\nfollow of "<<ch<<" is: \n"<<endl;

follow(a,ch);

while(1)

{

char c;

cout<<"\nenter y to continue \n n to terminate\n";

cin>>c;

if(c=='y')

{

cout<<"\nenter the variable whose follow is to be found out\n";

cin>>ch;

cout<<"\n follow of "<<ch<<" is: \n"<<endl;

follow(a,ch);

}

else

{

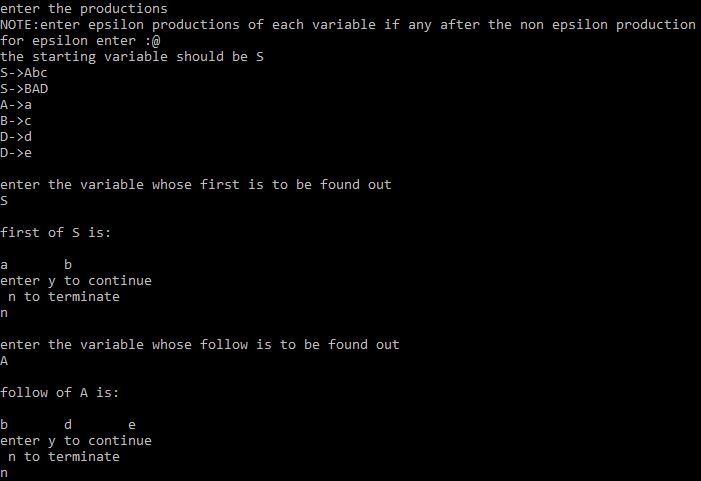
break;

}

}

}

Output:



A6.3 Implementation of lr0 algorithm.

Problem:

Parsingtable.h

//Parsing Table.

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

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");

exit(-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");

exit(-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();

print\_table();

}

//End of Parsing Table.

---------------------------------------------------------------------------

Parser.c

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#include"closure\_goto.h"

#include"parsingtable.h"

#include"first\_follow.h"

#include"parse.h"

int main() {

start(); //Compute closure and goto.

initialize\_first\_follow();

compute\_first();

compute\_follow();

create\_parsing\_table();

parse(); //Parse the input string.

return 0;

}

---------------------------------------------------------------------------

Parse.h

struct Stack { //Holds states.

int states[100];

int top;

} stack;

void push(int a) {

stack.top++;

stack.states[stack.top] = a;

}

void pop() {

int a = stack.states[stack.top];

stack.top--;

}

int get\_top() { //Returns top of stack state.

return stack.states[stack.top];

}

void initialize\_stack() { //Initialize stack to have state 0 on top.

stack.top = -1;

push(0);

}

int get\_int(char \*s) { //Get integer part of the strings found in table entries.

int i, j;

char temp[10];

for(i = 0; s[i] != ':'; i++);

i++;

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

temp[i-j] = s[i];

temp[i-j] = '\0';

return atoi(temp);

}

int get\_length(char \*production) { //Returns length of string in the production body.

int i, j;

for(i = 0; production[i] != '>'; i++);

i++;

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

return j;

}

//Start of functions meant only for displaying the result. (Doesn't affect the actual string parsing)

void get\_stack\_contents(char \*t) { //Stores stack contents in t.

int i;

char c[5];

strcpy(t, "$");

for(i = 0; i <= stack.top; i++) {

int n = stack.states[i];

sprintf(c, "%d", n);

strcat(t, c);

}

}

void get\_remaining\_input(char \*string, int index, char \*t) { //Stores remaining Input string in t.

int i, j;

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

t[j] = string[i];

t[j] = '\0';

}

void print\_contents(char \*string, int index, char \*matched\_string) { //Prints the required stuff.

char t1[20], t2[20];

get\_stack\_contents(t1);

get\_remaining\_input(string, index, t2);

printf("\t| %-25s | %-25s | %25s | \t", t1, matched\_string, t2);

}

//End of functions meant only for displaying the result.

void parse() {

char string[100];

char matched\_string[100];

initialize\_stack();

printf("\nEnter a string: ");

scanf("%s", string);

strcat(string, "$"); //Appending $ to end of input string.

matched\_string[0] = '\0';

printf("\nThe reduction steps for the given string are as follows...\n\n");

printf("\t| %-25s | %-25s | %25s | \t%-30s\n\n", "Stack", "Matched String", "Input String", "Action");

int index = 0, m\_index = 0;

while(1) {

char a = string[index];

print\_contents(string, index, matched\_string);

if(table.ACTION[get\_top()][get\_pos(0, a)][0] == 'S') { //Shift Action. (Table entry starts with char 'S')

int t = get\_int(table.ACTION[get\_top()][get\_pos(0, a)]);

push(t); //Push state t onto stack.

index++;

//Printing the result.

char t1[20];

char state[5];

strcpy(t1, "Shift ");

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

strcat(t1, state);

matched\_string[m\_index++] = a;

matched\_string[m\_index] = '\0';

printf("%-30s\n", t1);

}

else if(table.ACTION[get\_top()][get\_pos(0, a)][0] == 'R') { //Reduce Action.

int i, j = get\_int(table.ACTION[get\_top()][get\_pos(0, a)]);

for(i = 0; i < get\_length(augmented\_grammar[j]); i++) //Pop "length of string" times, w.r.t production 'j'.

pop();

int t = get\_top();

char A = augmented\_grammar[j][0]; //Production head of 'j'th production. (non-terminal)

push(table.GOTO[t][get\_pos(1, A)]); //Push state using GOTO of the table.

//Printing the result.

m\_index -= get\_length(augmented\_grammar[j]);

matched\_string[m\_index++] = A;

matched\_string[m\_index] = '\0';

char t1[20];

strcpy(t1, "Reduce by ");

strcat(t1, augmented\_grammar[j]);

printf("%-30s\n", t1);

}

else if(table.ACTION[get\_top()][get\_pos(0, a)][0] == 'a') { //Acceptance.

printf("%-30s\n", "Accept!!");

break;

}

else { //Error.

printf("%-30s\n", "Error!!\n\n");

printf("String doesn't belong to the language of the particular grammar!\n");

exit(0);

}

}

printf("\nString accepted!\n");

}

First\_follow.h

int epsilon\_flag = 0;

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[1][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], "$");

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])]);

}

}

}

}

}

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])]);

}

--------------------------------------------------------------------------------------------------------------------------------------

Closure\_goto.h

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]);

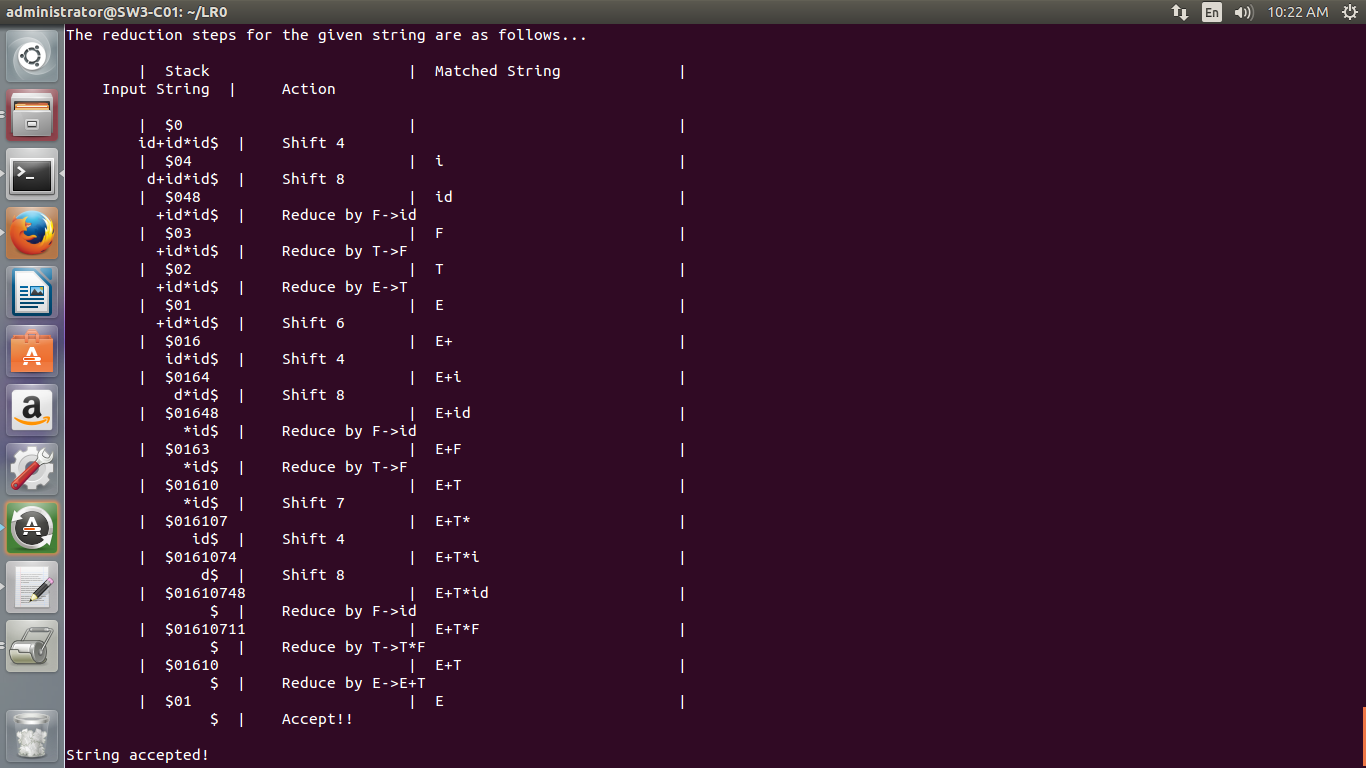
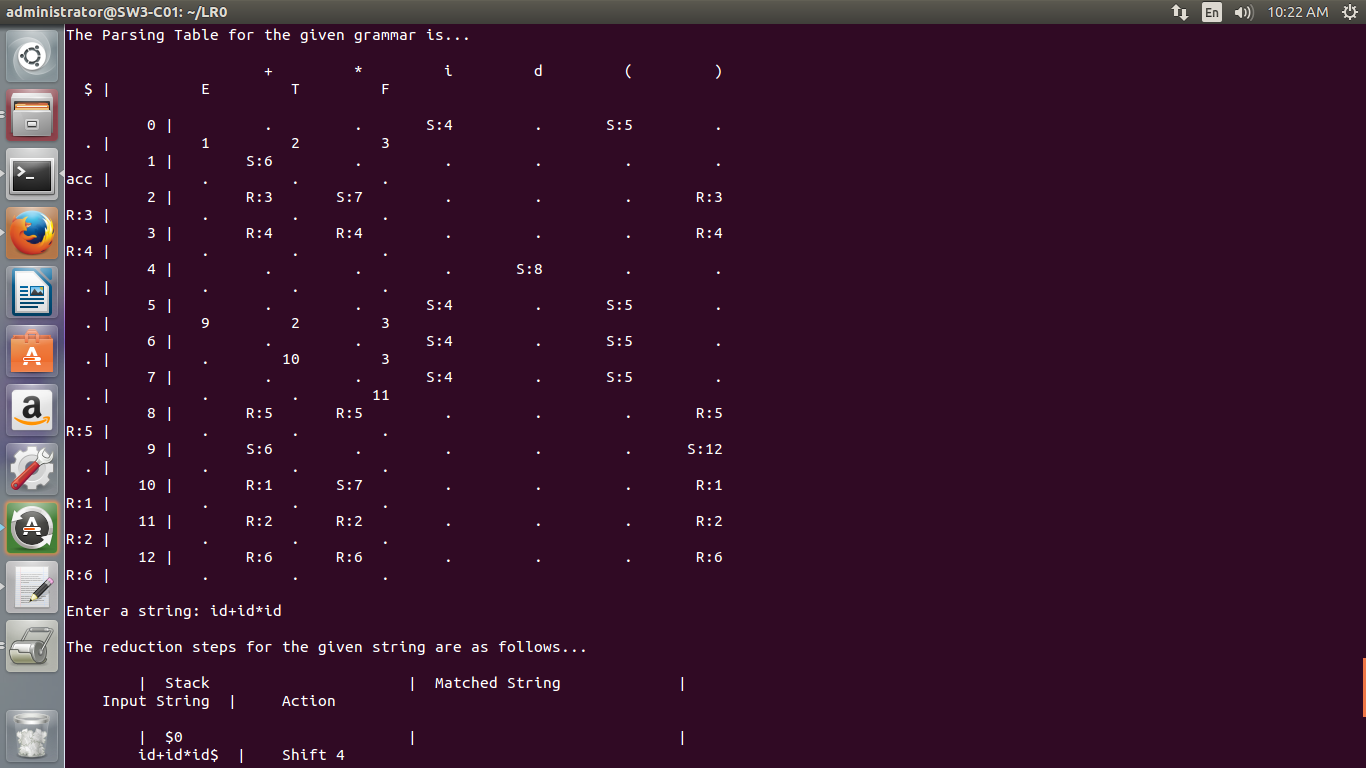
initialize\_items();

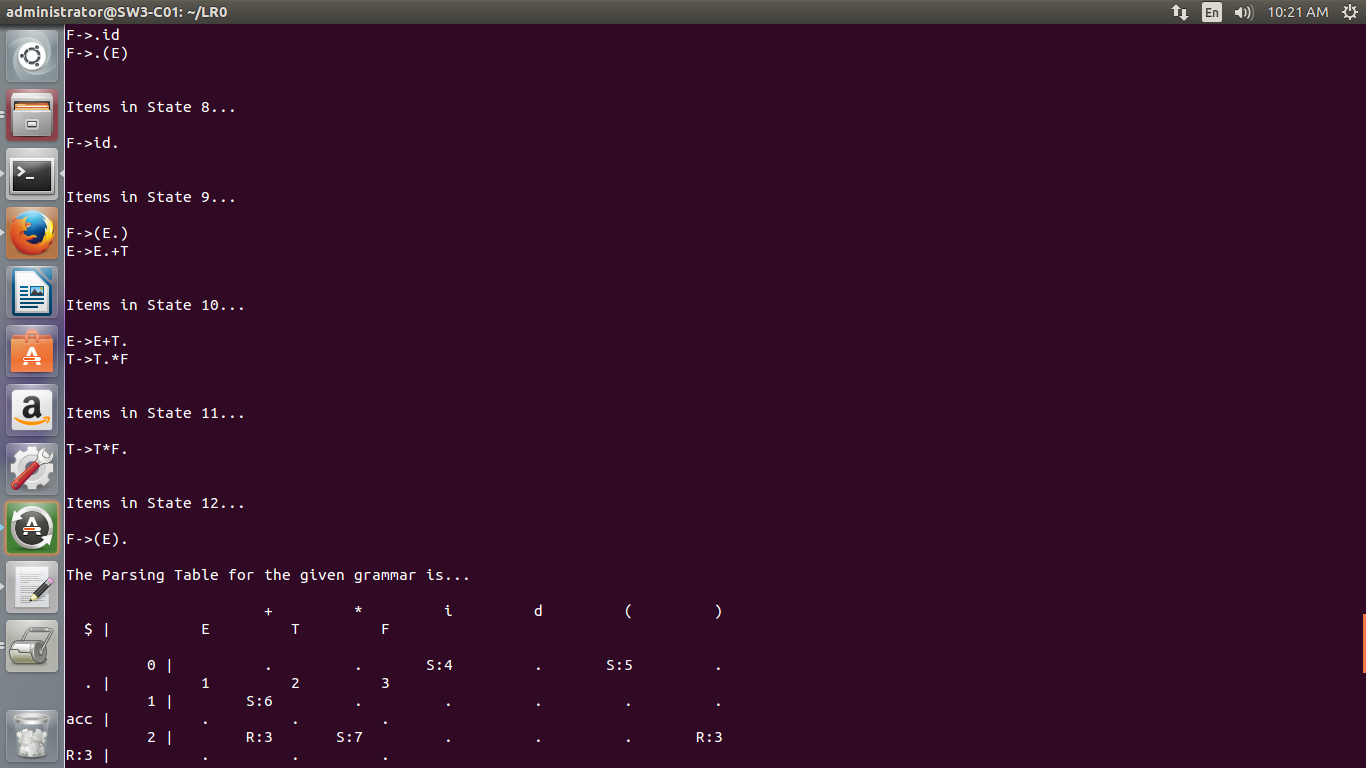
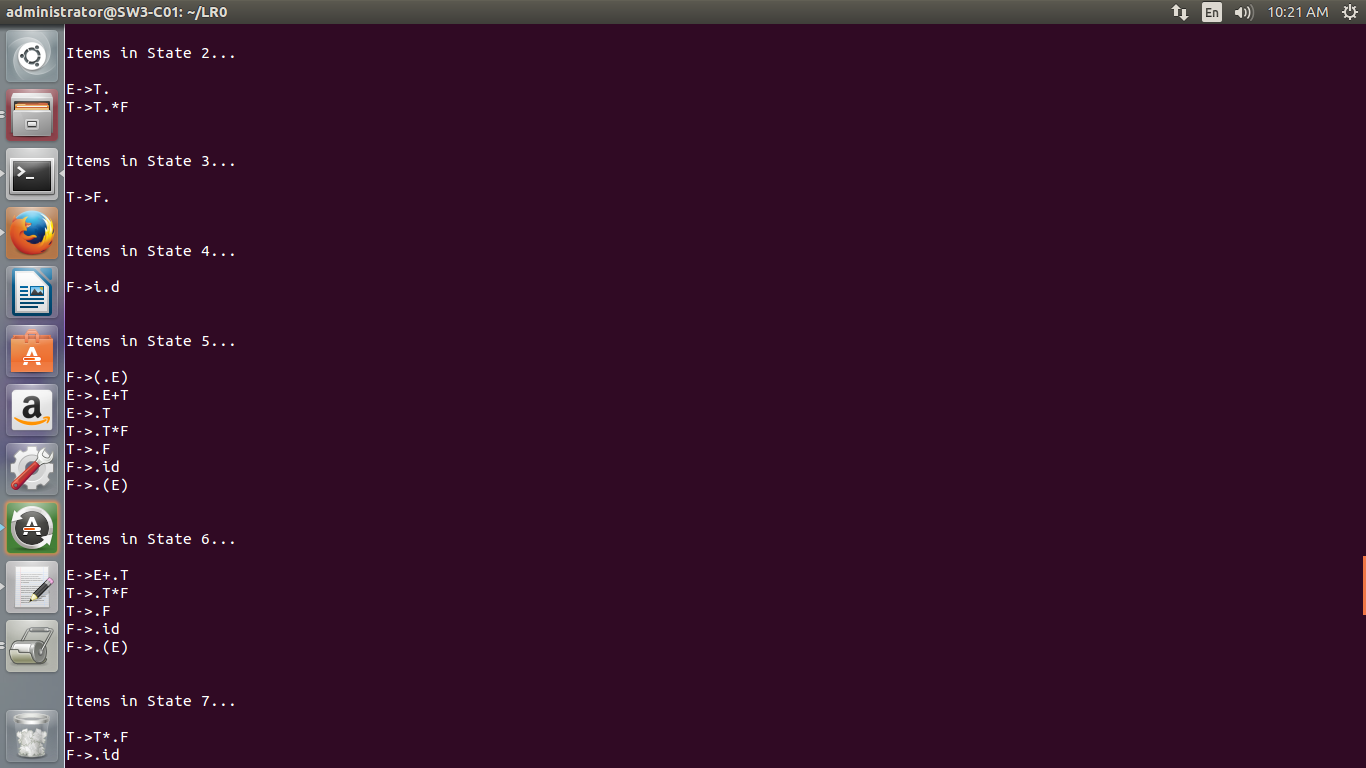
compute\_closure\_goto();

print();

}

Output:





A6.4 Implementation of LL1 parser.

Program:

/\*Find first and follow sets of a given grammar\*/

/\*To check whether the grammar is LL1 or not\*/

#include<stdio.h>

int n,m=0,p,i=0,j=0,npro,b;

char a[10][10],f[10];

int limit;

void follow(char c);

void first(char c);

//For finding first

void Find\_First(char\* array, char ch);

void Array\_Manipulation(char array[], char value);

void Find\_First(char\* array, char ch)

{

int count1, j, k;

char temporary\_result[20];

int x;

temporary\_result[0] = '\0';

array[0] = '\0';

if(!(isupper(ch)))

{

Array\_Manipulation(array, ch);

return ;

}

for(count1 = 0; count1 < limit; count1++)

{

if(a[count1][0] == ch)

{

if(a[count1][3] == '^')

{

Array\_Manipulation(array, '^');

}

else

{

j = 3;

while(a[count1][j] != '\0')

{

x = 0;

Find\_First(temporary\_result, a[count1][j]);

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

{

Array\_Manipulation(array,temporary\_result[k]);

}

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

{

if(temporary\_result[k] == '^')

{

x = 1;

break;

}

}

if(!x)

{

break;

}

j++;

}

}

}

}

return;

}

void Array\_Manipulation(char array[], char value)

{

int temp;

for(temp = 0; array[temp] != '\0'; temp++)

{

if(array[temp] == value)

{

return;

}

}

array[temp] = value;

array[temp + 1] = '\0';

}

void first(char c)

{

int k;

if(!isupper(c))

f[m++]=c;

for(k=0;k<n;k++)

{

if(a[k][0]==c)

{

if(a[k][3]=='^')

follow\_fun(a[k][0]);

else if(islower(a[k][3]))

f[m++]=a[k][3];

else first(a[k][3]);

}

}

}

void follow\_fun(char c)

{

if(a[0][0]==c)

f[m++]='$';

for(b=0;b<npro;b++)

{

for(j=3;j<strlen(a[b]);j++)

{

//printf("\nINSIDE IF for %c",c);

if(a[b][j]==c)

{

if(a[b][j+1]!='\0')

first(a[b][j+1]);

if(a[b][j+1]=='\0' && c!=a[b][0])

follow\_fun(a[b][0]);

}

}

}

}

void main()

{

char pro[10][10],first[10][10],follow[10][10],nt[10],ter[10],res[10][10][10],temp[10];

int noter=0,nont=0,k,flag=0,count[10][10],row,col,l,index;

char c,ch;

char array[25];

//clrscr();

for(i=0;i<10;i++)

{

for(j=0;j<10;j++)

{

count[i][j]=NULL;

for(k=0;k<10;k++)

{

res[i][j][k]=NULL;

}

}

}

printf("Enter the no of productions:");

scanf("%d",&npro);

printf("Enter the productions in the form of X->xy and individual rules separately:\n");

for(i=0;i<npro;i++)

{

//scanf("%s",pro[i]);

scanf("%s%c",a[i],&ch);

strcpy(pro[i],a[i]);

}

limit = npro;

n = npro;

for(i=0;i<npro;i++)

{

flag=0;

for(j=0;j<nont;j++)

{

if(nt[j]==pro[i][0])

flag=1;

}

if(flag==0)

{

nt[nont]=pro[i][0];

nont++;

}

}

for(i=0;i<nont;i++)

{

m=0;

Find\_First(array, nt[i] );

strcpy(first[i],array);

m=0;

follow\_fun(nt[i]);

strcpy(follow[i],f);

}

for(k=0;k<nont;k++)

{

printf("\nFirst Value of %c:\t{ ", nt[k]);

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

{

printf(" %c ", first[k][i]);

}

printf("}\n");

printf("Follow of %c:\t{ ",nt[k]);

for(i=0;i<m;i++)

printf(" %c ",follow[k][i]);

}

for(i=0;i<nont;i++)

{

flag=0;

for(j=0;j<strlen(first[i]);j++)

{

for(k=0;k<noter;k++)

{

if(ter[k]==first[i][j])

{

flag=1;

}

}

if(flag==0)

{

if(first[i][j]!='^')

{

ter[noter]=first[i][j];

noter++;

}

}

}

}

for(i=0;i<nont;i++)

{

flag=0;

for(j=0;j<strlen(follow[i]);j++)

{

for(k=0;k<noter;k++)

{

if(ter[k]==follow[i][j])

{

flag=1;

}

}

if(flag==0)

{

ter[noter]=follow[i][j];

noter++;

}

}

}

for(i=0;i<nont;i++)

{

for(j=0;j<strlen(first[i]);j++)

{

flag=0;

if(first[i][j]=='^')

{

col=i;

for(m=0;m<strlen(follow[col]);m++)

{

for(l=0;l<noter;l++)

{

if(ter[l]==follow[col][m])

{

row=l;

}

}

temp[0]=nt[col];

temp[1]='-' ;

temp[2]='>';

temp[3]='^';

temp[4]='\0';

printf("\ntemp %s",temp);

strcpy(res[col][row],temp);

count[col][row]+=1;

for(k=0;k<10;k++){

temp[k]=NULL; }

}

}

else{

for(l=0;l<noter;l++)

{

if(ter[l]==first[i][j])

{

row=l;

}

}

for(k=0;k<npro;k++){

if(nt[i]==pro[k][0])

{

col=i;

if((pro[k][3]==first[i][j])&&(pro[k][0]==nt[col]))

{

strcpy(res[col][row],pro[k]);

count[col][row]+=1;

}

else

{

if((isupper(pro[k][3]))&&(pro[k][0]==nt[col]))

{

flag=0;

for(m=0;m<nont;m++)

{

if(nt[m]==pro[k][3]){index=m;flag=1;}

}

if(flag==1){

for(m=0;m<strlen(first[index]);m++)

{if(first[i][j]==first[index][m])

{strcpy(res[col][row],pro[k]);

count[col][row]+=1;}

}

}

}}}}}

}}

printf("LL1 Table\n\n");

flag=0;

for(i=0;i<noter;i++)

{

printf("\t%c",ter[i]);

}

for(j=0;j<nont;j++)

{

printf("\n\n%c",nt[j]);

for(k=0;k<noter;k++)

{

printf("\t%s",res[j][k]);

if(count[j][k]>1){flag=1;}

}

}

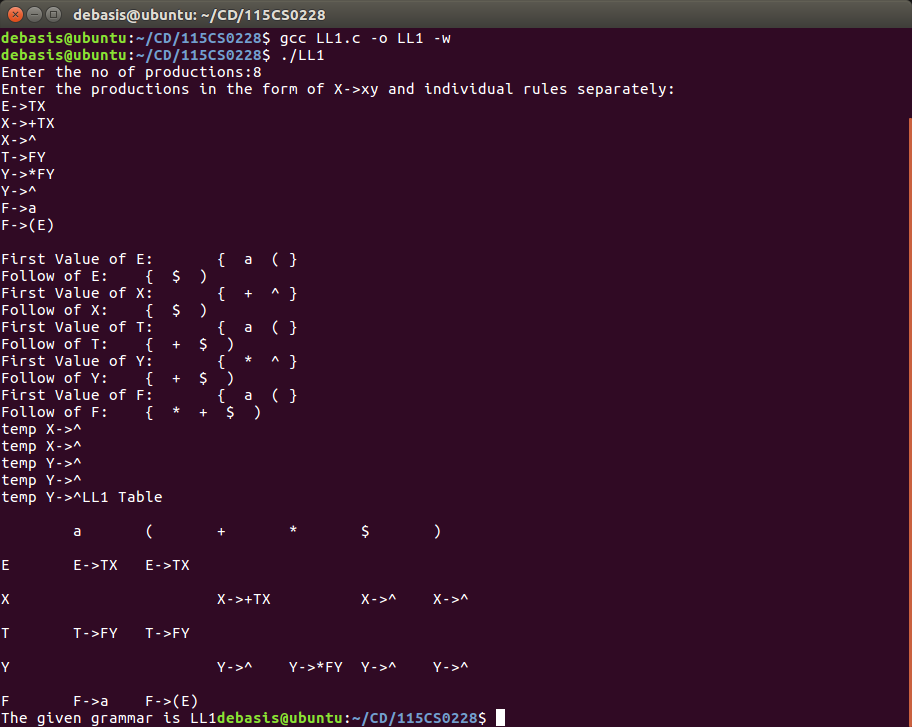
if(flag==1){printf("\nThe given grammar is not LL1\n");}

else{printf("\nThe given grammar is LL1\n");}

//getch();

}

Code:



A6.5 Implementation of LALR grammar.

Program:

/\*LALR PARSER \*/

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

void push(char \*,int \*,char);

char stacktop(char \*);

void isproduct(char,char);

int ister(char);

int isnter(char);

int isstate(char);

void error();

void isreduce(char,char);

char pop(char \*,int \*);

void printt(char \*,int \*,char [],int);

void rep(char [],int);

struct action

{

char row[6][5];

};

const struct action A[12]={

{"sf","emp","emp","se","emp","emp"},

{"emp","sg","emp","emp","emp","acc"},

{"emp","rc","sh","emp","rc","rc"},

{"emp","re","re","emp","re","re"},

{"sf","emp","emp","se","emp","emp"},

{"emp","rg","rg","emp","rg","rg"},

{"sf","emp","emp","se","emp","emp"},

{"sf","emp","emp","se","emp","emp"},

{"emp","sg","emp","emp","sl","emp"},

{"emp","rb","sh","emp","rb","rb"},

{"emp","rb","rd","emp","rd","rd"},

{"emp","rf","rf","emp","rf","rf"}

};

struct gotol

{

char r[3][4];

};

const struct gotol G[12]={

{"b","c","d"},

{"emp","emp","emp"},

{"emp","emp","emp"},

{"emp","emp","emp"},

{"i","c","d"},

{"emp","emp","emp"},

{"emp","j","d"},

{"emp","emp","k"},

{"emp","emp","emp"},

{"emp","emp","emp"},

};

char ter[6]={'i','+','\*',')','(','$'};

char nter[3]={'E','T','F'};

char states[12]={'a','b','c','d','e','f','g','h','m','j','k','l'};

char stack[100];

int top=-1;

char temp[10];

struct grammar

{

char left; char right[5]; };

const struct grammar rl[6]={

{'E',"e+T"},

{'E',"T"},

{'T',"T\*F"},

{'T',"F"},

{'F',"(E)"},

{'F',"i"},

};

void main()

{

char inp[80],x,p,dl[80],y,bl='a';

int i=0,j,k,l,n,m,c,len;

printf(" Enter the input :");

scanf("%s",inp);

len = strlen(inp);

inp[len]='$';

inp[len+1]='\0';

push(stack,&top,bl);

printf("\n stack \t\t\t input");

printt(stack,&top,inp,i);

do

{

x=inp[i]; p=stacktop(stack);

isproduct(x,p);

if(strcmp(temp,"emp")==0)

error();

if(strcmp(temp,"acc")==0)

break;

else

{

if(temp[0]=='s')

{

push(stack,&top,inp[i]); push(stack,&top,temp[1]);

i++;

}

else

{

if(temp[0]=='r')

{

j=isstate(temp[1]);

strcpy(temp,rl[j-2].right);

dl[0]=rl[j-2].left;

dl[1]='\0';

n=strlen(temp);

for(k=0;k<2\*n;k++)

pop(stack,&top);

for(m=0;dl[m]!='\0';m++) push(stack,&top,dl[m]);

l=top;

y=stack[l-1];

isreduce(y,dl[0]); for(m=0;temp[m]!='\0';m++) push(stack,&top,temp[m]);

}

}

}

printt(stack,&top,inp,i);

}

while(inp[i]!='\0');

if(strcmp(temp,"acc")==0)

printf("\nThe string is accepted ");

else

printf("\nThe string is not accepted");

}

void push(char \*s,int \*sp,char item)

{

if(\*sp==100) printf("The stack is full ");

else

{

\*sp=\*sp+1;

s[\*sp]=item;

}

}

char stacktop(char \*s)

{

char i; i=s[top];

return i;

}

void isproduct(char x,char p)

{

int k,l;

k=ister(x);

l=isstate(p);

strcpy(temp,A[l-1].row[k-1]);

}

int ister(char x)

{

int i;

for(i=0;i<6;i++)

if(x==ter[i])

return i+1;

return 0;

}

int isnter(char x)

{

int i;

for(i=0;i<3;i++)

if(x==nter[i])

return i+1;

return 0;

}

int isstate(char p)

{

int i;

for(i=0;i<12;i++)

if(p==states[i])

return i+1;

return 0;

}

void error()

{

printf(" error in the input ");

exit(0);

}

void isreduce(char x,char p)

{

int k,l; k=isstate(x);

l=isnter(p);

strcpy(temp,G[k-1].r[l-1]);

}

char pop(char \*s,int \*sp)

{

char item; if(\*sp==-1)

printf(" stack is empty ");

else

{

item=s[\*sp];

\*sp=\*sp-1;

}

return item;

}

void printt(char \*t,int \*p,char inp[],int i)

{

int r;

printf("\n");

for(r=0;r<=\*p;r++)

rep(t,r);

printf("\t\t\t");

for(r=i;inp[r]!='\0';r++)

printf("%c",inp[r]);

}

void rep(char t[],int r)

{

char c;

c=t[r];

switch(c)

{

case 'a': printf("0");

break;

case 'b': printf("1");

break;

case 'c': printf("2");

break;

case 'd': printf("3");

break;

case 'e': printf("4");

break;

case 'f': printf("5");

break;

case 'g': printf("6");

break;

case 'h': printf("7");

break;

case 'm': printf("8");

break;

case 'j': printf("9");

break;

case 'k': printf("10");

break;

case 'l': printf("11");

break;

default :printf("%c",t[r]);

break;

}

}

/\* Defined grammar

E->E+T

E->T

T->T\*F

T->F

F->(E)

F->i\*/

Output:

