**CS 6109**

**COMPILER DESIGN**

**LAB – 5**

**12-10-2021**

**DONE BY**

**SUDARSAN KUMAR N**

**2019503564**

**1. Write a C program and construct predictive parsing table for the grammar and find moves made by predictive parser on input id + id \* id and find FIRST and FOLLOW.**

**E --> E + T**

**E --> T**

**T --> T \* F**

**T --> F**

**F --> (E) | id**

**CODE**

#include<stdio.h>

#include<ctype.h>

#include<string.h>

void followfirst(char , int , int);

void findfirst(char , int , int);

void follow(char c);

int count,n=0;

char calc\_first[10][100];

char calc\_follow[10][100];

int m=0;

char production[10][10], first[10];

char f[10];

int k;

char ck;

int e;

int main(int argc,char \*\*argv)

{

int jm=0;

int km=0;

int i,choice;

char c,ch;

printf("How many productions ? :");

scanf("%d",&count);

printf("\nEnter %d productions in form A=B where A and B are grammar symbols :\n\n",count);

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

{

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

}

int kay;

char done[count];

int ptr = -1;

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

for(kay=0;kay<100;kay++){

calc\_first[k][kay] = '!';

}

}

int point1 = 0,point2,xxx;

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

{

c=production[k][0];

point2 = 0;

xxx = 0;

for(kay = 0; kay <= ptr; kay++)

if(c == done[kay])

xxx = 1;

if (xxx == 1)

continue;

findfirst(c,0,0);

ptr+=1;

done[ptr] = c;

printf("\n First(%c)= { ",c);

calc\_first[point1][point2++] = c;

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

int lark = 0,chk = 0;

for(lark=0;lark<point2;lark++){

if (first[i] == calc\_first[point1][lark]){

chk = 1;

break;

}

}

if(chk == 0){

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

calc\_first[point1][point2++] = first[i];

}

}

printf("}\n");

jm=n;

point1++;

}

printf("\n");

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

char donee[count];

ptr = -1;

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

for(kay=0;kay<100;kay++){

calc\_follow[k][kay] = '!';

}

}

point1 = 0;

int land = 0;

for(e=0;e<count;e++)

{

ck=production[e][0];

point2 = 0;

xxx = 0;

for(kay = 0; kay <= ptr; kay++)

if(ck == donee[kay])

xxx = 1;

if (xxx == 1)

continue;

land += 1;

follow(ck);

ptr+=1;

donee[ptr] = ck;

printf(" Follow(%c) = { ",ck);

calc\_follow[point1][point2++] = ck;

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

int lark = 0,chk = 0;

for(lark=0;lark<point2;lark++){

if (f[i] == calc\_follow[point1][lark]){

chk = 1;

break;

}

}

if(chk == 0){

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

calc\_follow[point1][point2++] = f[i];

}

}

printf(" }\n\n");

km=m;

point1++;

}

char ter[10];

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

ter[k] = '!';

}

int ap,vp,sid = 0;

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

for(kay=0;kay<count;kay++){

if(!isupper(production[k][kay]) && production[k][kay]!= '#' && production[k][kay] != '=' && production[k][kay] != '\0'){

vp = 0;

for(ap = 0;ap < sid; ap++){

if(production[k][kay] == ter[ap]){

vp = 1;

break;

}

}

if(vp == 0){

ter[sid] = production[k][kay];

sid ++;

}

}

}

}

ter[sid] = '$';

sid++;

printf("\n\t\t\t\t\t\t\t The LL(1) Parsing Table for the above grammer :-");

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

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

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

for(ap = 0;ap < sid; ap++){

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

}

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

char first\_prod[count][sid];

for(ap=0;ap<count;ap++){

int destiny = 0;

k = 2;

int ct = 0;

char tem[100];

while(production[ap][k] != '\0'){

if(!isupper(production[ap][k])){

tem[ct++] = production[ap][k];

tem[ct++] = '\_';

tem[ct++] = '\0';

k++;

break;

}

else{

int zap=0;

int tuna = 0;

for(zap=0;zap<count;zap++){

if(calc\_first[zap][0] == production[ap][k]){

for(tuna=1;tuna<100;tuna++){

if(calc\_first[zap][tuna] != '!'){

tem[ct++] = calc\_first[zap][tuna];

}

else

break;

}

break;

}

}

tem[ct++] = '\_';

}

k++;

}

int zap = 0,tuna;

for(tuna = 0;tuna<ct;tuna++){

if(tem[tuna] == '#'){

zap = 1;

}

else if(tem[tuna] == '\_'){

if(zap == 1){

zap = 0;

}

else

break;

}

else{

first\_prod[ap][destiny++] = tem[tuna];

}

}

}

char table[land][sid+1];

ptr = -1;

for(ap = 0; ap < land ; ap++){

for(kay = 0; kay < (sid + 1) ; kay++){

table[ap][kay] = '!';

}

}

for(ap = 0; ap < count ; ap++){

ck = production[ap][0];

xxx = 0;

for(kay = 0; kay <= ptr; kay++)

if(ck == table[kay][0])

xxx = 1;

if (xxx == 1)

continue;

else{

ptr = ptr + 1;

table[ptr][0] = ck;

}

}

for(ap = 0; ap < count ; ap++){

int tuna = 0;

while(first\_prod[ap][tuna] != '\0'){

int to,ni=0;

for(to=0;to<sid;to++){

if(first\_prod[ap][tuna] == ter[to]){

ni = 1;

}

}

if(ni == 1){

char xz = production[ap][0];

int cz=0;

while(table[cz][0] != xz){

cz = cz + 1;

}

int vz=0;

while(ter[vz] != first\_prod[ap][tuna]){

vz = vz + 1;

}

table[cz][vz+1] = (char)(ap + 65);

}

tuna++;

}

}

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

for(kay=0;kay<100;kay++){

if(calc\_first[k][kay] == '!'){

break;

}

else if(calc\_first[k][kay] == '#'){

int fz = 1;

while(calc\_follow[k][fz] != '!'){

char xz = production[k][0];

int cz=0;

while(table[cz][0] != xz){

cz = cz + 1;

}

int vz=0;

while(ter[vz] != calc\_follow[k][fz]){

vz = vz + 1;

}

table[k][vz+1] = '#';

fz++;

}

break;

}

}

}

for(ap = 0; ap < land ; ap++){

printf("\t\t\t %c\t|\t",table[ap][0]);

for(kay = 1; kay < (sid + 1) ; kay++){

if(table[ap][kay] == '!')

printf("\t\t");

else if(table[ap][kay] == '#')

printf("%c=#\t\t",table[ap][0]);

else{

int mum = (int)(table[ap][kay]);

mum -= 65;

printf("%s\t\t",production[mum]);

}

}

printf("\n");

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

printf("\n");

}

int j;

printf("\n\nPlease enter the desired INPUT STRING = ");

char input[100];

scanf("%s%c",input,&ch);

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

printf("\t\t\t\t\t\tStack\t\t\tInput\t\t\tAction");

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

int i\_ptr = 0,s\_ptr = 1;

char stack[100];

stack[0] = '$';

stack[1] = table[0][0];

while(s\_ptr != -1){

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

int vamp = 0;

for(vamp=0;vamp<=s\_ptr;vamp++){

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

}

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

vamp = i\_ptr;

while(input[vamp] != '\0'){

printf("%c",input[vamp]);

vamp++;

}

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

char her = input[i\_ptr];

char him = stack[s\_ptr];

s\_ptr--;

if(!isupper(him)){

if(her == him){

i\_ptr++;

printf("POP ACTION\n");

}

else{

printf("\nString Not Accepted by LL(1) Parser !!\n");

exit(0);

}

}

else{

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

if(ter[i] == her)

break;

}

char produ[100];

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

if(him == table[j][0]){

if (table[j][i+1] == '#'){

printf("%c=#\n",table[j][0]);

produ[0] = '#';

produ[1] = '\0';

}

else if(table[j][i+1] != '!'){

int mum = (int)(table[j][i+1]);

mum -= 65;

strcpy(produ,production[mum]);

printf("%s\n",produ);

}

else{

printf("\nString Not Accepted by LL(1) Parser !!\n");

exit(0);

}

}

}

int le = strlen(produ);

le = le - 1;

if(le == 0){

continue;

}

for(j=le;j>=2;j--){

s\_ptr++;

stack[s\_ptr] = produ[j];

}

}

}

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

if (input[i\_ptr] == '\0'){

printf("\t\t\t\t\t\t\t\tYOUR STRING HAS BEEN ACCEPTED !!\n");

}

else

printf("\n\t\t\t\t\t\t\t\tYOUR STRING HAS BEEN REJECTED !!\n");

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

}

void follow(char c)

{

int i ,j;

if(production[0][0]==c){

f[m++]='$';

}

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

{

for(j=2;j<10;j++)

{

if(production[i][j]==c)

{

if(production[i][j+1]!='\0'){

followfirst(production[i][j+1],i,(j+2));

}

if(production[i][j+1]=='\0'&&c!=production[i][0]){

follow(production[i][0]);

}

}

}

}

}

void findfirst(char c ,int q1 , int q2)

{

int j;

if(!(isupper(c))){

first[n++]=c;

}

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

{

if(production[j][0]==c)

{

if(production[j][2]=='#'){

if(production[q1][q2] == '\0')

first[n++]='#';

else if(production[q1][q2] != '\0' && (q1 != 0 || q2 != 0))

{

findfirst(production[q1][q2], q1, (q2+1));

}

else

first[n++]='#';

}

else if(!isupper(production[j][2])){

first[n++]=production[j][2];

}

else {

findfirst(production[j][2], j, 3);

}

}

}

}

void followfirst(char c, int c1 , int c2)

{

int k;

if(!(isupper(c)))

f[m++]=c;

else{

int i=0,j=1;

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

{

if(calc\_first[i][0] == c)

break;

}

while(calc\_first[i][j] != '!')

{

if(calc\_first[i][j] != '#'){

f[m++] = calc\_first[i][j];

}

else{

if(production[c1][c2] == '\0'){

follow(production[c1][0]);

}

else{

followfirst(production[c1][c2],c1,c2+1);

}

}

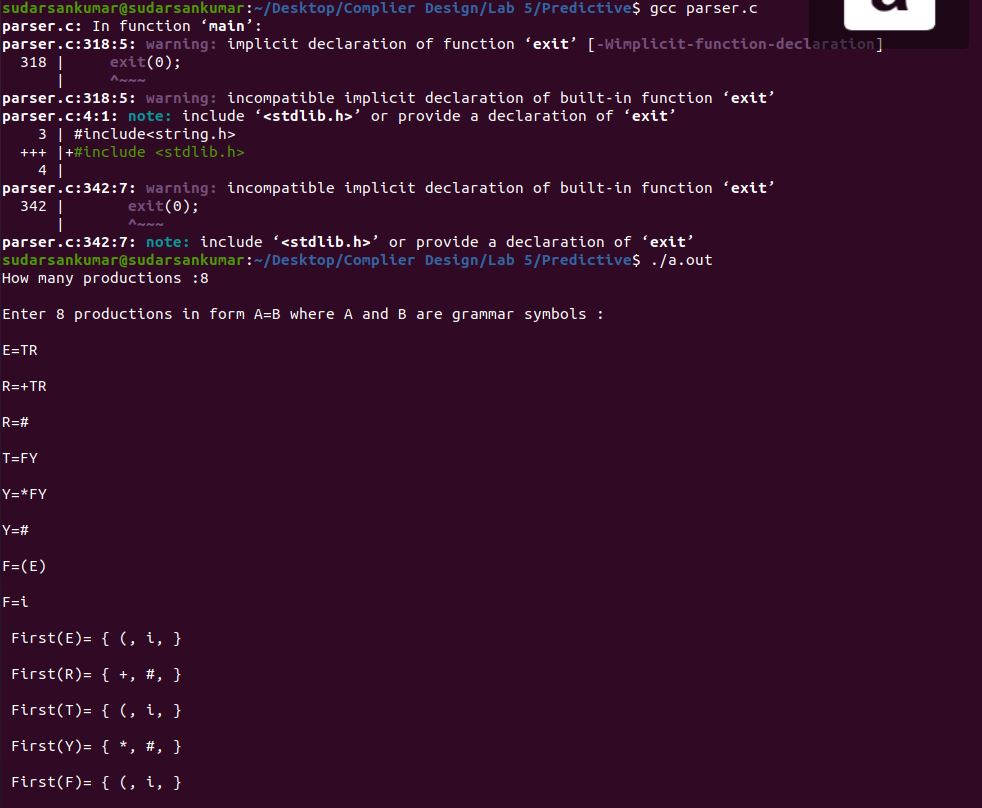
j++;

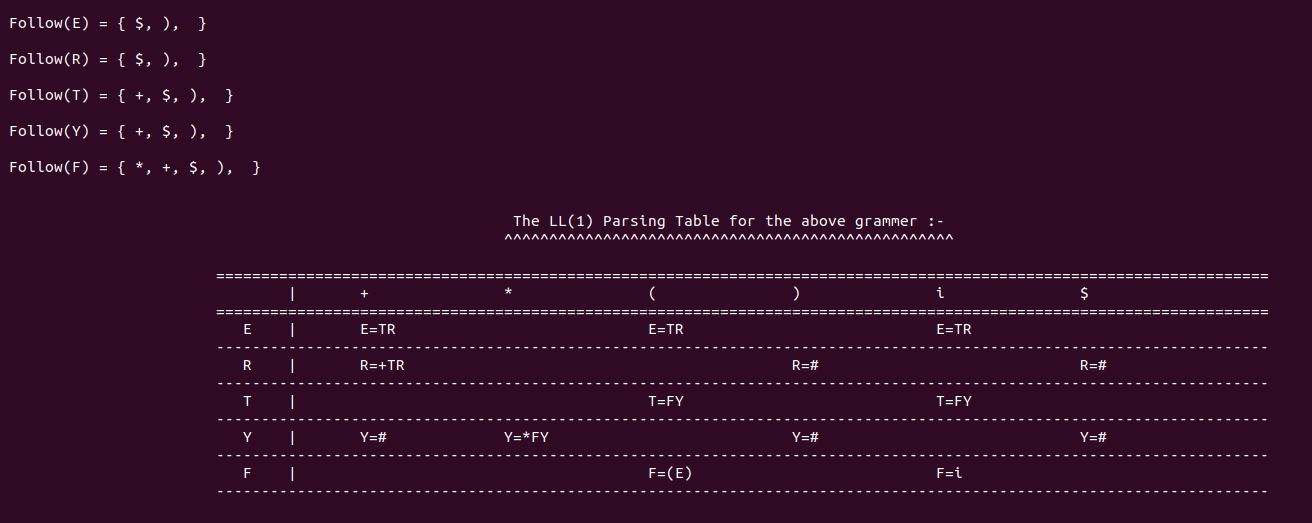
}

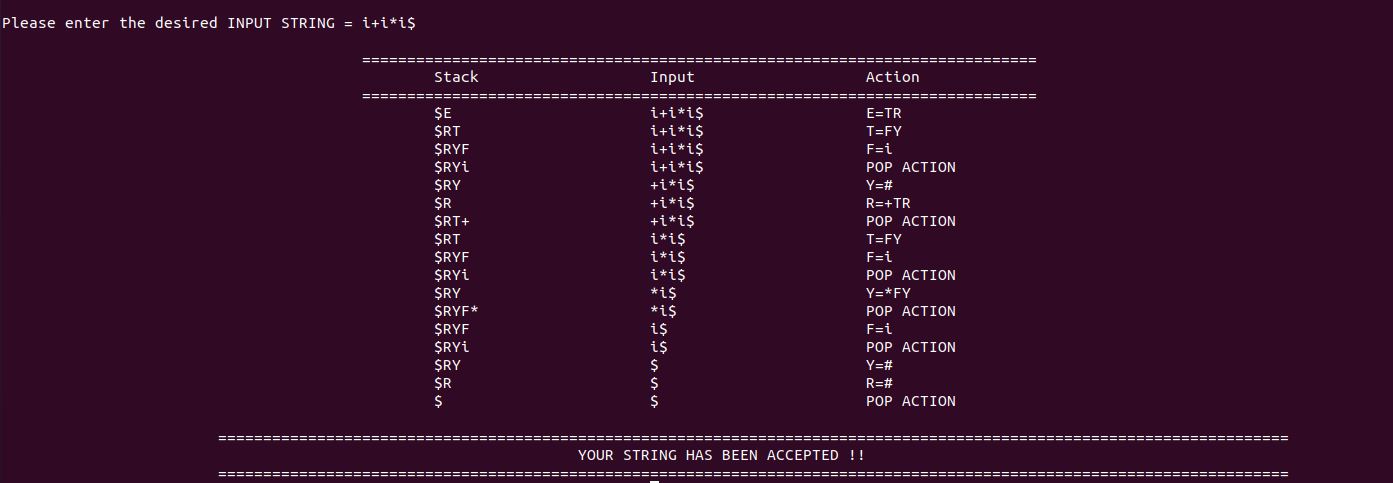
}

}

**OUTPUT**







**2. Write a C program and construct LR parsing table for the grammar and find moves made by predictive parser on input id + id \* id and find FIRST and FOLLOW.**

**E --> E + T**

**E --> T**

**T --> T \* F**

**T --> F**

**F --> (E) | id**

**CODE**

**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();

}

**FISRT\_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])]);

}

**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");

}

**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;

}

**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();

}

**OUTPUT**

