COMPILER DESIGN

**Construct predictive parsing table for the grammar,**

E → E + T | T

T → T ∗ F | F

F → (E) | id

and parse the input id + id∗id.

**AIM:**

To Find out the predictive parsing table using C

**PROCEDURE**:

Write the program in an online compiler

Create a new file and write the CFG into it

Run the main .c file

Output will be displayed as per the given grammar

**Method :**

Step 1 Construct an NFA with Null moves from the given regular

**Expression:**

Step 2 Remove Null transition from the NFA and convert it into its

equivalent DFA.

**Algorithm to construct LL(1) Parsing Table:**

Step 1:

First check for left recursion in the grammar, if there is left

recursion in the grammar remove that and go to step 2.

Step 2:

Calculate First() and Follow() for all non-terminals.

1.  First(): If there is a variable, and from that variable, if we

try to drive all the strings then the beginning Terminal

Symbol is called the First.

2. Follow(): What is the Terminal Symbol which follows a

variable in the process of derivation

Step 3:

For each production A –&amp;gt; α. (A tends to alpha)

1. Find First(α) and for each terminal in First(α), make entry

A –&amp;gt; α in the table.

2. If First(α) contains ε (epsilon) as terminal than, find the

Follow(A) and for each terminal in Follow(A), make entry A

–&amp;gt; α in the table.

3. If the First(α) contains ε and Follow(A) contains $ as terminal, then

make entry A –&amp;gt; α in the table for the $.

**To construct the parsing table, we have two functions:**

In the table, rows will contain the Non-Terminals and the column

will contain the Terminal Symbols. All the Null Productions of the

Grammars will go under the Follow elements and the remaining

productions will lie under the elements of the First set.

**CODE:**

#include<stdio.h>

#include<string.h>

#define TSIZE 128

// table[i][j] stores

// the index of production that must be applied on

// ithvarible if the input is

// jth nonterminal

int table[100][TSIZE];

// stores all list of terminals

// the ASCII value if use to index terminals

// terminal[i] = 1 means the character with

// ASCII value is a terminal

char terminal[TSIZE];

// stores all list of terminals

// only Upper case letters from 'A' to 'Z'

// can be nonterminals

// nonterminal[i] means ith alphabet is present as

// nonterminal is the grammar

char nonterminal[26];

// structure to hold each production

// str[] stores the production

// len is the length of production

struct product {

char str[100];

int len;

}pro[20];

// no of productions in form A->ß

int no\_pro;

char first[26][TSIZE];

char follow[26][TSIZE];

// stores first of each production in form A->ß

char first\_rhs[100][TSIZE];

// check if the symbol is nonterminal

int isNT(char c) {

return c >= 'A' && c <= 'Z';

}

// reading data from the file

void readFromFile() {

FILE\* fptr;

fptr = fopen("123.txt", "r");

char buffer[255];

int i;

int j;

while (fgets(buffer, sizeof(buffer), fptr)) {

printf("%s", buffer);

j = 0;

nonterminal[buffer[0] - 'A'] = 1;

for (i = 0; i<strlen(buffer) - 1; ++i) {

if (buffer[i] == '|') {

++no\_pro;

pro[no\_pro - 1].str[j] = '\0';

pro[no\_pro - 1].len = j;

pro[no\_pro].str[0] = pro[no\_pro - 1].str[0];

pro[no\_pro].str[1] = pro[no\_pro - 1].str[1];

pro[no\_pro].str[2] = pro[no\_pro - 1].str[2];

j = 3;

}

else {

pro[no\_pro].str[j] = buffer[i];

++j;

if (!isNT(buffer[i]) && buffer[i] != '-' && buffer[i] != '>') {

terminal[buffer[i]] = 1;

}

}

}

pro[no\_pro].len = j;

++no\_pro;

}

}

void add\_FIRST\_A\_to\_FOLLOW\_B(char A, char B) {

int i;

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

if (i != '^')

follow[B - 'A'][i] = follow[B - 'A'][i] || first[A - 'A'][i];

}

}

void add\_FOLLOW\_A\_to\_FOLLOW\_B(char A, char B) {

int i;

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

if (i != '^')

follow[B - 'A'][i] = follow[B - 'A'][i] || follow[A - 'A'][i];

}

}

void FOLLOW() {

int t = 0;

int i, j, k, x;

while (t++ <no\_pro) {

for (k = 0; k < 26; ++k) {

if (!nonterminal[k]) continue;

char nt = k + 'A';

for (i = 0; i<no\_pro; ++i) {

for (j = 3; j < pro[i].len; ++j) {

if (nt == pro[i].str[j]) {

for (x = j + 1; x < pro[i].len; ++x) {

char sc = pro[i].str[x];

if (isNT(sc)) {

add\_FIRST\_A\_to\_FOLLOW\_B(sc, nt);

if (first[sc - 'A']['^'])

continue;

}

else {

follow[nt - 'A'][sc] = 1;

}

break;

}

if (x == pro[i].len)

add\_FOLLOW\_A\_to\_FOLLOW\_B(pro[i].str[0], nt);

}

}

}

}

}

}

void add\_FIRST\_A\_to\_FIRST\_B(char A, char B) {

int i;

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

if (i != '^') {

first[B - 'A'][i] = first[A - 'A'][i] || first[B - 'A'][i];

}

}

}

void FIRST() {

int i, j;

int t = 0;

while (t <no\_pro) {

for (i = 0; i<no\_pro; ++i) {

for (j = 3; j < pro[i].len; ++j) {

char sc = pro[i].str[j];

if (isNT(sc)) {

add\_FIRST\_A\_to\_FIRST\_B(sc, pro[i].str[0]);

if (first[sc - 'A']['^'])

continue;

}

else {

first[pro[i].str[0] - 'A'][sc] = 1;

}

break;

}

if (j == pro[i].len)

first[pro[i].str[0] - 'A']['^'] = 1;

}

++t;

}

}

void add\_FIRST\_A\_to\_FIRST\_RHS\_\_B(char A, int B) {

int i;

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

if (i != '^')

first\_rhs[B][i] = first[A - 'A'][i] || first\_rhs[B][i];

}

}

// Calculates FIRST(ß) for each A->ß

void FIRST\_RHS() {

int i, j;

int t = 0;

while (t <no\_pro) {

for (i = 0; i<no\_pro; ++i) {

for (j = 3; j < pro[i].len; ++j) {

char sc = pro[i].str[j];

if (isNT(sc)) {

add\_FIRST\_A\_to\_FIRST\_RHS\_\_B(sc, i);

if (first[sc - 'A']['^'])

continue;

}

else {

first\_rhs[i][sc] = 1;

}

break;

}

if (j == pro[i].len)

first\_rhs[i]['^'] = 1;

}

++t;

}

}

int main() {

readFromFile();

follow[pro[0].str[0] - 'A']['$'] = 1;

FIRST();

FOLLOW();

FIRST\_RHS();

int i, j, k;

// display first of each variable

printf("\n");

for (i = 0; i<no\_pro; ++i) {

if (i == 0 || (pro[i - 1].str[0] != pro[i].str[0])) {

char c = pro[i].str[0];

printf("FIRST OF %c: ", c);

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

if (first[c - 'A'][j]) {

printf("%c ", j);

}

}

printf("\n");

}

}

// display follow of each variable

printf("\n");

for (i = 0; i<no\_pro; ++i) {

if (i == 0 || (pro[i - 1].str[0] != pro[i].str[0])) {

char c = pro[i].str[0];

printf("FOLLOW OF %c: ", c);

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

if (follow[c - 'A'][j]) {

printf("%c ", j);

}

}

printf("\n");

}

}

// display first of each variable ß

// in form A->ß

printf("\n");

for (i = 0; i<no\_pro; ++i) {

printf("FIRST OF %s: ", pro[i].str);

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

if (first\_rhs[i][j]) {

printf("%c ", j);

}

}

printf("\n");

}

// the parse table contains '$'

// set terminal['$'] = 1

// to include '$' in the parse table

terminal['$'] = 1;

// the parse table do not read '^'

// as input

// so we set terminal['^'] = 0

// to remove '^' from terminals

terminal['^'] = 0;

// printing parse table

printf("\n");

printf("\n\t\*\*\*\*\*\* LL(1) PARSING TABLE \*\*\*\*\*\*\*\n");

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

printf("%-10s", "");

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

if (terminal[i]) printf("%-10c", i);

}

printf("\n");

int p = 0;

for (i = 0; i<no\_pro; ++i) {

if (i != 0 && (pro[i].str[0] != pro[i - 1].str[0]))

p = p + 1;

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

if (first\_rhs[i][j] &&j != '^') {

table[p][j] = i + 1;

}

else if (first\_rhs[i]['^']) {

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

if (follow[pro[i].str[0] - 'A'][k]) {

table[p][k] = i + 1;

}

}

}

}

}

k = 0;

for (i = 0; i<no\_pro; ++i) {

if (i == 0 || (pro[i - 1].str[0] != pro[i].str[0])) {

printf("%-10c", pro[i].str[0]);

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

if (table[k][j]) {

printf("%-10s", pro[table[k][j] - 1].str);

}

else if (terminal[j]) {

printf("%-10s", "");

}

}

++k;

printf("\n");

}

}

}