**COMPILER CONSTRUCTION LAB 8**

# Task 1: Remove Left Factoring

#define \_CRT\_SECURE\_NO\_WARNINGS

#include<stdio.h>

#include<string.h>

int main() {

char gram[20], part1[20], part2[20], modifiedGram[20], newGram[20], tempGram[20];

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

printf("Enter Production : A->");

scanf("%s", gram);

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

part1[j] = gram[i];

part1[j] = '\0';

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

part2[i] = gram[j];

part2[i] = '\0';

for (i = 0; i<strlen(part1) || i<strlen(part2); i++) {

if (part1[i] == part2[i]) {

modifiedGram[k] = part1[i];

k++;

pos = i + 1;

}

}

for (i = pos, j = 0; part1[i] != '\0'; i++, j++)

newGram[j] = part1[i];

newGram[j++] = '|';

for (i = pos; part2[i] != '\0'; i++, j++)

newGram[j] = part2[i];

modifiedGram[k] = 'X';

modifiedGram[++k] = '\0';

newGram[j] = '\0';

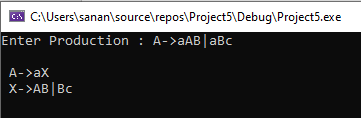
printf("\n A->%s", modifiedGram);

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

getchar();

getchar();

}



# Task 2: Eliminate Left Recursion

#define \_CRT\_SECURE\_NO\_WARNINGS

#include<stdio.h>

#include<string.h>

#define SIZE 10

int main() {

char non\_terminal;

char beta, alpha, bb;

int num;

char production[10][SIZE];

int index = 3; /\* starting of the string following "->" \*/

printf("Enter Number of Production : ");

scanf("%d", &num);

printf("Enter the grammar as E->E-A :\n");

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

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

}

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

printf("\nGRAMMAR : : : %s", production[i]);

non\_terminal = production[i][0];

if (non\_terminal == production[i][index]) {

alpha = production[i][index + 1];

bb = production[i][index + 2];

printf(" is left recursive.\n");

while (production[i][index] != 0 && production[i][index] != '|')

index++;

if (production[i][index] != 0) {

beta = production[i][index + 1];

printf("Grammar without left recursion:\n");

printf("%c->%c%c\'", non\_terminal, beta, non\_terminal);

printf("\n%c\'->%c%c%c\'|E\n", non\_terminal, alpha, bb, non\_terminal);

}

else

printf(" can't be reduced\n");

}

else

printf(" is not left recursive.\n");

index = 3;

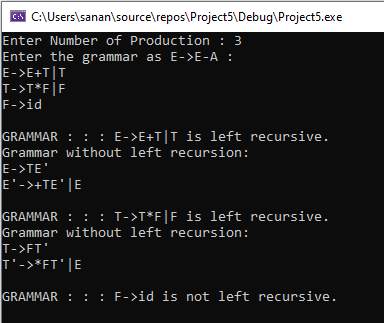
}

getchar();

getchar();

return 0;

}



# Task 3: Recursive Descent Parser

#define \_CRT\_SECURE\_NO\_WARNINGS

#include"stdio.h"

#include"conio.h"

#include"string.h"

#include"stdlib.h"

#include"ctype.h"

char ip\_sym[15], ip\_ptr = 0, op[50], tmp[50];

void e\_prime();

void e();

void t\_prime();

void t();

void f();

void advance();

int n = 0;

void e()

{

strcpy(op, "TE'");

printf("E=%-25s", op);

printf("E->TE'\n");

t();

e\_prime();

}

void e\_prime()

{

int i, n = 0, l;

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

if (op[i] != 'e')

tmp[n++] = op[i];

strcpy(op, tmp);

l = strlen(op);

for (n = 0; n < l && op[n] != 'E'; n++);

if (ip\_sym[ip\_ptr] == '+')

{

i = n + 2;

do

{

op[i + 2] = op[i];

i++;

} while (i <= l);

op[n++] = '+';

op[n++] = 'T';

op[n++] = 'E';

op[n++] = 39;

printf("E=%-25s", op);

printf("E'->+TE'\n");

advance();

t();

e\_prime();

}

else

{

op[n] = 'e';

for (i = n + 1; i <= strlen(op); i++)

op[i] = op[i + 1];

printf("E=%-25s", op);

printf("E'->e");

}

}

void t()

{

int i, n = 0, l;

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

if (op[i] != 'e')

tmp[n++] = op[i];

strcpy(op, tmp);

l = strlen(op);

for (n = 0; n < l && op[n] != 'T'; n++);

i = n + 1;

do

{

op[i + 2] = op[i];

i++;

} while (i < l);

op[n++] = 'F';

op[n++] = 'T';

op[n++] = 39;

printf("E=%-25s", op);

printf("T->FT'\n");

f();

t\_prime();

}

void t\_prime()

{

int i, n = 0, l;

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

if (op[i] != 'e')

tmp[n++] = op[i];

strcpy(op, tmp);

l = strlen(op);

for (n = 0; n < l && op[n] != 'T'; n++);

if (ip\_sym[ip\_ptr] == '\*')

{

i = n + 2;

do

{

op[i + 2] = op[i];

i++;

} while (i < l);

op[n++] = '\*';

op[n++] = 'F';

op[n++] = 'T';

op[n++] = 39;

printf("E=%-25s", op);

printf("T'->\*FT'\n");

advance();

f();

t\_prime();

}

else

{

op[n] = 'e';

for (i = n + 1; i <= strlen(op); i++)

op[i] = op[i + 1];

printf("E=%-25s", op);

printf("T'->e\n");

}

}

void f()

{

int i, n = 0, l;

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

if (op[i] != 'e')

tmp[n++] = op[i];

strcpy(op, tmp);

l = strlen(op);

for (n = 0; n < l && op[n] != 'F'; n++);

if ((ip\_sym[ip\_ptr] == 'i') || (ip\_sym[ip\_ptr] == 'I'))

{

op[n] = 'i';

printf("E=%-25s", op);

printf("F->i\n");

advance();

}

else

{

if (ip\_sym[ip\_ptr] == '(')

{

advance();

e();

if (ip\_sym[ip\_ptr] == ')')

{

advance();

i = n + 2;

do

{

op[i + 2] = op[i];

i++;

} while (i <= l);

op[n++] = '(';

op[n++] = 'E';

op[n++] = ')';

printf("E=%-25s", op);

printf("F->(E)\n");

}

}

else

{

printf("\n\t syntax error");

getchar();

exit(1);

}

}

}

void advance()

{

ip\_ptr++;

}

void main()

{

int i;

printf("\nGrammar without left recursion");

printf("\n\t\t E->TE' \n\t\t E'->+TE'|e \n\t\t T->FT' ");

printf("\n\t\t T'->\*FT'|e \n\t\t F->(E)|i");

printf("\n Enter the input expression:");

gets(ip\_sym);

printf("Expressions");

printf("\t Sequence of production rules\n");

e();

for (i = 0; i < strlen(ip\_sym); i++)

{

if (ip\_sym[i] != '+'&&ip\_sym[i] != '\*'&&ip\_sym[i] != '('&&

ip\_sym[i] != ')'&&ip\_sym[i] != 'i'&&ip\_sym[i] != 'I')

{

printf("\nSyntax error");

break;

}

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

if (op[i] != 'e')

tmp[n++] = op[i];

strcpy(op, tmp);

printf("\nE=%-25s", op);

}

getchar();

}

