**Week-1: Lexical analysis using lex tool**

**1.1) Write a lex program whose output is same as input.**

**Aim:** To write a lex program whose output is same as input.

**Description:** The FLEX tool is a modern version of LEX tool. It takes the input as set of input specifications and gives the output as a complete c program. One of the other tool which is developed in java is JLEX.

**Procedure:**

1. Create a file with LEX specifications.

2. Save the above file with **.LEX or .l** extension.

3. Compile the program on LEX compiler using command. (lex <filename>).

4. Compile the C program which is generated by LEX tool.

**Program:**

%%

. ECHO;

%%

int yywrap(void) {

return 1;

}

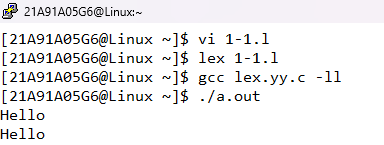
int main(void) {

yylex();

return 0;

}

**Output:**



**1.2) Write a lex program which removes white spaces from its input file.**

**Aim:** To write a lex program which removes white spaces from its input file.

**Description:** The FLEX tool is a modern version of LEX tool. It takes the input as set of input specifications and gives the output as a complete c program. One of the other tool which is developed in java is JLEX.

**Procedure:**

1. Create a file with LEX specifications.

2. Save the above file with **.LEX or .l** extension.

3. Compile the program on LEX compiler using command. (lex <filename>).

4. Compile the C program which is generated by LEX tool.

**Program:**

%{

#include<stdio.h>

%}

%%

[\n\t ' '] {};

%%

main()

{

yyin=fopen("myfile.txt","r");

yylex();

}

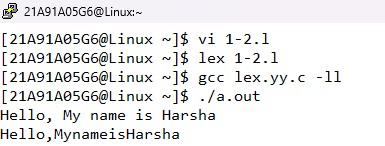
int yywrap()

{

return1;

}

**Output:**



**Week-2: Lexical analysis using lex tool**

**2.1) Write a lex program to identify the patterns in the input file.**

**Aim:** To write a lex program to identify the patterns in the input file.

**Description:** The FLEX tool is a modern version of LEX tool. It takes the input as set of input specifications and gives the output as a complete c program. One of the other tool which is developed in java is JLEX.

**Procedure:**

1. Create a file with LEX specifications.

2. Save the above file with **.LEX or .l** extension.

3. Compile the program on LEX compiler using command. (lex <filename>).

4. Compile the C program which is generated by LEX tool.

**Program:**

%{

#include<stdio.h>

%}

%%

["int""char""for""if""while""then""return""do"] {printf("keyword : %s\n");}

[\*%+\-] {printf("Operator : %s ", yytext);}

[(){};] {printf("Special Character: %s\n", yytext);}

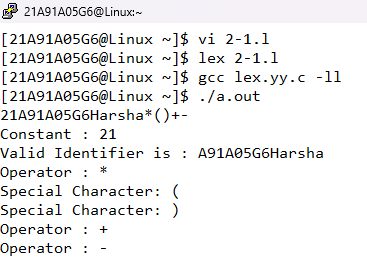
[0-9]+ {printf("Constant : %s\n", yytext);}

[a-zA-Z\_][a-zA-Z0-9\_]\* {printf("Valid Identifier is : %s\n", yytext);}

^[^a-zA-Z\_] {printf("Invalid Indentifier \n");}

%%

**Output:**



**2.2) Design a lexical analyzer for given language and the lexical analyzer should ignore redundant spaces, tabs and new lines.**

**Aim:** To implement a lexical analyzer for given language and the lexical analyzer should ignore redundant spaces, tabs and new lines.

**Description:** Lexical analyzer reads characters from the input, groups them into lexemes and passes the tokens formed by the lexemes. Lexical analysis is the processing of an input sequence of characters to produce, as output, a sequence of symbols called lexical tokens or just "tokens". For example, lexers for many programming languages convert the character sequence 123 abc into two tokens: 123 and abc (white space is not a token in most languages). The purpose of producing these tokens is usually to forward them.

**Program:**

%{

#include<stdio.h>

int i=0,id=0;

%}

%%

[#].\*[<].\*[>]\n {}

[ \t\n]+ {}

\/\/.\*\n {}

\/\\*(.\*\n)\*.\*\\*\/ {}

auto|break|case|char|const|continue|default|do|double|else|enum|extern|float|for|goto|if|int|long|register|return|short|signed|sizeof|static|struct|switch|typedef|union|unsigned|void|volatile|while {printf("token: %d < keyword, %s >\n",++i,yytext);}

[+\-\\*\/%<>] {printf("token: %d < operator, %s >\n",++i,yytext);}

[();{}] {printf("token: %d < special char, %s >\n",++i,yytext);}

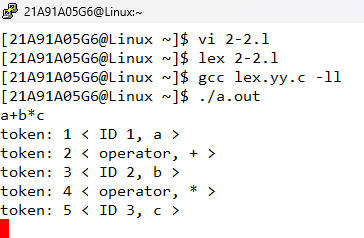
[0-9]+ {printf("token: %d < constant, %s >\n",++i,yytext);}

[a-zA-Z\_][a-zA-Z0-9\_]\* {printf("token: %d < ID %d, %s >\n",++i,++id,yytext);}

^[^a-zA-Z\_] {printf("ERROR INVALID TOKEN %s\n",yytext);}

%%

**Output:**



**Week-3: First and Follow**

**3.1) Simulate First and Follow of a Grammar**

**Aim:** To implement First and Follow of a Grammar.

**Description:**

**Rules for calculating FIRST:**

1. If X is a terminal symbol, then FIRST(X)={X}

2. If X 🡪 ɛ is a production, then FIRST(X)={ɛ}

3. If X is a nonterminal and X->Y1Y2…Yk is a production for some k>=1, then place a in First(X) if for some i a is in First(Yi) and ɛ is in all of First(Y1),…,First(Yi-1) that is Y1…Yi-1 => ɛ. If ɛ is in First(Yj) for j=1,…,k then add ɛ to First(X).

**Rules for calculating FOLLOW:**

1. If S is the start symbol, then add $ to FOLLOW(S).

2. If A 🡪 αB is a production then FOLLOW(B) =FIRST(β) if FIRST(β) contains ɛ then add FOLLOW(A) to FOLLOW(B).

3. If A🡪 αB is a production FOLLOW(B) = FOLLOW(A).

We apply these rules until nothing more can be added to any follow set.

**Program:**

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

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

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

void follow(char c){

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

f[m++] = '$';

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

for(j = 2;j<strlen(a[i]);j++){

if(a[i][j] == c){

if(a[i][j+1] != '\0') first(a[i][j+1]);

if(a[i][j+1] == '\0' && c != a[i][0]) follow(a[i][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][2] == '$') follow(a[i][0]);

else if(islower(a[k][2])) f[m++] = a[k][2];

else first(a[k][2]);

}

}

}

int main(){

int i, z;

char c, ch;

printf("enter the no. of productions:");

scanf("%d", &n);

printf("enter the productions(epsilon = $):\n");

for(i = 0; i < n; i++) scanf("%s%c", a[i], &ch);

do{

m = 0;

printf("enter the element whose FIRST & FOLLOW is to be found:");

scanf("%c", &c);

first(c);

printf("FIRST(%c) = {", c);

for(i = 0;i<m;i++) printf("%c", f[i]);

printf("}\n");

follow(c);

printf("FOLLOW(%c) = {", c);

for(;i<m;i++) printf("%c", f[i]);

printf("}\n");

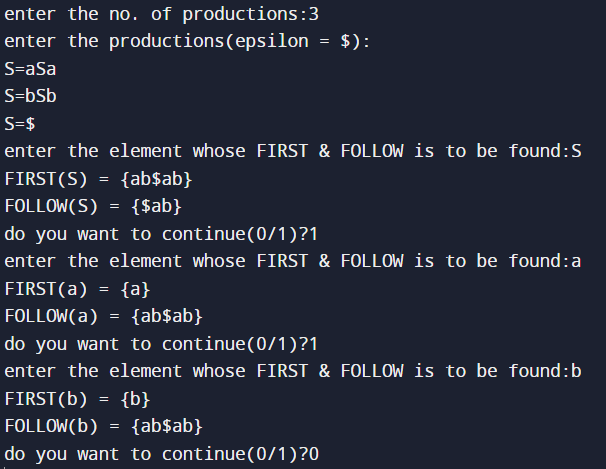
printf("do you want to continue(0/1)?");

scanf("%d%c", &z, &ch);

}while(z == 1);

}

**Output:**



**3.2) Implement the lexical analyzer using JLex, flex or lex or other lexical analyzer generating tools.**

**Aim:** To implement the lexical analyzer using JLex, flex or lex or other lexical analyzer generating tools.

**Description:** The FLEX tool is a modern version of LEX tool. It takes the input as set of input specifications and gives the output as a complete c program. One of the other tool which is developed in java is JLEX.

a. Create a file with LEX specifications.

b. Save the above file with **.LEX or .l** extension.

c. Compile the program on LEX compiler using command. (lex <filename>).

d. Compile the C program which is generated by LEX tool.

**Program:**

%{

#include<stdio.h>

int i=0,id=0;

%}

%%

[#].\*[<].\*[>]\n {}

[ \t\n]+ {}

\/\/.\*\n {}

\/\\*(.\*\n)\*.\*\\*\/ {}

auto|break|case|char|const|continue|default|do|double|else|enum|extern|float|for|goto|if|int|long|register|return|short|signed|sizeof|static|struct|switch|typedef|union|unsigned|void|volatile|while {printf("token: %d < keyword, %s >\n",++i,yytext);}

[+\-\\*\/%<>] {printf("token: %d < operator, %s >\n",++i,yytext);}

[();{}] {printf("token: %d < special char, %s >\n",++i,yytext);}

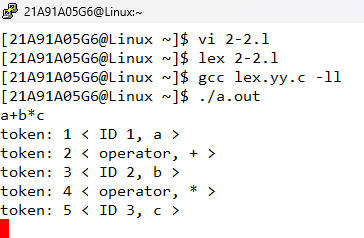
[0-9]+ {printf("token: %d < constant, %s >\n",++i,yytext);}

[a-zA-Z\_][a-zA-Z0-9\_]\* {printf("token: %d < ID %d, %s >\n",++i,++id,yytext);}

^[^a-zA-Z\_] {printf("ERROR INVALID TOKEN %s\n",yytext);}

%%

**Output:**



**Week-4:** **Top-Down Parsing**

**4.1) Develop an operator precedence parser for a given language.**

**Aim:** To develop an operator precedence parser for a given language.

**Description:** Bottom-up parsers for a large class of context free grammars can be easily developed using operator grammars. Operator grammars have the property that no production rule can have:

* ɛ at the right side
* two adjacent non terminals at the right side.

This property enables the implementation of efficient operator

precedence parsers. These

parser rely on the following three precedence rela tions:

Relation Meaning:

a <· b 🡪 a yields precedence to b

a =· b 🡪 a has the same precedence as b

a ·> b 🡪 a takes precedence over b

**Program:**

#include<stdio.h>

#include<string.h>

char stack[20],temp;

int top=-1;

void push(char item){

if(top>=20){

printf("STACK OVERFLOW");

return;

}

stack[++top]=item;

}

char pop(){

if(top<=-1){

printf("STACK UNDERFLOW");

return;

}

char c;

c=stack[top--];

printf("Popped element:%c\n",c);

return c;

}

char TOS(){

return stack[top];

}

int convert(char item){

switch(item){

case 'i':return 0;

case '+':return 1;

case '\*':return 2;

case '$':return 3;

}

}

int main(){

char pt[4][4]={

{'-','>','>','>'},

{'<','>','<','>'},

{'<','>','>','>'},

{'<','<','<','1'}};

char input[20];

int lkh=0;

printf("Enter input with $ at the end\n");

scanf("%s",input);

push('$');

while(lkh<=strlen(input)){

if(TOS()=='$'&&input[lkh]=='$'){

printf("SUCCESS\n");

return 1;

}

else if(pt[convert(TOS())][convert(input[lkh])]=='<'){

push(input[lkh]);

printf("Push---%c\n",input[lkh]);

lkh++;

}

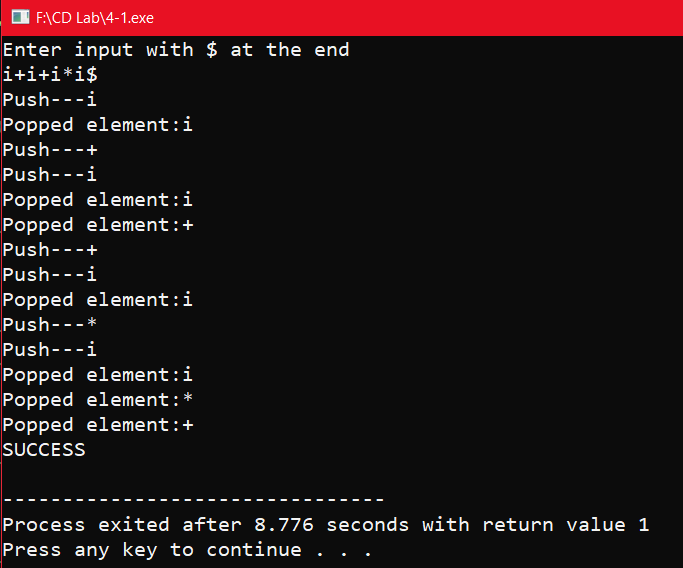
else pop();

}

return 0;

}

**Output:**



**4.2) Construct a recursive descent parser for an expression.**

**Aim:** To construct a recursive descent parser for an expression.

**Description:** A top-down parser that executes a set of recursive procedures to process the input without backtracking is called Recursive Descent Parser. Typically, top down parsers are implemented as a set of recursive functions that descent through a parse tree for a string. This approach is known as recursive descent parsing.

**Program:**

#include<stdio.h>

#include<ctype.h>

#include<string.h>

void Tp();

void Ep();

void E();

void T();

void check();

int count,flag;

char expr[10];

int main(){

count=0;

flag=0;

printf("\nEnter an Algebraic Expression:\t");

scanf("%s",expr);

E();

if((strlen(expr)==count)&&(flag==0))

printf("\nThe expression %s is valid\n",expr);

else

printf("\nThe expression %s is invalid\n",expr);

return 0;

}

void E(){

T();

Ep();

}

void T(){

check();

Tp();

}

void Tp(){

if(expr[count]=='\*'){

count++;

check();

Tp();

}

}

void check(){

if(isalnum(expr[count]))

count++;

else if(expr[count]=='('){

count++;

E();

if(expr[count]==')') count++;

else flag=1;

}

else flag=1;

}

void Ep(){

if(expr[count]=='+'){

count++;

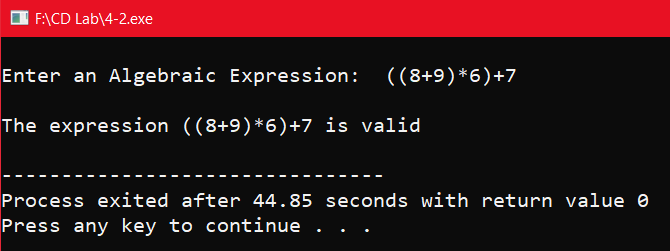
T();

Ep();

}

}

**Output:**



**Week-5:** **Bottom-up Parsing**

**5.1) Construct an LL(1) parser for an expression.**

**Aim:** To construct an LL(1) parser for an expression.

**Description:** A grammar whose parsing table has no multiply defined entries is said to be LL(1). It can be shown that the above algorithm can be used to produce for every LL(1) grammar G a parsing table M that parses all and only the sentences of G. LL(1) grammars have several distinctive properties. No ambiguous or left recursive grammar can be LL(1). There remains a question of what should be done in case of multiply defined entries. One easy solution is to eliminate all left recursion and left factoring, hoping to produce a grammar which will produce no multiply defined entries in the parse tables. Unfortunately there are some grammars which will give an LL(1) grammar after any kind of alteration. In general, there are no universal rules to convert multiply defined entries into single valued entries without affecting the language recognized by the parser.

**Program:**

#include<stdio.h>

#include<string.h>

int stack[20],top=-1;

void push(int item){

if(top>=20){

printf("stack overflow");

return;

}

stack[++top]=item;

}

int pop(){

int ch;

if(top<=-1){

printf("underflow");

return;

}

ch=stack[top--];

return ch;

}

char convert(int item){

char ch;

switch(item){

case 0:return('E');

case 1:return('e');

case 2:return('T');

case 3:return('t');

case 4:return('F');

case 5:return('i');

case 6:return('+');

case 7:return('\*');

case 8:return('(');

case 9:return(')');

case 10:return('$');

}

}

void main(){

int m[10][10],i,j,k;

char ips[20];

int ip[10],a,b,t;

m[0][0]=m[0][3]=21;

m[1][1]=621;

m[1][4]=m[1][5]=-2;

m[2][0]=m[2][3]=43;

m[3][1]=m[3][4]=m[3][5]=-2;

m[3][2]=743;

m[4][0]=5;

m[4][3]=809;

printf("\nenter the input string with $ at the end (Ex: i+i\*i$): ");

scanf("%s",ips);

for(i=0;i<strlen(ips);i++){

switch(ips[i]){

case 'E':k=0;break;

case 'e':k=1;break;

case 'T':k=2;break;

case 't':k=3;break;

case 'F':k=4;break;

case 'i':k=5;break;

case '+':k=6;break;

case '\*':k=7;break;

case '(':k=8;break;

case ')':k=9;break;

case '$':k=10;break;

}

ip[i]=k;

}

ip[i]=-1;

push(10);

push(0);

i=0;

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

while(1){

printf("\t");

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

printf("%c",convert(stack[j]));

printf("\t\t");

for(k=i;ip[k]!=-1;k++)

printf("%c",convert(ip[k]));

printf("\n");

if(stack[top]==ip[i]){

if(ip[i]==10){

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

return;

}

else{

top--;

i++;

}

}

else if(stack[top]<=4&&stack[top]>=0){

a=stack[top];

b=ip[i]-5;

t=m[a][b];

top--;

while(t>0){

push(t%10);

t=t/10;

}

}

else{

printf("error\n");

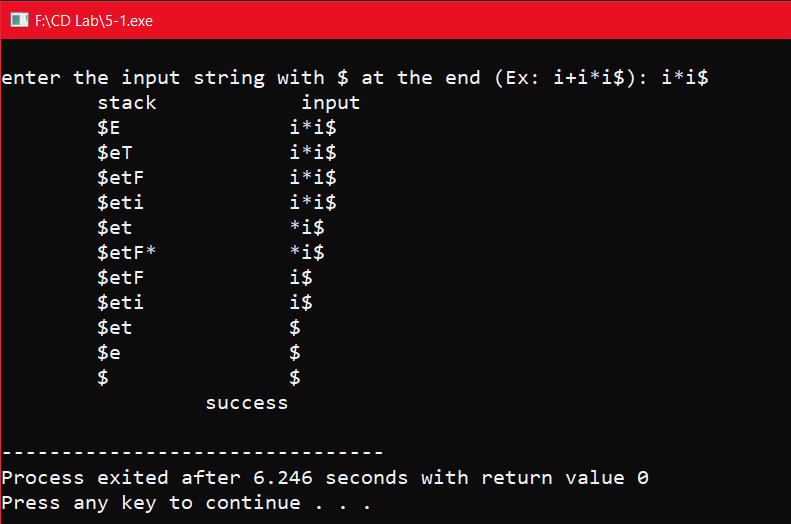
return;

}

}

}

**Output:**



**5.2) Design a LALR bottom up parser for the given language.**

**Aim:** To design a LALR bottom up parser for the given language.

**Description:** LALR Parser is lookahead LR parser. It is the most powerful parser which can handle large classes of grammar. The size of CLR parsing table is quite large as compared to other parsing table. LALR reduces the size of this table.LALR works similar to CLR. The only difference is , it combines the similar states of CLR parsing table into one single state.

The general syntax becomes [A->∝.B, a ]

where A->∝.B is production and a is a terminal or right end marker $

LR(1) items=LR(0) items + look ahead

**Program:**

#include<stdio.h>

#include<string.h>

int st[20],top=-1;

char input[20];

int encode(char ch){

switch(ch) {

case 'i':return 0;

case '+':return 1;

case '\*':return 2;

case '(':return 3;

case ')':return 4;

case '$':return 5;

case 'E':return 6;

case 'T':return 7;

case 'F':return 8;

}

return -1;

}

char decode(int n){

switch(n){

case 0:return('i');

case 1:return('+');

case 2:return('\*');

case 3:return('(');

case 4:return(')');

case 5:return('$');

case 6:return('E');

case 7:return('T');

case 8:return('F');

}

return 'z';

}

void push(int n){

st[++top]=n;

}

int pop(){

return(st[top--]);

}

void display(int p,char \*ptr){

int l;

for(l=0;l<=top;l++){

if(l%2==1)

printf("%c",decode(st[l]));

else

printf("%d",st[l]);

}

printf("\t");

for(l=p;ptr[l];l++)

printf("%c",ptr[l]);

printf("\n");

}

int main(){

char t1[20][20],pr[20][20],xy;

int inp[20],t2[20][20],gt[20][20],i,k,x,y,tx=0,ty=0,len;

strcpy(pr[1],"E E+T");

strcpy(pr[2],"E T");

strcpy(pr[3],"T T\*F");

strcpy(pr[4],"T F");

strcpy(pr[5],"F (E)");

strcpy(pr[6],"F i");

t2[2][1]=t2[2][4]=t2[2][5]=2;

t2[3][1]=t2[3][2]=t2[3][4]=t2[3][5]=4;

t2[5][1]=t2[5][2]=t2[5][4]=t2[5][5]=6;

t2[9][1]=t2[9][4]=t2[9][5]=1;

t2[10][1]=t2[10][2]=t2[10][4]=t2[10][5]=3;

t2[11][2]=t2[11][1]=t2[11][4]=t2[11][5]=5;

t1[2][1]=t1[2][4]=t1[2][5]='r';

t1[3][1]=t1[3][2]=t1[3][4]='r';

t1[3][5]=t1[5][1]=t1[5][2]='r';

t1[5][4]=t1[5][5]=t1[9][1]=t1[9][4]='r';

t1[9][5]=t1[10][1]=t1[10][2]=t1[10][4]=t1[10][5]='r';

t1[11][1]=t1[11][4]=t1[11][2]=t1[11][5]='r';

t1[0][0]=t1[4][0]=t1[6][0]=t1[7][0]=t1[0][3]=t1[4][3]=t1[6][3]='s';

t1[2][2]=t1[9][2]=t1[8][4]=t1[1][1]=t1[8][1]=t1[7][3]='s';

t1[1][5]='a';

t2[0][0]=t2[4][0]=t2[6][0]=t2[7][0]=5;

t2[0][3]=t2[4][3]=t2[6][3]=t2[7][3]=4;

t2[2][2]=t2[9][2]=7;

t2[8][4]=11;

t2[1][1]=t2[8][1]=6;

gt[0][6]=1;

gt[0][7]=gt[4][7]=2;

gt[0][8]=gt[4][8]=gt[6][8]=3;

gt[4][6]=8;gt[6][7]=9;gt[7][8]=10;

printf("Enter String: ");

scanf("%s",input);

for(k=0;input[k];k++){

inp[k]=encode(input[k]);

if(input[k]<0||inp[k]>5)

printf("\n error in input");

}

push(0);

i=0;

while(1){

x=st[top];y=inp[i];

display(i,input);

if(t1[x][y]=='a'){

printf("String is Accepted \n");

return 0;

}

else if(t1[x][y]=='s'){

push(inp[i]);

push(t2[x][y]);

i++;

}

else if(t1[x][y]=='r'){

len=strlen(pr[t2[x][y]])-2;

xy=pr[t2[x][y]][0];

ty=encode(xy);

for(k=1;k<=2\*len;k++) pop();

tx=st[top];

push(ty);

push(gt[tx][ty]);

}

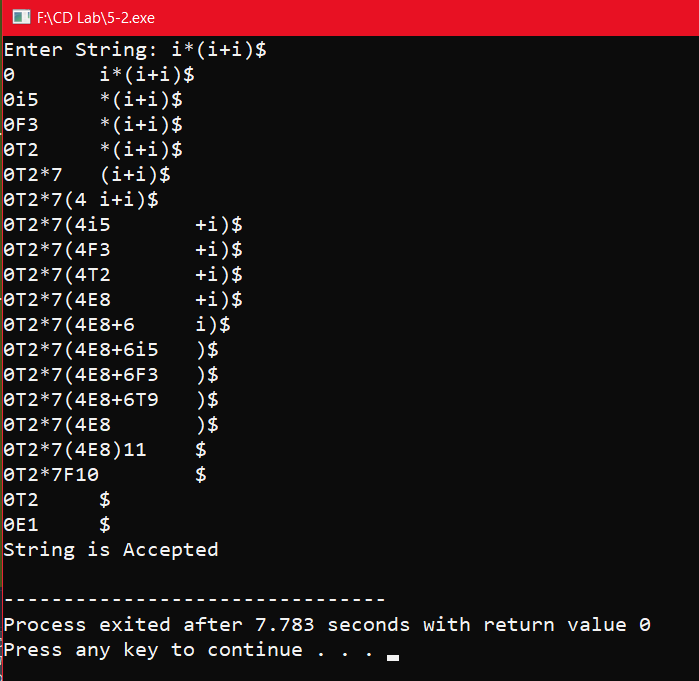
else

printf("\n error in parsing");

}

}

**Output:**



**Week-6: Optimization Phase**

**6.1) Write a program to perform loop unrolling.**

**Aim:** To write a program to perform loop unrolling.

**Description:** Loop unrolling is a loop transformation technique that helps to optimize the execution time of a program. We basically remove or reduce iterations. Loop unrolling increases the program’s speed by eliminating loop control instruction and loop test instructions.

This increases Program Efficiency, Reduces Loop Overhead, Parallel Execution.

**Program:**

#include<stdio.h>

#define TOGETHER (8)

int main(void){

int i = 0,entries = 15,repeat,left = 0;

repeat = (entries / TOGETHER);

left = (entries % TOGETHER);

while (repeat--){

printf("process(%d)\n", i);

printf("process(%d)\n", i + 1);

printf("process(%d)\n", i + 2);

printf("process(%d)\n", i + 3);

printf("process(%d)\n", i + 4);

printf("process(%d)\n", i + 5);

printf("process(%d)\n", i + 6);

printf("process(%d)\n", i + 7);

i += TOGETHER;

}

switch (left){

case 7 : printf("process(%d)\n", i + 6);

case 6 : printf("process(%d)\n", i + 5);

case 5 : printf("process(%d)\n", i + 4);

case 4 : printf("process(%d)\n", i + 3);

case 3 : printf("process(%d)\n", i + 2);

case 2 : printf("process(%d)\n", i + 1);

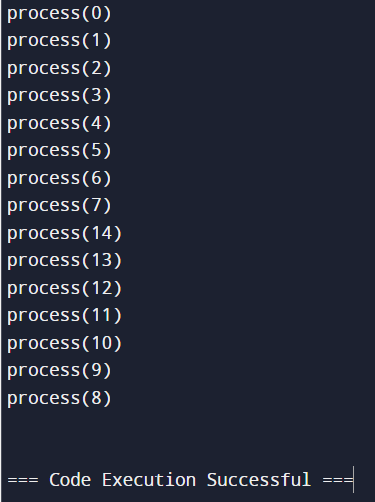
case 1 : printf("process(%d)\n", i);

case 0 : ;

}

}

**Output:**



**6.2) Write a program for constant propagation.**

**Aim:** To write a program for constant propagation.

**Description:** Constant Propagation is one of the local code optimization techniques in Compiler Design. It can be defined as the process of replacing the constant value of variables in the expression. In simpler words, we can say that if some value is assigned a known constant, than we can simply replace the that value by constant. Constants assigned to a variable can be propagated through the flow graph and can be replaced when the variable is used.

**Program:**

#include<stdio.h>

#include<string.h>

#include<stdlib.h>

#include<ctype.h>

void input();

void output();

void change(int p,char \*res);

void constant();

struct expr{

char op[2],op1[5],op2[5],res[5];

int flag;

}arr[10];

int n;

void main(){

input();

constant();

output();

}

void input(){

int i;

printf("\n\nEnter the maximum number of expressions : ");

scanf("%d",&n);

printf("\nEnter the input : \n");

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

scanf("%s",arr[i].op);

scanf("%s",arr[i].op1);

scanf("%s",arr[i].op2);

scanf("%s",arr[i].res);

arr[i].flag=0;

}

}

void constant(){

int i;

int op1,op2,res;

char op,res1[5];

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

if(isdigit(arr[i].op1[0]) && isdigit(arr[i].op2[0]) || strcmp(arr[i].op,"=")==0){

op1=atoi(arr[i].op1);

op2=atoi(arr[i].op2);

op=arr[i].op[0];

switch(op){

case '+':res=op1+op2;

break;

case '-':res=op1-op2;

break;

case '\*':res=op1\*op2;

break;

case '/':res=op1/op2;

break;

case '=':res=op1;

break;

}

sprintf(res1,"%d",res);

arr[i].flag=1;

change(i,res1);

}

}

}

void output(){

int i=0;

printf("\nOptimized code is : ");

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

if(!arr[i].flag)

printf("\n%s %s %s %s",arr[i].op,arr[i].op1,arr[i].op2,arr[i].res);

}

}

void change(int p,char \*res){

int i;

for(i=p+1;i<n;i++){

if(strcmp(arr[p].res,arr[i].op1)==0)

strcpy(arr[i].op1,res);

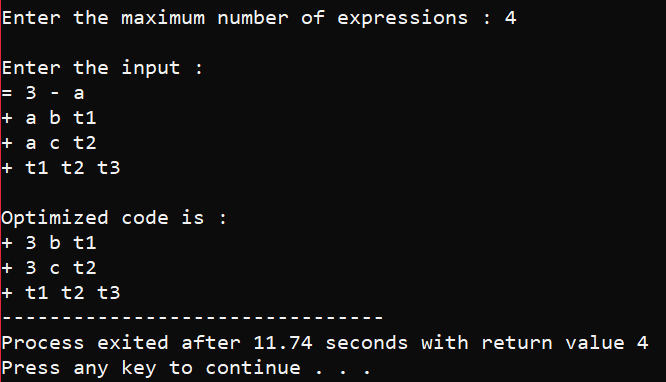
else if(strcmp(arr[p].res,arr[i].op2)==0)

strcpy(arr[i].op2,res);

}

}

**Output:**



**AUGMENTED EXPERIMENTS:**

**13) Write a C program to identify whether a given line is a comment or not.**

**Program:**

#include<stdio.h>

#include<stdlib.h>

void main() {

char com[30];

int i=2,a=0;

printf("\nEnter comment: ");

gets(com);

if(com[0]=='/') {

if(com[1]=='/')

printf("\n It is a comment");

else if(com[1]=='\*') {

for(i=2;i<=30;i++){

if(com[i]=='\*'&&com[i+1]=='/'){

printf("\n It is a comment");

a=1;

break;

}

else

continue;

}

if(a==0)

printf("\n It is not a comment");

}

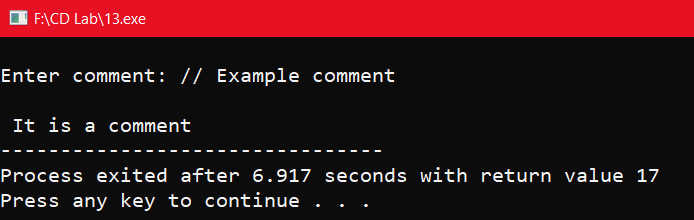
else printf("\n It is not a comment");

}

else printf("\n It is not a comment");

}

**Output:**



**14) Write a C program to simulate lexical analyzer for validating operators.**

**Program:**

#include<stdio.h>

#include<stdlib.h>

void main(){

char s[5];

printf("\nEnter any operator: ");

gets(s);

switch(s[0]){

case'>': if(s[1]=='=') printf("\n Greater than or equal");

else printf("\n Greater than");

break;

case'<': if(s[1]=='=') printf("\n Less than or equal");

else printf("\nLess than");

break;

case'=': if(s[1]=='=') printf("\nEqual to");

else printf("\nAssignment");

break;

case'!': if(s[1]=='=') printf("\nNot Equal");

else printf("\n Bit Not");

break;

case'&': if(s[1]=='&') printf("\nLogical AND");

else printf("\n Bitwise AND");

break;

case'|': if(s[1]=='|') printf("\nLogical OR");

else printf("\nBitwise OR");

break;

case'+': printf("\n Addition");

break;

case'-': printf("\nSubstraction");

break;

case'\*': printf("\nMultiplication");

break;

case'/': printf("\nDivision");

break;

case'%': printf("Modulus");

break;

default: printf("\n Not a operator");

}

}

**Output:**

