**COLLEGE OF APPLIED BUSINESS AND TECHNOLOGY**

Kathmandu, Nepal

(Affiliated to Tribhuvan University)

****

**Practical Report**

**Compiler Design and Construction**

Submitted by: Submitted to:

Name: Lotus Khetri (114) Mr.Ashok Pandey

Semester: Six CDC Instructor

Faculty: Science and Technology

Level: Bachelor

Program: Computer Science and Information Technology

July, 2024

**INDEX**

|  |  |  |
| --- | --- | --- |
| **S.N** | **Title of Experiment** | **Sign** |
| 1 | Write a program to implement Lexical Analyzer to identify token. |  |
| 2 | Write a program to implement First of grammar. |  |
| 3 | Write a program to implement Follow of grammar. |  |
| 4 | Write a program to implement Follow of grammar. |  |
| 5 | Write a program to implement LR Parser. |  |
| 6 | Write a program to implement Intermediate code generation. |  |
| 7 | Write a program to implement Final code generation. |  |
| 8 | Write a program to implement Type Conversion. |  |
| 9 | Write a program to check whether a given identifier is valid or not. |  |
| 10 | Write a program to check whether a given string is within valid comment section or not. |  |

**Experiment 1**

**Write a program to implement Lexical Analyzer to identify token.**

**Source Code:**

#include <stdbool.h>

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

// Returns 'true' if the character is a DELIMITER.

bool isDelimiter(char ch)

{

if (ch == ' ' || ch == '+' || ch == '-' || ch == '\*' ||

ch == '/' || ch == ',' || ch == ';' || ch == '>' ||

ch == '<' || ch == '=' || ch == '(' || ch == ')' ||

ch == '[' || ch == ']' || ch == '{' || ch == '}')

return (true);

return (false);

}

// Returns 'true' if the character is an OPERATOR.

bool isOperator(char ch)

{

if (ch == '+' || ch == '-' || ch == '\*' ||

ch == '/' || ch == '>' || ch == '<' ||

ch == '=')

return (true);

return (false);

}

// Returns 'true' if the string is a VALID IDENTIFIER.

bool validIdentifier(char\* str)

{

if (str[0] == '0' || str[0] == '1' || str[0] == '2' ||

str[0] == '3' || str[0] == '4' || str[0] == '5' ||

str[0] == '6' || str[0] == '7' || str[0] == '8' ||

str[0] == '9' || isDelimiter(str[0]) == true)

return (false);

return (true);

}

// Returns 'true' if the string is a KEYWORD.

bool isKeyword(char\* str)

{

if (!strcmp(str, "if") || !strcmp(str, "else") ||

!strcmp(str, "while") || !strcmp(str, "do") ||

!strcmp(str, "break") ||

!strcmp(str, "continue") || !strcmp(str, "int")

|| !strcmp(str, "double") || !strcmp(str, "float")

|| !strcmp(str, "return") || !strcmp(str, "char")

|| !strcmp(str, "case") || !strcmp(str, "char")

|| !strcmp(str, "sizeof") || !strcmp(str, "long")

|| !strcmp(str, "short") || !strcmp(str, "typedef")

|| !strcmp(str, "switch") || !strcmp(str, "unsigned")

|| !strcmp(str, "void") || !strcmp(str, "static")

|| !strcmp(str, "struct") || !strcmp(str, "goto"))

return (true);

return (false);

}

// Returns 'true' if the string is an INTEGER.

bool isInteger(char\* str)

{

int i, len = strlen(str);

if (len == 0)

return (false);

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

if (str[i] != '0' && str[i] != '1' && str[i] != '2'

&& str[i] != '3' && str[i] != '4' && str[i] != '5'

&& str[i] != '6' && str[i] != '7' && str[i] != '8'

&& str[i] != '9' || (str[i] == '-' && i > 0))

return (false);

}

return (true);

}

// Returns 'true' if the string is a REAL NUMBER.

bool isRealNumber(char\* str)

{

int i, len = strlen(str);

bool hasDecimal = false;

if (len == 0)

return (false);

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

if (str[i] != '0' && str[i] != '1' && str[i] != '2'

&& str[i] != '3' && str[i] != '4' && str[i] != '5'

&& str[i] != '6' && str[i] != '7' && str[i] != '8'

&& str[i] != '9' && str[i] != '.' ||

(str[i] == '-' && i > 0))

return (false);

if (str[i] == '.')

hasDecimal = true;

}

return (hasDecimal);

}

// Extracts the SUBSTRING.

char\* subString(char\* str, int left, int right)

{

int i;

char\* subStr = (char\*)malloc(

sizeof(char) \* (right - left + 2));

for (i = left; i <= right; i++)

subStr[i - left] = str[i];

subStr[right - left + 1] = '\0';

return (subStr);

}

// Parsing the input STRING.

void parse(char\* str)

{

int left = 0, right = 0;

int len = strlen(str);

while (right <= len && left <= right) {

if (isDelimiter(str[right]) == false)

right++;

if (isDelimiter(str[right]) == true && left == right) {

if (isOperator(str[right]) == true)

printf("'%c' IS AN OPERATOR\n", str[right]);

right++;

left = right;

} else if (isDelimiter(str[right]) == true && left != right

|| (right == len && left != right)) {

char\* subStr = subString(str, left, right - 1);

if (isKeyword(subStr) == true)

printf("'%s' IS A KEYWORD\n", subStr);

else if (isInteger(subStr) == true)

printf("'%s' IS AN INTEGER\n", subStr);

else if (isRealNumber(subStr) == true)

printf("'%s' IS A REAL NUMBER\n", subStr);

else if (validIdentifier(subStr) == true

&& isDelimiter(str[right - 1]) == false)

printf("'%s' IS A VALID IDENTIFIER\n", subStr);

else if (validIdentifier(subStr) == false

&& isDelimiter(str[right - 1]) == false)

printf("'%s' IS NOT A VALID IDENTIFIER\n", subStr);

left = right;

}

}

return;

}

// DRIVER FUNCTION

int main()

{

char str[100]; // Maximum length of string is 100

printf("Enter a string: ");

// Use fgets to read the input including spaces, up to 99 characters

if (fgets(str, sizeof(str), stdin) != NULL) {

// Remove the trailing newline character, if present

size\_t len = strlen(str);

if (len > 0 && str[len-1] == '\n') {

str[len-1] = '\0';

}

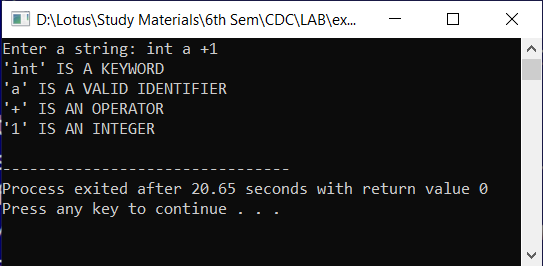
}

parse(str); // Calling the parse function

return 0;

}

**Output:**



**Experiment 2**

**Write a program to implement First of grammar**

**Source Code:**

#include <ctype.h>

#include <stdio.h>

#include <string.h>

// Function to calculate First

void findfirst(char, int, int);

int count, n = 0;

// Stores the final result of the First Sets

char calc\_first[10][100];

// Stores the production rules

char production[10][10];

char first[10];

int k;

char ck;

int e;

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

int jm = 0;

int i;

char c;

count = 8;

// The Input grammar

strcpy(production[0], "X=TnS");

strcpy(production[1], "X=Rm");

strcpy(production[2], "T=q");

strcpy(production[3], "T=#");

strcpy(production[4], "S=p");

strcpy(production[5], "S=#");

strcpy(production[6], "R=om");

strcpy(production[7], "R=ST");

int kay;

char done[count];

int ptr = -1;

// Initializing the calc\_first array

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;

// Checking if First of c has already been calculated

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

if (c == done[kay])

xxx = 1;

if (xxx == 1)

continue;

// Function call

findfirst(c, 0, 0);

ptr += 1;

// Adding c to the calculated list

done[ptr] = c;

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

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

// Printing the First Sets of the grammar

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++;

}

return 0;

}

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

int j;

// The case where we encounter a Terminal

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)) {

// Recursion to calculate First of New

// Non-Terminal we encounter after epsilon

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

} else {

first[n++] = '#';

}

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

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

} else {

// Recursion to calculate First of

// New Non-Terminal we encounter at the beginning

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

}

}

}

}

**Output:**

A screenshot of a computer program

Description automatically generated

**Experiment 3**

**Write a program to implement Follow of grammar.**

**Source Code**

#include <ctype.h>

#include <stdio.h>

#include <string.h>

// Functions to calculate Follow

void followfirst(char, int, int);

void follow(char c);

int count, n = 0;

// Stores the final result of the Follow Sets

char calc\_follow[10][100];

int m = 0;

// Stores the production rules

char production[10][10];

char f[10];

char ck;

int e;

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

{

int km = 0;

int i;

count = 8;

// The Input grammar

strcpy(production[0], "X=TnS");

strcpy(production[1], "X=Rm");

strcpy(production[2], "T=q");

strcpy(production[3], "T=#");

strcpy(production[4], "S=p");

strcpy(production[5], "S=#");

strcpy(production[6], "R=om");

strcpy(production[7], "R=ST");

int kay;

char done[count];

int ptr = -1;

// Initializing the calc\_follow array

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

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

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

}

}

int point1 = 0;

int point2;

int xxx;

int land = 0;

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

ck = production[e][0];

point2 = 0;

xxx = 0;

// Checking if Follow of ck has already been calculated

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

if (ck == done[kay])

xxx = 1;

if (xxx == 1)

continue;

land += 1;

// Function call

follow(ck);

ptr += 1;

// Adding ck to the calculated list

done[ptr] = ck;

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

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

// Printing the Follow Sets of the grammar

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++;

}

}

void follow(char c)

{

int i, j;

// Adding "$" to the follow set of the start symbol

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') {

// Calculate the first of the next Non-Terminal in the production

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

}

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

// Calculate the follow of the Non-Terminal in the L.H.S. of the production

follow(production[i][0]);

}

}

}

}

}

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

{

int k;

// The case where we encounter a Terminal

if (!(isupper(c)))

f[m++] = c;

else {

int i = 0, j = 1;

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

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

break;

}

// Including the First set of the Non-Terminal in the Follow of the original query

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

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

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

} else {

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

// Case where we reach the end of a production

follow(production[c1][0]);

} else {

// Recursion to the next symbol in case we encounter a "#"

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

}

}

j++;

}

}

}

**Output:**

A screenshot of a computer program

Description automatically generated

**Experiment 4**

**Write a program to implement Shift Reduce Parser**.

**Source Code:**

//Including Libraries

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

//Global Variables

int z = 0, i = 0, j = 0, c = 0;

// Modify array size to increase

// length of string to be parsed

char a[16], ac[20], stk[15], act[10];

// This Function will check whether

// the stack contain a production rule

// which is to be Reduce.

// Rules can be E->2E2 , E->3E3 , E->4

void check()

{

// Copying string to be printed as action

strcpy(ac,"REDUCE TO E -> ");

// c=length of input string

for(z = 0; z < c; z++)

{

//checking for producing rule E->4

if(stk[z] == '4')

{

printf("%s4", ac);

stk[z] = 'E';

stk[z + 1] = '\0';

//printing action

printf("\n$%s\t%s$\t", stk, a);

}

}

for(z = 0; z < c - 2; z++)

{

//checking for another production

if(stk[z] == '2' && stk[z + 1] == 'E' &&

stk[z + 2] == '2')

{

printf("%s2E2", ac);

stk[z] = 'E';

stk[z + 1] = '\0';

stk[z + 2] = '\0';

printf("\n$%s\t%s$\t", stk, a);

i = i - 2;

}

}

for(z=0; z<c-2; z++)

{

//checking for E->3E3

if(stk[z] == '3' && stk[z + 1] == 'E' &&

stk[z + 2] == '3')

{

printf("%s3E3", ac);

stk[z]='E';

stk[z + 1]='\0';

stk[z + 1]='\0';

printf("\n$%s\t%s$\t", stk, a);

i = i - 2;

}

}

return ; //return to main

}

//Driver Function

int main()

{

printf("GRAMMAR is -\nE->2E2 \nE->3E3 \nE->4\n");

// a is input string

strcpy(a,"32423");

// strlen(a) will return the length of a to c

c=strlen(a);

// "SHIFT" is copied to act to be printed

strcpy(act,"SHIFT");

// This will print Labels (column name)

printf("\nstack \t input \t action");

// This will print the initial

// values of stack and input

printf("\n$\t%s$\t", a);

// This will Run upto length of input string

for(i = 0; j < c; i++, j++)

{

// Printing action

printf("%s", act);

// Pushing into stack

stk[i] = a[j];

stk[i + 1] = '\0';

// Moving the pointer

a[j]=' ';

// Printing action

printf("\n$%s\t%s$\t", stk, a);

// Call check function ..which will

// check the stack whether its contain

// any production or not

check();

}

// Rechecking last time if contain

// any valid production then it will

// replace otherwise invalid

check();

// if top of the stack is E(starting symbol)

// then it will accept the input

if(stk[0] == 'E' && stk[1] == '\0')

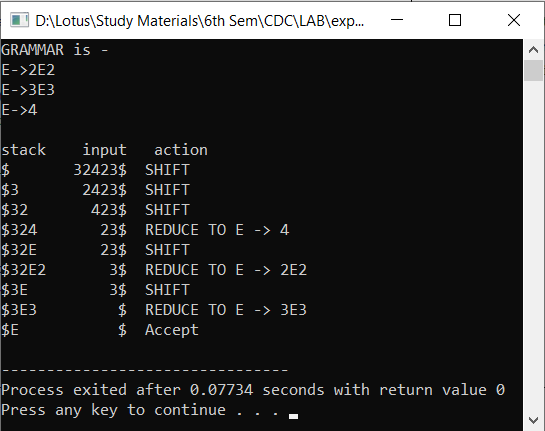
printf("Accept\n");

else //else reject

printf("Reject\n");

}

**Output:**



Experiment 5

Write a program to implement LR Parser.

**Source Code:**

//SLR paresr

#include<stdio.h>

#include<string.h>

int axn[][6][2]={

{{100,5},{-1,-1},{-1,-1},{100,4},{-1,-1},{-1,-1}},

{{-1,-1},{100,6},{-1,-1},{-1,-1},{-1,-1},{102,102}},

{{-1,-1},{101,2},{100,7},{-1,-1},{101,2},{101,2}},

{{-1,-1},{101,4},{101,4},{-1,-1},{101,4},{101,4}},

{{100,5},{-1,-1},{-1,-1},{100,4},{-1,-1},{-1,-1}},

{{-1,-1},{101,6},{101,6},{-1,-1},{101,6},{101,6}},

{{100,5},{-1,-1},{-1,-1},{100,4},{-1,-1},{-1,-1}},

{{100,5},{-1,-1},{-1,-1},{100,4},{-1,-1},{-1,-1}},

{{-1,-1},{100,6},{-1,-1},{-1,-1},{100,1},{-1,-1}},

{{-1,-1},{101,1},{100,7},{-1,-1},{101,1},{101,1}},

{{-1,-1},{101,3},{101,3},{-1,-1},{101,3},{101,3}},

{{-1,-1},{101,5},{101,5},{-1,-1},{101,5},{101,5}}

};//Axn Table

int gotot[12][3]={1,2,3,-1,-1,-1,-1,-1,-1,-1,-1,-1,8,2,3,-1,-1,-1,

-1,9,3,-1,-1,10,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1}; //GoTo table

int a[10];

char b[10];

int top=-1,btop=-1,i;

void push(int k)

{

if(top<9)

a[++top]=k;

}

void pushb(char k)

{

if(btop<9)

b[++btop]=k;

}

char TOS()

{

return a[top];

}

void pop()

{

if(top>=0)

top--;

}

void popb()

{

if(btop>=0)

b[btop--]='\0';

}

void display()

{

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

printf("%d%c",a[i],b[i]);

}

void display1(char p[],int m) //Displays The Present Input String

{

int l;

printf("\t\t");

for(l=m;p[l]!='\0';l++)

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

printf("\n");

}

void error()

{

printf("Syntax Error");

}

void reduce(int p)

{

int len,k,ad;

char src,\*dest;

switch(p)

{

case 1:dest="E+T";

src='E';

break;

case 2:dest="T";

src='E';

break;

case 3:dest="T\*F";

src='T';

break;

case 4:dest="F";

src='T';

break;

case 5:dest="(E)";

src='F';

break;

case 6:dest="i";

src='F';

break;

default:dest="\0";

src='\0';

break;

}

for(k=0;k<strlen(dest);k++)

{

pop();

popb();

}

pushb(src);

switch(src)

{

case 'E':ad=0;

break;

case 'T':ad=1;

break;

case 'F':ad=2;

break;

default: ad=-1;

break;

}

push(gotot[TOS()][ad]);

}

int main()

{

int j,st,ic;

char ip[20]="\0",an;

// clrscr();

printf("Enter any String\n");

+

scanf("%s",ip);

push(0);

display();

printf("\t%s\n",ip);

for(j=0;ip[j]!='\0';)

{

st=TOS();

an=ip[j];

if(an>='a'&&an<='z') ic=0;

else if(an=='+') ic=1;

else if(an=='\*') ic=2;

else if(an=='(') ic=3;

else if(an==')') ic=4;

else if(an=='$') ic=5;

else {

error();

break;

}

if(axn[st][ic][0]==100)

{

pushb(an);

push(axn[st][ic][1]);

display();

j++;

display1(ip,j);

}

if(axn[st][ic][0]==101)

{

reduce(axn[st][ic][1]);

display();

display1(ip,j);

}

if(axn[st][ic][1]==102)

{

printf("Given String is accepted \n");

// getch();

break;

}

/\* else

{

printf("Given String is rejected \n");

break;

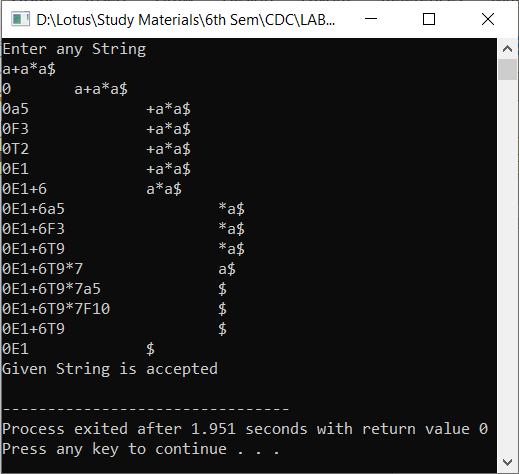
}\*/

}

return 0;

}

**Output:**



**Experiment 6**

**Write a program to implement Intermediate code generation.**

**Source Code:**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

// Struct Definitions

typedef struct ASTNode {

enum { BIN\_OP, NUM, VARIABLE } type;

union {

struct { struct ASTNode \*left; char op; struct ASTNode \*right; } bin\_op;

int num;

char \*var\_name;

} data;

} ASTNode;

typedef struct {

int temp\_counter;

int label\_counter;

char \*\*instructions;

int instruction\_count;

} IntermediateCodeGenerator;

// Function Prototypes

ASTNode\* create\_bin\_op(ASTNode \*left, char op, ASTNode \*right);

ASTNode\* create\_num(int value);

ASTNode\* create\_variable(char \*name);

IntermediateCodeGenerator\* create\_generator();

char\* generate\_temp(IntermediateCodeGenerator \*gen);

char\* generate\_label(IntermediateCodeGenerator \*gen);

void add\_instruction(IntermediateCodeGenerator \*gen, const char \*instruction);

void generate\_code(IntermediateCodeGenerator \*gen, ASTNode \*ast);

char\* visit(IntermediateCodeGenerator \*gen, ASTNode \*node);

char\* visit\_bin\_op(IntermediateCodeGenerator \*gen, ASTNode \*node);

void free\_ast(ASTNode \*node);

void free\_generator(IntermediateCodeGenerator \*gen);

ASTNode\* parse\_expression(const char \*expr, int \*pos);

// Function Implementations

ASTNode\* create\_bin\_op(ASTNode \*left, char op, ASTNode \*right) {

ASTNode \*node = (ASTNode\*)malloc(sizeof(ASTNode));

node->type = BIN\_OP;

node->data.bin\_op.left = left;

node->data.bin\_op.op = op;

node->data.bin\_op.right = right;

return node;

}

ASTNode\* create\_num(int value) {

ASTNode \*node = (ASTNode\*)malloc(sizeof(ASTNode));

node->type = NUM;

node->data.num = value;

return node;

}

ASTNode\* create\_variable(char \*name) {

ASTNode \*node = (ASTNode\*)malloc(sizeof(ASTNode));

node->type = VARIABLE;

node->data.var\_name = strdup(name);

return node;

}

IntermediateCodeGenerator\* create\_generator() {

IntermediateCodeGenerator \*gen = (IntermediateCodeGenerator\*)malloc(sizeof(IntermediateCodeGenerator));

gen->temp\_counter = 0;

gen->label\_counter = 0;

gen->instructions = NULL;

gen->instruction\_count = 0;

return gen;

}

char\* generate\_temp(IntermediateCodeGenerator \*gen) {

char \*temp = (char\*)malloc(10 \* sizeof(char));

sprintf(temp, "t%d", gen->temp\_counter++);

return temp;

}

char\* generate\_label(IntermediateCodeGenerator \*gen) {

char \*label = (char\*)malloc(10 \* sizeof(char));

sprintf(label, "L%d", gen->label\_counter++);

return label;

}

void add\_instruction(IntermediateCodeGenerator \*gen, const char \*instruction) {

gen->instructions = (char\*\*)realloc(gen->instructions, (gen->instruction\_count + 1) \* sizeof(char\*));

gen->instructions[gen->instruction\_count] = strdup(instruction);

gen->instruction\_count++;

}

void generate\_code(IntermediateCodeGenerator \*gen, ASTNode \*ast) {

visit(gen, ast);

}

char\* visit(IntermediateCodeGenerator \*gen, ASTNode \*node) {

if (node->type == BIN\_OP) {

return visit\_bin\_op(gen, node);

} else if (node->type == NUM) {

char \*buffer = (char\*)malloc(20 \* sizeof(char));

sprintf(buffer, "%d", node->data.num);

return buffer;

} else if (node->type == VARIABLE) {

return strdup(node->data.var\_name);

}

return NULL;

}

char\* visit\_bin\_op(IntermediateCodeGenerator \*gen, ASTNode \*node) {

char \*left = visit(gen, node->data.bin\_op.left);

char \*right = visit(gen, node->data.bin\_op.right);

char \*result = generate\_temp(gen);

char instruction[100];

sprintf(instruction, "%s = %s %c %s", result, left, node->data.bin\_op.op, right);

add\_instruction(gen, instruction);

free(left);

free(right);

return result;

}

void free\_ast(ASTNode \*node) {

if (node->type == BIN\_OP) {

free\_ast(node->data.bin\_op.left);

free\_ast(node->data.bin\_op.right);

} else if (node->type == VARIABLE) {

free(node->data.var\_name);

}

free(node);

}

void free\_generator(IntermediateCodeGenerator \*gen) {

for (int i = 0; i < gen->instruction\_count; i++) {

free(gen->instructions[i]);

}

free(gen->instructions);

free(gen);

}

// Simple Parser for the Expression

ASTNode\* parse\_factor(const char \*expr, int \*pos) {

while (isspace(expr[\*pos])) (\*pos)++;

if (isdigit(expr[\*pos])) {

int value = 0;

while (isdigit(expr[\*pos])) {

value = value \* 10 + (expr[\*pos] - '0');

(\*pos)++;

}

return create\_num(value);

} else if (expr[\*pos] == '(') {

(\*pos)++;

ASTNode \*node = parse\_expression(expr, pos);

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

(\*pos)++;

}

return node;

} else {

printf("Error: Unexpected character '%c'\n", expr[\*pos]);

exit(EXIT\_FAILURE);

}

}

ASTNode\* parse\_term(const char \*expr, int \*pos) {

ASTNode \*left = parse\_factor(expr, pos);

while (isspace(expr[\*pos])) (\*pos)++;

while (expr[\*pos] == '\*') {

char op = expr[\*pos];

(\*pos)++;

ASTNode \*right = parse\_factor(expr, pos);

left = create\_bin\_op(left, op, right);

}

return left;

}

ASTNode\* parse\_expression(const char \*expr, int \*pos) {

ASTNode \*left = parse\_term(expr, pos);

while (isspace(expr[\*pos])) (\*pos)++;

while (expr[\*pos] == '+') {

char op = expr[\*pos];

(\*pos)++;

ASTNode \*right = parse\_term(expr, pos);

left = create\_bin\_op(left, op, right);

}

return left;

}

int main() {

char input[256];

printf("Enter an arithmetic expression: ");

fgets(input, sizeof(input), stdin);

input[strcspn(input, "\n")] = '\0'; // Remove the newline character

int pos = 0;

ASTNode \*ast = parse\_expression(input, &pos);

// Generate intermediate code

IntermediateCodeGenerator \*generator = create\_generator();

generate\_code(generator, ast);

// Print intermediate code

for (int i = 0; i < generator->instruction\_count; i++) {

printf("%s\n", generator->instructions[i]);

}

// Free allocated memory

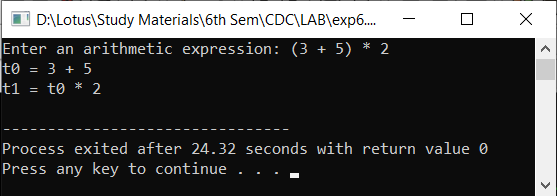
free\_ast(ast);

free\_generator(generator);

return 0;

}

**Output:**



**Experiment 7**

**Write a program to implement Final code generation.**

**Source Code:**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

// Define a struct for the instruction

typedef struct {

char op[10];

char arg1[10];

char arg2[10];

char result[10];

} Instruction;

// Function to generate the final code

void generateFinalCode(char \*expression) {

int tempCount = 1;

Instruction instructions[10];

int instrCount = 0;

// Parse the expression and generate instructions

for (int i = 0; i < strlen(expression); i++) {

if (isdigit(expression[i])) {

// If it's a digit, it's an operand

char operand[2] = { expression[i], '\0' };

strcpy(instructions[instrCount].result, operand);

} else {

// If it's an operator, create an instruction

instructions[instrCount].op[0] = expression[i];

instructions[instrCount].op[1] = '\0';

sprintf(instructions[instrCount].arg1, "t%d", tempCount - 1);

sprintf(instructions[instrCount].arg2, "%c", expression[i + 1]);

sprintf(instructions[instrCount].result, "t%d", tempCount);

tempCount++;

instrCount++;

i++;

}

}

// Print the generated instructions (final code)

printf("Final Code Generation:\n");

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

printf("%s = %s %s %s\n", instructions[i].result, instructions[i].arg1, instructions[i].op, instructions[i].arg2);

}

}

int main() {

char expression[100];

printf("Enter an arithmetic expression (e.g., 2+3\*4): ");

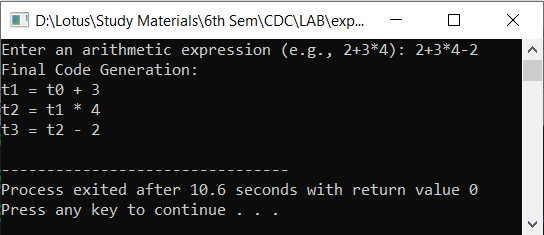
scanf("%s", expression);

generateFinalCode(expression);

return 0;

}

**Output:**



**Experiment 8**

**Write a program to implement Type Conversion.**

**Source Code:**

#include <stdio.h>

#include <stdlib.h>

int main() {

char input[100];

printf("Enter a number: ");

scanf("%s", input);

// Convert to int

int intValue = atoi(input);

// Convert to float

float floatValue = atof(input);

// Convert to double

double doubleValue = strtod(input, NULL);

// Display the results

printf("Input as int: %d\n", intValue);

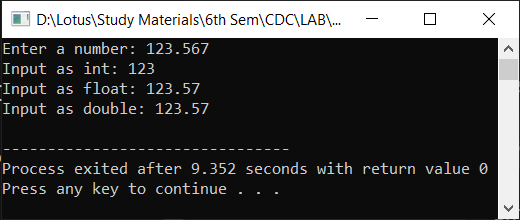
printf("Input as float: %.2f\n", floatValue);

printf("Input as double: %.2f\n", doubleValue);

return 0;

}

**Output:**



**Experiment 9**

**Write a program to check whether a given identifier is valid or not.**

**Source Code:**

#include<stdio.h>

#include<conio.h>

#include<ctype.h>

void main()

{

char a[10];

int flag, i=1;

printf("\n Enter an identifier:");

gets(a);

if(isalpha(a[0]))

flag=1;

else

printf("\n Not a valid identifier");

while(a[i]!='\0')

{

if(!isdigit(a[i])&&!isalpha(a[i]))

{

flag=0;

break;

}

i++;

}

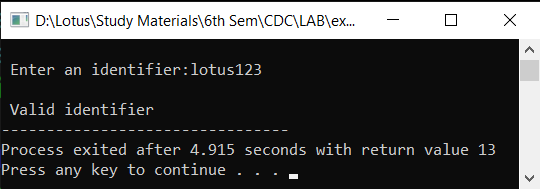
if(flag==1)

printf("\n Valid identifier");

getch();

}

**Output:**



**Experiment 10**

**Write a program to check whether a given string is within valid comment section or not.**

**Source Code:**

#include <stdio.h>

#include <string.h>

// Function to check if the given string is a comment or not

void isComment(char \*line) {

int len = strlen(line);

// Check for single-line comment

if (len >= 2 && line[0] == '/' && line[1] == '/') {

printf("It is a single-line comment");

return;

}

// Check for multi-line comment

if (len >= 4 && line[len - 2] == '\*' && line[len - 1] == '/' && line[0] == '/' && line[1] == '\*') {

printf("It is a multi-line comment");

return;

}

printf("It is not a comment");

}

// Driver code

int main() {

// Variable to hold the input string

char line[256];

// Prompt the user for input

printf("Enter a string: ");

fgets(line, sizeof(line), stdin);

// Remove the newline character at the end if present

line[strcspn(line, "\n")] = 0;

// Function call to check whether the given string is a comment or not

isComment(line);

return 0;

}

**Output:**

