# **Laboratory-8**

## **Question**

## Implement DAG Based Code Generation.

### **Dag.c:**

#include<stdlib.h>

#include<stdio.h>

/\* We will implement DAG as Strictly Binary Tree where each node has zero or two children \*/

struct bin\_tree {

char data;

int label;

struct bin\_tree \* right, \* left;

};

typedef struct bin\_tree node;

/\* R is stack for storing registers \*/

int R[10];

int top;

/\* op will be used for opcode name w.r.t. arithmetic operator e.g. ADD for + \*/

char \* op;

/\* insertnode() and insert() functions are for adding nodes to tree(DAG) \*/

void insertnode(node \*\* tree, char val) {

node \* temp = NULL;

if (!( \* tree)) {

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

temp -> left = temp -> right = NULL;

temp -> data = val;

temp -> label = -1;

\* tree = temp;

}

}

void insert(node \*\* tree, char val) {

char l, r;

int numofchildren;

insertnode(tree, val);

printf("\nEnter number of children of %c:", val);

scanf("%d", & numofchildren);

if (numofchildren == 2) {

printf("\nEnter Left Child of %c:", val);

scanf("%s", & l);

insertnode( & ( \* tree) -> left, l);

printf("\nEnter Right Child of %c:", val);

scanf("%s", & r);

insertnode( & ( \* tree) -> right, r);

insert( & ( \* tree) -> left, l);

insert( & ( \* tree) -> right, r);

}

}

/\* findleafnodelabel() will find out the label of leaf nodes of tree(DAG) \*/

void findleafnodelabel(node \* tree, int val) {

if (tree -> left != NULL && tree -> right != NULL) {

findleafnodelabel(tree -> left, 1);

findleafnodelabel(tree -> right, 0);

} else {

tree -> label = val;

}

}

/\* findinteriornodelabel() will find out the label of interior nodes of tree(DAG) \*/

void findinteriornodelabel(node \* tree) {

if (tree -> left -> label == -1) {

findinteriornodelabel(tree -> left);

} else if (tree -> right -> label == -1) {

findinteriornodelabel(tree -> right);

} else {

if (tree -> left != NULL && tree -> right != NULL) {

if (tree -> left -> label == tree -> right -> label) {

tree -> label = (tree -> left -> label) + 1;

} else {

if (tree -> left -> label > tree -> right -> label) {

tree -> label = tree -> left -> label;

} else {

tree -> label = tree -> right -> label;

}

}

}

}

}

/\* function print\_inorder() will print inorder of nodes. Here we are also printing label of each node of tree(DAG) \*/

void print\_inorder(node \* tree) {

if (tree) {

print\_inorder(tree -> left);

printf("%c with Label %d\n", tree -> data, tree -> label);

print\_inorder(tree -> right);

}

}

/\* function swap() will swap the top and second top elements of Register stack R \*/

void swap() {

int temp;

temp = R[0];

R[0] = R[1];

R[1] = temp;

}

/\* function pop() will remove and return topmost element of stack \*/

int pop() {

int temp = R[top];

top--;

return temp;

}

/\* function push() will increment top by one and will insert element at top position of Register stack \*/

void push(int temp) {

top++;

R[top] = temp;

}

/\* nameofoperation() will return opcode w.r.t. arithmetic operator \*/

char \* nameofoperation(char temp) {

switch (temp) {

case '+':

return "ADD";

break;

case '-':

return "SUB";

break;

case '\*':

return "MUL";

break;

case '/':

return "DIV";

break;

}

}

/\* gencode() will generate Assembly code w.r.t. labels of tree(DAG) \*/

void gencode(node \* tree) {

if (tree -> left != NULL && tree -> right != NULL) {

if (tree -> left -> label == 1 && tree -> right -> label == 0 && tree -> left -> left == NULL && tree -> left -> right == NULL && tree -> right -> left == NULL && tree -> right -> right == NULL) {

printf("MOV %c,R[%d]\n", tree -> left -> data, R[top]);

op = nameofoperation(tree -> data);

printf("%s %c,R[%d]\n", op, tree -> right -> data, R[top]);

} else if (tree -> left -> label >= 1 && tree -> right -> label == 0) {

gencode(tree -> left);

op = nameofoperation(tree -> data);

printf("%s %c,R[%d]\n", op, tree -> right -> data, R[top]);

} else if (tree -> left -> label < tree -> right -> label) {

int temp;

swap();

gencode(tree -> right);

temp = pop();

gencode(tree -> left);

push(temp);

swap();

op = nameofoperation(tree -> data);

printf("%s R[%d],R[%d]\n", op, R[top - 1], R[top]);

} else if (tree -> left -> label >= tree -> right -> label) {

int temp;

gencode(tree -> left);

temp = pop();

gencode(tree -> right);

push(temp);

op = nameofoperation(tree -> data);

printf("%s R[%d],R[%d]\n", op, R[top - 1], R[top]);

}

} else if (tree -> left == NULL && tree -> right == NULL && tree -> label == 1) {

printf("MOV %c,R[%d]\n", tree -> data, R[top]);

}

}

/\* deltree() will free the memory allocated for tree(DAG) \*/

void deltree(node \* tree) {

if (tree) {

deltree(tree -> left);

deltree(tree -> right);

free(tree);

}

}

/\* Program execution will start from main() function \*/

void main() {

node \* root;

root = NULL;

node \* tmp;

char val;

int i, temp;

/\* Inserting nodes into tree(DAG) \*/

printf("\nEnter root of tree:");

scanf("%c", & val);

insert( & root, val);

/\* Finding Labels of Leaf nodes \*/

findleafnodelabel(root, 1);

/\* Finding Labels of Interior nodes \*/

while (root -> label == -1)

findinteriornodelabel(root);

/\* value of top = index of topmost element of stack R = label of Root of tree(DAG) minus one \*/

top = root -> label - 1;

/\* Allocating Stack Registers \*/

temp = top;

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

R[i] = temp;

temp--;

}

/\* Printing inorder of nodes of tree(DAG) \*/

printf("\nInorder Display:\n");

print\_inorder(root);

/\* Printing assembly code w.r.t. labels of tree(DAG) \*/

printf("\nAssembly Code:\n");

gencode(root);

/\* Deleting all nodes of tree \*/

deltree(root);

}

### **Output:**

