Laboratory-7

Question

Perform DAG construction and simple code generation.

Dag.cpp:

```
#include <bits/stdc++.h>
using namespace std;

struct label_list
{
    char value;
    struct label_list *next;
};

struct dag_node
{
    char value;
    int index;
    int label;
    int parent_count;
    struct label_list *labels;
    struct dag_node *left, *right;
};

struct root_list
{
    struct dag_node *root;
    struct root_list *next;
};

struct three_address
{
    char op;
    char op1;
    char op2;
    char lhs;
};

int index_global = 0, n;
three_address tac[100];
bool visited_left_most_child = 0;
stack<int> rstack, tstack;

label_list *insert_label(label_list *curr, char c)
{
    label_list *temp = new label_list;
    temp->value = c;
    temp->next = NULL;
    if (curr == NULL)
    {
        return temp;
    }
    label_list *start = curr;
```

```
    while (start->next)
    {
        start = start->next;
    }
    start->next = temp;
    return start;
}

dag_node *create_dag_node(char op, char result)
{

    dag_node *temp = new dag_node;
    temp->index = index_global++;
    temp->value = op;
    temp->left = temp->right = NULL;
    temp->labels = NULL;
    temp->labels = insert_label(temp->labels, result);

    return temp;
}

bool search_labels(dag_node *x, char value)
{
    if (x->value == value)
    {
        return 1;
    }
    label_list *start = x->labels;
    while (start)
    {
        if (start->value == value)
        {
            return 1;
        }
        start = start->next;
    }
    return 0;
}

bool check(dag_node *n1, dag_node *n2, char c1, char c2)
{
    if (search_labels(n1, c1) && search_labels(n2, c2))
    {
        return 1;
    }
    return 0;
}

dag_node *search_for_value(root_list *root, char value)
{
    int recent = -1;
    queue<dag_node *> q;
    dag_node *x, *result = NULL;
    vector<bool> visited(index_global, 0);

    root_list *start = root;
    while (start)
    {
        q.push(start->root);
        start = start->next;
```

```
    }

    while (!q.empty())
    {
        x = q.front();
        q.pop();
        visited[x->index] = 1;
        if (search_labels(x, value))
        {
            if (x->index > recent)
            {
                recent = x->index;
                result = x;
            }
        }

        dag_node *left = x->left;
        dag_node *right = x->right;
        if (left && !visited[left->index])
            q.push(left);

        if (right && !visited[right->index])
            q.push(right);
    }

    return result;
}

dag_node *search_for_similar(root_list *root, char op, char op1, char op2)
{
    int recent = -1;
    queue<dag_node *> q;
    dag_node *x, *result = NULL;
    vector<bool> visited(index_global, 0);

    root_list *start = root;
    while (start)
    {
        q.push(start->root);
        start = start->next;
    }

    while (!q.empty())
    {
        x = q.front();

        q.pop();
        visited[x->index] = 1;
        if (search_labels(x, op))
        {
            if (check(x->left, x->right, op1, op2))
            {
                if (x->index > recent)
                {
                    result = x;
                    recent = x->index;
                }
            }
        }
```

```
        dag_node *left = x->left;
        dag_node *right = x->right;
        if (left && !visited[left->index])
            q.push(left);

        if (right && !visited[right->index])
            q.push(right);
    }

    return result;
}

root_list *add_to_end(root_list *curr, dag_node *root)
{
    root_list *temp = new root_list;
    temp->root = root;
    temp->next = NULL;
    if (curr == NULL)
    {
        return temp;
    }

    root_list *start = curr;
    while (start->next)
    {
        start = start->next;
    }
    start->next = temp;
    return curr;
}
root_list *create_dag(root_list *root, int curr)
{
    if (curr == n)
    {
        return root;
    }

    if (root == NULL)
    {
        dag_node *parent, *left, *right;
        parent = create_dag_node(tac[curr].op, tac[curr].lhs);
        left = create_dag_node(tac[curr].op1, tac[curr].op1);
        right = create_dag_node(tac[curr].op2, tac[curr].op2);
        parent->left = left;
        parent->right = right;
        root = new root_list;
        root->root = parent;
        root->next = NULL;
        return create_dag(root, curr + 1);
    }

    if (tac[curr].op == '=')
    {
        dag_node *temp = search_for_value(root, tac[curr].op1);
        temp->labels = insert_label(temp->labels, tac[curr].lhs);
        return create_dag(root, curr + 1);
    }
    else
    {
```

```
        dag_node *parent = search_for_similar(root, tac[curr].op,
tac[curr].op1, tac[curr].op2);
        dag_node *left = search_for_value(root, tac[curr].op1);
        dag_node *right = search_for_value(root, tac[curr].op2);


{

    if (left && right && parent)
    {
        if (parent->left->index == left->index && parent->right->index ==
right->index)

            parent->labels = insert_label(parent->labels, tac[curr].lhs);
        return create_dag(root, curr + 1);
    }
}

parent = create_dag_node(tac[curr].op, tac[curr].lhs);

if (left == NULL)
{
    left = create_dag_node(tac[curr].op1, tac[curr].op1);
}
if (right == NULL)
{
    right = create_dag_node(tac[curr].op2, tac[curr].op2);
}

parent->left = left;
parent->right = right;

root_list *start = root;
root_list *temp = NULL;
while (start)
{
    if (start->root != left && start->root != right)
    {
        temp = add_to_end(temp, start->root);
    }
    start = start->next;
}
temp = add_to_end(temp, parent);
root = temp;
return create_dag(root, curr + 1);
    }
}

void inorder(dag_node *curr, vector<bool> &visited)
{
    curr->parent_count++;
    if (visited[curr->index])
    {
        if (curr->left != NULL)
        {
            inorder(curr->left, visited);
        }

        if (curr->right != NULL)
        {
```

```
            inorder(curr->right, visited);
        }

        return;
    }

    visited[curr->index] = 1;

    if (curr->left == NULL && curr->right == NULL)
    {
        cout << "Leaf with Index: " << curr->index << " ,Value: " << curr-
>value << " ,Label: " << curr->label << "\n\n";
    }
    else
    {
        cout << "Index: " << curr->index << " ,Value: " << curr->value << "
,Label: " << curr -> label << '\n';
        label_list *temp = curr->labels;
        cout << "Labels are: ";
        while (temp)
        {
            cout << temp->value << ' ';
            temp = temp->next;
        }
        cout << '\n';
        cout << "Left child has index " << curr->left->index << '\n';
        cout << "Right child has index " << curr->right->index << '\n';
        cout << '\n';
        inorder(curr->left, visited);
        inorder(curr->right, visited);
    }
}

dag_node *assign_labels(dag_node *curr, bool left_child, vector<bool>
&visited)
{
    if (visited[curr->index])
    {
        return curr;
    }

    visited[curr->index] = 1;

    if (curr->left == NULL && curr->right == NULL)
    {
        if (!left_child)
        {
            curr->label = 0;
        }
        else
        {
            curr->label = 1;
        }
        return curr;
    }

    curr->left = assign_labels(curr->left, 1, visited);
    curr->right = assign_labels(curr->right, 0, visited);

    if (curr->left->label == curr->right->label)
```

```
    {
        curr->label = curr->left->label + 1;
    }
    else
    {
        curr->label = max(curr->left->label, curr->right->label);
    }

    return curr;
}

void swap_registers()
{

    int temp1, temp2;
    temp1 = rstack.top();
    rstack.pop();
    temp2 = rstack.top();
    rstack.pop();

    rstack.push(temp1);
    rstack.push(temp2);
}

void gen_code(dag_node *curr, bool left_child)
{

    if (curr->left == NULL && curr->right == NULL)
    {
        if (left_child)
        {
            printf("MOV %c R%d\n", curr->value, rstack.top());
        }
    }
    else
    {
        int left_label = curr->left->label;
        int right_label = curr->right->label;

        if (right_label == 0)
        {
            gen_code(curr->left, 1);
            printf("%c %c R%d\n", curr->value, curr->right->value,
rstack.top());
        }
        else if (right_label > left_label && left_label < rstack.size())
        {
            swap_registers();
            gen_code(curr->right, 0);
            int R = rstack.top();
            rstack.pop();
            gen_code(curr->left, 1);
            printf("%c R%d R%d\n", curr->value, R, rstack.top());
            rstack.push(R);
            swap_registers();
        }
        else if (left_label >= right_label && right_label < rstack.size())
        {
            gen_code(curr->left, 1);
            int R = rstack.top();
```

```
            rstack.pop();
            gen_code(curr->right, 0);
            printf("%c R%d R%d\n", curr->value, rstack.top(), R);
            rstack.push(R);
        }
        else if (left_label >= right_label && left_label > rstack.size() &&
right_label > rstack.size())
        {
            gen_code(curr->right, 0);
            int T = tstack.top();
            printf("MOV R%d T%d", rstack.top(), T);
            gen_code(curr->left, 1);
            tstack.push(T);
            printf("%c T%d R%d", curr->value, T, rstack.top());
        }
    }
}

int main()
{
    char lhs, op, op1, op2;
    string s;

    int i = 0;

    while (getline(cin, s))
    {
        if (s.size() <= 4)
        {
            tac[i].lhs = s[0];
            tac[i].op1 = s[2];
            tac[i].op = s[1];
            tac[i].op2 = ' ';
        }
        else
        {
            tac[i].lhs = s[0];
            tac[i].op1 = s[2];
            tac[i].op = s[3];
            tac[i].op2 = s[4];
        }
        i++;
    }
    n = i;

    root_list *root = NULL;
    root = create_dag(root, 0);

    root_list *start = root;
    vector<bool> visited(index_global, 0);
    while (start)
    {
        start->root = assign_labels(start->root, 1, visited);
        start = start->next;
    }

    fill(visited.begin(), visited.end(), 0);
    start = root;
    while (start)
    {
```

```
        inorder(start->root, visited);
        start = start->next;
    }
    rstack.push(0);
    rstack.push(1);

    for (int i = 0; i < 10; i++)
    {
        tstack.push(i);
    }

    gen_code(root->root, 1);
}
```

Output:

```
kal-el@mos-13:~/Desktop/Compilers/Compilers Lab/lab7$ g++ dag.cpp
kal-el@mos-13:~/Desktop/Compilers/Compilers Lab/lab7$ ./a.out
Enter the input :
a=b*c;
d=b*c;
g=a/d;
f=g+a;
Index: 4 ,Value: + ,Label: 2
Labels are: f
Left child has index 3
Right child has index 0

Index: 3 ,Value: / ,Label: 2
Labels are: g
Left child has index 0
Right child has index 0

Index: 0 ,Value: * ,Label: 1
Labels are: a d
Left child has index 1
Right child has index 2

Leaf with Index: 1 ,Value: b ,Label: 1

Leaf with Index: 2 ,Value: c ,Label: 0

MOV b R1
* c R1
MOV b R0
* c R0
/ R0 R1
MOV b R0
* c R0
+ R0 R1
kal-el@mos-13:~/Desktop/Compilers/Compilers Lab/lab7$
```

Result:

DAG construction and Simple Code Generation was performed successfully.