

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Date : 20/04/2022 | | | | | | | |
|  | CSPC62 : COMPILER DESIGN  **LAB-7** | | | | | |  |
|  |  | | | | |  | |
|  | | |  |  | | | |
|  | | | Roll no. : 106119100Name : Rajneesh PandeySection : CSE-B |  | | | |
|  | |  | | |  | | |

DAG construction and Simple code generation.

Code:

Construct\_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);

}

Lexer.l

%{

    #include"parser.tab.h"

%}

%%

[\t ] ;

[0-9]**+** {yylval.*symbol* = (char)(yytext[0]);return NUMBER;}

[a-z] {yylval.*symbol* = (char)(yytext[0]);return LETTER;}

. {return yytext[0];}

\n\n {return 0;}

%%

*int* yywrap(){

    return 1;

}

Parser.y

%{

#include<string.h>

#include<stdio.h> int yylex(void);

*int* yyerror(const *char* **\*s**);

struct node{

char value;

struct node \*left, \*right;

};

struct start\_node{ char value;

struct node \*left, \*right; struct start\_node \*next;

};

struct label\_list{ char value;

struct label\_list\* next;

};

struct dag\_node{

char value; int index;

struct label\_list \*labels; struct dag\_node \*left, \*right;

};

struct root\_list{

struct dag\_node\* root; struct root\_list\* next;

};

struct three\_address{ char operator; char operand1; char operand2; char lhs;

struct three\_address\* next;

};

struct node\* make\_node(char);

struct start\_node\* create\_start\_node();

struct start\_node \*start\_ptr = NULL;

struct three\_address \*start\_three\_address\_ptr = NULL; struct root\_list \*roots = NULL;

char label = 'A' - 1; int index = 0;

%}

%union{

char symbol;

struct node \*sub\_expr;

struct start\_node \*start\_expr;

}

%left '+' '-'

%left '/' '\*'

%token <symbol> LETTER NUMBER

%type <sub\_expr> exp

%type <sub\_expr> L

%type <start\_expr> stmts

%type <start\_expr> statement

%start S

%%

S: stmts {

start\_ptr = $1;

};

stmts: statement stmts {

struct start\_node\* curr = $1;

}

| statement {

    $$ =$1;

    $$ = curr;

};

statement: L '=' exp ';' {

struct start\_node \*curr = create\_start\_node(); curr->left = $1;

curr->right = $3;

$$ = curr;

};

exp: exp '+' exp { struct node\* curr = make\_node('+'); curr->left = $1;

curr->right = $3;

$$ = curr;

}

|exp '-' exp { struct node\* curr = make\_node('-');

curr->left = $1; curr->right = $3;

$$ = curr;

}

|exp '/' exp { struct node\* curr = make\_node('/');

curr->left = $1; curr->right = $3;

$$ = curr;

}

|exp '\*' exp {

struct node \*curr = make\_node('\*');

curr->left = $1;

curr->right = $3;

$$ = curr;

}

| '(' exp ')' {$$ = $2;}

|NUMBER {

    struct node\* curr = make\_node((char)$1);

    $$ = curr;

}

|LETTER {

struct node\* curr = make\_node((char)$1);

$$ = curr;

}

;

L: LETTER {

struct node\* curr = make\_node((char)$1);

$$ = curr;

};

%%

*int* yyerror(const *char* **\*s**){

printf("%s",s);

}

struct start\_node \*create\_start\_node(){

struct start\_node \*temp\_node = (struct start\_node\*)malloc( sizeof(struct start\_node) );

temp\_node->value = '=';

temp\_node->left = temp\_node->right = NULL; temp\_node->next = NULL;

return temp\_node;

}

struct node\* make\_node(char s){

struct node\* temp\_node = (struct node\*)malloc(sizeof(struct node)); temp\_node->left = NULL;

temp\_node->right = NULL; temp\_node->value = s; return temp\_node;

}

struct three\_address\* add\_to\_end(struct three\_address\* start, struct three\_address\* curr){

if(start == NULL){ return curr;

}

struct three\_address\* temp = start; while(temp-> next != NULL){

temp = temp->next;

}

temp->next = curr; return start;

}

struct three\_address\* make\_three\_address\_node(char lhs, char op, char op1, char op2){

struct three\_address\* temp\_node = (struct three\_address\*)malloc(sizeof(struct three\_address));

temp\_node->lhs = lhs; temp\_node->operator = op; temp\_node->operand1 = op1; if(op != '='){

temp\_node->operand2 = op2;

}

temp\_node->next = NULL; return temp\_node;

}

char postfix(struct node\* curr){

if(curr->left == NULL && curr->right == NULL){ return curr->value;

}

char lhs = postfix(curr->left); char rhs = postfix(curr->right); char curr\_char = ++label;

struct three\_address \*temp\_node = make\_three\_address\_node(curr\_char, curr-

>value, lhs, rhs);

start\_three\_address\_ptr = add\_to\_end(start\_three\_address\_ptr, temp\_node);

return curr\_char;

}

void traverse(struct start\_node \*curr){ if(curr==NULL){

return;

}

char lhs = postfix(curr->left); char rhs = postfix(curr->right);

struct three\_address \*temp\_node = make\_three\_address\_node(lhs, '=', rhs, ' '); start\_three\_address\_ptr = add\_to\_end(start\_three\_address\_ptr, temp\_node);

traverse(curr->next);

}

void print\_three\_addr(){

struct three\_address\* temp = start\_three\_address\_ptr; while(temp){

if(temp->operator == '='){

printf("%c %c %c\n", temp->lhs, temp->operator, temp->operand1);

}

else{

printf("%c = %c %c %c\n", temp->lhs, temp->operand1, temp->operator, temp->operand2);

}

temp = temp->next;

}

}

struct label\_list\* insert\_label(struct label\_list\* curr, char c){ struct label\_list\* temp = (struct label\_list\*)malloc(sizeof(struct

label\_list));

temp->value = c; if(curr == NULL){

return temp;

}

struct label\_list\* t = curr; while(t->next != NULL){

t = t->next;

}

t->next = temp; return t;

}

struct dag\_node\* search\_value(char value){ struct start = roots;

while(start){

}

}

struct dag\_node\* create\_dag\_node(char op, char res){

struct dag\_node\* temp = (struct dag\_node\*)malloc(sizeof(struct dag\_node)); temp->labels = NULL;

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

temp->labels = insert\_label(temp->labels, res); temp->index = index++;

temp->value = op; return temp;

}

void create\_dag(struct three\_address\* curr){ if(roots == NULL){

struct dag\_node\* curr = create\_dag\_node(curr->operator, curr->lhs); struct dag\_node\* left = create\_dag\_node(curr->operand1, ' '); struct dag\_node\* right = create\_dag\_node(curr->operand2, ' '); roots = curr;

}

else{

if(curr->operator == '='){

struct dag\_node \*temp = search\_value(curr->operand1); temp->labels = insert\_label(temp->labels, curr->lhs);

}

else{

struct dag\_node \*temp

}

}

create\_dag(curr->next);

}

int main(){

printf("Enter the expression: "); yyparse();

traverse(start\_ptr); print\_three\_addr(); printf("\n"); return 0;

}

Input:

Graphical user interface, text, application

Description automatically generated

Output:

Text

Description automatically generated