Отчет по лабораторной работе № 24 по курсу “Фундаментальная информатика”

Студент группы М80-103Б-21 Катин Иван Вячеславович, № по списку 12

Контакты e-mail: ikatin.2003.sokol@gmail.com, telegram: @Dazz1e

Работа выполнена: «2» мая 2022г.

Преподаватель: каф. 806 Севастьянов Виктор Сергеевич

Отчет сдан « » _____20__ г., итоговая оценка _____

Подпись преподавателя_____

Тема: Дерево выражений

1. **Цель работы:** Научиться реализовать деревья, записывать выражение в бин деререво и преоброзовывать их.
2. **Задание:** 24. Перемножить дроби.

3. **Оборудование** (студента):
   Процессор *Intel® Core™ i5-9300H CPU @ 2.40GHz × 8* с ОП 7,6 GiB, НМД *1024* Гб. Монитор *1920x1080*

4. **Программное обеспечение (**студента**):**
   Операционная система семейства: *linux*, наименование: *ubuntu*, версия *20.04.3 LTS*
   интерпретатор команд: *bash* версия *4.4.20(1)-release*.
   Система программирования -- CLion--, редактор текстов *emacs* версия *25.2.2*
   Утилиты операционной системы --
   Прикладные системы и программы – **LibreOffice**
   Местонахождение и имена файлов программ и данных на домашнем компьютере – *home*/dazzle

6. **Идея, метод, алгоритм.** Реализовать дерево, вектор и токен. Токену содержит информацию о типе+имя оператора, значение или имя переменной.
Типы: operator, variable, bracket, floating,integer, final.
Структура дерева: токен, левое поддерево, правое поддерево.
Вектор нужен, чтобы при парсинге выражения хранить токены,
Если мы в дереве встречаем, что значение оператора = “/”, значение левого поддерева = “*” и левого-левого поддерева равняется = “/” , то это перемножение дробей => сделать преобразование.

7. **Сценарий выполнения работы**

1.Создание структур.

2.Парсинг выражения.

3.Создание дерева выражения.

4.Преобразование над деревом выражений.

8. Распечатка протокола (подклеить листинг окончательного варианта программы с тестовыми примерами,подписанныйпреподавателем).

TOKEN.H

```
//
// Created by machine on 30.04.2022.
//

#ifndef LAB24_TOKEN_H
#define LAB24_TOKEN_H
#include "stdbool.h"

typedef enum {
    FINAL,
    INTEGER,
    FLOATING,
    OPERATOR,
    VARIABLE,
    BRACKET
}TokenType;

typedef struct {
    TokenType type;
    union {
        int   value_int;
        float value_float;
        char  operator_name;
        bool  is_left_bracket;
        char  variable_name;
    } data;
} Token;

void token_print(Token *t, int depth);
void token_next(Token *t);

#endif //LAB24_TOKEN_H
```

TREE.H

```
//
// Created by machine on 30.04.2022.
//
#ifndef LAB24_TREE_H
#define LAB24_TREE_H
#include "token.h"
#include "malloc.h"
#include "vector.h"

typedef struct Tree{
    Token data;
    struct Tree* left;
    struct Tree* right;
}Tree;

Tree* tree_create(vector* v, int idx_left, int idx_right);
void tree_print(Tree* t, size_t depth);
void tree_infix(Tree* t);
```

```
#endif //LAB24_TREE_H
```

VECTOR.H

```
//
// Created by machine on 01.05.2022.
//

#ifndef LAB24_VECTOR_H
#define LAB24_VECTOR_H
#include "token.h"
#include <malloc.h>
#include <stdlib.h>

typedef struct {
    int size;
    int ptr;
    Token* t;
}vector;


void create(vector* v);
void push(vector* v, Token t);
void pop(vector *v);
Token take(vector*v, int idx);

#endif //LAB24_VECTOR_H
```

TRANSFORM.H

```
//
// Created by machine on 01.05.2022.
//

#ifndef LAB24_TRANSFORM_H
#define LAB24_TRANSFORM_H

#include "tree.h"
void tree_transform(Tree* t);

#endif //LAB24_TRANSFORM_H
```

TREE.C

```
//
// Created by machine on 30.04.2022.
//
```

```
#include "tree.h"
#include <stdlib.h>
#include <stdio.h>

int get_priority(char c){
    switch (c) {
        case '+':
        case '-':
            return 1;
        case '*':
        case '/':
            return 2;
        case '^':
            return 3;
    }
    return 100;
}

Tree* tree_create(vector* v, int idx_left, int idx_right){
    Tree* t = malloc(sizeof(Tree));
    if(idx_left > idx_right){
        return NULL;
    }
    if(idx_right == idx_left){
        t->data = take(v, idx_left);
        t->left = NULL;
        t->right = NULL;
        return t;
    }
    int priority_min = get_priority('a');
    int brackets = 0;
    int priority;
    int op_pos;
    for (int i = idx_left; i < idx_right; ++i) {
        if (take(v, i).type == BRACKET && take(v, i).data.is_left_bracket) {
            ++brackets;
        } else if (take(v, i).type == BRACKET && take(v, i).data.is_left_bracket
== 0) {
            --brackets;
        } else if(brackets > 0){
            continue;
        } else if(take(v,i).type == OPERATOR){
            priority = get_priority(take(v,i).data.operator_name);
            if(priority <= priority_min){
                priority_min = priority;
                op_pos = i;
            }
        }
    }
    if(priority_min == 100 && take(v,idx_left).type == BRACKET &&
            take(v,idx_left).data.is_left_bracket && take(v,idx_right).type ==
BRACKET &&
            !(take(v,idx_right).data.is_left_bracket)){
        free(t);
        return tree_create(v,idx_left + 1, idx_right-1);
    }

    if(take(v,op_pos).data.operator_name == '^'){
        brackets=0;
        for (int i = op_pos; i >= idx_left ; --i) {
            if(take(v,i).type == BRACKET && take(v,i).data.is_left_bracket==0){
                ++brackets;
```

```
            }
            if (take(v,i).type == BRACKET && take(v,i).data.is_left_bracket) {
                --brackets;
            }
            if(brackets > 0) continue;
            if(take(v,i).type == OPERATOR){
                priority = get_priority(take(v,i).data.operator_name);
                if(priority == 3){
                    op_pos = i;
                }
            }
        }
    }
    t->data = take(v,op_pos);
    t->left = tree_create(v, idx_left, op_pos-1);
    t->right = tree_create(v,op_pos+1, idx_right);
    if(t->right == NULL){
        printf("error");
        exit(1);
    }
    return t;
}

void tree_delete(Tree *t){
    if(t != NULL){
        tree_delete(t->left);
        tree_delete(t->right);
    }
    free(t);
}

void tree_infix(Tree* t){
    if(t != NULL){
        if(t->left && t->right) printf("(");
        tree_infix(t->left);
        token_print(&t->data,1);
        tree_infix(t->right);
        if(t->right && t->left){
            printf(")");
        }
    }
}

void tree_print(Tree* t, size_t depth){
    if(t != NULL){
        tree_print(t->left, depth + 10);
        token_print(&(t->data),depth);
        printf("\n");
        tree_print(t->right, depth + 10);
    }
}
```

VECTOR.C

```
//
// Created by machine on 01.05.2022.
//
```

```
#include "vector.h"

void create(vector* v){
    v->ptr=0;
    v->size = 200;
    v->t = malloc(sizeof(Token)* v->size);
}

void push(vector* v, Token t){
    if(v->size == v->ptr) {
        v->size *= 2;
        v->t = realloc(v,sizeof(Token)* v->size);
    }
    v->t[v->ptr++] = t;
}

void pop(vector* v ){
    v->ptr--;
}

Token take(vector* v, int idx){
//    if(idx >= v->ptr){
//        exit(1);
//    }
    return v->t[idx];
}
```

TRANSFORM.C

```
#include "token.h"
#include "token.c"
#include "transform.h"

int match_frac(Tree *t)
{
    return (t->data.data.operator_name == '/' &&
t->left->data.data.operator_name == '*' &&
    t->left->left->data.data.operator_name == '/');
}

void transform_frac(Tree* t)
{
    Tree* tmpright = malloc(sizeof(Tree));
    Tree* tmpleft = malloc(sizeof(Tree));
    tmpright->data = t->left->data;
    tmpright->left = t->left->left->right;
    tmpright->right = t->right;
    tmpleft->data = t->left->data;
    tmpleft -> right = t->left->right;
    tmpleft ->left = t->left->left->left;
    t->left = tmpleft;
    t->right = tmpright;
}

void tree_transform(Tree *t){
    if (t != NULL) {
        tree_transform(t->left);
        tree_transform(t->right);
```

```
        if (match_frac(t)) {
            transform_frac(t);
        }
    }
}
```

TOKEN.C

```
#include "token.h"
#include <stdlib.h>
#include <ctype.h>
#include <stdio.h>

void token_next(Token* t){
    static bool mb_unary = true;
    char c;
    do {
        c = fgetc(stdin);
    } while (isspace(c));
    if (c == EOF) {
        t->type = FINAL;
    }
    else if(isalpha(c) || c == '_'){
        mb_unary = false;
        t->type=VARIABLE;
        t->data.variable_name=c;
    }
    else if(isdigit(c)){
        mb_unary=false;
        ungetc(c,stdin);
        float f;
        scanf("%f", &f);
        if(f == (int)f){
            t->type=INTEGER;
            t->data.value_int=(int)f;
        } else {
            t->type=FLOATING;
            t->data.value_float = f;
        }
    }
    else if(c== '(' || c ==')'){
        t->type=BRACKET;
        if(c=='('){
            mb_unary = true;
            t->data.is_left_bracket=1;
        } else {
            mb_unary = false;
            t->data.is_left_bracket = 0;
        }
    }

    else if(mb_unary && (c == '+' || c == '-')){
        int znak = (c == '+') ? 1 : -1;
        do {
            c = fgetc(stdin);
        } while (isspace(c));
        if(isdigit(c)) {
            ungetc(c, stdin);
```

```
            token_next(t);
            if (t->type == INTEGER) {
                t->data.value_int = t->data.value_int * znak;
            } else {
                t->data.value_float = t->data.value_float * znak;
            }
        } else {
            ungetc(c,stdin);
            t->type = OPERATOR;
            t->data.operator_name = '-';
            mb_unary = true;
        }
    }
    else {
        t->type = OPERATOR;
        t->data.operator_name = c;
        mb_unary = true;
    }
}

void token_print(Token* t, int depth){
    switch (t->type) {
        case FINAL:
            break;
        case INTEGER:
            printf("%*d", depth, t->data.value_int);
            break;
        case FLOATING:
            printf("%*f", depth, t->data.value_float);
            break;
        case VARIABLE:
            printf("%*c", depth, t->data.variable_name);
            break;
        case BRACKET:
            printf("%*c", depth, t->data.is_left_bracket == 1 ? '(' : ')');
            break;
        case OPERATOR:
            printf("%*c", depth, t->data.operator_name);
            break;
    }
}
```

MAIN.C

```
#include "vector.h"
#include "tree.h"
#include "token.h"
#include "transform.h"
#include "vector.c"
#include "tree.c"
#include "transform.c"
int main() {
    vector* tokens = malloc(sizeof(vector));
    create(tokens);
    size_t tokens_qty = 0;
```

```
    Token token;
    token_next(&token);

    while (token.type != FINAL){
        push(tokens, token);
        token_next(&token);
        tokens_qty++;
    }
    Tree* tree = tree_create(tokens,0,(int)tokens_qty-1);
    printf("\nExpression tree:\n");
    tree_print(tree,0);
    tree_transform(tree);
    printf("\nSemitransformed expression tree:\n");
    tree_print(tree,0);
    return 0;
}
```

9. Дневник отладки

№	Лаб. или дом.	Дата	Время	Событие	Действие по исправлению	Примечание

10. Замечания автора

11. Выводы

Узнал, что такое дерево выражений и о множестве его реализаций. Реализовал инфиксное дерево выражений, вектор и токен дерева выражений. Узнал о таких функциях, как isdigit, isalpha, isspace.

Подпись студента _____