**Отчет по лабораторной работе № 24** по курсу по курсу “Практикум на ЭВМ”

Студент группы [**М8О-107Б-20**](https://vk.com/im?sel=c51) Чекменев Вячеслав Алексеевич, № по списку 27

Контакты e-mail: [chekmenev031@gmail.com](mailto:chekmenev031@gmail.com), telegram: @suraba03

Работа выполнена: «21» мая 2021 г.

Преподаватель: каф. 806 Найденов Иван Евгеньевич

Отчет сдан « » \_\_\_\_\_\_\_\_\_20 \_\_\_ г., итоговая оценка \_\_\_\_\_

Подпись преподавателя \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

* 1. **Тема:** Деревья выражений.

1. **Цель работы:** Составить программу на языке Си выполнения заданных преобразований арифметических выражений с применением деревьев.
2. **Задание:** (вариант 35) Подсчитать количество операндов (переменных, констант) в выражении.
3. **Оборудование** (студента):

Процессор *Intel Core i5-8265U* с ОП 7851 Мб, НМД 256 Гб. Монитор *1920x1080*

1. **Программное обеспечение (студента):**

Операционная система семейства UNIX: linux, наименование: manjaro, версия: 20.1 Mikah

интерпретатор команд: bash, версия: 5.0.18.

текствый редактор: atom, версия: 5.2

Утилиты операционной системы --

Прикладные системы и программы --

Местонахождение и имена файлов программ и данных –

**6. Идея, метод, алгоритм** решения задачи(в формах:словесной,псевдокода,графической[блок-схема,диаграмма,рисунок,таблица] или формальные спецификации с пред- и постусловиями)

1) использовать файлы tree.h tree.c из 23 ЛР

2) написать lexer.h lexer.c файлы, функции для считывания и преобразования в инфиксную запись введенного выражения.

3) Написать transform.h transform.c, с функцей из задания. Моя функция тождественна подсчету количества листьев, функция рекурсиная, исполняет проход в глубину и, встречая терминальную вершину возвращает 1, в переменную result прибаляется по одной единице, при исполнении функций из callstack. В итоге, вывод – кол-во терминальных вершин (листьев)

**7. Сценарий выполнения работы** [план работы,первоначальный текст программы в черновике(можно на отдельном листе)итесты либо соображения по тестированию].

**тесты:**

|  |  |  |
| --- | --- | --- |
| **Входные данные** | **Выходные данные** | **Описание тестируемого случая** |
| (a+b)\*c | (a+b)\*c  Expression tree:  \*  +  a  b  c  **вот столько операндов: 3**  Tree's infix linearization:  ((a+b)\*c) | Простой случай |
| a + b + c + d \* e \* x / f | Expression tree:  +  +  +  a  b  c  /  \*  \*  d  e  x  f  **вот столько операндов: 7**  Tree's infix linearization:  (((a+b)+c)+(((d\*e)\*x)/f)) | Простой случай 2 |
| -a | Expression tree:  -  a  вот столько операндов: 1  Tree's infix linearization:  -a | Один унарный оператор |

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

**Tree.h --------------------------------------------------------------------------------------------------------------------------------------------------------**

#ifndef \_TREE\_H\_

#define \_TREE\_H\_

#include <stdlib.h>

#include "lexer.h"

typedef struct tree\_node \*Tree;

struct tree\_node {

Token node;

Tree left;

Tree right;

};

Tree tree\_create(Token tokens[], int idx\_left, int idx\_right);

void tree\_print(Tree t, size\_t depth);

void tree\_infix(Tree t);

#endif

#endif

**tree.c --------------------------------------------------------------------------------------------------------------------------------------------------------**

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#include <math.h>

#include "tree.h"

int get\_priority(char c)

{

switch (c) { // Приоритеты операций

case '+': case '-': return 1;

case '\*': case '/': return 2;

case '^': return 3;

}

return 100; // Приоритет чисел, скобок, переменных, etc.

}

Tree tree\_create(Token tokens[], int idx\_left, int idx\_right)

{

Tree t = (Tree) malloc(sizeof(struct tree\_node));

if (idx\_left > idx\_right) { // Для корректной обработки унарного минуса

return NULL;

}

if (idx\_left == idx\_right) { // Тривиальный случай рекурсии (число, переменная)

t->node = tokens[idx\_left];

t->left = NULL;

t->right = NULL;

return t;

}

int priority;

int priority\_min = get\_priority('a');

int brackets = 0; // Счётчик открытых скобок

int op\_pos;

// Поиск позиции последнего оператора с наименьшим приоритетом

for (int i = idx\_left; i <= idx\_right; ++i) {

if ((tokens[i].type == BRACKET) && (tokens[i].data.is\_left\_bracket)) {

++brackets;

continue;

}

if ((tokens[i].type == BRACKET) && !(tokens[i].data.is\_left\_bracket)) {

--brackets;

continue;

}

if (brackets > 0) { // Пропуск того, что в скобках

continue;

}

if (tokens[i].type == OPERATOR) {

priority = get\_priority(tokens[i].data.operator\_name);

if (priority <= priority\_min) {

priority\_min = priority;

op\_pos = i;

}

}

}

if ((priority\_min == get\_priority('a')) &&

(tokens[idx\_left].type == BRACKET) &&

(tokens[idx\_left].data.is\_left\_bracket) &&

(tokens[idx\_right].type == BRACKET) &&

!(tokens[idx\_right].data.is\_left\_bracket)) {

free(t);

return tree\_create(tokens, idx\_left + 1, idx\_right - 1);

}

// Операции +, -, \* и т.п. -- левоассоциативные, а степень -- правоассоц.,

// т.е. 2^3^4 == 2^(3^4), а не (2^3)^4, как если бы была левоассоц.

if (tokens[op\_pos].data.operator\_name == '^') {

// Поэтому ищем самый левый оператор степени, связанный с текущим

brackets = 0;

for (int i = op\_pos; i >= idx\_left; --i) {

if ((tokens[i].type == BRACKET) && !(tokens[i].data.is\_left\_bracket)) {

++brackets;

continue;

}

if ((tokens[i].type == BRACKET) && (tokens[i].data.is\_left\_bracket)) {

--brackets;

continue;

}

if (brackets > 0) {

continue;

}

if (tokens[i].type == OPERATOR) {

priority = get\_priority(tokens[i].data.operator\_name);

if (priority == 3) {

op\_pos = i;

}

}

}

}

t->node = tokens[op\_pos]; // Запись оператора

t->left = tree\_create(tokens, idx\_left, op\_pos - 1);

t->right = tree\_create(tokens, op\_pos + 1, idx\_right);

if (t->right == NULL) {

printf("Epic fail: operator at the expression's end.");

exit(1);

}

return t;

}

void tree\_delete(Tree \*t)

{

if ((\*t) != NULL) {

tree\_delete(&((\*t)->left));

tree\_delete(&((\*t)->right));

}

free(\*t);

\*t = NULL;

}

void tree\_infix(Tree t)

{

if (t != NULL) {

if (t->left && t->right)

printf("(");

tree\_infix(t->left);

token\_print(&(t->node));

tree\_infix(t->right);

if (t->right && t->left)

printf(")");

}

}

void tree\_print(Tree t, size\_t depth)

{

if (t != NULL) {

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

printf("\t");

}

token\_print(&(t->node)); printf("\n");

tree\_print(t->left, depth + 1);

tree\_print(t->right, depth + 1);

}

}

**lexer.h --------------------------------------------------------------------------------------------------------------------------------------------------------**

#ifndef \_\_LEXER\_H\_\_

#define \_\_LEXER\_H\_\_

#include <stdbool.h>

typedef enum {

FINAL, // Идентификатор конца входной строки

INTEGER,

FLOATING,

OPERATOR,

VARIABLE,

BRACKET

} TokenType;

typedef struct {

TokenType type;

union {

int value\_int;

float value\_float;

char operator\_name;

bool is\_left\_bracket; // Левая скобка - истина, правая - ложь

char variable\_name;

} data;

} Token; // Именно они будут в узлах дерева выражений

void token\_print(Token \*t);

void token\_next(Token \*t); // Считать в \*t следующий "кусок" входной строки

#endif

**lexer.c**

**--------------------------------------------------------------------------------------------------------------------------------------------------------**

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#include <ctype.h>

#include "lexer.h"

void token\_next(Token \*t)

{

static bool can\_be\_unary = true; // http://en.wikipedia.org/wiki/Static\_variable

char c;

do { // Избавление от пробельных литер

c = fgetc(stdin);

} while (isspace(c));

if (c == EOF) { // The end

t->type = FINAL;

}

else if (isalpha(c) || c == '\_') { // Переменные

t->type = VARIABLE;

t->data.variable\_name = c;

can\_be\_unary = false;

}

else if (isdigit(c)) { // Числа

float result;

ungetc(c, stdin);

scanf("%f", &result);

if (result == (int) result) {

t->type = INTEGER;

t->data.value\_int = (int) result;

} else {

t->type = FLOATING;

t->data.value\_float = result;

}

can\_be\_unary = false;

}

else if (c == '(' || c == ')') {

t->type = BRACKET;

t->data.is\_left\_bracket = (c == '(');

can\_be\_unary = t->data.is\_left\_bracket;

}

else if (can\_be\_unary && (c == '-' || c == '+')) { // Учёт минуса перед числом

int m = (c == '+') ? +1 : -1; // Знак

do {

c = fgetc(stdin);

} while (isspace(c));

if (isdigit(c)) {

ungetc(c, stdin);

token\_next(t); // После минуса и т.д. надо число считать

if (t->type == INTEGER) {

t->data.value\_int = m \* (t->data.value\_int);

} else {

t->data.value\_float = m \* (t->data.value\_float);

}

} else {

ungetc(c, stdin);

t->type = OPERATOR;

t->data.operator\_name = '-';

can\_be\_unary = true;

}

}

else {

t->type = OPERATOR;

t->data.operator\_name = c;

can\_be\_unary = true;

}

}

void token\_print(Token \*t)

{

switch (t->type) {

case FINAL:

break;

case INTEGER:

printf("%d", t->data.value\_int);

break;

case FLOATING:

printf("%lg", t->data.value\_float);

break;

case VARIABLE:

printf("%c", t->data.variable\_name);

break;

case OPERATOR:

printf("%c", t->data.operator\_name);

break;

case BRACKET:

printf("%c", (t->data.is\_left\_bracket) ? '(' : ')');

break;

}

}

**transform.h --------------------------------------------------------------------------------------------------------------------------------------------------------**

#ifndef \_\_TRANSFORM\_H\_\_

#define \_\_TRANSFORM\_H\_\_

#include "tree.h"

int list\_count(Tree root);

#endif

**transform.c**

**--------------------------------------------------------------------------------------------------------------------------------------------------------**

#include "tree.h"

#include "transform.h"

#include <math.h>

int list\_count(Tree root)

{

int result;

if ((root->left == NULL) && (root->right == NULL)) {

result = 1;

} else {

result = 0;

} if (root->left) {

result += list\_count(root->left);

} if (root->right) {

result += list\_count(root->right);

}

return result;

}

**main.c --------------------------------------------------------------------------------------------------------------------------------------------------------**

#include <stdio.h>

#include <stdlib.h>

#include "lexer.h"

#include "tree.h"

#include "transform.h"

int main(void)

{

Token tokens[256];

size\_t tokens\_qty = 0;

Token token;

token\_next(&token);

while (token.type != FINAL) {

tokens[tokens\_qty++] = token;

token\_next(&token);

}

Tree tree = tree\_create(tokens, 0, tokens\_qty - 1);

printf("\nExpression tree:\n");

tree\_print(tree, 0);

printf("вот столько операндов: %d\n", list\_count(tree));

printf("\nTree's infix linearization:\n");

tree\_infix(tree);

printf("\n");

return 0;

}

**9. Дневник отладки** должен содержать дату и время сеансов отладки и основные события(ошибки в сценарии и программе,нестандартные ситуации) и краткие комментарии к ним. В дневнике отладки приводятся сведения об использовании других ЭВМ, существенном участии преподавателя и других лиц в написании и отладке программы.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| № | Лаб. | Дата | Время | Событие | Действие по исправлению | Примечание |
|  | или |  |  |  |  |  |
|  | дом. |  |  |  |  |  |
| 24 | дома | 25.05 | 02:30 | закончил | никаких |  |

1. **Замечания автора** по существу работы
2. **Выводы**

В лабораторной работе я познакомлся с разными видами записи алгебраических выражений и деревом выражений. Работа была интересной, но сложной при работе с динамической памятью, при отображении обратной польской нотации в дерево выражений, при реализации различных алгоритмов для деревьев и при отладке программы, ведь с памятью нужно работать аккуратно. Деревья выражений имеют огромное значения для хранения, представления различных алгебраических выражений + для работы калькуляторов. Я уверен, что полученные знания мне пригодятся в будущем.

Подпись студента: