**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра МО ЭВМ**

отчет

**по лабораторной работе №2**

**по дисциплине «Алгоритмы и структуры данных»**

**Тема: «Упрощение префиксного выражения и проверка деления на 0 с использованием иерархических списков»**

Студент гр. 7382 Лящевская А. П.

Преподаватель Фирсов М. А.

Санкт-Петербург

2018

**Задание.**

Пусть выражение (логическое, арифметическое, алгебраическое\*) представлено иерархическим списком. В выражение входят константы и переменные, которые являются атомами списка. Операции представляются в префиксной форме ( (<операция> <аргументы>) ), либо в постфиксной форме (<аргументы> <операция>) ). Аргументов может быть 1, 2 и более. Например (в префиксной форме): (+ a (\* b (- c))) или (OR a (AND b (NOT c))).

19) Арифметическое выражение, упрощение и проверка деления на 0 (простая), префиксная форма выражения.

**Пояснение задания.**

На вход программе подаётся последовательность чисел, символов и знаков арифметических операций (арифметическое выражение) в форме иерархического списка в префиксной форме. Итогом завершения программы должно быть преобразованное выражение по принципу упрощения, где под принципом упрощения подразумевается сокращение числа лишних операций.

**Описание алгоритма.**

Введённое пользователем выражение прежде всего подлежит проверке на пустоту. Далее в процессе обработки выражения может быть выявлена ошибка пропуска знака в начале выражения.

Программа анализирует структуру самого выражения, компактно записывает ее в иерархический список, обрабатывает на сократимость и выводит на экран результат в соответствии с условиями сокращения. При обнаружении в выражении или подвыражении деления на 0, программа сообщает об этом и заканчивает работу.

Процесс записи в иерархический список представлен следующим образом. Список инициализируется первым элементом и подается в функцию его заполнения. Далее идет процесс поиска индексов аргументов. Для того чтобы ограничить его, находится нужная закрывающая скобка выражения. И поиск за ее границу не пойдет. Ожидается, что первым аргументом в префиксной записи будет знак. И исходя из этого, если после нахождения знака встречается "-", то он уже интерпретируется программой как минус перед числом. Если же по ходу просмотра нашего основного выражения встречается скобка, то поиск аргументов продолжается для найденного подвыражения рекурсивно.

Процесс упрощения выражения так же начинается с верхушки списка. Проходя по выражению, определяя его знак, программа делает возможные сокращения. И если выражение после этого осталось пусто или в нем присутствует лишь один аргумент, то происходит повышение его уровня на 1. И, благодаря рекурсивному упрощению, это получается сделать.

**Описание функций**

|  |  |  |  |
| --- | --- | --- | --- |
| **Название** | **Выходной параметр** | **Входные данные** | **Описание** |
| **abort\_** | - | **сonst char \* –** анализируемая строка | Выводит сообщение об ошибке и завершает программу аварийно. |
| **fill\_list** | - | **int –** глубина рекурсии  **Node \* –** указатель на элемент списка;  **сhar \* –** анализируемая строка | Заполняет иерархический список. |
| **find\_sec\_bracket** | int | **сhar \* –** анализируемая строка | Ищет нужную закрывающую скобку для найденной открытой скобки. |
| **reduct** | - | **int –** глубина рекурсии;  **Node \* –** указатель на элемент списка; | Редактирует список полученный на вход, производя возможные сокращения для исходного выражения. |
| **check\_division** | - | **Node \* –** указатель на элемент списка; | Проверяет отредактированное выражение на деление на 0. |
| **print\_list** | - | **Node \* –** указатель на элемент списка; | Печатает новое сокращенное выражение. |

**Описание структур данных**

|  |  |  |
| --- | --- | --- |
| **Название** | **Поле** | **Описание** |
| Node | int type; | Принимает значение 1, если элемент хранит символ, 2 – число и 3 – ссылается на выражение. |
| struct Node\* next; | Здесь хранится указатель на следующий элемент списка. |
| char symbol; | Встретившийся символ. |
| struct Node\* expr; | Указатель на встретившееся выражение. |
| int digit; | Встретившееся число. |

**Тестирование.**

|  |  |
| --- | --- |
| Исходное выражение: | Результат: |
| (+ (- 0 4) (/ (+ 0 (\* 5 1)) 1) (+ (\* (/ 6))) ) | (+ (- 0 4 ) 5 6 ) |
| (\*(- 0 -3)(+ 0 k)) | (\* (- 0 -3 ) k ) |
| (- 0 4 (\* 1 t ) (/ 1 (+ i 0) ) (- 0 0 0 (\* l 1) (\* l 0 0))) | (- 0 4 t (/ 1 i ) (- 0 l ) ) |
| (- 0 0 (\* 8 0 (+ -7 0))) | 0 |
|  | Expression is empty. |
| (+ 1 2 0 r y t(- 1 6) (/ 1 m 2 3 \* 1 4 k )) | Incorrect expression, needs more brackets |
| (/ (- 1 2) (\* i 0)) | Error - deviision by zero! |

**Вывод.**

В процессе выполнения лабораторной работы были продуманы, созданы и реализованы на практике алгоритмы и методы работы по построению иерархического списка, а также по работе с ним. С помощью построения списка в памяти и вывода его на стандартный поток мы смогли проверить на корректность, сократить и вывести на экран сокращенное префиксное арифметическое выражение.

**Приложение 1. Код программы.**

* [**Run\_test.sh**](https://github.com/makometr/AiSD/pull/37/files#diff-240c2792fd92b595432e18629f6e16b6)

#!/bin/bash

Gcc ./Sourse/Lab2.1.c -o Lab2

echo -e '\_\_\_\_\_\_\_\nTest 1:'

cat ./Tests/Test1.txt

echo -e '\_\_\_\_\_\_\_\nTesting:\n'

./Sourse/Lab2 < ./Tests/Test1.txt

echo -e ''

echo -e '\_\_\_\_\_\_\_\nTest 2:'

cat ./Tests/Test2.txt

echo -e '\_\_\_\_\_\_\_\nTesting:\n'

./Sourse/Lab2 < ./Tests/Test2.txt

echo -e ''

echo -e '\_\_\_\_\_\_\_\nTest 3:'

cat ./Tests/Test3.txt

echo -e '\_\_\_\_\_\_\_\nTesting:\n'

./Sourse/Lab2 < ./Tests/Test3.txt

echo -e ''

echo -e '\_\_\_\_\_\_\_\nTest 4:'

cat ./Tests/Test4.txt

echo -e '\_\_\_\_\_\_\_\nTesting:\n'

./Sourse/Lab2 < ./Tests/Test4.txt

echo -e ''

echo -e '\_\_\_\_\_\_\_\nTest 5:'

cat ./Tests/Test5.txt

echo -e '\_\_\_\_\_\_\_\nTesting:\n'

./Sourse/Lab2 < ./Tests/Test5.txt

echo -e ''

echo -e '\_\_\_\_\_\_\_\nTest 6:'

cat ./Tests/Test6.txt

echo -e '\_\_\_\_\_\_\_\nTesting:\n'

./Sourse/Lab2 < ./Tests/Test6.txt

echo -e ''

echo -e '\_\_\_\_\_\_\_\nTest 7:'

cat ./Tests/Test7.txt

echo -e '\_\_\_\_\_\_\_\nTesting:\n'

./Sourse/Lab2 < ./Tests/Test7.txt

* [**Lab2.1.c**](https://github.com/makometr/AiSD/pull/37/files#diff-240c2792fd92b595432e18629f6e16b6)

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <stdarg.h>

#include <ctype.h>

typedef struct Node {

int type; //Хранит: 1 - символ, 2 - число, 3 - выражение

struct Node \*next;

char symbol;

struct Node \*expr;

int digit;

} Node;

void abort\_(const char \* s, ...); //Экстренное завершение программы с сообщением

void fill\_list(int, Node \*, char \*);

int find\_sec\_bracket(char \*);

void check\_division(Node \*);

void print\_list(Node \*node);

int reduct(int, Node \*node); //Возвращает инфу об аргументе после аргумента-знака

int main() {

char \*str = (char \*)malloc(100 \* sizeof(char));

fgets(str, 100, stdin);

if (!strcmp(str, "\n"))

abort\_("Expression is empty.");

printf("Before: %s\n", str);

printf("Args in expr with levels:\n");

Node \*top = (Node\*)malloc(sizeof(Node));

fill\_list(0, top, str); //Заполняет список, выводя аргументы на своих уровнях

printf("\nReduction process:\n");

reduct(0, top); //Редактирует список, выводя сообщения о каждом сокращении

printf("\nAfter: ");

if(top->next->next == NULL) { //Выражение обнулилось

printf("0\n");

return 0;

}

if(top->next->next->next != NULL)

print\_list(top);

else { //Во внешнем выражении остался один аргумент

if(top->next->type == 1)

printf("%c", top->next->symbol);

if(top->next->type == 2)

printf("%d", top->next->digit);

if(top->next->type == 3)

print\_list(top->next->expr);

}

check\_division(top); //Проверка деления на ноль

printf("\n");

return 0;

}

void abort\_(const char \* s, ...) {

va\_list args;

va\_start(args, s);

vfprintf(stderr, s, args);

fprintf(stderr, "\n");

va\_end(args);

abort();

}

void fill\_list(int deep, Node \*node, char \*str) {

int sign = 0;

int end\_expr = find\_sec\_bracket(str + 1);

for(int i = 0; i < deep; i++) //Отступы при глубине рекурсии deep

printf("%d|\t", i);

printf("Start of expr.\n");

for(int i = 1; i < end\_expr; i++) { //Обработка выражения до его закрывающей скобки

if(isalpha(str[i]) || str[i] == '\*' || str[i] == '/' || str[i] == '-' || str[i] == '+') {

if(!sign || isalpha(str[i])) { //Если это буква или знак, но не минус перед числом, то

node->type = 1; //инициализируем элемент-символ

node->symbol = str[i];

for(int i = 0; i < deep; i++)

printf("%d|\t", i);

printf("%d| %c\n", deep, str[i]);

node->next = (Node \*)malloc(sizeof(Node));

node = node->next; //Указатель перемещается к следующему элементу списка

if(str[i] == '-' || str[i] == '+' || str[i] == '\*' || str[i] == '/')

sign++; //В следующий раз в этом подвыражении будут только минусы перед числами

continue;

}

}

if(isdigit(str[i]) || (str[i] == '-' && sign)) { //Если это число или минус перед ним,то

node->type = 2; //инициализируем элемент-число

node->digit = atoi(str + i);

while(str[i] == '-')

i++;

while(isdigit(str[i + 1]))

i++;

for(int i = 0; i < deep; i++)

printf("%d|\t", i);

printf("%d| %d\n", deep, node->digit);

node->next = (Node \*)malloc(sizeof(Node));

node = node->next;

}

if(str[i] == '(') { //Если встретили подвыражение, то

node->type = 3; //инициализируем элемент-выраженияе

node->expr = (Node \*)malloc(sizeof(Node));

fill\_list(deep + 1, node->expr, str + i); //И переходим к его заполнению рекурсивно

node->next = (Node \*)malloc(sizeof(Node));

node = node->next;

i += find\_sec\_bracket(str + i + 1); //Перемещаем индекс хождения по выражению i

}

}

node->next = NULL; //Обозначим границу списка

for(int i = 0; i < deep; i++)

printf("%d|\t", i);

printf("End of expr.\n");

}

int find\_sec\_bracket(char \*str) { //Ищет парную скобку в выражении

int left\_br = 1, right\_br = 0;

int i;

for (i = 0; left\_br != right\_br && i < strlen(str); i++) {

if (str[i] == '(')

left\_br++;

if (str[i] == ')')

right\_br++;

}

if(i > strlen(str))

abort\_("\nIncorrect expression, needs more brackets");

return i;

}

void check\_division(Node \*node) { //Проверка деления на ноль

while(node->next) {

if (node->type == 1 && node->symbol == '/') {

node = node->next;

while(node->next->next) { //Переходим от аргумента к аргументу

node = node->next;

if(node->type == 2 && node->digit == 0) //Пока не встретим нужный элемент в выражении с делением

abort\_("\nError - deviision by zero!");

}

}

if(node->type == 3)

check\_division(node->expr);

node = node->next;

}

}

void print\_list(Node \*node) { //Последовательный вывод выражения за выражением

printf("(");

while(node->next ) {

if(node->type == 1)

printf("%c", node->symbol);

if(node->type == 2)

printf("%d", node->digit);

if(node->type == 3)

if(node->expr->next)

print\_list(node->expr);

printf(" ");

node = node->next;

}

printf(")");

}

int reduct(int deep, Node \*node) { //Редактирует содержимое списка

char sign;

Node \*start\_node = node;

if(node->type == 1)

sign = node->symbol;

else

abort\_("\nError - pass mark"); //Если первый элемент не символ, вывод ошибки об этом

for(int i = 0; i < deep; i++) //Отступ по глубине

printf("%d|\t", i);

printf("Start of expr.\n");

for(int i = 0; i < deep; i++)

printf("%d|\t", i);

printf("%d| %c\n", deep, sign);

while(node->next) {

for(int i = 0; i < deep && (node->next->type == 1 || node->next->type == 2); i++)

printf("%d|\t", i); //Печатаем отступы если символ или цифра

if(node->next->type != 3 && node->next->next)

printf("%d| ", deep);

if(node->next->type == 1)

printf("%c ", node->next->symbol);

if(node->next->type == 2)

printf("%d ", node->next->digit);

if(node->next->type == 3) { //Встретилось подвыражение

if(reduct(deep + 1, node->next->expr)) { //Если в нем все сократилось:

if(sign == '+' || sign == '-') { //для выражения со знаком "+" или "-"

node->next = node->next->next; //перемещаем указатель, стирая элемент пустого подвыражения

}

if(sign == '/') { //для выражения со знаком "/"

node->next->type = 2; //элемент-подвыражение становится элементом-числом (0)

node->next->digit = 0;

}

if(sign == '\*') { //для выражения со знаком "\*"

start\_node->next->next = NULL; //Все выражение обнуляется

break;

}

}

else //Если же не все сократилось

if(node->next->expr->next->next->next == NULL) {//и остался один элемент

if(node->next->expr->next->type == 1) { //если это символ

node->next->type = 1; //поднимаем его на уровень выше

node->next->symbol = node->next->expr->next->symbol;

for(int i = 0; i < deep; i++)

printf("%d|\t", i);

printf("%d| level up with %c", deep, node->next->symbol);

}

if(node->next->expr->next->type == 2) { //если это цифра

node->next->type = 2; //поднимаем ее на уровень выше

node->next->digit = node->next->expr->next->digit;

for(int i = 0; i < deep; i++)

printf("%d|\t", i);

printf("%d| level up with %d", deep, node->next->digit);

}

if(node->next->expr->next->type == 3) { //если это другое выражение

node->next->expr = node->next->expr->next->expr;//поднимаем его на уровень выше

for(int i = 0; i < deep; i++)

printf("%d|\t", i);

printf("%d| level up with expr", deep);

}

}

}

if(sign == '+' || sign == '-') { //Если знак текущего выражения "+" или "-"

if(node->next->type == 2) { //и встретился элемент-число 0

if(node->next->digit == 0 && (node->symbol != '-' && (node->next->next->next || node->symbol != '\*'))) {

printf("delete\n"); //и притом он не первый при "-"

node->next = node->next->next; //- удаляем его

continue;

}

}

}

if((sign == '/' || sign == '\*') && node->next->type == 2) { //Если знак текущего выражения "/" или "\*"

if(node->next->digit == 1 && (node->symbol != '/' && (node->next->next->next || node->symbol != '\*'))) {

printf("delete\n"); //и встретился элемент-число 1

node->next = node->next->next; //и притом он не первый при делении - удаляем его

continue;

}

}

if(sign == '\*' || node->symbol == '/') //Если знак текущего выражения "/" или "\*"

if(node->next->type == 2 && node->next->digit == 0) { //и мы встречаем элемент-число 0

printf("delete expr\n"); //обнуляем выражение

start\_node->next->type = 2;

start\_node->next->digit = 0;

start\_node->next->next = NULL;

break; //заканчиваем его обработку

}

if(node->next->type != 0 && node->next->type != 3)

printf("\n");

node = node->next; //переходим к следующему выражению

}

for(int i = 0; i < deep; i++)

printf("%d|\t", i);

printf("End of expr.\n");

return (start\_node->next->next == NULL); //Возврат инфы о существовании второго элемента

}