

**Отчет по лабораторной работе № 24** по курсу Фундаментальная информатика

Студент группы М8О-204Б-22, Филиппов Фёдор Иванович, № по списку 18

Контакты: [gooselinjk@yandex.ru](mailto:gooselinjk@yandex.ru)

Работа выполнена: “28” сентября 2023 года

Преподаватель: Потенко М.А., каф.806

Входной контроль знаний с оценкой \_\_\_\_\_

Отчёт сдан “29” сентября 2023 года, ИО \_\_\_\_\_

Подпись преподавателя \_\_\_\_\_

**1. Тема:** Алгоритмы и структуры данных

**2. Цель работы:** Составить и отладить программу выполнения заданных преобразований арифметических выражений с применением деревьев.

**3. Задание (вариант № 18):** вынести из произведений унарные минусы  $a*(-b)^3 - (a*b^3)$ ,  $a*(-b)^4*(-5) \rightarrow a*b^4*5$

**4. Оборудование**

ЭВМ — ноутбук HP 255 G8

Процессор — Ryzen 5500U

ОП — 16384 МБ

НМД — 1048576 МБ

Терминал Windows Powershell (с возможностью переключения на UNIX)

**5. Программное обеспечение**

Операционная система семейства Windows, наименование Windows 11 Home, версия 22H2

Редактор текстов — Sublime Text

Утилиты операционной системы — терминал Windows Powershell

Прикладные системы и программы — Visual Studio Code, Visual Studio

**6. Идея, метод, алгоритм** решения задачи (в формах: словесной, псевдокода, графической или формальные с пред- и постусловиями)

Моя программа будет делиться на 3 файла: заголовочный файл математического парсера (parse.h), его реализация (parse.c) и основной исходный файл команды(main.c).

### **parse.h:**

Определение перечисления token: Здесь объявлено перечисление token, которое определяет три типа токенов: OPERATOR, NUMBER, и VARIABLE. Эти типы используются для классификации элементов математического выражения.

Определение структуры node: node представляет узел синтаксического дерева. Он содержит информацию о типе узла (type), операторе (oper), числовом значении (val), переменной (var), наличии унарного минуса (minus), указателях на левого и правого потомков (l\_ch и r\_ch), указателе на родителя (parent) и указателе на следующий узел для использования в очереди (next).

Определение структуры tree: tree представляет собой синтаксическое дерево и содержит указатель на корневой узел (root).

Определение структуры queue: queue представляет очередь, используемую для разбора выражения. Она содержит указатели на начало и конец очереди (front и back).

Прототипы функций: В файле parse.h также объявлены прототипы всех функций, которые будут реализованы в файле parse.c. Эти функции включают в себя создание узла, создание очереди, добавление и извлечение элементов из очереди, парсинг строки, создание синтаксического дерева, печать дерева, упрощение дерева и другие операции.

### **parse.c:**

Реализации функций: В файле parse.c реализованы все функции, объявленные в parse.h. Эти функции выполняют различные операции, связанные с разбором математических выражений и работой с синтаксическим деревом. Например, create\_node создает узел, create\_queue создает очередь, enqueue добавляет узел в очередь, dequeue извлекает узел из очереди, parse выполняет разбор строки и создание очереди обратной польской записи, create\_tree строит синтаксическое дерево, print\_tree выводит дерево в отформатированном виде, reduce\_minuses выполняет сокращение унарных минусов в дереве, и так далее.

## **main.c:**

Функция `main`: Файл `main.c` содержит функцию `main`, которая является точкой входа в программу. Эта функция выполняет следующие действия:

Пользователь вводит математическое выражение с клавиатуры, и оно сохраняется в строке `str`.

Длина строки `str` вычисляется в переменной `len`.

Вызывается функция `parse` для разбора строки и создания очереди обратной польской записи (RPN).

Создается синтаксическое дерево с помощью функции `create_tree`.

Синтаксическое дерево упрощается с помощью функции `reduce_minuses`.

Выводятся результаты разбора: синтаксическое дерево в отформатированном виде и инфиксное выражение, полученное из дерева с помощью функции `infix`.

Затем происходит освобождение выделенной памяти с использованием функций `free_node` и `delete_tree`.

Завершение программы: Функция `main` завершается с кодом возврата 0, указывая на успешное выполнение программы.

## **7. Сценарий выполнения работы** (план работы, первоначальный текст программы в черновике и тесты, либо соображения по тестам)

Когда я начинал разработку данной программы для разбора математических выражений, мой первый шаг был определить структуры данных и функции, которые мне понадобятся для реализации алгоритма Shunting Yard и построения синтаксического дерева. Для этого я создал файл `parse.h`, в котором объявил перечисления, структуры и прототипы функций, необходимые для работы программы.

Затем я перешел к реализации самих функций в файле `parse.c`. Начал с создания узла синтаксического дерева (`create_node`), функции для работы с очередью (`create_queue`, `enqueue`, `dequeue`), и других вспомогательных функций.

Следующим важным этапом было реализовать алгоритм Shunting Yard. Я создал функцию `parse`, которая принимает входную строку и преобразует её в обратную польскую запись, используя стек операторов и выходную очередь RPN. При этом, я следил за правильным приоритетом операторов и их ассоциативностью, как это описано в тексте.

Далее, я реализовал функцию `buildSyntaxTree`, которая на основе RPN очереди строит синтаксическое дерево. Это было одним из самых важных шагов в разработке программы.

После успешной реализации алгоритма `Shunting Yard` и построения синтаксического дерева, я перешел к функциям, которые упрощают синтаксическое дерево (`reduce_minuses`) и преобразуют его обратно в инфиксную форму (`infix`).

Теперь, когда основная логика программы была готова, я приступил к тестированию. Вот предварительный тестовый сценарий:

Введите выражение:

`2 + 3 * 4`

Ожидаемый результат:

=== Дерево исходного выражения ===

```
      +
     /\
    2  *
     /\
    3  4
```

=== Дерево преобразованного выражения ===

```
      +
     /\
    2  12
```

=== Инфиксная запись ===

`2 + 12`

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

C main.c > ...

```
1
2  #include <ctype.h>
3  #include <stdio.h>
4  #include <string.h>
5  #include "parse.h"
6
7  #define MAX_LEN 250
8
9  int main() {
10     char str[MAX_LEN];
11     printf("Введите выражение:\n");
12     fgets(str, MAX_LEN, stdin);
13     int len;
14     for(len = 0; str[len] != '\0'; ++len);
15     queue* q = parse(str, len);
16
17     printf("\n=== Дерево исходного выражения ===\n");
18     tree* t = create_tree(q);
19     print_tree(t->root, 0);
20
21     printf("\n=== Дерево преобразованного выражения ===\n");
22     reduce_minuses(t->root);
23     print_tree(t->root, 0);
24
25     printf("\n=== Инфиксная запись ===\n");
26     char* infix_str = infix(t->root);
27     printf("%s\n", infix_str);
28
29     free(infix_str);
30     delete_tree(t);
31     free(q);
32
33     return 0;
34 }
```

```
parce.c > ...
1  #include "parse.h"
2
3  // Функция для создания узла дерева
4  node* create_node(token type, char op, int val, char var, bool minus) {
5      node* n = (node*)malloc(sizeof(node));
6      n->type = type;
7      n->oper = op;
8      n->minus = minus;
9      n->val = val;
10     n->var = var;
11     n->l_ch = NULL;
12     n->r_ch = NULL;
13     n->next = NULL;
14     return n;
15 }
16
17 // ===== ОЧЕРЕДЬ =====
18 // Функция для создания очереди
19 queue* create_queue() {
20     queue* q = (queue*)malloc(sizeof(queue));
21     q->back = NULL;
22     q->front = NULL;
23     return q;
24 }
25
26 // Функция для добавления узла в очередь
27 void enqueue(queue* q, node* n) {
28     n->next = NULL; // Инициализируем указатель на следующий узел нового узла
29     if (q->back == NULL) {
30         // Если очередь пуста, устанавливаем и back, и front на новый узел
31         q->back = n;
32         q->front = n;
33     } else {
34         // Если очередь не пуста, обновляем указатель на следующий узел текущего последнего элемента
35         q->back->next = n;
36         // Обновляем указатель back на новый узел
37         q->back = n;
38     }
39 }
```

```

41 // Функция для извлечения узла из очереди
42 node* dequeue(queue* q) {
43     if (q->front == NULL) {
44         return NULL;
45     }
46     // Получаем узел из начала очереди
47     node* node = q->front;
48     // Обновляем указатель front на следующий узел
49     q->front = node->next;
50     // Если очередь становится пустой, также обновляем указатель back
51     if (q->front == NULL) {
52         q->back = NULL;
53     }
54     // Возвращаем извлеченный узел
55     return node;
56 }
57
58 // Функция для удаления очереди
59 void delete_queue(queue* q) {
60     if (q == NULL) {
61         return;
62     }
63
64     while (q->front != NULL) {
65         node* temp = q->front; // Получаем узел из начала очереди
66         q->front = q->front->next; // Перемещаем указатель front к следующему узлу
67         free(temp); // Освобождаем память удаляемого узла
68     }
69
70     // После удаления всех узлов обязательно устанавливаем back в NULL
71     q->back = NULL;
72 }
73 // =====
74
75 // Функция, которая анализирует строку выражения
76 // и преобразует ее в очередь обратной польской записи (!)
77 queue* parse(char* str, int l) {
78     char tokens[MAX_SIZE];
79     int t_idx = 0;

```

```

118     int t_idx = 0;
80     queue* q = create_queue();
81     char s[MAX_SIZE]; // СТЕК
82     int s_top = -1;
83
84     for (int i = 0; i < l; i++) {
85         char c = str[i];
86         bool unary_minus = false;
87
88         if (isdigit(c)) { // ЕСЛИ ЧИСЛО
89             int num = 0;
90             // Продолжаем считать символы, пока не встретится нецифровой символ
91             while (i < l && isdigit(str[i])) {
92                 num = num * 10 + (str[i] - '0');
93                 i++;
94             }
95             i--; // Уменьшаем итератор, чтобы не пропустить символ после числа
96
97             // Есть ли унарный минус?
98             if (t_idx > 0 && tokens[t_idx - 1] == '-') {
99                 if (t_idx - 2 == -1 || tokens[t_idx - 2] == '(') {
100                     s_top--; // Убираем унарный минус из стека
101                     unary_minus = true;
102                 }
103             }
104
105             node* node = create_node(NUMBER, ' ', num, ' ', unary_minus);
106             enqueue(q, node);
107
108             tokens[t_idx++] = 'v'; // 'v' - значит значение (число или переменная)
109         }
110         else if (isalpha(c)) { // ЕСЛИ ПЕРЕМЕННАЯ
111             // Есть ли унарный минус?
112             if (t_idx > 0 && tokens[t_idx - 1] == '-') {
113                 if (t_idx - 2 == -1 || tokens[t_idx - 2] == '(') {
114                     s_top--; // Убираем унарный минус из стека
115                     unary_minus = true;
116                 }
117             }
118         }

```



```

119     node* node = create_node(VARIABLE, ' ', 0, c, unary_minus);
120     enqueue(q, node);
121
122     tokens[t_idx++] = 'v'; // 'v' - значит значение (число или переменная)
123 }
124 else { // ЕСЛИ ОПЕРАТОР
125     char op = str[i];
126     if (op == '+' || op == '-' || op == '*' || op == '/' || op == '(' || op == ')') {
127         tokens[t_idx++] = op; // Добавляем оператор в токены
128     }
129     switch (op) {
130         case '+':
131         case '-':
132             // Обрабатываем операторы + и -
133             while (s_top >= 0 &&
134                 (s[s_top] == '+' || s[s_top] == '-'
135                 || s[s_top] == '*' || s[s_top] == '/'))
136             {
137                 // Извлекаем операторы из стека и добавляем их в очередь
138                 node* op_node = create_node(OPERATOR, s[s_top], 0, ' ', unary_minus);
139                 enqueue(q, op_node);
140                 s_top--;
141             }
142             // Помещаем текущий оператор в стек
143             s_top++;
144             s[s_top] = op;
145             break;
146         case '*':
147         case '/':
148             // Обрабатываем операторы * и /
149             while (s_top >= 0
150                 && (s[s_top] == '*' || s[s_top] == '/'))
151             {
152                 // Извлекаем операторы из стека и добавляем их в очередь
153                 node* op_node = create_node(OPERATOR, s[s_top], 0, ' ', unary_minus);
154                 enqueue(q, op_node);
155                 s_top--;
156             }
157             // Помещаем текущий оператор в стек

```

```

158         s_top++;
159         s[s_top] = op;
160         break;
161     case '(':
162         s_top++;
163         s[s_top] = op;
164         break;
165     case ')':
166         while (s_top >= 0 && (s[s_top] != '(')) {
167             node* op_node = create_node(OPERATOR, s[s_top], 0, ' ', unary_minus);
168             enqueue(q, op_node);
169             s_top--;
170         }
171         s_top--; // Убираем открывающую скобку
172         break;
173     default:
174
175         break;
176     }
177 }
178 }
179 // Переключаем остальные операторы
180 while (s_top >= 0) {
181     if (s[s_top] != '(') {
182         node* op_node = create_node(OPERATOR, s[s_top], 0, ' ', false);
183         enqueue(q, op_node);
184     }
185     s_top--;
186 }
187 return q;
188 }
189
190 // Функция для добавления узлов в дерево
191 node* add_to_tree(node* n, queue* q) {
192     if (n == NULL) {
193         return NULL;
194     }
195     if (n->type == NUMBER || n->type == VARIABLE) {
196         return n;

```

```

197     }
198     node* temp = n;
199     temp->l_ch = add_to_tree(dequeue(q), q);
200     temp->r_ch = add_to_tree(dequeue(q), q);
201     return temp;
202 }
203
204 // Функция для создания синтаксического дерева
205 tree* create_tree(queue* q) {
206     // РАЗВОРАЧИВАЕМ ОЧЕРЕДЬ
207     queue* reverse_q = create_queue();
208     node* stack[MAX_SIZE];
209     int top = -1;
210     while (q->front != NULL) {
211         top++;
212         stack[top] = dequeue(q);
213     }
214     while (top >= 0) {
215         enqueue(reverse_q, stack[top]);
216         top--;
217     }
218     // =====
219     tree* t = (tree*)malloc(sizeof(tree));
220     t->root = dequeue(reverse_q);
221     t->root->l_ch = add_to_tree(dequeue(reverse_q), reverse_q);
222     t->root->r_ch = add_to_tree(dequeue(reverse_q), reverse_q);
223
224     return t;
225 }
226
227 // Функция для печати дерева в отформатированном виде
228 void print_tree(node* root, int n) {
229     if (root == NULL) {
230         return;
231     }
232     print_tree(root->r_ch, n + 1);
233     for (int i = 0; i < n; i++) printf("    ");
234     print_node(root);
235     printf("\n");

```

```

236     print_tree(root->l_ch, n + 1);
237 }
238
239 // функция для вывода дерева в форматированном виде
240 void print_node(node* n) {
241     char minus;
242     if (n->minus) {
243         minus = '-';
244     } else {
245         minus = ' ';
246     }
247     switch (n->type) {
248         case NUMBER:
249             printf("%c%d ", minus, n->val);
250             break;
251         case VARIABLE:
252             printf("%c%c ", minus, n->var);
253             break;
254         case OPERATOR:
255             if (n->minus) {
256                 printf("-(%c) ", n->oper);
257             } else {
258                 printf("(%c) ", n->oper);
259             }
260             break;
261         // Обрабатываем другие типы, если необходимо
262         default:
263             // Обрабатываем недопустимый тип или предоставляем соответствующую обработку ошибок
264             break;
265     }
266 }
267
268 // Функция для сокращения минусов в дереве
269 bool reduce_minuses(node* root) {
270     if (root->type == NUMBER || root->type == VARIABLE) {
271         return root->minus;
272     }
273     if (root->oper == '*') {
274         bool unary = (reduce_minuses(root->l_ch) + reduce_minuses(root->r_ch)) % 2;

```

```

275     root->minus = unary;
276     root->l_ch->minus = false;
277     root->r_ch->minus = false;
278     return unary;
279 }
280 reduce_minuses(root->l_ch);
281 reduce_minuses(root->r_ch);
282 return false;
283 }
284
285 // Функция для инфиксного представления дерева
286 char* infix(node* root) {
287     char* str = (char*)malloc(MAX_SIZE);
288
289     if (root->type == NUMBER) {
290         if (root->minus) {
291             sprintf(str, "(-%d)", root->val);
292             return str;
293         }
294         sprintf(str, "%d", root->val);
295         return str;
296     }
297     if (root->type == VARIABLE) {
298         if (root->minus) {
299             sprintf(str, "(-%d)", root->val);
300             return str;
301         }
302         sprintf(str, "%c", root->var);
303         return str;
304     }
305
306     char* left = infix(root->l_ch);
307     char* right = infix(root->r_ch);
308     char op = root->oper;
309
310     if (root->minus) {
311         sprintf(str, "-(%s %c %s)", left, op, right);
312     } else if (op == '-' || op == '+') {
313         sprintf(str, "(%s %c %s)", left, op, right);

```

```
314     } else {
315         sprintf(str, "%s %c %s", left, op, right);
316     }
317
318     free(left);
319     free(right);
320
321     return str;
322 }
323
324 // Функция для освобождения узла и его детей
325 void free_node(node* n) {
326     if (n == NULL) {
327         return;
328     }
329     if (n->l_ch != NULL)
330         free_node(n->l_ch);
331     if (n->r_ch != NULL)
332         free_node(n->r_ch);
333     free(n);
334 }
335
336 // Функция для освобождения дерева
337 void delete_tree(tree* t) {
338     free_node(t->root);
339     t->root = NULL;
340 }
341
```

```

1  #ifndef PARSE
2  #define PARSE
3
4  #include <stdio.h>
5  #include <stdbool.h>
6  #include <stdlib.h>
7  #include <string.h>
8  #include <ctype.h>
9
10 #define MAX_SIZE 250
11
12 typedef enum {
13     OPERATOR,
14     NUMBER,
15     VARIABLE
16 } token;
17
18 typedef struct node {
19     struct node* parent;
20     struct node* l_ch;
21     struct node* r_ch;
22     token type;
23     char oper;
24     bool minus;
25     int val;
26     char var;
27     struct node* next; // Для очереди
28 } node;
29
30 typedef struct tree {
31     node* root;
32 } tree;
33
34 typedef struct {
35     node* back;
36     node* front;
37 } queue;
38
39 node* create_node(token type, char op, int val, char var, bool minus);

```

```
40
41 //===== ОЧЕРЕДЬ =====
42 queue* create_queue();
43
44 void enqueue(queue* q, node* n);
45
46 node* dequeue(queue* q);
47
48 void delete_queue(queue* q);
49 //=====
50
51 queue* parse(char* str, int l);
52
53 node* add_to_tree(node* n, queue* q);
54
55 tree* create_tree(queue* q);
56
57 void print_tree(node* root, int n);
58
59 void print_node(node* n);
60
61 bool reduce_minuses(node* root);
62
63 char* infix(node* root);
64
65 void free_node(node* n);
66
67 void delete_tree(tree* t);
68
69 #endif
```



Enter the expression:

$a*(-b)*4*(-5)$

=== the tree of the original expression ===

```
      a
     *
    -b
   *
  4
 *
-5
```

=== tree of the transformed expression ===

```
      a
     *
    b
   *
  4
 *
 5
```

=== infix entry ===

$5 * 4 * b * a$

Enter the expression:

$a*(-b)*3$

=== the tree of the original expression ===

```
      a
     *
    -b
   *
  3
```

=== tree of the transformed expression ===

```
      a
     *
    b
  -(*)
   3
```

=== infix entry ===

$-(3 * b * a)$

PS C:\Users\theo\_rvn\Desktop\coding\24> █

## 10. Замечания автора по существу работы

---

---

---

---

Недочеты при выполнении работы могут быть устранены следующим образом: \_\_\_\_\_

---

**11. Выводы:** Эта лабораторная работа была для меня отличной возможностью погрузиться в разработку программ, связанных с анализом математических выражений и построением синтаксических деревьев. В ходе выполнения работы, я изучил и применил алгоритм Shunting Yard, который позволяет эффективно преобразовывать инфиксные выражения в обратную польскую нотацию (RPN) и строить синтаксические деревья. В дальнейшей работе эта лабораторная работа может пригодиться, когда потребуется разрабатывать приложения, которые обрабатывают математические выражения. Например, это может быть полезно при создании калькуляторов, систем компьютерной алгебры, анализа данных или при разработке приложений для научных расчетов. Также, понимание алгоритмов анализа выражений и работа с синтаксическими деревьями полезны при работе с компиляторами или интерпретаторами языков программирования.

Кроме того, умение работать с алгоритмами разбора и обработки данных может стать незаменимым навыком в программировании в области искусственного интеллекта, обработки естественного языка и многих других областях, где требуется анализ текстовых данных.