Отчет по лабораторной работе № 24 по курсу Фундаментальная информатика

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

Контакты: 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) → 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 (parce.h), его реализация (parce.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)

main.c > ...

```c

#include <ctype.h>
#include <stdio.h>
#include <string.h>
#include "parse.h"

#define MAX_LEN 250

int main() {
    char str[MAX_LEN];
    printf("Введите выражение:\n");
    fgets(str, MAX_LEN, stdin);
    int len;
    for(len = 0; str[len] != '\0'; ++len);
    queue* q = parse(str, len);

    printf("\n=== Дерево исходного выражения ===\n");
    tree* t = create_tree(q);
    print_tree(t->root, 0);

    printf("\n=== Дерево преобразованного выражения ===\n");
    reduce_minuses(t->root);
    print_tree(t->root, 0);

    printf("\n=== Инфиксная запись ===\n");
    char* infix_str = infix(t->root);
    printf("%s\n", infix_str);

    free(infix_str);
    delete_tree(t);
    free(q);

    return 0;
}
```

parce.c > ...

```c
#include "parse.h"

// Функция для создания узла дерева
node* create_node(token type, char op, int val, char var, bool minus) {
    node* n = (node*)malloc(sizeof(node));
    n->type = type;
    n->oper = op;
    n->minus = minus;
    n->val = val;
    n->var = var;
    n->l_ch = NULL;
    n->r_ch = NULL;
    n->next = NULL;
    return n;
}

// =============== ОЧЕРЕДЬ =================
// Функция для создания очереди
queue* create_queue() {
    queue* q = (queue*)malloc(sizeof(queue));
    q->back = NULL;
    q->front = NULL;
    return q;
}

// Функция для добавления узла в очередь
void enqueue(queue* q, node* n) {
    n->next = NULL; // Инициализируем указатель на следующий узел нового узла
    if (q->back == NULL) {
        // Если очередь пуста, устанавливаем и back, и front на новый узел
        q->back = n;
        q->front = n;
    } else {
        // Если очередь не пуста, обновляем указатель на следующий узел текущего последнего элемента
        q->back->next = n;
        // Обновляем указатель back на новый узел
        q->back = n;
    }
}
```

```
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;
```

```
    queue* q = create_queue();
    char s[MAX_SIZE]; // СТЕК
    int s_top = -1;

    for (int i = 0; i < l; i++) {
        char c = str[i];
        bool unary_minus = false;

        if (isdigit(c)) { // ЕСЛИ ЧИСЛО
            int num = 0;
            // Продолжаем считывать символы, пока не встретится нецифровой символ
            while (i < l && isdigit(str[i])) {
                num = num * 10 + (str[i] - '0');
                i++;
            }
            i--; // Уменьшаем итератор, чтобы не пропустить символ после числа

            // Есть ли унарный минус?
            if (t_idx > 0 && tokens[t_idx - 1] == '-') {
                if (t_idx - 2 == -1 || tokens[t_idx - 2] == '(') {
                    s_top--; // Убираем унарный минус из стека
                    unary_minus = true;
                }
            }

            node* node = create_node(NUMBER, ' ', num, ' ', unary_minus);
            enqueue(q, node);

            tokens[t_idx++] = 'v'; // 'v' - значит значение (число или переменная)
        }
        else if (isalpha(c)) { // ЕСЛИ ПЕРЕМЕННАЯ
            // Есть ли унарный минус?
            if (t_idx > 0 && tokens[t_idx - 1] == '-') {
                if (t_idx - 2 == -1 || tokens[t_idx - 2] == '(' ) {
                    s_top--; // Убираем унарный минус из стека
                    unary_minus = true;
                }
            }

```

```
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     // Перекладываем 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");
```

```
    print_tree(root->l_ch, n + 1);
}

// функция для вывода дерева в форматированном виде
void print_node(node* n) {
    char minus;
    if (n->minus) {
        minus = '-';
    } else {
        minus = ' ';
    }
    switch (n->type) {
        case NUMBER:
            printf("%c%d ", minus, n->val);
            break;
        case VARIABLE:
            printf("%c%c ", minus, n->var);
            break;
        case OPERATOR:
            if (n->minus) {
                printf("-(%c) ", n->oper);
            } else {
                printf("%c ", n->oper);
            }
            break;
        // Обрабатываем другие типы, если необходимо
        default:
            // Обрабатываем недопустимый тип или предоставляем соответствующую обработку ошибок
            break;
    }
}

// Функция для сокращения минусов в дереве
bool reduce_minuses(node* root) {
    if (root->type == NUMBER || root->type == VARIABLE) {
        return root->minus;
    }
    if (root->oper == '*') {
        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 
```

```c
#ifndef PARSE
#define PARSE

#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

#define MAX_SIZE 250

typedef enum {
    OPERATOR,
    NUMBER,
    VARIABLE
} token;

typedef struct node {
    struct node* parent;
    struct node* l_ch;
    struct node* r_ch;
    token type;
    char oper;
    bool minus;
    int val;
    char var;
    struct node* next; // Для очереди
} node;

typedef struct tree {
    node* root;
} tree;

typedef struct {
    node* back;
    node* front;
} queue;

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
```

```
PS C:\Users\theo_rvn\Desktop\coding\24> ./a.exe
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
```

```
PS C:\Users\theo_rvn\Desktop\coding\24> ./a.exe
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.