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

Практикум программирование

Студент группы: М8О - 110Б - 23, Гула Дмитрий Александрович, № по списку: 6,

Контакты: guladmitriy1331@gmail.com

Работа выполнена: ««31» мая 2024г.

Преподаватель: каф. 806 Почечура А. А.

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

Отчет сдан «31» мая 2024 г., итоговая оценка \_\_\_\_\_

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

1. **Тема:** Реализация и анализ бинарного дерева выражений.
2. **Цель работы:** Разработать программу на языке C для построения бинарного дерева выражений и выполнения различных операций с ним, включая подсчет числа уровней дерева, количества операндов, переменных, целых констант и бинарных операций.
3. **Задание**

Программа должна выполнять следующие операции:

* Построение бинарного дерева выражений из токенов.
* Подсчет числа уровней дерева.
* Подсчет количества операндов (переменных и констант) в выражении.
* Подсчет количества переменных, используемых в данном выражении.
* Подсчет количества целых констант, используемых в выражении.
* Подсчет количества бинарных операций в выражении.

1. **Оборудование** (лабораторное):

ЭВМ , процессор . имя узла сети с ОП Мб, НМД Мб. Терминал

адрес . Принтер Другие устройства



*Оборудование ПЭВМ студента, если использовалось:*

Процессор \_Intel Core i3-12500H \_ с ОП 8 Гб НМД 256 Гб. Монитор 1920x1080~144Hz Другие устройства



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

Операционная система семейства , наименование версия



интерпретатор команд версия

Система программирования версия

Редактор текстов

версия

Утилиты операционной системы



Прикладные системы и программы



Местонахождение и имена файлов программ и

данных

*Программное обеспечение ЭВМ студента, если использовалось:*

Операционная система семейства \_Linux\_, наименование \_Ubuntu\_ версия 22.04.2 интерпретатор команд \_GNU bash\_ версия 5.1.16.

Система программирования С. Редактор текстов emacs версия 29.1

Утилиты операционной системы

Прикладные системы и программы Emacs

Местонахождение и имена файлов программ и данных на домашнем компьютере /home/

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

**Идея**

Идея данной работы заключается в реализации бинарного дерева выражений для хранения и обработки математических выражений. Бинарное дерево выражений позволяет эффективно представлять и вычислять арифметические выражения, а также выполнять различные операции над выражением, такие как подсчет уровней дерева, количества операндов, переменных, констант и операций. Использование дерева также упрощает реализацию функций для анализа выражений.

**Метод**

Метод включает в себя несколько этапов:

1. **Разбор выражения на токены:** Преобразование строки с выражением в последовательность токенов (чисел, переменных, операторов).
2. **Построение дерева выражений:** Использование стеков для создания бинарного дерева на основе токенов.
3. **Анализ дерева:** Реализация функций для подсчета уровней дерева, количества операндов, переменных, констант и операций.

**Алгоритм**

1. **Разбор выражения на токены:**
   * Определение типа токена (число, переменная, оператор) и создание структуры Token.
   * Чтение выражения посимвольно и создание соответствующих токенов.
2. **Построение дерева выражений:**
   * Использование двух стеков: один для операций, другой для узлов дерева.
   * Для каждого токена:
     + Если токен - операнд (число или переменная), создать узел дерева и поместить его в стек узлов.
     + Если токен - открывающая скобка, поместить его в стек операций.
     + Если токен - закрывающая скобка, извлекать операции из стека операций и создавать узлы дерева, пока не будет найдена открывающая скобка.
     + Если токен - оператор, извлекать операции из стека операций и создавать узлы дерева, пока приоритет верхней операции в стеке не станет меньше приоритета текущей операции, затем поместить текущий оператор в стек операций.
3. **Анализ дерева:**
   * Подсчет уровней дерева: Рекурсивный обход дерева для определения максимальной глубины.
   * Подсчет количества операндов: Рекурсивный обход дерева для подсчета всех листовых узлов (переменных и констант).
   * Подсчет количества переменных: Рекурсивный обход дерева для подсчета всех узлов, содержащих переменные.
   * Подсчет количества целых констант: Рекурсивный обход дерева для подсчета всех узлов, содержащих числа.
   * Подсчет количества бинарных операций: Рекурсивный обход дерева для подсчета всех узлов, содержащих бинарные операторы.
4. **Сценарий выполнения работы** [план работы, первоначальный текст программы в черновике (можно на отдельном листе) и тесты либо соображения по тестированию].
5. **Ввод выражения:** Пользователь вводит математическое выражение, которое необходимо обработать. Выражение может содержать целые числа, переменные, скобки и бинарные операторы (например, +, -, \*, /, ^).
6. **Разбор выражения на токены:** Программа считывает выражение посимвольно и преобразует его в последовательность токенов с помощью функции create\_token.
7. **Построение дерева выражений:** Программа создает бинарное дерево на основе последовательности токенов, используя функцию work\_with\_token и два стека: стек операций и стек узлов дерева.
8. **Подсчет уровней дерева:** Программа вычисляет количество уровней дерева с помощью функции **count\_levels**.
9. **Подсчет количества операндов:** Программа подсчитывает количество операндов (переменных и констант) с помощью функции **count\_operands**.
10. **Подсчет количества переменных:** Программа подсчитывает количество переменных в выражении с помощью функции **count\_variables**.
11. **Подсчет количества целых констант:** Программа подсчитывает количество целых констант в выражении с помощью функции **count\_constants**.
12. **Подсчет количества операций:** Программа подсчитывает количество бинарных операций в выражении с помощью функции **count\_binary\_operations**.
13. **Вывод результатов:** Программа выводит результаты анализа выражения: количество уровней дерева, количество операндов, переменных, констант и операций.
14. **Очистка памяти:** Программа очищает память, выделенную для дерева и стеков, чтобы предотвратить утечки памяти.
15. **Распечатка протокола** (подклеить листинг окончательного варианта программы с тестовыми примерами, подписанный преподавателем).

#include <stdio.h>  
#include <stdbool.h>  
#include <stdlib.h>  
#include <assert.h>  
#include <string.h>  
#include <errno.h>  
#include <ctype.h>  
  
#ifndef C\_TREE\_H  
#define C\_TREE\_H  
  
typedef enum {  
 *NONE*,  
 *INT*,  
 *COMBINED*,  
 *PLUS*,  
 *MINUS*,  
 *MULTIPLICATION*,  
 *DIVISION*,  
 *RAISE\_TO\_POWER*,  
 *UNARY\_MINUS*,  
 *OPENING\_BRACKET*,  
 *CLOSING\_BRACKET*,  
 *LAST\_SPACE*} Token\_type;  
  
typedef struct {  
 Token\_type type;  
 int value;  
 char name[50];  
}Token;  
  
  
typedef struct {  
 Token token;  
 struct TreeNode \*left;  
 struct TreeNode \*right;  
} TreeNode;  
  
char\* remove\_spaces(const char\* input\_string);  
Token\_type determine\_type\_of\_char(char symbol);  
int str\_to\_int(const char \*str);  
int char\_to\_int(char c);  
Token create\_token(const char \*\*expr\_ptr);  
TreeNode\* create\_tree\_node(Token token);  
bool is\_valid\_expression(const char \*expression);  
bool is\_int\_or\_comb(Token\_type tokenType);  
  
#endif //C\_TREE\_H

#include "tree.h"  
  
char\* remove\_spaces(const char\* input\_string) {  
 int input\_length = strlen(input\_string);  
 int count = 0;  
 for (int i = 0; i < input\_length; i++) {  
 if (input\_string[i] != ' ') {  
 count++;  
 }  
 }  
  
 char\* result = (char\*)malloc(count + 2); // Добавляем 2 для символа '\0' и дополнительного пробела в конце  
 if (result == NULL) {  
 perror("Ошибка выделения памяти");  
 exit(EXIT\_FAILURE);  
 }  
  
 int j = 0;  
 for (int i = 0; i < input\_length; i++) {  
 if (input\_string[i] != ' ') {  
 result[j++] = input\_string[i];  
 }  
 }  
 result[j++] = '\_';  
 result[j] = '\0';  
  
 return result;  
}  
  
Token\_type determine\_type\_of\_char(char symbol) {  
 if (symbol == '+') {  
 return *PLUS*;  
 } else if (symbol == '-') {  
 return *MINUS*;  
 } else if (symbol == '\*') {  
 return *MULTIPLICATION*;  
 } else if (symbol == '/') {  
 return *DIVISION*;  
 } else if (symbol == '^') {  
 return *RAISE\_TO\_POWER*;  
 } else if (symbol == '!') {  
 return *UNARY\_MINUS*;  
 } else if (symbol == '(') {  
 return *OPENING\_BRACKET*;  
 } else if (symbol == ')') {  
 return *CLOSING\_BRACKET*;  
 } else if (isdigit(symbol)) {  
 return *INT*;  
 } else if (isalpha(symbol)) {  
 return *COMBINED*;  
 }else if (symbol == '\_') {  
 return *LAST\_SPACE*;  
 }else {  
 return *NONE*;  
 }  
}  
  
int str\_to\_int(const char \*str) {  
 char \*endptr;  
 errno = 0;  
 long result = strtol(str, &endptr, 10);  
 if (endptr == str) {  
 fprintf(stderr, "Ошибка: в строке не найдены целочисленные данные\n");  
 exit(EXIT\_FAILURE);  
 }  
 return (int) result;  
}  
  
int char\_to\_int(char c) {  
 if (!isdigit(c)) {  
 fprintf(stderr, "Ошибка: символ не является цифрой\n");  
 exit(EXIT\_FAILURE);  
 }  
 return c - '0';  
}  
  
Token create\_token(const char \*\*expr\_ptr) {  
 const char \*expr = \*expr\_ptr;  
 Token token;  
 token.name[0] = '\0';  
 token.value = 0;  
  
 if (isdigit(\*expr)) {  
 char buffer[50];  
 int i = 0;  
 while (isdigit(\*expr)) {  
 buffer[i++] = \*expr;  
 expr++;  
 }  
 buffer[i] = '\0';  
 token.type = *INT*;  
 token.value = strtol(buffer, NULL, 10);  
 } else if (isalpha(\*expr)) {  
 int i = 0;  
 while (isalnum(\*expr)) {  
 token.name[i++] = \*expr;  
 expr++;  
 }  
 token.name[i] = '\0';  
 token.type = *COMBINED*;  
 } else if(\*expr == ' '){  
 token.type = determine\_type\_of\_char(\*expr);  
 }  
 else {  
 token.type = determine\_type\_of\_char(\*expr);  
 if (token.type != *NONE*) {  
 token.name[0] = \*expr;  
 token.name[1] = '\0';  
 expr++;  
 }  
 }  
  
 \*expr\_ptr = expr;  
 return token;  
}  
  
TreeNode\* create\_tree\_node(Token token) {  
 TreeNode \*node = (TreeNode \*)malloc(sizeof(TreeNode));  
 if (node == NULL) {  
 perror("Ошибка выделения памяти для TreeNode");  
 exit(EXIT\_FAILURE);  
 }  
 node->token = token;  
 node->left = NULL;  
 node->right = NULL;  
 return node;  
}  
  
  
bool is\_valid\_expression(const char \*expression) {  
 int open\_brackets = 0;  
 bool last\_was\_operator = false;  
 bool last\_was\_open\_bracket = false;  
 char prev\_char = '\0';  
  
 for (const char \*ptr = expression; \*ptr != '\0'; ++ptr) {  
 char current\_char = \*ptr;  
 Token\_type type = determine\_type\_of\_char(current\_char);  
  
 if (type == *NONE* && !isspace(current\_char)) {  
 fprintf(stderr, "Ошибка: недопустимый символ '%c'\n", current\_char);  
 return false;  
 }  
  
 if (type == *OPENING\_BRACKET*) {  
 open\_brackets++;  
 last\_was\_operator = false;  
 last\_was\_open\_bracket = true;  
 } else if (type == *CLOSING\_BRACKET*) {  
 if (open\_brackets == 0 || last\_was\_operator || last\_was\_open\_bracket) {  
 fprintf(stderr, "Ошибка: неправильное использование скобок\n");  
 return false;  
 }  
 open\_brackets--;  
 last\_was\_operator = false;  
 last\_was\_open\_bracket = false;  
 } else if (type == *PLUS* || type == *MINUS* || type == *MULTIPLICATION* || type == *DIVISION* ||  
 type == *RAISE\_TO\_POWER*) {  
 if (last\_was\_operator || last\_was\_open\_bracket) {  
 fprintf(stderr, "Ошибка: два оператора подряд или оператор после открывающей скобки\n");  
 return false;  
 }  
 last\_was\_operator = true;  
 } else if (type == *INT* || type == *COMBINED*) {  
 last\_was\_operator = false;  
 last\_was\_open\_bracket = false;  
 }  
  
 prev\_char = current\_char;  
 }  
  
 if (open\_brackets != 0) {  
 fprintf(stderr, "Ошибка: несоответствие количества открывающихся и закрывающихся скобок\n");  
 return false;  
 }  
  
 if (last\_was\_operator) {  
 fprintf(stderr, "Ошибка: выражение не может заканчиваться оператором\n");  
 return false;  
 }  
  
 return true;  
}  
  
bool is\_int\_or\_comb(Token\_type tokenType){  
 return tokenType == *INT* || tokenType == *COMBINED*;  
}

#include "tree.h"  
  
#ifndef C\_STACK\_H  
#define C\_STACK\_H  
  
  
typedef struct {  
 Token \*tokens;  
 size\_t size;  
 size\_t capacity;  
} Operation\_stack;  
  
typedef Operation\_stack\* Operation\_stack\_handle;  
  
Operation\_stack\_handle operationStack\_init(int cap);  
void destroy\_operationStack(Operation\_stack\_handle operationStack);  
bool operationStack\_is\_empty(Operation\_stack\_handle operationStack);  
bool need\_to\_grow(Operation\_stack\_handle operationStack);  
void grow\_operationStack\_buffer(Operation\_stack\_handle operationStack);  
void operationStack\_push(Operation\_stack\_handle operationStack, Token token);  
void shrink\_operationStack\_buffer(Operation\_stack\_handle operationStack);  
bool need\_to\_shrink(Operation\_stack\_handle operationStack);  
Token operationStack\_pop(Operation\_stack\_handle operationStack);  
Token get\_last\_token(Operation\_stack\_handle operationStack);  
void print\_operation\_stack(Operation\_stack\_handle stack);  
  
  
#endif //C\_STACK\_H

#include "stack.h"  
  
Operation\_stack\_handle operationStack\_init(int cap){  
 Operation\_stack\_handle operationStack = malloc(sizeof(Operation\_stack));  
 operationStack->capacity = cap;  
 operationStack->size = 0;  
 operationStack->tokens = malloc(sizeof(Token) \* cap);  
 return operationStack;  
}  
  
void destroy\_operationStack(Operation\_stack\_handle operationStack) {  
 if (operationStack) {  
 free(operationStack->tokens);  
 free(operationStack);  
 }  
}  
  
bool operationStack\_is\_empty(Operation\_stack\_handle operationStack){  
 return operationStack->size == 0;  
}  
  
bool need\_to\_grow(Operation\_stack\_handle operationStack){  
 return operationStack->capacity < operationStack->size \* 2;  
}  
  
void grow\_operationStack\_buffer(Operation\_stack\_handle operationStack){  
 operationStack->capacity \*= 2;  
 operationStack->tokens = realloc(operationStack->tokens, sizeof(Token) \* operationStack->capacity);  
 if(operationStack->tokens == NULL){  
 printf("Ошибка в записи памяти\n");  
 exit(EXIT\_FAILURE);  
 }  
}  
  
void operationStack\_push(Operation\_stack\_handle operationStack, Token token){  
 if(need\_to\_grow(operationStack)){  
 grow\_operationStack\_buffer(operationStack);  
 }  
 operationStack->tokens[operationStack->size] = token;  
 operationStack->size++;  
}  
  
void shrink\_operationStack\_buffer(Operation\_stack\_handle operationStack) {  
 size\_t new\_capacity = operationStack->capacity / 2;  
 if (new\_capacity < 1) {  
 new\_capacity = 1;  
 }  
  
 operationStack->tokens = realloc(operationStack->tokens, sizeof(Token) \* new\_capacity);  
 if (operationStack->tokens == NULL) {  
 printf("Ошибка в записи памяти при уменьшении буфера\n");  
 exit(EXIT\_FAILURE);  
 }  
 operationStack->capacity = new\_capacity;  
}  
  
bool need\_to\_shrink(Operation\_stack\_handle operationStack){  
 return operationStack->size < operationStack->capacity / 4;  
}  
  
Token operationStack\_pop(Operation\_stack\_handle operationStack) {  
 if (operationStack->size == 0) {  
 printf("Ошибка: попытка извлечения из пустого стека\n");  
 exit(EXIT\_FAILURE);  
 }  
 Token token = operationStack->tokens[operationStack->size - 1];  
 operationStack->size--;  
 if (need\_to\_shrink(operationStack) && !operationStack\_is\_empty(operationStack)) {  
 shrink\_operationStack\_buffer(operationStack);  
 }  
 return token;  
}  
  
Token get\_last\_token(Operation\_stack\_handle operationStack){  
 return operationStack->tokens[operationStack->size - 1];  
}  
  
void print\_operation\_stack(Operation\_stack\_handle stack) {  
 printf("Operation Stack (size: %zu, capacity: %zu):\n", stack->size, stack->capacity);  
 for (size\_t i = 0; i < stack->size; ++i) {  
 Token token = stack->tokens[i];  
 printf("Token %zu: ", i + 1);  
 switch (token.type) {  
 case *INT*:  
 printf("Type: INT, Value: %d\n", token.value);  
 break;  
 case *PLUS*:  
 printf("Type: PLUS, Symbol: +\n");  
 break;  
 case *MINUS*:  
 printf("Type: MINUS, Symbol: -\n");  
 break;  
 case *MULTIPLICATION*:  
 printf("Type: MULTIPLICATION, Symbol: \*\n");  
 break;  
 case *DIVISION*:  
 printf("Type: DIVISION, Symbol: /\n");  
 break;  
 case *RAISE\_TO\_POWER*:  
 printf("Type: RAISE\_TO\_POWER, Symbol: ^\n");  
 break;  
 case *UNARY\_MINUS*:  
 printf("Type: UNARY\_MINUS, Symbol: !\n");  
 break;  
 case *OPENING\_BRACKET*:  
 printf("Type: OPENING\_BRACKET, Symbol: (\n");  
 break;  
 case *CLOSING\_BRACKET*:  
 printf("Type: CLOSING\_BRACKET, Symbol: )\n");  
 break;  
 case *COMBINED*:  
 printf("Type: COMBINED, Name: %s\n", token.name);  
 break;  
 default:  
 printf("Type: NONE\n");  
 break;  
 }  
 }  
}

#include "tree.h"  
#ifndef C\_TREE\_STACK\_H  
#define C\_TREE\_STACK\_H  
  
  
  
typedef struct {  
 TreeNode \*\*node;  
 size\_t size;  
 size\_t capacity;  
}TreeStack;  
  
typedef TreeStack\* TreeStack\_handle;  
  
TreeStack\_handle treeStack\_init(int cap);  
void destroy\_treeStack(TreeStack\_handle treeStack);  
bool treeStack\_is\_empty(TreeStack\_handle treeStack);  
bool need\_to\_grow\_treeStack(TreeStack\_handle treeStack);  
void grow\_treeStack\_buffer(TreeStack\_handle treeStack);  
void treeStack\_push(TreeStack\_handle treeStack, TreeNode \*node);  
void shrink\_treeStack\_buffer(TreeStack\_handle treeStack);  
bool need\_to\_shrink\_treeStack(TreeStack\_handle treeStack);  
TreeNode \*treeStack\_pop(TreeStack\_handle treeStack);  
TreeNode \*get\_last\_treeNode(TreeStack\_handle treeStack);  
TreeNode \*get\_penultimate\_treeNode(TreeStack\_handle treeStack);  
void print\_tree\_stack(TreeStack\_handle treeStack);  
void print\_tree(TreeNode\* node, int level);  
  
  
#endif //C\_TREE\_STACK\_H

#include "tree\_stack.h"  
  
TreeStack\_handle treeStack\_init(int cap){  
 TreeStack\_handle treeStack = malloc(sizeof(TreeStack));  
 if (treeStack == NULL) {  
 fprintf(stderr, "Ошибка выделения памяти для стека\n");  
 exit(EXIT\_FAILURE);  
 }  
 treeStack->node = malloc(sizeof(TreeNode) \* cap);  
 if (treeStack->node == NULL) {  
 fprintf(stderr, "Ошибка выделения памяти для массива узлов\n");  
 free(treeStack);  
 exit(EXIT\_FAILURE);  
 }  
 treeStack->capacity = cap;  
 treeStack->size = 0;  
 return treeStack;  
}  
  
void destroy\_treeStack(TreeStack\_handle treeStack){  
 if(treeStack){  
 free(treeStack->node);  
 free(treeStack);  
 }  
}  
  
bool treeStack\_is\_empty(TreeStack\_handle treeStack){  
 return treeStack->size == 0;  
}  
  
bool need\_to\_grow\_treeStack(TreeStack\_handle treeStack) {  
 return treeStack->capacity < treeStack->size \* 2;  
}  
  
  
void grow\_treeStack\_buffer(TreeStack\_handle treeStack) {  
 treeStack->capacity \*= 2;  
 treeStack->node = realloc(treeStack->node, sizeof(TreeNode \*) \* treeStack->capacity);  
 if (treeStack->node == NULL) {  
 printf("Ошибка в записи памяти\n");  
 exit(EXIT\_FAILURE);  
 }  
}  
  
void treeStack\_push(TreeStack\_handle treeStack, TreeNode \*node) {  
 if (need\_to\_grow\_treeStack(treeStack)) {  
 grow\_treeStack\_buffer(treeStack);  
 }  
 treeStack->node[treeStack->size] = node;  
 treeStack->size++;  
}  
  
void shrink\_treeStack\_buffer(TreeStack\_handle treeStack) {  
 size\_t new\_capacity = treeStack->capacity / 2;  
 if (new\_capacity < 1) {  
 new\_capacity = 1;  
 }  
  
 treeStack->node = realloc(treeStack->node, sizeof(TreeNode \*) \* new\_capacity);  
 if (treeStack->node == NULL) {  
 printf("Ошибка в записи памяти при уменьшении буфера\n");  
 exit(EXIT\_FAILURE);  
 }  
 treeStack->capacity = new\_capacity;  
}  
  
bool need\_to\_shrink\_treeStack(TreeStack\_handle treeStack) {  
 return treeStack->size < treeStack->capacity / 4;  
}  
  
TreeNode \*treeStack\_pop(TreeStack\_handle treeStack) {  
 if (treeStack->size == 0) {  
 printf("Ошибка: попытка извлечения из пустого стека\n");  
 exit(EXIT\_FAILURE);  
 }  
 TreeNode \*node = treeStack->node[treeStack->size - 1];  
 treeStack->size--;  
 if (need\_to\_shrink\_treeStack(treeStack) && !treeStack\_is\_empty(treeStack)) {  
 shrink\_treeStack\_buffer(treeStack);  
 }  
 return node;  
}  
  
TreeNode \*get\_last\_treeNode(TreeStack\_handle treeStack) {  
 if (treeStack->size == 0) {  
 printf("Ошибка: стек пуст\n");  
 exit(EXIT\_FAILURE);  
 }  
 return treeStack->node[treeStack->size - 1];  
}  
  
TreeNode \*get\_penultimate\_treeNode(TreeStack\_handle treeStack) {  
 if (treeStack->size == 0) {  
 printf("Ошибка: стек пуст\n");  
 exit(EXIT\_FAILURE);  
 }  
 return treeStack->node[treeStack->size - 2];  
}  
  
  
  
  
void print\_tree\_stack(TreeStack\_handle treeStack) {  
 printf("Tree Stack (size: %zu, capacity: %zu):\n", treeStack->size, treeStack->capacity);  
 for (size\_t i = 0; i < treeStack->size; ++i) {  
 TreeNode \*tree = treeStack->node[i];  
 printf("Tree %zu:\n", i + 1);  
 print\_tree(tree, 0);  
 printf("\n");  
 }  
}  
  
void print\_tree(TreeNode \*node, int level) {  
 if (node == NULL) {  
 return;  
 }  
  
 print\_tree(node->right, level + 1);  
  
 for (int i = 0; i < level; i++) {  
 printf("\t");  
 }  
  
 if (node->token.type == *INT*) {  
 printf("%d\n", node->token.value);  
 } else if (node->token.type == *COMBINED*) {  
 printf("%s\n", node->token.name);  
 } else {  
 printf("%s\n", node->token.name);  
 }  
  
 print\_tree((TreeNode \*) node->left, level + 1);  
}

#include "stack.h"  
#include "tree\_stack.h"  
  
#ifndef C\_HOOD\_H  
#define C\_HOOD\_H  
  
void work\_with\_token(Token token, Operation\_stack\_handle operationStack, TreeStack\_handle treeStack);  
TreeNode \* create\_tree\_from\_stack(TreeNode \*poped\_node, TreeStack\_handle treeStack);  
Token create\_zero\_token();  
int count\_levels(TreeNode \*root);  
int count\_operands(TreeNode \*root);  
int count\_variables(TreeNode \*root);  
int count\_constants(TreeNode \*root);  
int count\_operations(TreeNode \*root);  
  
#endif //C\_HOOD\_H

#include "hood.h"  
  
  
void work\_with\_token(Token token, Operation\_stack\_handle operationStack, TreeStack\_handle treeStack) {  
 if (token.type == *INT* || token.type == *COMBINED*) {  
 TreeNode \*current\_node = create\_tree\_node(token);  
 treeStack\_push(treeStack, current\_node);  
 return;  
 }  
  
 if (token.type == *CLOSING\_BRACKET*) {  
 while (!operationStack\_is\_empty(operationStack)) {  
 Token last\_token = operationStack\_pop(operationStack);  
 if (last\_token.type == *OPENING\_BRACKET*) {  
 break;  
 }  
 TreeNode \*poped\_node = create\_tree\_node(last\_token);  
 create\_tree\_from\_stack(poped\_node, treeStack);  
 treeStack\_push(treeStack, poped\_node);  
  
 }  
 }else if(token.type == *LAST\_SPACE*){  
 while (!operationStack\_is\_empty(operationStack)){  
 Token last\_token = operationStack\_pop(operationStack);  
 TreeNode \*poped\_node = create\_tree\_node(last\_token);  
 create\_tree\_from\_stack(poped\_node, treeStack);  
 treeStack\_push(treeStack, poped\_node);  
  
 }  
 }  
 else {  
 while (!operationStack\_is\_empty(operationStack)) {  
 Token last\_token = get\_last\_token(operationStack);  
 if (last\_token.type == *OPENING\_BRACKET* || last\_token.type == *NONE*) {  
 break;  
 }  
 if ((token.type == *PLUS* || token.type == *MINUS*) &&  
 (last\_token.type == *PLUS* || last\_token.type == *MINUS* || last\_token.type == *MULTIPLICATION* ||  
 last\_token.type == *DIVISION* || last\_token.type == *RAISE\_TO\_POWER* || last\_token.type == *UNARY\_MINUS*)) {  
 TreeNode \*poped\_node = create\_tree\_node(last\_token);  
 operationStack\_pop(operationStack);  
 create\_tree\_from\_stack(poped\_node, treeStack);  
 treeStack\_push(treeStack, poped\_node);  
 } else if ((token.type == *MULTIPLICATION* || token.type == *DIVISION*) &&  
 (last\_token.type == *MULTIPLICATION* || last\_token.type == *DIVISION* ||  
 last\_token.type == *RAISE\_TO\_POWER* || last\_token.type == *UNARY\_MINUS*)) {  
 TreeNode \*poped\_node = create\_tree\_node(last\_token);  
 operationStack\_pop(operationStack);  
 create\_tree\_from\_stack(poped\_node, treeStack);  
 treeStack\_push(treeStack, poped\_node);  
 } else if (token.type == *RAISE\_TO\_POWER* && (last\_token.type == *RAISE\_TO\_POWER* || last\_token.type == *UNARY\_MINUS*)) {  
 TreeNode \*poped\_node = create\_tree\_node(last\_token);  
 operationStack\_pop(operationStack);  
 create\_tree\_from\_stack(poped\_node, treeStack);  
 treeStack\_push(treeStack, poped\_node);  
 }else if (token.type == *UNARY\_MINUS* && last\_token.type == *UNARY\_MINUS*) {  
 TreeNode \*poped\_node = create\_tree\_node(last\_token);  
 operationStack\_pop(operationStack);  
 create\_tree\_from\_stack(poped\_node, treeStack);  
 treeStack\_push(treeStack, poped\_node);  
 }  
 else {  
 break;  
 }  
 }  
 operationStack\_push(operationStack, token);  
 }  
}  
  
  
TreeNode \* create\_tree\_from\_stack(TreeNode \*poped\_node, TreeStack\_handle treeStack){  
 TreeNode \*last\_tree\_node = get\_last\_treeNode(treeStack);  
 TreeNode \*penultimate = get\_penultimate\_treeNode(treeStack);  
 Token\_type last\_tree\_node\_type = last\_tree\_node->token.type;  
 Token\_type penultimate\_type = penultimate->token.type;  
 if(poped\_node->token.type == *UNARY\_MINUS*){  
 Token zeroToken = create\_zero\_token();  
 TreeNode \*zeroTokenNode = create\_tree\_node(zeroToken);  
 poped\_node->left = zeroTokenNode;  
 poped\_node->right = last\_tree\_node;  
 treeStack\_pop(treeStack);  
 treeStack\_pop(treeStack);  
 }  
 else if (is\_int\_or\_comb(last\_tree\_node\_type) && is\_int\_or\_comb(penultimate\_type)){  
 poped\_node->left = penultimate;  
 poped\_node->right = last\_tree\_node;  
 treeStack\_pop(treeStack);  
 treeStack\_pop(treeStack);  
 } else if(!is\_int\_or\_comb(last\_tree\_node\_type) && is\_int\_or\_comb(penultimate\_type)){  
 if(last\_tree\_node->left != NULL && last\_tree\_node->right != NULL){  
 poped\_node->left = penultimate;  
 poped\_node->right = last\_tree\_node;  
 treeStack\_pop(treeStack);  
 treeStack\_pop(treeStack);  
 }  
 }  
 else if(is\_int\_or\_comb(last\_tree\_node\_type) && !is\_int\_or\_comb(penultimate\_type)){  
 if(penultimate->left != NULL && penultimate->right != NULL){  
 poped\_node->left = penultimate;  
 poped\_node->right = last\_tree\_node;  
 treeStack\_pop(treeStack);  
 treeStack\_pop(treeStack);  
 }  
 }  
 else if(!is\_int\_or\_comb(last\_tree\_node\_type) && !is\_int\_or\_comb(penultimate\_type)){  
 if(last\_tree\_node->left != NULL && last\_tree\_node->right != NULL && penultimate->left != NULL && penultimate->right != NULL){  
 poped\_node->left = penultimate;  
 poped\_node->right = last\_tree\_node;  
 treeStack\_pop(treeStack);  
 treeStack\_pop(treeStack);  
 }  
 }  
 return poped\_node;  
}  
  
  
Token create\_zero\_token() {  
 Token zero\_token;  
 zero\_token.type = *INT*;  
 zero\_token.value = 0;  
 zero\_token.name[0] = '\0';  
 return zero\_token;  
}  
  
int count\_levels(TreeNode \*root) {  
 if (root == NULL) {  
 return 0;  
 }  
 int left\_depth = count\_levels(root->left);  
 int right\_depth = count\_levels(root->right);  
 return (left\_depth > right\_depth ? left\_depth : right\_depth) + 1;  
}  
  
int count\_operands(TreeNode \*root) {  
 if (root == NULL) {  
 return 0;  
 }  
 if (root->token.type == *INT* || root->token.type == *COMBINED*) {  
 return 1;  
 }  
 return count\_operands(root->left) + count\_operands(root->right);  
}  
  
int count\_variables(TreeNode \*root) {  
 if (root == NULL) {  
 return 0;  
 }  
 if (root->token.type == *COMBINED*) {  
 return 1;  
 }  
 return count\_variables(root->left) + count\_variables(root->right);  
}  
  
int count\_constants(TreeNode \*root) {  
 if (root == NULL) {  
 return 0;  
 }  
 if (root->token.type == *INT*) {  
 return 1;  
 }  
 return count\_constants(root->left) + count\_constants(root->right);  
}  
  
int count\_operations(TreeNode \*root) {  
 if (root == NULL) {  
 return 0;  
 }  
 if (root->token.type == *PLUS* || root->token.type == *MINUS* || root->token.type == *MULTIPLICATION* ||  
 root->token.type == *DIVISION* || root->token.type == *RAISE\_TO\_POWER* || root->token.type == *UNARY\_MINUS*) {  
 return 1 + count\_operations(root->left) + count\_operations(root->right);  
 }  
 return count\_operations(root->left) + count\_operations(root->right);  
}

#include "hood.h"  
  
int main() {  
  
// char \*current\_expression = "(A + 10) - 155 \* B ^ 2";  
// char \*current\_expression = "(A + 10) - (155 \* B) ^ 2";  
// char \*current\_expression = "1 + 10 - 155 \* 2 ^ 2";  
 char \*current\_expression = "A \* 12 + 3 - (B / 4) \* 3 + 42 / 2 - 10 + (C + (D - 4))";  
 if (is\_valid\_expression(current\_expression)) {  
 printf("Выражение корректное.\n");  
 } else {  
 printf("Выражение некорректное.\n");  
 return 0;  
 }  
  
 char \*expression\_without\_spaces = remove\_spaces(current\_expression);  
 const char \*ptr = expression\_without\_spaces;  
 Operation\_stack\_handle operationStack = operationStack\_init(10);  
 TreeStack\_handle treeStack = treeStack\_init(10);  
  
 while (\*ptr != '\0') {  
 Token token = create\_token(&ptr);  
 work\_with\_token(token, operationStack, treeStack);  
 }  
  
 print\_operation\_stack(operationStack);  
 print\_tree\_stack(treeStack);  
  
 int levels = count\_levels(\*treeStack->node);  
 int operands = count\_operands(\*treeStack->node);  
 int variables = count\_variables(\*treeStack->node);  
 int constants = count\_constants(\*treeStack->node);  
 int operations = count\_operations(\*treeStack->node);  
  
 printf("Число уровней дерева выражения: %d\n", levels);  
 printf("Количество операндов: %d\n", operands);  
 printf("Количество переменных: %d\n", variables);  
 printf("Количество констант: %d\n", constants);  
 printf("Количество операций: %d\n", operations);  
  
 destroy\_operationStack(operationStack);  
 destroy\_treeStack(treeStack);  
 free(expression\_without\_spaces);  
 return 0;  
}

ПРИМЕРЫ РАБОТЫ ПРОГРАММЫ  
  
ПРИМЕР 1

char \*current\_expression = "(A + 10) - 155 \* B ^ 2";

Выражение корректное.

Operation Stack (size: 0, capacity: 5):

Tree Stack (size: 1, capacity: 5):

Tree 1:

2

^

B

\*

155

-

10

+

A

Число уровней дерева выражения: 4

Количество операндов: 5

Количество переменных: 2

Количество констант: 3

Количество операций: 4

Process finished with exit code 0

ПРИМЕР 2

char \*current\_expression = "1 + 10 - 155 \* 2 ^ 2";

C:\Users\Дима\CLionProjects\C\LB24\cmake-build-debug\LB24.exe

Выражение корректное.

Operation Stack (size: 0, capacity: 5):

Tree Stack (size: 1, capacity: 5):

Tree 1:

2

^

2

\*

155

-

10

+

1

Число уровней дерева выражения: 4

Количество операндов: 5

Количество переменных: 0

Количество констант: 5

Количество операций: 4

Process finished with exit code 0

ПРИМЕР 3

char \*current\_expression = "A \* 12 + 3 - (B / 4) \* 3 + 42 / 2 - 10 + (C + (D - 4))";

C:\Users\Дима\CLionProjects\C\LB24\cmake-build-debug\LB24.exe

Выражение корректное.

Operation Stack (size: 0, capacity: 5):

Tree Stack (size: 1, capacity: 5):

Tree 1:

4

-

D

+

C

+

10

-

2

/

42

+

3

\*

4

/

B

-

3

+

12

\*

A

Число уровней дерева выражения: 7

Количество операндов: 12

Количество переменных: 4

Количество констант: 8

Количество операций: 11

Process finished with exit code 0

ПРИМЕР 4

char \*current\_expression = "A \* 12) ++ 3 - (B / 4) \* 3 + 42 / 2 AA - 10 + (C + (D - 4))";

C:\Users\Дима\CLionProjects\C\LB24\cmake-build-debug\LB24.exe

Ошибка: неправильное использование скобок

Выражение некорректное.

Process finished with exit code 0

**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.  или дом. | Дата | Время | Событие | Действие по исправлению | Примечание |
|  |  |  |  |  |  |  |

1. **Замечания автора** по существу работы



1. **Выводы**

В ходе выполнения данной работы была разработана программа на языке Си для анализа математических выражений с использованием дерева выражений. Программа успешно разбирает входное выражение, строит соответствующее дерево, и производит анализ дерева, подсчитывая количество уровней, операндов, переменных, целых констант и бинарных операций.

Подпись студента