Отчет по лабораторной работе N 24 по курсу

"Фундаментальная информатика"

Студент группы: М80-107Б-21, Павлов Иван Дмитриевич
Контакты: pavlov.id.2003@gmail.com
Работа выполнена: 08.04.2022
Преподаватель: Найденов Иван Евгеньевич

1 Тема

Деревья выражений

2 Цель работы

Составить программу выполнения заданных арифметических преобразований с использованием деревьев.

3 Задание

№22: Выполнить сложение и вычитание дробей.

$\frac{a}{b} + \frac{c}{d} \rightarrow \frac{(a*d+b*c)}{b*d}$

4 Оборудование

Процессор: AMD Ryzen 5 4600H with Radeon Graphics
ОП: 7851 Мб
НМД: 256 Гб
Монитор: 1920x1080

5 Программное обеспечение

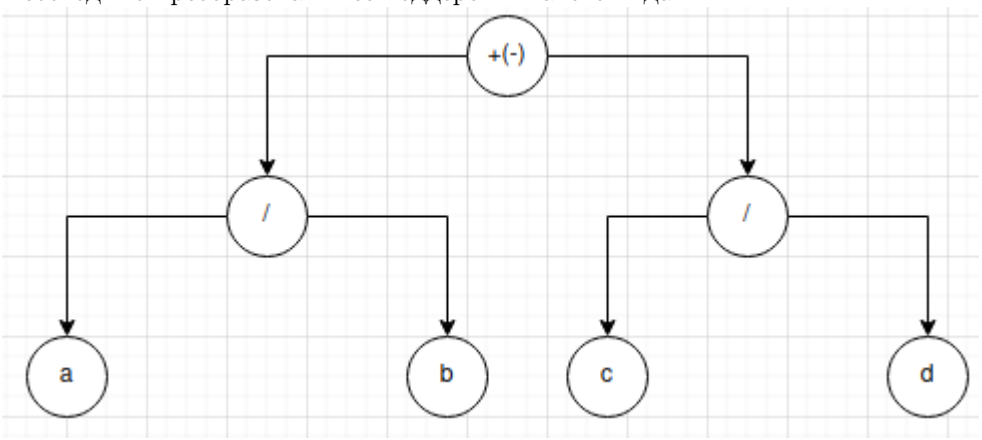
Операционная система семейства: linux (ubuntu), версия 20.04.3 LTS
Интерпретатор команд: bash, версия 5.0.17(1)-release.
Система программирования: gcc*, версия 17
Редактор текстов: emacs, версия 25.2.2
Утилиты операционной системы: subl, make
Прикладные системы и программы: sublime text, bash
Местонахождение и имена файлов программ и данных на домашнем компьютере: /home/ggame/newlabs/

6 Идея, метод, алгоритм решения задачи

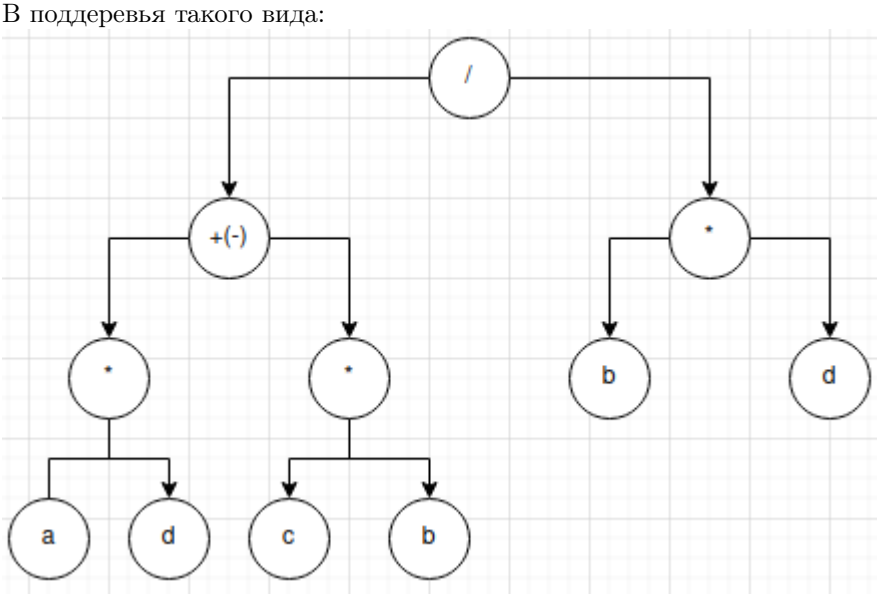
За основу я взял Простую реализаацию лексического и синтаксического разбора выражения и изменил main.c, tree.c и transform.c;

6.1 Функции преобразования дерева

Необходимо преобразовать все поддеревья такого вида:



В поддеревья такого вида:



Для этого необходимы:

- функция поиска поддерева;
- функция, создающая копии поддеревьев (на схеме узлы b и d);
- функция преобразования данного поддерева;
- функция, изменяющая дерево.

Функция поиска поддерева возвращает 0 или 1 в зависимости от того, является ли значение узла дерева плюсом (или минусом), а в левом и правом его поддереве находятся знаки умножения.
Функция *Tree* *Tree_copy рекурсивно обходит поддерево и создает его копию, присваивая всем элементам соответствующий тип и значения.
Функция *void* transform_div преобразует все поддеревья вида как на схеме 1 в поддеревья как на схеме 2. Для этого создаем копии поддеревьев b и d (так как в результате их должно быть 2) и реконструируем все поддерево, добавляя новые узлы умножения и используя старые поддеревья a, b, c, d и 2 копии.

6.2 Дополнительное задание

Перевести арифметическое выражение из инфиксной записи в постфиксную. Для этого изменим функцию *void* tree_infix из файла tree.c: уберем скобки и напечатаем все узлы дерева только после его обхода (снизу вверх). Таким образом получается постфиксная запись.

7 Сценарий выполнения работы

Тесты:

Листинг 1: test1

```
ggame@ggame:~/newlabs/lab24$ cat test.in
(a / b) - (c / d)
ggame@ggame:~/newlabs/lab24$ ./main < test.in
Tree's infix linearization:
(((a*d)-(c*b))/(b*d))
Tree's postfix linearization:
a d * c b * - b d * /
```

Листинг 2: test2

```
ggame@ggame:~/newlabs/lab24$ cat test.in
((17 * 82) / b) - (c / (4 + 4))
ggame@ggame:~/newlabs/lab24$ ./main < test.in
Tree's infix linearization:
((((17*82)*+)-(c*b))/(b*(4+4)))
Tree's postfix linearization:
17 82 * + * c b * - b 4 4 + * /
```

Листинг 3: test3

```
ggame@ggame:~/newlabs/lab24$ cat test.in
( ( e / f + k / g)  / b) - (c / d)
ggame@ggame:~/newlabs/lab24$ ./main < test.in
Tree's infix linearization:
((((((e*g)+(k*f))/(f*g))*d)-(c*b))/(b*d))
Tree's postfix linearization:
e g * k f * + f g * / d * c b * - b d * /
```

8 Распечатка протокола

Листинг 4: tree.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <math.h>

#include "tree.h"

int get_priority(char c)
{
  switch (c) {
    case '+': case '-': return 1;
    case '*': case '/': return 2;
    case '^': return 3;
  }
  return 100;
}

Tree tree_create(Token tokens[], int idx_left, int idx_right)
{
  Tree t = (Tree) malloc(sizeof(struct tree_node));
  if (idx_left > idx_right) {
    return NULL;
  }

  if (idx_left == idx_right) {
    t->node  = tokens[idx_left];
    t->left  = NULL;
    t->right = NULL;
    return t;
  }

```

```
32        int priority;
33        int priority_min = get_priority('a');
34        int brackets = 0;
35        int op_pos;
36        for (int i = idx_left; i <= idx_right; ++i) {
37          if ((tokens[i].type == BRACKET) && (tokens[i].data.is_left_bracket)) {
38            ++brackets;
39            continue;
40          }
41          if ((tokens[i].type == BRACKET) && !(tokens[i].data.is_left_bracket)) {
42            --brackets;
43            continue;
44          }
45          if (brackets > 0) {
46            continue;
47          }
48          if (tokens[i].type == OPERATOR) {
49            priority = get_priority(tokens[i].data.operator_name);
50            if (priority <= priority_min) {
51              priority_min = priority;
52              op_pos = i;
53            }
54          }
55        }
56        if ((priority_min == get_priority('a')) &&
57        (tokens[idx_left].type == BRACKET) &&
58        (tokens[idx_left].data.is_left_bracket) &&
59        (tokens[idx_right].type  == BRACKET) &&
60        !(tokens[idx_right].data.is_left_bracket)) {
61          free(t);
62          return tree_create(tokens, idx_left + 1, idx_right - 1);
63        }
64
65        if (tokens[op_pos].data.operator_name == '^') {
66          brackets = 0;
67          for (int i = op_pos; i >= idx_left; --i) {
68            if ((tokens[i].type == BRACKET) && !(tokens[i].data.is_left_bracket)) {
69              ++brackets;
70              continue;
71            }
72            if ((tokens[i].type == BRACKET) && (tokens[i].data.is_left_bracket)) {
73              --brackets;
74              continue;
75            }
76            if (brackets > 0) {
77              continue;
78            }
79            if (tokens[i].type == OPERATOR) {
80              priority = get_priority(tokens[i].data.operator_name);
81              if (priority == 3) {
82                op_pos = i;
83              }
84            }
85          }
86        }
87
88        t->node  = tokens[op_pos];
89        t->left  = tree_create(tokens, idx_left, op_pos - 1);
90        t->right = tree_create(tokens, op_pos + 1, idx_right);
91        if (t->right == NULL) {
92          printf("Epic fail: operator at the expression's end.");
93          exit(1);
94        }
95        return t;
96      }
97
```

```
98      void tree_delete(Tree *t)
99      {
100       if ((*t) != NULL) {
101         tree_delete(&((*t)->left));
102         tree_delete(&((*t)->right));
103       }
104       free(*t);
105       *t = NULL;
106     }
107
108     void tree_destroy(Tree t) {
109       if (t != NULL) {
110         if (t->left != NULL && t->right != NULL) {
111           if (t->left != NULL)
112           tree_destroy(t->left);
113
114           if (t->right != NULL) {
115             tree_destroy(t->right);
116           }
117         }
118         free(t);
119       }
120     }
121
122     void tree_infix(Tree t)
123     {
124       if (t != NULL) {
125         if (t->left != NULL && t->right != NULL) { //
126
127           if (t->left && t->right)
128           printf("(");
129
130           if (t->left != NULL)
131           tree_infix(t->left);
132         }
133
134         token_print(&(t->node));
135
136         if (t->left != NULL && t->right != NULL) {
137
138           if (t->right != NULL) //
139           tree_infix(t->right);
140
141           if (t->right && t->left)
142           printf(")");
143         }
144       }
145     }
146
147     void tree_postfix(Tree t)
148     {
149       if (t != NULL) {
150         if (t->left != NULL && t->right != NULL) {
151           if (t->left != NULL) //
152           tree_postfix(t->left);
153           if (t->right != NULL) //
154           tree_postfix(t->right);
155         }
156         token_print(&(t->node));
157         printf(" ");
158       }
159     }
160
161     void tree_print(Tree t, size_t depth)
162     {
163       if (t != NULL) {
```

```
164         for (int i = 0; i < depth; ++i) {
165           printf("\t");
166         }
167         token_print(&(t->node)); printf("\n");
168         tree_print(t->left,  depth + 1);
169         tree_print(t->right, depth + 1);
170       }
171     }
```

Листинг 5: tree.h

```
1     #ifndef _TREE_H_
2     #define _TREE_H_
3
4     #include <stdlib.h>
5     #include "lexer.h"
6
7     typedef struct tree_node *Tree;
8     struct tree_node {
9       Token node;
10      Tree left;
11      Tree right;
12    };
13
14    Tree tree_create(Token tokens[], int idx_left, int idx_right);
15    void tree_print(Tree t, size_t depth);
16    void tree_infix(Tree t);
17    void tree_delete(Tree *t);
18    void tree_postfix(Tree t);
19    //void tree_destroy(Tree t);
20
21    #endif
```

Листинг 6: transform.c

```
1     #include "tree.h"
2     #include "transform.h"
3     #include "lexer.h"
4     #include <math.h>
5
6     Tree *tree_copy(Tree t)
7     {
8       if (t == NULL) {
9         return NULL;
10      }
11      Tree left = NULL;
12      Tree right = NULL;
13      if (t->left != NULL) {
14        Tree left = tree_copy(t->left);
15      }
16      if (t->right != NULL) {
17        Tree right = tree_copy(t->right);
18      }
19      Tree new_t = (Tree) malloc(sizeof(struct tree_node));
20      new_t->node.type = t->node.type;
21      if (new_t->node.type == INTEGER) {
22        new_t->node.data.value_int = t->node.data.value_int;
23      } else
24      if (new_t->node.type == FLOATING) {
25        new_t->node.data.value_float = t->node.data.value_float;
26      } else
27      if (new_t->node.type == OPERATOR) {
28        new_t->node.data.operator_name = t->node.data.operator_name;
29      } else
30      if (new_t->node.type == VARIABLE) {
31        new_t->node.data.variable_name = t->node.data.variable_name;
```

```
32        }
33
34        if (left != NULL) {
35          new_t->left = left;
36        }
37        if (right != NULL) {
38          new_t->right = right;
39        }
40        return new_t;
41      }
42
43      int match_plus(Tree *t)
44      {
45        return ((*t) != NULL) && ((*t)->node.type == OPERATOR)
46        && (((*t)->node.data.operator_name == '+') || ((*t)->node.data.operator_name == '
              -'))
47        && (((*t)->left->node.type == OPERATOR) && ((*t)->left->node.data.operator_name
              == '/'))
48        && (((*t)->right->node.type == OPERATOR) && ((*t)->right->node.data.operator_name
               == '/'));
49      }
50
51      void transform_div(Tree *t)
52      {
53        Tree b = tree_copy((*t)->left->right);
54        Tree d = tree_copy((*t)->right->right);
55
56        char oper = (*t)->node.data.operator_name;
57        (*t)->node.data.operator_name = '/';
58        (*t)->right->node.data.operator_name = '*';
59
60
61        (*t)->left->node.data.operator_name = oper;
62        Tree left_mul = (Tree) malloc(sizeof(struct tree_node));
63        left_mul->node.type = OPERATOR;
64        left_mul->node.data.operator_name = '*';
65        left_mul->left = (*t)->left->left;
66        left_mul->right = d;
67
68        Tree right_mul = (Tree) malloc(sizeof(struct tree_node));
69        right_mul->node.type = OPERATOR;
70        right_mul->node.data.operator_name = '*';
71        right_mul->left = (*t)->right->left;
72        right_mul->right = b;
73
74        (*t)->right->left = (*t)->left->right;
75        (*t)->left->left = left_mul;
76        (*t)->left->right = right_mul;
77
78        return;
79      }
80
81      void tree_transform(Tree *t)
82      {
83        if ((*t) != NULL) {
84          tree_transform(&((*t)->left));
85          tree_transform(&((*t)->right));
86          if (match_plus(t)) {
87            transform_div(t);
88            //              printf("YES\n");
89          }
90        }
91      }
```

Листинг 7: transform.h

```
#ifndef __TRANSFORM_H__
#define __TRANSFORM_H__

#include "tree.h"

void tree_transform(Tree *t);

#endif
```

Листинг 8: 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;
  char c;

  do {
    c = fgetc(stdin);
  } while (isspace(c));

  if (c == EOF) { // The end
    t->type = FINAL;
  }

  else if (isalpha(c) || c == '_') {
    t->type = VARIABLE;
    t->data.variable_name = c;
    can_be_unary = false;
  }
  // stdin:   [123]
  //           ^
  else if (isdigit(c)) {
    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 == '+')) {
    int sign = (c == '+') ? +1 : -1;

    do {
      c = fgetc(stdin);
    } while (isspace(c));

```

```
56          if (isdigit(c)) {
57            ungetc(c, stdin);
58            token_next(t);
59            if (t->type == INTEGER) {
60              t->data.value_int = sign * (t->data.value_int);
61            } else {
62              t->data.value_float = sign * (t->data.value_float);
63            }
64          } else {
65            ungetc(c, stdin);
66            t->type = OPERATOR;
67            t->data.operator_name = '-';
68            can_be_unary = true;
69          }
70        }
71
72        else {
73          t->type = OPERATOR;
74          t->data.operator_name = c;
75          can_be_unary = true;
76        }
77      }
78
79      void token_print(Token *t)
80      {
81        switch (t->type) {
82          case FINAL:
83          break;
84          case INTEGER:
85          printf("%d", t->data.value_int);
86          break;
87          case FLOATING:
88          printf("%lg", t->data.value_float);
89          break;
90          case VARIABLE:
91          printf("%c", t->data.variable_name);
92          break;
93          case OPERATOR:
94          printf("%c", t->data.operator_name);
95          break;
96          case BRACKET:
97          printf("%c", (t->data.is_left_bracket) ? '(' : ')');
98          break;
99        }
100     }
```

Листинг 9: lexer.h

```
1     #ifndef __LEXER_H__
2     #define __LEXER_H__
3
4     #include <stdbool.h>
5
6     typedef enum {
7       FINAL,
8       INTEGER,
9       FLOATING,
10      OPERATOR,
11      VARIABLE,
12      BRACKET
13    } TokenType;
14
15    typedef struct {
16      TokenType type;
17      union {
18        int   value_int;
```

```
19          float value_float;
20          char  operator_name;
21          bool  is_left_bracket;
22          char  variable_name;
23        } data;
24      } Token;
25
26      void token_print(Token *t);
27      void token_next(Token *t);
28
29      #endif
```

Листинг 10: main.c

```
1     #include <stdio.h>
2     #include <stdlib.h>
3
4     #include "lexer.h"
5     #include "tree.h"
6     #include "transform.h"
7
8     int main(void)
9     {
10      Token tokens[256];
11      size_t tokens_qty = 0;
12
13      Token token;
14      token_next(&token);
15
16      while (token.type != FINAL) {
17        tokens[tokens_qty++] = token;
18        token_next(&token);
19      }
20
21      Tree tree = tree_create(tokens, 0, tokens_qty - 1);
22
23      //    printf("\nExpression tree:\n");
24      //    tree_print(tree, 0);
25
26      tree_transform(&tree);
27
28      //    printf("\nSemitransformed expression tree:\n");
29      //    tree_print(tree, 0);
30
31      printf("Tree's infix linearization:\n");
32      tree_infix(tree);
33      printf("\nTree's postfix linearization:\n");
34      tree_postfix(tree);
35      printf("\n");
36      tree_destroy(&tree);
37      return 0;
38    }
```

9 Вывод

В ходе выполнения данной лабораторной работы я познакомился с лексическим и синтаксическим анализом, научился преобразовывать деревья выражений и узнал о постфиксной записи.