1. Министерство образования и науки Российской Федерации
2. Санкт-Петербургский Политехнический Университет Петра Великого
3. —
4. Институт кибербезопасности и защиты информации

**ЛАБОРАТОРНАЯ РАБОТА № 4**

1. «**Анализ идентификаторов в исходном коде**»
2. по дисциплине «Структуры данных»
3. Выполнил
4. студент гр.5151001/40001 Волошкевич М.А.

<*подпись*>

1. Преподаватель Семьянов П.В.

<*подпись*>

1. Санкт-Петербург
2. 2025 г.
3. **Цель работы**

Реализовать программу для анализа идентификаторов в исходном коде на языке C. Программа должна находить и подсчитывать количество вхождений переменных и функций, исключая ключевые слова, с использованием префиксного дерева и хеш-таблицы.

**2. Постановка задачи**

Написать программу, которая:

1. Удаляет комментарии из исходного кода.

2. Находит все идентификаторы (имена переменных и функций) в коде, исключая ключевые слова.

3. Подсчитывает количество вхождений каждого идентификатора.

4. Реализует поиск по префиксному дереву и хеш-таблице.

5. Обеспечивает интерактивное взаимодействие с пользователем.

**3. Теоретические исследования**

Для выполнения работы были изучены:

* + - * + Префиксное дерево (Trie):

Структура данных для эффективного хранения строк.

Позволяет быстро находить все строки с заданным префиксом.

* + - * + Хеш-таблица:

Используется для подсчета количества вхождений идентификаторов.

Реализована с ресайзингом для оптимизации производительности.

* + - * + Удаление комментариев:

Обработка однострочных (//) и многострочных (/\* ... \*/) комментариев.

Учет строковых и символьных литералов для корректного удаления.

**4. Описание решения**

Программа состоит из следующих модулей:

4.1. Удаление комментариев (remove\_comment.h)

Используется конечный автомат для обработки различных состояний:

STATE\_DEFAULT — обычный режим, ожидание комментариев или строк.

STATE\_IN\_STRING — внутри строки ("..."), игнорирует комментарии.

STATE\_IN\_CHAR — внутри символьного литерала ('...').

STATE\_IN\_SINGLE\_COMMENT — внутри однострочного комментария (//).

STATE\_IN\_MULTI\_COMMENT — внутри многострочного комментария (/\* ... \*/).

Комментарии удаляются, а остальной код сохраняется в новом файле.

4.2. Поиск идентификаторов (main.c)

Алгоритм поиска работает следующим образом:

* + - * + Пропускаются ключевые слова (например, if, for, return).
        + Игнорируются строки и символьные литералы (чтобы не считать их за идентификаторы).
        + Определяются объявления переменных и функций по синтаксису C:
        + Если после типа данных следует допустимый идентификатор (начинается с буквы или \_), он считается переменной или функцией.
        + Проверяется контекст (например, объявление внутри параметров функции или глобально).
        + Найденные идентификаторы добавляются в префиксное дерево (для быстрого поиска по префиксу) и хеш-таблицу (для подсчета вхождений).

4.3. Префиксное дерево (prefix\_tree.h)

Назначение:

Быстрый поиск идентификаторов по префиксу (например, поиск всех переменных, начинающихся на var\_).

Структура:

Каждый узел содержит массив указателей на дочерние узлы (по символам a-z, A-Z, 0-9, \_).

Если узел помечен как is\_value = true, он соответствует концу идентификатора.

Функции:

* + - * + trie\_insert() — добавляет строку в дерево.
        + trie\_search() — ищет все строки с заданным префиксом.
        + print\_tree() — выводит дерево в удобном виде (для отладки).

4.4. Хеш-таблица (hash.h)

Назначение: Хранение количества вхождений каждого идентификатора.

Принцип работы:

1. Хеширование:

Используется хеш-функция DJB2 (hash \* 33 + c), которая преобразует строку в индекс таблицы.

2. Разрешение коллизий:

Метод цепочек: если два ключа попадают в одну ячейку, они хранятся в виде связного списка.

3. Динамическое расширение:

Если коэффициент заполнения (LOAD\_FACTOR = 0.7) превышен, таблица увеличивается вдвое (hash\_resize()).

4. Операции:

hash\_insert() — добавляет или обновляет счетчик идентификатора.

hash\_search() — возвращает количество вхождений строки.

hash\_remove() — удаляет элемент из таблицы.

hash\_free() — освобождает память.

**5. Тестирование и результаты работы программы**

Программа была протестирована на различных исходных файлах:

* + - * + Корректно удаляются все виды комментариев.
        + Идентификаторы успешно обнаруживаются и подсчитываются.
        + Префиксное дерево и хеш-таблица возвращают корректные результаты.
        + Ресайзинг хеш-таблицы работает без ошибок.

**6. Вывод**

В ходе работы была успешно реализована программа для анализа идентификаторов в исходном коде. Использование префиксного дерева и хеш-таблицы обеспечило эффективность и удобство поиска, в связи с тем что количество структур данных не конечно и поэтому для быстрого извлечения количества идентификатороф, а префиксное дерево обеспечивает быстрый поиск за счет своей структуры. Программа готова к дальнейшему расширению, например, для поддержки большего количества языков или интеграции с другими инструментами анализа кода.

**7. Приложения**

**Приложение1:**

#include <stdio.h>

#include <stdbool.h>

#include <ctype.h>

#include "prefix\_tree.h"

#include "hash.h"

#include "remove\_comment.h"

#define BUFFER\_SIZE 1024

const char \*ignore\_keywords[] = {

// Ключевые слова

"typedef", "return", "else", "if", "for", "while", "switch",

"case", "default", "sizeof", "struct", "enum", "union",

// Модификаторы хранения

"static", "extern", "auto", "register", "inline", "restrict",

// Квалификаторы типа

"const", "volatile",

// Спецификаторы типа

"signed", "unsigned", "short", "long"

};

const int num\_ignore\_keywords = sizeof(ignore\_keywords) / sizeof(ignore\_keywords[0]);

void find\_all(const char \*filename, char \*search\_str, prefix\_tree \*pt, hash\_table \*ht) {

FILE \*file = fopen(filename, "r");

size\_t search\_len = strlen(search\_str);

int c;

size\_t pos = 0;

int prev\_char = 0;

int next\_char;

while ((c = fgetc(file)) != EOF) {

if (pos == 0) {

next\_char = fgetc(file);

ungetc(next\_char, file);

}

if (c == search\_str[pos]) {

pos++;

if (pos == search\_len) {

if (search\_len == 1) {

if ((prev\_char == 0 || !isalpha(prev\_char)) &&

(next\_char == EOF || !isalpha(next\_char))) {

hash\_insert(ht, search\_str);

trie\_insert(pt, search\_str);

}

} else {

hash\_insert(ht, search\_str);

trie\_insert(pt, search\_str);

}

pos = 0;

}

} else {

pos = (c == search\_str[0]) ? 1 : 0;

}

prev\_char = c;

}

fclose(file);

}

bool is\_valid\_id\_char(char c) {

return isalnum((unsigned char)c) || c == '\_';

}

bool is\_ignore\_keyword(const char \*str, size\_t len) {

for (int i = 0; i < num\_ignore\_keywords; i++) {

size\_t kw\_len = strlen(ignore\_keywords[i]);

if (len == kw\_len && strncmp(str, ignore\_keywords[i], len) == 0) {

return true;

}

}

return false;

}

void find\_ids(FILE \*file, prefix\_tree \*pt, hash\_table \*ht) {

char line[1024];

int line\_num = 0;

bool in\_function\_params = false;

while (fgets(line, sizeof(line), file)) {

line\_num++;

char \*ptr = line;

if (\*ptr == '#') continue;

while (\*ptr) {

while (\*ptr && isspace(\*ptr)) ptr++;

if (!\*ptr) break;

if (\*ptr == '"') {

ptr++;

while (\*ptr && \*ptr != '"') ptr++;

if (\*ptr) ptr++;

continue;

}

if (\*ptr == '\'') {

ptr++;

while (\*ptr && \*ptr != '\'') ptr++;

if (\*ptr) ptr++;

continue;

}

if (\*ptr == '(' && !in\_function\_params) {

in\_function\_params = true;

ptr++;

continue;

}

if (\*ptr == ')' && in\_function\_params) {

in\_function\_params = false;

ptr++;

continue;

}

char \*word\_start = ptr;

while (\*ptr && is\_valid\_id\_char(\*ptr)) ptr++;

size\_t word\_len = ptr - word\_start;

if (word\_len > 0 && is\_ignore\_keyword(word\_start, word\_len)) {

continue;

}

if (word\_len > 0) {

ptr = word\_start;

}

char \*type\_start = ptr;

while (\*ptr && is\_valid\_id\_char(\*ptr)) ptr++;

if (type\_start == ptr) {

ptr++;

continue;

}

while (\*ptr && isspace(\*ptr)) ptr++;

if (!\*ptr) break;

while (\*ptr == '\*') ptr++;

while (\*ptr && isspace(\*ptr)) ptr++;

if (!\*ptr) break;

char \*name\_start = ptr;

while (\*ptr && is\_valid\_id\_char(\*ptr)) ptr++;

if (name\_start == ptr) continue;

bool valid\_decl = false;

if (in\_function\_params) {

if (\*ptr == ',' || \*ptr == ')' || \*ptr == '=' || isspace(\*ptr)) {

valid\_decl = true;

}

} else {

if (\*ptr == ';' || \*ptr == '=' || \*ptr == '(' || \*ptr == '[' || \*ptr == ',' || isspace(\*ptr)) {

valid\_decl = true;

}

}

if (valid\_decl) {

char name[256];

size\_t name\_len = ptr - name\_start;

if (name\_len >= sizeof(name)) name\_len = sizeof(name) - 1;

strncpy(name, name\_start, name\_len);

name[name\_len] = '\0';

if (isdigit(name[0])) continue;

if(hash\_search(ht, name) == 0)

{

printf("Found ID: %s (line %d)\n", name, line\_num);

find\_all("test.c", name, pt, ht);

}

}

}

}

}

int main()

{

FILE \*f\_in = fopen("test.c", "r");

FILE \*f\_out = fopen("test1.c", "w");

remove\_comments(f\_in, f\_out);

fclose(f\_in);

fclose(f\_out);

FILE \*f = fopen("test1.c", "r");

hash\_table ht;

hash\_init(&ht);

prefix\_tree pt;

init\_tree(&pt);

find\_ids(f, &pt, &ht);

fclose(f);

while(true)

{

int command;

printf("Выберите цифру для действия:\n"

"1 - поиск по дереву\n"

"2 - общее количество слов\n"

"3 - вывести дерево\n"

"4 - поиск по хэш таблице\n"

"5 - выход\n");

scanf("%d", &command);

switch (command)

{

case 1:

printf("Введите название переменной/функции: ");

char str[100];

if (scanf("%s", str) == 1)

{

trie\_search(&pt, str);

}

break;

case 2:

printf("%d\n", pt.word\_counts);

break;

case 3:

print\_tree(&pt);

break;

case 4:

printf("Введите название переменной/функции: ");

char str\_hash[100];

if (scanf("%s", str\_hash) == 1)

{

printf("%d\n", hash\_search(&ht, str\_hash));

}

break;

case 5:

hash\_free(&ht);

return 0;

default:

printf("Неизвестная команда\n");

}

}

}

**Приложение2:**

#include <stdio.h>

#define STATE\_DEFAULT 0

#define STATE\_IN\_STRING 1

#define STATE\_IN\_CHAR 2

#define STATE\_IN\_SINGLE\_COMMENT 3

#define STATE\_IN\_MULTI\_COMMENT 4

void remove\_comments(FILE\* input, FILE\* output) {

int c, next;

int state = STATE\_DEFAULT;

int count\_sleshik = 0;

while ((c = fgetc(input)) != EOF) {

if (c == '\\')count\_sleshik++;

switch (state) {

case STATE\_DEFAULT:

if (c == '/') {

next = fgetc(input);

if (next == EOF) {

fputc(c, output);

return;

}

if (next == '/') {

state = STATE\_IN\_SINGLE\_COMMENT;

}

else if (next == '\*') {

state = STATE\_IN\_MULTI\_COMMENT;

}

else {

fputc(c, output);

fseek(input, -1, SEEK\_CUR);

}

}

else if (c == '"') {

state = STATE\_IN\_STRING;

fputc(c, output);

}

else if (c == '\'') {

state = STATE\_IN\_CHAR;

fputc(c, output);

}

else {

fputc(c, output);

}

break;

case STATE\_IN\_STRING:

fputc(c, output);

if (count\_sleshik%2==0) {

if (c == '"') {

state = STATE\_DEFAULT;

}

else if (c == '\n') {

state = STATE\_DEFAULT;

}

}

break;

case STATE\_IN\_CHAR:

fputc(c, output);

if (c == '\n') state = STATE\_DEFAULT;

else if (c == '\'' && (count\_sleshik % 2 == 0)) state = STATE\_DEFAULT;

break;

case STATE\_IN\_SINGLE\_COMMENT:

if (c == '\n') {

if (count\_sleshik % 2 == 0) {

fputc('\n', output);

state = STATE\_DEFAULT;

}

}

else if (c == EOF) {

return;

}

break;

case STATE\_IN\_MULTI\_COMMENT:

if (c == '\*') {

next = fgetc(input);

if (next == EOF)return;

else if (next == '/') {

state = STATE\_DEFAULT;

}

else {

fseek(input, -1, SEEK\_CUR);

}

}

break;

default:

state = STATE\_DEFAULT;

break;

}

if (c != '\\' || c == '\n')count\_sleshik = 0;

}

}

**Приложение3:**

#ifndef \_PREFIX\_TREE\_H

#define \_PREFIX\_TREE\_H

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

typedef struct node

{

struct node\* next[63];

char value;

bool is\_value;

int count\_of\_values;

}node;

typedef struct

{

node\* start;

int word\_counts;

}prefix\_tree;

int char\_to\_index(char ch) {

if (ch >= 'a' && ch <= 'z') {

return ch - 'a'; // 0-25

} else if (ch >= 'A' && ch <= 'Z') {

return 26 + (ch - 'A'); // 26-51

} else if (ch >= '0' && ch <= '9') {

return 52 + (ch - '0'); // 52-61

} else if (ch == '\_') {

return 62; // 62

} else {

return -1; // Недопустимый символ

}

}

int has\_child(node\* nd, char ch)

{

int pos = char\_to\_index(ch);

if(!nd->next[pos]) return false;

return true;

}

void add\_child(node\* nd, char ch)

{

int pos = char\_to\_index(ch);

if(nd->next[pos]) return;

node \*new\_node = (node\*)malloc(sizeof(node));

new\_node->count\_of\_values = 0;

new\_node->value = ch;

new\_node->is\_value = false;

// Обнуляем все указатели next

for (int i = 0; i < 63; i++) {

new\_node->next[i] = NULL;

}

nd->next[pos] = new\_node;

}

void print\_descendants(node \*nd, char\* start, char\* str)

{

if(nd->is\_value) printf("%s%s %d\n", start, str, nd->count\_of\_values);

for(int i = 0; i < 63; i++)

{

if(nd->next[i])

{

strncat(str, &nd->next[i]->value, 1);

print\_descendants(nd->next[i], start, str);

str[strlen(str) - 1] = '\0';

}

}

}

node\* find\_node(prefix\_tree \*pt, char\* str, bool is\_insert)

{

node\* nd = pt->start;

for (int i = 0; i < strlen(str); i++)

{

if(!has\_child(nd, str[i]))

{

if(is\_insert)

{

add\_child(nd, str[i]);

}

else return nullptr;

}

int pos = char\_to\_index(str[i]);

nd = nd->next[pos];

}

return nd;

}

void init\_tree(prefix\_tree \*pt)

{

node \*root\_node = (node\*)malloc(sizeof(node));

root\_node->value = '\0';

root\_node->is\_value = false;

root\_node->count\_of\_values = 0;

// Обнуляем все указатели next

for (int i = 0; i < 63; i++) {

root\_node->next[i] = NULL;

}

pt->start = root\_node;

pt->word\_counts = 0;

}

void print\_node(node\* nd, int level) {

if (!nd) return;

// Отступ в зависимости от уровня

for (int i = 0; i < level; i++) {

printf(" ");

}

// Выводим текущий узел

printf("'%c'", nd->value);

if (nd->is\_value) {

printf(" (end, count=%d)", nd->count\_of\_values);

}

printf("\n");

// Рекурсивно выводим детей

for (int i = 0; i < 63; i++) {

if (nd->next[i]) {

print\_node(nd->next[i], level + 1);

}

}

}

void print\_tree(prefix\_tree \*pt) {

if (!pt || !pt->start) {

printf("(empty tree)\n");

return;

}

printf("Prefix Tree (root):\n");

for (int i = 0; i < 63; i++) {

if (pt->start->next[i]) {

print\_node(pt->start->next[i], 1);

}

}

}

void trie\_insert(prefix\_tree \*pt, char\* str)

{

node\* nd = find\_node(pt, str, true);

pt->word\_counts++;

nd->count\_of\_values++;

nd->is\_value = true;

}

int trie\_lookup(prefix\_tree \*pt, char\* str)

{

node\* nd = find\_node(pt, str, false);

if(!nd) return 0;

if(nd->is\_value) return nd->count\_of\_values;

else return 0;

}

void trie\_search(prefix\_tree \*pt, char\* prefix)

{

node\* nd = find\_node(pt, prefix, false);

if(!nd)

{

printf("No value\n");

return;

}

char test[100] = "\0";

print\_descendants(nd, prefix, test);

}

#endif

**Приложение5:**

#ifndef \_HASH\_H

#define \_HASH\_H

#include <string.h>

#include <stdlib.h>

#define INITIAL\_SIZE 2 // Начальный размер таблицы

#define LOAD\_FACTOR 0.7 // Максимальный коэффициент заполнения перед ресайзинг

typedef struct hash\_node

{

struct hash\_node\* next;

char\* key;

int value;

}hash\_node;

typedef struct

{

hash\_node\*\* table;

int size;

int count;

}hash\_table;

int hash(char\* value, int size)

{

unsigned long hash = 5381;

int c;

while (c = \*value++)

hash = ((hash << 5) + hash) + c; /\* hash \* 33 + c \*/

return hash % size;

}

void hash\_init(hash\_table\* ht)

{

ht->size = INITIAL\_SIZE;

ht->table = (hash\_node\*\*)calloc(ht->size, sizeof(hash\_node\*));

}

void hash\_resize(hash\_table\* ht, int new\_size) {

hash\_node\*\* new\_table = (hash\_node\*\*)calloc(new\_size, sizeof(hash\_node\*));

for (int i = 0; i < ht->size; i++) {

hash\_node\* current = ht->table[i];

while (current != NULL) {

hash\_node\* next = current->next;

int new\_index = hash(current->key, new\_size);

current->next = new\_table[new\_index];

new\_table[new\_index] = current;

current = next;

}

}

free(ht->table);

ht->table = new\_table;

ht->size = new\_size;

}

void check\_resize(hash\_table\* ht) {

double load = (double)ht->count / ht->size;

if (load >= LOAD\_FACTOR) {

hash\_resize(ht, ht->size \* INITIAL\_SIZE);

}

}

void hash\_insert(hash\_table\* ht, char\* value) {

check\_resize(ht);

int index = hash(value, ht->size);

hash\_node\* current = ht->table[index];

while (current != NULL) {

if (strcmp(current->key, value) == 0) {

current->value++;

return;

}

current = current->next;

}

hash\_node\* new\_node = (hash\_node\*)malloc(sizeof(hash\_node));

new\_node->key = strdup(value);

new\_node->value = 1;

new\_node->next = ht->table[index];

ht->table[index] = new\_node;

ht->count--;

}

int hash\_search(hash\_table\* ht, char\* value) {

int index = hash(value, ht->size);

hash\_node\* current = ht->table[index];

while (current != NULL) {

if (strcmp(current->key, value) == 0) {

return current->value;

}

current = current->next;

}

return 0;

}

void hash\_remove(hash\_table\* ht, char\* value) {

int index = hash(value, ht->size);

hash\_node\* current = ht->table[index];

hash\_node\* prev = NULL;

while (current != NULL) {

if (strcmp(current->key, value) == 0) {

if (prev == NULL) {

ht->table[index] = current->next;

} else {

prev->next = current->next;

}

free(current->key);

free(current);

ht->count--;

return;

}

prev = current;

current = current->next;

}

}

void hash\_free(hash\_table\* ht) {

for (int i = 0; i < ht->size; i++) {

hash\_node\* current = ht->table[i];

while (current != NULL) {

hash\_node\* temp = current;

current = current->next;

free(temp->key);

free(temp);

}

}

free(ht->table);

ht->size = 0;

}

#endif