VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

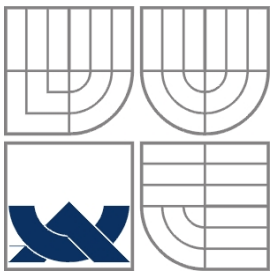
APLIKACE PRO ODHALOVÁNÍ PLAGIÁTŮ

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

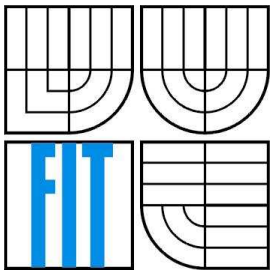
AUTOR PRÁCE
AUTHOR

Bc. PAVEL ŠALPLACHTA

BRNO 2009



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

APLIKACE PRO ODHALOVÁNÍ PLAGIÁTŮ

APPLICATION FOR DETECTION OF PLAGIARISM

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. PAVEL ŠALPLACHTA

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. ROMAN LUKÁŠ, Ph.D.

BRNO 2009

Zadání

1. Prostudujte pečlivě veškeré konstrukce programovacího jazyka C/C++.
2. Zamyslete se nad způsoby detekce dvou velmi podobných programů napsaných v jazyce C/C++.
3. Navrhněte strukturu aplikace, která rozpozná dva velmi podobné programy napsané v programovacím jazyce C/C++.
4. Danou aplikaci implementujte.
5. Funkčnost aplikace otestujte na vzorcích projektů odevzdaných studenty z minulých let.
6. Zhodnoťte dosažené výsledky, porovnejte vaši aplikaci s již existujícími aplikacemi a navrhněte další možné rozšíření do budoucna.

Abstrakt

Tato práce se zabývá programovacími jazyky C a C++, různými způsoby zápisu jejich konstrukcí a vývojem aplikace, která rozpozná velmi podobné programy napsané v těchto programovacích jazycích. Aplikace je určena pro kontrolu plagiátů ve školních projektech, ve kterých mají studenti za úkol vytvořit program v jazyce C nebo C++. Aplikace dokáže zkontrolovat jak krátké programy, tak i rozsáhlé programy rozdělené do několika modulů.

Abstract

This thesis is dealing with programming languages C and C++, various methods writing their constructions and development of application which detects very similar programs written in these languages. The application is intended to control plagiarism in school projects in which students have to create a program in C or C++. The application can check short programs as well as large programs divided into several modules.

Klíčová slova

jazyk C, jazyk C++, plagiát, odhalování plagiátů, preprocesor, statistika

Keywords

C language, C++ language, plagiarism, detection of plagiarism, preprocessor, statistics

Citace

Šalplachta Pavel: Aplikace pro odhalování plagiátů, diplomová práce, Brno, FIT VUT v Brně, 2009

Aplikace pro odhalování plagiátů

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Ing. Romana Lukáše, Ph.D.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Pavel Šalplachta
25. května 2009

Poděkování

Děkuji svému vedoucímu Ing. Romanovi Lukášovi, Ph.D. za odborné vedení a podnětné rady při tvorbě této práce.

Obsah

1 Úvod

„Plagiát je umělecké nebo vědecké dílo, jež někdo jiný než skutečný autor neprávem vydává za své. Při tvorbě plagiátu je zcela nebo z části použito dílo jiného autora. Tento původní autor je úmyslně nebo neúmyslně zatajen.“ [18]

Plagování se také často objevuje na akademické půdě, kde studenti kopírují svoje projekty mezi sebou. Týká se to také projektů, kde studenti mají za úkol něco naprogramovat. Někteří studenti si s tím neví rady, avšak místo toho, aby šli na konzultaci, zkopírují si program svého známého, nepatrně jej poupraví a odevzdají v přesvědčení, že se na to nepřijde. Při dnešním počtu studentů není v lidských silách kontrolovat tyto projekty ručně a je nutné tento proces automatizovat. Z tohoto důvodu vznikla tato diplomová práce, ve které bude navržena a implementována aplikace, která rozpozná velmi podobné programy napsané v programovacích jazycích C nebo C++.

Druhá kapitola se zabývá programovacím jazykem C, jeho preprocesorem a lexikálními a syntaktickými prvky. Třetí kapitola se pak zabývá programovacím jazykem C++. Protože jazyk C++ je nadstavba jazyka C, zabývá se pouze prvky, které jsou v jazyce C++ nové či jiné. Čtvrtá kapitola analyzuje problémy detekce plagiátů na základě stavebních prvků jazyků. Pátá kapitola se zabývá návrhem aplikace, která rozpozná velmi podobné programy napsané v jazycích C nebo C++. V šesté kapitole jsou uvedeny výsledky testování této aplikace, její zhodnocení a možná rozšíření do budoucna.

Tato práce volně navazuje na bakalářské práce [7] a [13], které se tímto tématem také zabývají. Využívá se zde některých principů uvedených v těchto bakalářských pracích, avšak aplikace je vyvíjena zcela od začátku.

V rámci semestrálního projektu bylo napsáno prvních pět kapitol, které však byly v této práci rozšířeny, zejména pátá kapitola.

2 Jazyk C

Programovací jazyk C vyvinuli Ken Thompson a Denis M. Ritchie pro potřeby operačního systému Unix. V současné době je to jeden z nejpopulárnějších jazyků, zřejmě nejčastější pro psaní systémového softwaru, ale velmi rozšířený i pro aplikace [16].

2.1 Historie a normy jazyka C

2.1.1 Počátky jazyka C

Vývoj jazyka C začal v Bellových laboratořích AT&T mezi léty 1969 a 1973. Ritchie tvrdí, že nejpřínosnější období bylo v roce 1972. Pojmenování „C“ zvolili, protože mnoho vlastností přebírali ze staršího jazyka zvaného „B“, jehož název byl zase odvozen od jazyka BCPL (ale to není jisté, neboť Thompson také vytvořil jazyk Bon na poctu své ženy Bonnie) [16].

2.1.2 Norma K&R C

V roce 1978 vydali Ritchie a Brian Kernighan první vydání knihy The C Programming Language. Tato kniha sloužila po mnoho let jako neformální specifikace jazyka. Verze jazyka C, kterou takto popsali, bývá označována jako norma „K&R C“. Tato norma je považována za základní normu, kterou musejí obsahovat všechny překladače jazyka C. Ještě mnoho let po uvedení další normy ANSI C to byl „nejmenší společný jmenovatel“, který využívali programátoři v jazyce C kvůli maximální přenositelnosti, protože zdaleka ne všechny překladače plně podporovaly ANSI C [16].

2.1.3 Norma ANSI C

V roce 1988 byl přijat nový standard, který vychází z K&R C. Jeho součástí je i přesná specifikace množiny knihovních funkcí a hlavičkových souborů, které musí každá implementace ANSI C kompilátoru obsahovat [4].

ANSI C je dnes podporováno téměř všemi rozšířenými překladači. Většina kódu psaného v současné době v C je založena na ANSI C. Jakýkoliv program napsaný pouze ve standardním C je přeložitelný a spustitelný na jakékoli platformě, která odpovídá tomuto standardu [16].

2.1.4 Norma C99

V březnu roku 2000 byla přijata rozšiřující norma jazyka C označena jako ISO 9899:1999 (obvykle nazývaná C99). Tato norma přináší mnoho nových vlastností, kterými jsou například [16]:

- inline funkce;
- pole s nekonstantní velikostí;
- datové typy long long int a bool;
- řádkové komentáře;
- proměnné mohou být deklarovány kdekoliv.

2.2 Preprocesor jazyka C

Před vlastním překladem zdrojového kódu v jazyce C je kód upraven tzv. preprocesorem. V mnoha implementacích se jedná o samostatný program spouštěný překladačem v rámci první fáze překladu. Preprocesor odstraní ze zdrojového kódu komentáře a interpretuje direktivy pro vložení zdrojového

kódu z jiného souboru, definici maker a podmíněné vložení kódu. Všechny direktivy preprocesoru začínají znakem # [1, 19].

Direktiva preprocesoru musí být první na novém řádku, před ní mohou být jen „bílé znaky“. Direktiva končí s koncem řádku. Rozdělit direktivu na více řádků lze napsáním zpětného lomítka před konec řádku [1].

2.2.1 Fáze překladu

Překlad zdrojového kódu v jazyce C se skládá ze čtyř fází [19]:

1. **Nahrazení trigramů** (v podstatě jde o escape sekvence, které umožňují zadávání potřebných 7-bit ASCII znaků i z jiných, většinou starších kódových tabulek).
2. **Spojení řádků** – fyzické řádky zdrojového textu, které jsou zakončeny escape-sekvencí nového řádku (v podstatě jde o obrácené lomítko \ použité jako poslední znak na řádku), jsou spojeny do jediného logického řádku.
3. **Tokenizace** – preprocesor zalomí výsledek do posloupnosti tzv. tokenů preprocesoru a bílých znaků (mezery, tabelátory a konce řádků). Komentáře jsou nahrazeny bílým znakem.
4. **Přepsání maker a provedení direktiv** – v této fázi jsou expandována makra a provedeny direktivy, jako např. vložení zdrojového kódu z jiného souboru nebo vynechání části zdrojového kódu v případě podmíněného překladu.

2.2.2 Makra

Používají se dva základní typy definice maker. Jednak lze definovat makra bez parametru, která expandují vždy do stejné sekvence znaků (resp. tokenů), jednak lze definovat makra s parametry, které se z praktického hlediska chovají podobně jako funkce. Makro se definuje za direktivou `#define` a lze jej zrušit direktivou `#undef`. Obecná syntaxe direktivy `#define` je:

```
#define identifikator seznam_nahrazujicich_tokenu
#define identifikator(seznam_parametru) seznam_nahrazujicich_tokenu
```

Speciálním případem je definice samotného identifikátoru, kterému není přiřazen žádný seznam tokenů – užívá se u podmíněného překladu [19].

Příklady maker:

```
#define DEBUG
#define POCET 128
#define MAX(a,b) ((a>b)?a:b)
```

Mezi nahrazujícími tokeny se mohou vyskytovat také operátory # a ##. Operátor # je unární a následuje za ním jméno parametru makra. Během rozvíjení makra je # a jméno parametru makra nahrazeno odpovídajícím skutečným parametrem, který je uzavřen v řetězcových uvozovkách. Například [3]:

```
#define TEST(a, b) printf( #a "<" #b "=%d\n", (a)<(b));
TEST(0, 0xFFFF)
```

se rozvine jako:

```
printf("0" "<" "0xFFFF" "=%d\n", (0)<(0xFFFF));
```

Operátor ## je binární a v průběhu rozvoje makra se každé dva tokeny okolo operátoru ## spojí do jednoho. Například [3]:

```
#define TEMP(i) temp ## i
TEMP(1) = TEMP(2);
```

se rozvine jako:

```
temp1 = temp2;
```

2.2.3 Podmíněný překlad

Pro podmíněný překlad se používají direktivy `#if`, `#ifdef`, `#ifndef`, `#else`, `#elif` a `#endif`. Direktiva `#if` vyhodnocuje podmínku, která za ní následuje (tedy preprocesor jí vyhodnocuje). Pokud je tato podmínka vyhodnocena jako FALSE, pak vše mezi `#if` a `#endif` je ze zdrojového kódu vypuštěno.

V podmínce musí být výrazy, které může vyhodnotit preprocesor (tj. nelze tam používat hodnoty proměnných atd.). Pomocí klíčového slova `defined` lze vyhodnocovat, zda existuje nějaké makro (zda bylo definováno). Výraz `#if defined` je možno zkrátit výrazem `#ifdef`, výraz `#if !defined` lze zkrátit výrazem `#ifndef`. Za podmínkou `#if` může být další podmínka `#elif`. Ta se vyhodnotí v případě, že předchozí podmínka nebyla splněna. Pokud nejsou splněny žádné podmínky, může být jako poslední direktiva `#else`, která již žádný výraz nevyhodnocuje a provede se právě v případě, že všechny podmínky za `#if` a `#elif` byly vyhodnoceny jako FALSE. Podmíněný překlad se ukončuje direktivou `#endif` [1].

2.2.4 Vkládání souborů

Nejčastěji využívanou direktivou preprocesoru je vložení jiného souboru. To se provádí pomocí direktivy `#include`, která má dvě varianty:

```
#include <soubor.h>
#include "soubor.h"
```

V prvním případě se vezme soubor z adresáře se standardními hlavičkovými soubory, ve druhém případě pak z adresáře se zdrojovým kódem programu.

Vkládané soubory mají obvykle příponu .h, ale jde pouze o programátorskou konvenci, nikoli o vlastnost jazyka C. Obsah vkládaného souboru není nijak omezen. Lze tedy například rozčlenit velký projekt do více zdrojových souborů a v jediném *.c souboru, který se bude překládat, mít jen několik `#include` příkazů. Takto se ovšem v C v praxi neprogramuje, vkládání souborů se obvykle používá pouze na zpřístupnění maker, typů funkcí a proměnných [9].

2.2.5 Ostatní direktivy

Direktiva #error

Tato direktiva vyvolá při překladu chybové hlášení, jež obsahuje parametry za touto direktivou. Parametry příkazu podléhají rozvoji maker. Nejčastěji se používá při zjišťování programátorských nesrovnalostí a porušení zásad během zpracování programu preprocesorem [3].

Příklad použití direktivy:

```
#if defined (PRAVDA) && defined (NE_PRAVDA)
  #error "Nesouhlas v pravdě!"
#endif
```

Direktiva #line

Direktiva `#line` naznačuje překladači, že zdrojový program byl vygenerován jiným programovým nástrojem a dává do spojitosti místa ve zdrojovém programu jazyka C a řádky v originálním uživatelském souboru, ze kterého se zdrojový program vygeneroval. Informace, kterou udává příkaz `#line`, se používá pro nastavení hodnot předdefinovaných maker `__LINE__` a `__FILE__`. Chování příkazu není dále specifikováno a překladače ho mohou ignorovat. Typicky se informace používají v diagnostických hlášeních [3].

Direktiva #pragma

Poslední direktiva se v implementaci používá pro přidání nových funkcí preprocesoru nebo pro předávání informací definovaných implementací do překladače. Informace, která následuje za jménem příkazu `#pragma`, není nijak omezena a implementace musí ignorovat informace, kterým nerozumí [3].

2.3 Lexikální elementy jazyka C

2.3.1 Identifikátory

Identifikátory jsou jména, která dáváme například **proměnným**, **funkcím** a **datovým typům**. Identifikátor se musí lišit od kteréhokoliv klíčového slova. Identifikátor je tvořen posloupností alfanumerických znaků a podtržítka, přičemž musí být splněna podmínka, že první symbol nesmí být číslice. Jazyk C je case sensitive, tzn. rozlišuje malá a velká písmena. V praxi to znamená, že: prom, Prom a PROM jsou tři různé identifikátory [13].

2.3.2 Klíčová slova

Klíčová slova jsou vyhrazené identifikátory, které slouží pro označení standardních typů dat, některých příkazů jazyka C apod. ANSI C definuje 32 klíčových slov, jimiž jsou: **auto**, **break**, **case**, **char**, **const**, **continue**, **default**, **do**, **double**, **else**, **enum**, **extern**, **float**, **for**, **goto**, **if**, **int**, **long**, **register**, **return**, **short**, **signed**, **sizeof**, **static**, **struct**, **switch**, **typedef**, **union**, **unsigned**, **void**, **volatile**, **while** [8]. Norma ANSI C99 přidává navíc klíčové slovo **inline** a při načtení hlavičkového souboru *stdbool.h* také klíčová slova **bool, true** a **false**.

2.3.3 Konstanty

Konstanty jsou symboly, reprezentující neměnnou číselnou nebo jinou hodnotu. Překladač jazyka jim přiřadí typ, který této hodnotě odpovídá. Z konstant odpovídajících si typů můžeme vytvářet konstantní výrazy. Tyto výrazy musí být regulární. Nesmí obsahovat přiřazení, inkrementace a dekrementace, funkční volání, operátor čárka. Konstantám s vhodně zvolenými identifikátory dáváme přednost například před přímým uvedením konstantní hodnoty jako meze cyklu nebo dimenze pole. Modifikace programu pak probíhá velmi snadno změnou hodnoty konstanty. Odpadá obtížné uvažování, zdali ta či jiná hodnota má být modifikována či nikoliv. Konstanty mají rovněž určen typ. Tím je umožněna typová kontrola. Konstanty definujeme po klíčovém slově **const**

a následovaném typem konstanty, jejím identifikátorem a po rovnítku její hodnotu ukončenou středníkem [13].

Celočíselné konstanty

V jazyce C máme tři druhy celočíselných konstant:

- **desítkové** (dekadické) – posloupnost číslic, z nichž první nesmí být 0 (nula);
- **osmičkové** (oktalové) – číslice 0 (nula) následovaná posloupností osmičkových číslic (0–7);
- **šestnáctkové** (hexadecimální) – číslice 0 následovaná znakem `x` (nebo `X`) a posloupností hexadecimálních číslic (0–9, a–f, A–F).

Příklady celočíselných konstant:

1, desítkové – `0`, `1`, `15`, `90`;
2, osmičkové – `0`, `01`, `015`, `065`;
3, šestnáctkové – `0x0`, `0x1`, `0xF`, `0xcd`, `0Xcd`.

Záporné konstanty označujeme znakem „−“ (mínus), například: `−10`. Pro konstanty typu long použijeme příponu `L` (nebo `l`), například `123L`. Další příponou je `U` (nebo `u`), která značí unsigned, například `50u`, `123LU` [13].

Reálné konstanty

Tvoří se podle běžných zvyklostí, mohou začínat a končit desetinnou tečkou a jsou implicitně typu double, např. `15.`, `5.6`, `.84`, `5e6`, `7E23`, `3E-20`.

Reálná konstanta typu float se definuje pomocí přípony `F` (nebo `f`), např. `3.14f`, a typu long double pomocí `L` (nebo `l`), např. `12e3L` [13].

Znakové konstanty

Hodnota znakových konstant (ordinální číslo) je odvozena z odpovídající tabulky – nejčastěji ASCII. Potřebujeme-li znakovou konstantu z neviditelného znaku, pak použijeme zápis ve tvaru `'\ddd'`, kde `ddd` je kód znaku složený ze tří osmičkových číslic, např. `'\012'`, `'\007'`. Počáteční nuly jsou nutné. Druhou možností, jak zapsat tyto konstanty, je použití zápisu `'\xhh'`, kde `hh` označuje hexadecimální číslice, např. `'x0A'`, `'XD'`, `'X1f'`.

Znak „\“ (zpětné lomítko – backsplash) se často nazývá escape character, protože je používán pro změnu běžného významu. Např. ‘`012`‘ je nedovolená konstrukce (tříznaková konstanta), protože znaková konstanta může mít jen jeden znak [13].

2.3.4 Operátory

Operátory v jazyce C lze rozdělit několika způsoby. Mimo jiné na unární (viz tabulka 2.1), binární (viz tabulka 2.2) a ternární, neboli na operátory s jedním, dvěma nebo třemi operandy. Operandy jsou data, se kterými operátory pracují.

Ternárním operátorem v jazyce C je podmíněný operátor (`? :`). Prvním operandem je výraz, který se vyhodnotí jako logický výraz (TRUE nebo FALSE). Pokud se vyhodnotí jako TRUE, výsledkem bude druhý operand (mezi `?` a `:`), jinak třetí operand.

Operátory se také rozdělují na aritmetické (`+`, `−`, `*`, `/`, `%`), bitové (`~`, `<<`, `>>`, `&`, `|`, `^`), relační (`<`, `<=`, `>`, `>=`, `==`, `!=`) a logické (`!`, `&&`, `||`) [1].

Operátor	Význam
`+, -`	unární plus a mínus
`&`	reference (získání adresy objektu)
`*`	dereference (získání objektu dle adresy)
`!`	logická negace
`~`	bitová negace
`++, --`	inkrementace a dekrementace hodnoty
`(typ)`	přetypování
`sizeof(typ)`	získání velikosti daného datového typu

Tabulka 2.1: Unární operátory

Operátor	Význam
`=`	přiřazení
`+, -, *, /, %`	plus, mínus, krát, děleno, modulo
`<<, >>`	bitový posun vlevo, vpravo
`&, |, ^`	bitový and, or ,xor
`&&, ||`	logické and, or
`.`	tečka, přímý přístup ke členu struktury
`->`	nepřímý přístup ke členu struktury
`,`	čárka, oddělení výrazů
`<, <=, >, >=`	menší než, menší nebo rovno, větší než, větší nebo rovno
`==, !=`	rovnost, nerovnost

Tabulka 2.2: Binární operátory

2.4 Syntaktické a sémantické prvky jazyka C

2.4.1 Základní datové typy

Základní typy dat dělíme na **aritmetické datové typy** (celočíselné a racionální), **znaky** a na **ukazatele**.

Celočíselnými datovými typy jsou:

- int,
- short int (též short),
- long int (též long),
- long long int (též long long).

Racionálními datovými typy jsou:

- float,
- double,
- long double.

Znaky jsou reprezentovány datovým typem **char**.

Typy **char**, **short int**, **long int**, **long long int** mohou být buď typu **signed** (znaménkový) nebo **unsigned** (neznaménkový). Typ **unsigned int** se často zkracuje jen na **unsigned.** Pro celočíselné typy je implicitní typ **signed**, pro znaky záleží na implementaci [13].

Typ ukazatel

K libovolnému typu T lze vytvořit typ „ukazatel na T“. Typ ukazatel se jinak nazývá ukazatel na objekt nebo ukazatel na funkci podle toho, je-li T typ nějakého objektu nebo typ funkce. Hodnota typu ukazatel je adresa typu T nebo adresa funkce typu T. Příklady [3]:

```
int *ip;      // ip: ukazatel na objekt typu int
char *cp;     // cp: ukazatel na objekt typu char
int (*fp)();  // fp: ukazatel na funkci vracející celé číslo
```

Ve spojení s ukazatelem jsou důležité dva operátory, kterými jsou:

- adresový operátor `&`, který získá adresu objektu;
- nepřímý operátor `*`, který získá obsah adresy, na kterou ukazatel míří.

2.4.2 Uživatelské datové typy

Uživatelské datové typy můžeme vytvářet pomocí příkazu **typedef**. Jsou to však spíše aliasy jiných typů. Syntaxe příkazu je:

```
typedef definice_typu identifikator;
```

Za klíčovým slovem **`typedef`** následuje `definice_typu`, kterou může být například struktura (viz kapitola 2.4.10) nebo nějaký základní typ jazyka C, a `identifikator`, který slouží k pojmenování nového datového typu [1].

2.4.3 Proměnné

Proměnné jsou paměťová místa přístupná prostřednictvím identifikátoru. Hodnotu proměnných můžeme během výpočtu měnit. Tím se proměnné zásadně odlišují od konstant, které mají po celou dobu chodu programu hodnotu neměnnou – konstantní. Proměnné deklarujeme uvedením datového typu, jež je následován identifikátorem, nebo seznamem identifikátorů navzájem oddělených čárkami. Deklarace končí středníkem. Současně s deklarací proměnné můžeme definovat i její počáteční hodnotu [13].

Příklady deklarací proměnných:

```
int a, b, c, pocet = 0;
float x, prumer = 0.0, odchylka;
double y;
```

Kromě identifikátoru a datového typu jsou proměnné určeny ještě paměťovou třídou, které náleží. Paměťová třída určuje, ve které části paměti bude proměnná kompilátorem umístěna, a také, kde všude bude proměnná viditelná. Jazyk C rozeznává tyto modifikátory paměťové třídy [4]:

- **auto** – je implicitní pro lokální proměnné a proměnná je uložená na zásobníku;
- **extern** – používá se pro zpřístupnění globální proměnné z jiného zdrojového souboru;
- **register** – proměnná se umístí do registru procesoru, je-li některý volný;

- **static** – proměnná je uložena v datové oblasti, lokální proměnná si ponechává hodnotu.

Libovolná proměnná určitého datového typu, která je zařazena do určité paměťové třídy, může být navíc ještě modifikována typovým modifikátorem. Jazyk C rozeznává dva [4]:

- **const** – proměnné po inicializaci již nelze měnit hodnotu;
- **volatile** – proměnná může být měněna pomocí některých systémových přerušení.

2.4.4 Složený příkaz, blok

Syntaxe bloku je velice jednoduchá. Blok začíná otevírací složenou závorkou a končí zavírací složenou závorkou. Mezi těmito závorkami mohou být libovolné jiné příkazy včetně dalších bloků. Blok je v C chápán jako jediný příkaz, a proto se také dá použít všude tam, kde je možné použít příkaz. Blok se také často označuje pojmem složený příkaz [11].

2.4.5 Podmíněný příkaz if

Příkaz `if` je základním příkazem, který slouží k větvení toku programu. Jeho syntaxe je:

```
if (vyraz1)
  prikaz1
```

Nejprve je vyhodnocen `vyraz1` (závorky kolem výrazu nelze vynechat). Pokud je hodnota, vzniklá jeho vyhodnocením, nenulová, provede se `prikaz1`. V opačném případě, kdy je `vyraz1` vyhodnocen jako nula, se `prikaz1` neprovede. Jak již je známo z předchozí kapitoly, místo příkazu lze použít i složený příkaz. Příkaz `if` lze ještě doplnit o nepovinnou část **`else`**:

```
if (vyraz1)
  prikaz1
else
  prikaz2
```

Rozdíl při použití `else` je v tom, že při nesplnění podmínky (`vyraz1` je vyhodnocen jako 0) se provede `prikaz2` [11].

2.4.6 Příkaz mnohonásobného větvení (switch)

Pokud potřebujeme tok programu větvit do více jak dvou směrů, můžeme místo několika do sebe vnořených příkazů `if-else` využít možností, které nám v jazyce C poskytuje příkaz `switch`. Jeho syntaxe je:

```
switch (vyraz1) {
  case konstantni_vyraz: prikazy
  case konstantni_vyraz: prikazy
  .
  .
  .
  default: prikazy
}
```

Při provádění příkazu **switch** je nejdříve vyhodnocen celočíselný `vyraz1` uzavřený v kulatých závorkách a pak se postupně vyhodnocují `konstantni_vyrazy` v návěští **case**. V případě, že je nalezeno návěští, kde je `konstantni_vyraz` roven hodnotě `vyraz1`, začnou se provádět všechny příkazy uvedené za tímto návěštím až do konce příkazu `switch`. To ale znamená, že se provedou i všechny příkazy v následujících větvích. Pokud tedy chceme, aby se provedla vždy jen jedna větev, lze vykonávání příkazu switch okamžitě zastavit uvedením příkazu **break**. Pokud není nalezena žádná vyhovující větev, skočí se na návěští **default**, které může být uvedeno kdekoliv v těle příkazu switch [11].

2.4.7 Cykly

Jazyk C umožňuje použít tři příkazy pro iteraci – **while**, **for** a **do-while**. Ve všech typech cyklů lze použít příkazy **break** a **continue**, které mění „normální“ průběh cyklu [4]:

- **break** – ukončuje nejvnitřnější neuzavřenou smyčku – opouští okamžitě cyklus;
- **continue** – skáče na konec nejvnitřnější neuzavřené smyčky a tím vynutí další iteraci smyčky.

Cyklus while

Tento cyklus bývá také nazýván cyklus s podmínkou na začátku. Používá se v případech, kdy dopředu neznáme počet iterací a kdy se také může stát, že cyklus nebude muset proběhnout ani jednou. Jeho syntaxe je:

```
while (vyraz1)
  prikaz1
```

Výraz `vyraz1`, podle kterého se rozhoduje o provádění (resp. neprovádění) iterace cyklu, se testuje ještě před prvním průchodem. Iterace se nezačne provádět, pokud je tento výraz nepravdivý. Výraz `vyraz1` se nazývá záhlaví cyklu a musí být vždy uveden v kulatých závorkách. Příkaz `prikaz1` se nazývá tělem cyklu [13].

Cyklus for

Tento cyklus bývá také nazýván cyklem se známým počtem průchodů. Používáme jej v případě, že už před jeho prvním průchodem víme, kolikrát se bude muset provést. Jeho syntaxe je:

```
for (vyraz1; vyraz2; vyraz3)
  prikaz1
```

Příkaz **for** provádí příkaz `prikaz1` jednou, vícekrát, nebo vůbec, dokud je hodnota nepovinného testovacího výrazu `vyraz2` nenulová. Výraz `vyraz1` je také nepovinný. Provádí se před prvním vyhodnocením testu. Typicky se používá pro inicializaci proměnných před cyklem. Po každém provedení těla cyklu se provede opět nepovinný výraz `vyraz3`. Typicky se jedná o přípravu další iterace cyklu. Pokud neuvedeme testovací výraz `vyraz2`, použije překladač hodnotu 1, a bude tedy provádět nekonečný cyklus. Avšak pro jeho ukončení můžeme použít příkaz **break** [13].

Cyklus do-while

Je to jediný cyklus, který zajišťuje alespoň jedno provedení těla cyklu. Jeho syntaxe je:

```
do prikaz1 while (vyraz);
```

Testovací výraz `vyraz1` je testován až po průchodu tělem cyklu. Pokud je test splněn, provádí se znovu `prikaz1` (tělo cyklu) [13].

2.4.8 Funkce

Funkce jsou základním stavebním kamenem jazyka C. Představují výhodný způsob zapouzdření nějakého výpočtu, aniž bychom se museli později starat o jeho implementaci. V jazyce C je používání funkcí jednoduché, šikovné a efektivní. Často narazíme na krátkou funkci, která je pouze jednou definovaná a pouze jednou zavolaná jenom proto, že to zjednodušuje chápání části programu.

Syntaxe deklarace funkce:

```
navratovy_typ jmeno_funkce (seznam_argumentu);
```

Syntaxe definice funkce:

```
navratovy_typ jmeno_funkce (seznam_argumentu)
{
  //telo funkce
}
```

Na návratový typ nejsou kladena žádná omezení. Pokud funkce nevrací žádnou hodnotu, použije se klíčové slovo **void**. Návratová hodnota se vrátí pomocí argumentu v příkazu **return**, který se dává do těla funkce.

Seznam argumentů je nepovinný. Funkce nemusí mít žádné argumenty, může mít jeden nebo více argumentů, nebo také můžeme určit, že funkce má proměnný počet argumentů. Deklarace všech formálních argumentů funkcí musí obsahovat datový typ. Není možné uvést společný typ pro více následujících identifikátorů.

Funkci voláme tak, že napíšeme název (identifikátor) funkce a do kulatých závorek dáme seznam předávaných argumentů, například [13]:

```
printf ("Hello World!");
prumer = vypocti_prumer(5.5);
```

2.4.9 Pole

Pole je kolekce proměnných stejného typu, které mohou být označovány společným jménem. K prvkům pole přistupujeme prostřednictvím identifikátoru pole a indexu. Pole v C obsazuje spojitou oblast operační paměti tak, že první prvek je umístěn na nejnižší přidělené adrese a poslední na nejvyšší přidělené adrese. Syntaktická deklarace pole je:

```
typ jmeno[rozsah];
```

`Typ` určuje typ prvků pole, `jmeno` představuje identifikátor pole a `rozsah` je kladný počet prvků pole (celočíselný konstantní výraz, který musí být vyčíslitelný za překladu).

V jazyce C je možné deklarovat pouze jednorozměrné pole. Jeho prvky ovšem mohou být libovolného typu. Mohou tedy být opět jednorozměrnými poli. To však již dostáváme vektor vektorů, tedy matici [13].

Deklarace dvourozměrného pole má syntaxi:

```
typ jmeno [rozsah1][rozsah2];
```

2.4.10 Struktura, union

Zatímco pole je homogenní datový typ – všechny jeho prvky jsou stejného typu – struktura je datový typ heterogenní (datový typ je složen z datových prvků různých typů, což je ale jeho vnitřní záležitostí, protože navenek vystupuje jako jednolitý objekt).

Jedna z možných definic struktury je:

```
struct {
  int vyska;
  float vaha;
} pavel, honza, karel;
```

Další možnosti definice struktury jsou:

```
stuct miry {
  int vyska;
  float vaha;
};
struct miry pavel;
struct miry honza, karel;

typedef struct {
  int vyska;
  float vaha;
} MIRY;
MIRY pavel, honza, karel;
```

Ještě další možností je ještě za **typedef stuct** přidat název struktury. Tato možnost se používá, když chceme, aby struktura mohla odkazovat sama na sebe. To se využívá například u spojovaných seznamů.

Přístup k prvkům struktury je pomocí tečkové notace:

```
pavel.vyska=175;
```

Pokud použijeme ukazatel na strukturu (např. `MIRY *pavel;`), můžeme k prvkům přistupovat dvěma způsoby:

```
1. (*pavel).vyska=175;
2.  pavel->vyska=175;
```

Podobnou syntaxi jako má struktura, má i **union**, který je obdobou pro **variantní záznam** v programovacím jazyce Pascal. Datový typ union znamená, že se vyhradí paměť pro největší položku ze všech položek v unionu definovaných. Všechny položky unionu se překrývají (ve struktuře by ležely v paměti za sebou), což znamená, že v něm může být v jednom okamžiku pouze jedna položka. V praxi se uniony používají málokdy [13].

2.4.11 Výčtový typ (enum)

Výčtový typ má podobnou konstrukci jako struktura či union, ale má zcela jinou filosofii práce. Pomocí něho lze snadno definovat seznam symbolických konstant, které mohou být vzájemně závislé. Podobně jako u struktury a unionu má i výčtový typ několik různých způsobů zápisu. Nejčastějším způsobem zápisu je:

```
typedef enum {
  MODRA, CERVENA, ZELENA, ZLUTA
} BARVY;
Barvy c,d;
c = MODRA;
d = CERVENA;
```

Pokud explicitně nepřiřadíme číselné hodnoty jednotlivým prvkům vyjmenovaného typu, pak mají tyto prvky implicitní hodnoty 0, 1, 2. Chceme-li jiné hodnoty, použijeme tento zápis:

```
typedef enum {
 MODRA=0, CERVENA=4, ZELENA=2, ZLUTA
} BARVY;
```

Barva `ZLUTA` není inicializována, má tedy hodnotu 3 [13].

3 Jazyk C++

„Jazyk C++, který je jakousi nástavbou jazyka C a s podporou OOP (objektově orientovaného programování) se stal důležitým jazykem 20. století a jeho důležitost pokračuje.“ [6]

3.1 Historie jazyka

Jazyk C++ vyvinul Bjarne Stroustrup počátkem 80. let v Bellových laboratořích. Jméno C++ pochází z přírůstového operátoru C++, který přičítá k hodnotě proměnné jedničku. Připomíná tedy jasně rozšířenou verzi jazyka C. Stroustrup založil C++ na stručnosti C, přidal rysy OOP aniž by významně změnil složku C. C++ je tedy nadstavba znamenající, že každý program napsaný v C je také platným programem pro C++. Programy v C++ mohou využívat softwarové knihovny C [6].

3.2 Přehled rozdílů oproti jazyku C

C++ je nadstavbou nad jazykem C90. Některá rozšíření jazyka C++ se později objevila v jazyce C v normě C99, jako jsou řádkové komentáře, datový typ bool a inline funkce. Dalšími rozšíření jsou například:

- typ reference;
- deklarace v bloku je příkaz;
- anonymní unie;
- přetěžování funkcí a operátorů;
- operátory `new`, `delete`, `new[]` a `delete[]`;
- třídy (`class`), abstraktní třídy:
  - ukrývání dat (`private`, `public`, `protected`);
  - automatická inicializace;
  - uživatelem definované konverze;
  - polymorfismus (virtuální funkce);
- jméno třídy a výčtu je jméno typu;
- dědičnost, násobná dědičnost;
- ukazatele na členy tříd;
- v inicializaci statických objektů je dovolen obecný výraz;
- generické datové typy - šablony (`template`, `typename`);
- obsluha výjimek (`try`, `catch`, `throw`);
- prostory jmen (`namespace`, `using`);
- nové způsoby přetypování (`static_cast`, `const_cast`, `reinterpret_cast`, `dynamic_cast`);
- informace o typu za běhu programu (`typeid`, `type_info`);
- klíčové slovo `mutable`.

Více o rozdílech mezi jazyky C a C++ se můžete dočíst na [10].

3.3 Přidané lexikální elementy v jazyce C++

3.3.1 Klíčová slova

Kromě klíčových slov uvedených v kapitole 2.32 jsou v jazyce C++ navíc klíčová slova: **asm**, **catch**, c**lass**, **const_cast**, **delete**, **dynamic_cast**, **explicit**, **export**, **friend**, **mutable, namespace**, **new, operator**, **private**, **protected**, **public**, **reinterpret_cast**, **static_cast**, **template**, **this**, **throw**, **try**, **typeid**, **typename**, **using**, **virtual** a **wchar_t**.

Jazyk C++ má dále vyhrazena klíčová slova pro názvy operátorů: **and**, **and_eq**, **bitand**, **bitor**, **compl**, **not**, **not_eq**, **or**, **or_eq**, **xor** a **xor_eq** [10].

3.3.2 Operátory

Jazyk C++ obsahuje všechny operátory, které se nachází v jazyce C, avšak je ještě rozšířen o několik nových operátorů. Jsou jimi operátory pro přetypování (`const_cast`, `static_cast`, `dynamic_cast`, `reinterpret_cast`), operátory pro přidělení (`new`) a uvolnění (`delete`) paměti, operátor rozlišení platnosti (`::`), dereference třídních ukazatelů (`.*`, `->*`) a identifikace typu (`typeid`) [10].

Jak již bylo zmíněno v předchozí kapitole, jazyk C++ vyhrazuje pro některé operátory místo klasického označení také alternativní reprezentaci klíčovým slovem (viz tabulka 3.1).

and	and_eq	bitand	bitor	compl	not	not_eq	or	or_eq	xor	xor_eq
&&	&=	&	|	~	!	!=	||	|=	^	^=

Tabulka 3.1: Alternativní reprezentace operátorů

3.4 Syntaktické a sémantické prvky jazyka C++

V této kapitole se budeme věnovat syntaktickým a sémantickým prvkům jazyka C++, které se nevyskytují v jazyce C nebo jsou jiné.

3.4.1 Prostory jmen

Prostor jmen dává programátorovi mocný nástroj pro řešení problémů s případným konfliktem v oblasti identifikátorů. Konflikt identifikátorů nastává například u týmových projektů či při používání knihoven tříd či funkcí, získaných z různých zdrojů. K definici prostoru jmen slouží klíčové slovo **namespace**. Deklaruje se následujících způsobem:

```
namespace identifikator {seznam_deklaraci}
```

Za `namespace` následuje jméno prostoru jmen a ve složených závorkách se nachází oblast platnosti identifikátorů.

Použitím prostoru jmen musíme použít operátor :: ve formě prostor_jmen::identifikator. Aby programátor nemusel vždy psát jméno prostoru společně i se jménem, které chce napsat, existuje klíčové slovo **using**. Napsáním **using namespace** *jmeno prostoru*, bude v následujícím zdrojovém textu přístup k identifikátorům tohoto prostoru jako kdybychom se nacházeli uvnitř tohoto prostoru [2, 9].

3.4.2 Třídy a objekty

Objekt je zjednodušeným pohledem na nějaký skutečný předmět, osobu, pojem. Objekty se navzájem liší svou jedinečností, i když mohou mít řadu stejných či podobných vlastností či způsobu chování. **Třída** je popisem objektů se společnými vlastnostmi. Objekt pak je instancí nějaké třídy a obsahuje data a metody. K vytvoření třídy slouží klíčové slovo **Class** [12]. Definice vypadá následovně:

```
class NazevTridy {
  private:
    clenska data a metoda
    // primo dostupne jen v ramci tridy
  protected:
    clenska data a metoda
    // primo dostupne jen z tridy a jejich potomku
  public:
    clenska data a metoda
    // primo dostupne odkudkoliv, tvori rozhrani tridy
};
```

Instance třídy se tvoří stejně, jako bychom definovali proměnnou, tzn. například:

```
moje_trida moje_instance;
```

Zaslání zprávy objektu se provádí následujícím způsobem:

```
Instance.NazevZpravy(parametry);
```

Každá instance je vytvořena pomocí speciální metody – konstruktoru. Konstruktor se musí jmenovat stejně jako třída, jejíž instance se bude vytvářet, nesmí mít návratovou hodnotu a měl by být veřejný. Každá třída může mít k dispozici více konstruktorů s různými parametry.

K likvidaci objektu se používá destruktor. Destruktor se jmenuje stejně jako třída, jen před názvem třídy je znak ~. Destruktor nesmí mít návratovou hodnotu a žádné parametry [2].

3.4.3 Výjimky

Výjimky patří k doporučovaným a moderním programátorským technikám. Pod pojmem výjimka se rozumí nějaká výjimečná situace, která nastane ve volané metodě, nebo funkci. Typickým příkladem může být zápis do souboru, když například během zápisu vytáhne uživatel disketu. V jazyce C i C++ se často používá návratových hodnot funkcí, které vracejí úspěšnost provádění nějaké operace. Chceme-li v posloupnosti několika řádků zdrojového textu takto ošetřit všechny možné chyby, vznikne nám velice nepřehledný program plný vnořených příkazů `if`. Jinou možnost nám nabízí mechanismus výjimek. Princip spočívá v tom, že označíme posloupnost příkazů do zvláštního bloku, říká se mu často hlídaný blok, ve kterém neošetřujeme žádné chyby. Právě v tomto bloku může vzniknout výjimka. Za tímto blokem označíme postupně jaké výjimky mohly v hlídaném bloku nastat, a jak je ošetřit. V C++ může být výjimkou proměnná jakéhokoliv primitivního datového typu, nebo instance jakékoliv třídy. Výjimka by v sobě měla nést nějakou informaci o situaci, která nastala a proč byla vyvolána.

Výjimka se vyvolá pomocí klíčového slova **throw**. Hlídaný blok se značí klíčovým slovem **try**. K odchycení vyvolaných výjimek slouží klíčové slovo **catch**:

```
try {
  //hlidany blok
   throw vyjimka;
}
catch (typ_vyjimky_id) {
  //zpracovani vyjimky
}
```

Více o výjimkách naleznete například v [2] nebo v [12].

3.4.4 Šablony funkcí a datových typů

Šablona funkce je konstrukce, která umožňuje vytvořit funkci na základě parametrů. Proto se také vytváření a používání šablon někdy nazývá parametrické programování, a v souvislosti s šablonami se hovoří o parametrismu. Význam šablon se často a nejlépe demonstruje na příkladu funkce swap, která vymění obsah dvou proměnných. Představme si situaci, kdy chceme mít možnost vyměnit hodnoty proměnných mnoha typů nebo i instancí mnoha tříd. Bez použití šablon bychom museli napsat mnoho funkcí swap, které by se od sebe lišili jen typem parametrů. Lepší řešení je vytvořit jednu šablonu funkce swap, podle které se dle potřeby vytvoří požadovaná funkce na základě parametru, kterým bude typ. Syntaxe šablony funkce je:

```
template<parametry sablony> deklarace
```

Šablona funkce swap:

```
template<class T> void swap(T &a, T &b)
{
  T c(a);
  a=b;
  b=c;
}
```

Parametrem šablony je typ T. Klíčové slovo *class* značí, že parametrem bude datový typ. Nemusí se jednat pouze o třídu, ale o jakýkoliv datový typ, i primitivní datový typ. Konkrétní funkci pro konkrétní typ (instanci šablony) vytvoříme dosazením typu za parametr T, například [2]:

```
swap<int>(a,b);
```

V jazyce C++ existují také šablony datových typů (struktur, tříd, unií). Syntaxe pro třídy je následující:

```
template <class Typ> class Obal
{
  private:
    Typ Promenna;
  public:
    Obal();
    Obal(Typ p);
};
```

Tato šablona třídy vytváří jakýsi „objektový obal“ pro jakoukoliv proměnnou. Vytvoříme-li instanci této šablony pro nějaký konkrétní typ, instance šablony třídy je třída. Šablona třídy by se dala považovat za něco jako metatřídu. Metody šablony třídy se implementují obdobně jako metody třídy. Jen je třeba si uvědomit, že i metoda šablony třídy je vlastně šablona. Proto musíme uvést klíčové slovo `template` i s parametry šablony.

Součástí jazyka C++ je i standardní knihovna šablon – STL, která obsahuje šablony datových kontejnerů, iterátory a šablony algoritmů, které pomocí iterátorů pracují s datovými kontejnery [2].

3.4.5 Přetěžování funkcí a operátorů

Další novinkou jazyka C++ je možnost přetěžovat funkce. To znamená, že je možnost deklarovat i definovat více funkcí stejného jména s různým počtem nebo s různými typy parametrů. Například můžeme definovat dvě funkce se stejným jménem `max`:

```
int max(int a, int b)
{
  return (a>b)? a:b;
}

float max(float a, float b)
{
  return (a>b)? a:b;
}
```

Zavoláme-li funkci `max` s parametry typu float, vrátí se větší z nich jako typ float. Zavoláme-li funkci `max` s parametry typu int, vrátí se větší z nich jako typ int [2].

Kromě přetěžování funkcí, lze přetěžovat také **standardní operátory**. Pro přetěžování operátorů platí tato pravidla [2]:

- Přetížené operátory se nemohou lišit pouze v typu návratové hodnoty, ale v typech parametrů.
- Přetížíme-li binární operátor, musí být zase binární. Přetížíme-li unární operátor, musí být zase unární. U operátorů tedy nelze měnit počet parametrů.
- Nelze vytvářet nové operátory.
- Nelze přetěžovat operátory: `.`, `.*`, `::`, `?:`, `sizeof` a operátory pro přetypování.
- Pro operátory platí stále stejná priorita, ať jsou přetížené jakkoliv. Prioritu operátorů v C++ nelze nijak změnit.

Operátory se v jazyce C++ chovají podobně jako funkce. Jméno operátorové funkce je tvořeno klíčovým slovem `operator`, za kterým následuje symbol tohoto operátoru. Například [2]:

```
Vektor Vektor::operator=(const Vektor &kopie)
{
  for (int i=0; i<3; i++)
  {
    pole[i]=kopie[i];
  }
  return kopie;
}
```

3.4.6 Přetypování

Jazyk C++ „zdědil“ po jazyce C operátor přetypování. V jazyce C se přetypuje výraz například takto:

```
int a=65;
char A = (char) a;
```

Použít tento operátor přetypování v C++ není vhodné a někdy je dokonce i nebezpečné. Jazyk C++ nabízí nové operátory k přetypování, kterými jsou: `const_cast`, `dynamic_cast`, `static_cast` a `reinterpret_cast`. Jejich syntaxe je [2]:

```
operator_pretypovani<cilovy_typ>(vyraz)
```

Příklad přetypování:

```
int a=65;
char A = static_cast<char>(a);
```

4 Analýza

V předchozích dvou kapitolách jsme se seznámili s programovacími jazyky C a C++ a různými zápisy jejich konstrukcí. Nyní se již tedy můžeme zaměřit na způsob rozpoznání dvou velmi podobných programů napsaných v těchto jazycích. Plagiátoři si většinou vezmou cizí zdrojový kód a snaží se v něm provést různé změny tak, aby nebylo poznat, že je to zdrojový kód někoho jiného. Nejčastějšími změnami jsou:

- změna komentářů;
- změna odřádkování, odsazení, mezer;
- změna názvu proměnných a funkcí;
- změna datových typů;
- nahrazení nějaké konstrukce jinou ekvivalentní;
- přidávání nadbytečných příkazů;
- proházení příkazů.

Někteří plagiátoři u rozsáhlejších projektů mohou pouze okopírovat některé funkce. Z toho důvodu je vhodné rozdělit zdrojový kód na jednotlivé funkce a tyto funkce navzájem porovnávat s funkcemi ostatních zdrojových kódů. Na základě počtu okopírovaných funkcí pak rozhodneme, zda celý projekt lze považovat za plagiát či nikoliv. V následujících podkapitolách probereme jednotlivé změny ve zdrojových kódech, které plagiátoři provádí, a jak tyto změny odhalit.

4.1 Komentáře a bílé znaky

Komentáře nemají žádný vliv na výsledný program. Při překladu programu v jazyku C i C++ jsou hned zpočátku odstraněny preprocesorem a jsou nahrazeny mezerou. Proto můžeme v této aplikaci použít stejný postup. Avšak analýza komentářů se může hodit jako následný důkaz v případě, že budou dva zdrojové kódy označeny jako plagiáty. Například když ve dvou zdrojových kódech bude v komentáři stejná pravopisná chyba, je to jasným důkazem, že skutečně jsou tyto zdrojové kódy plagiáty. Z toho důvodu budeme komentáře také ukládat, ale zvlášť od funkčního kódu testovaných programů, a v případě, že kód dvou programů bude označen jako možný plagiát, bude možnost porovnat také komentáře a poté shodné či velmi podobné vypsat.

Bílé znaky slouží k oddělování identifikátorů (resp. klíčových slov), popřípadě i operátorů. Avšak ve zdrojovém kódu se může nacházet velké množství nadbytečných bílých znaků, které nemají na překlad žádný vliv. Z toho důvodu je vhodné všechny bílé znaky odstranit a nechat je pouze tam, kde je to bezpodmínečně nutné.

4.2 Přejmenování názvů proměnných a funkcí

Přejmenování názvů proměnných a funkcí se velmi často používá k maskování dvou stejných zdrojových kódů. Protože zdrojový kód rozdělíme po jednotlivých funkcí, které se budou následně porovnávat, přejmenování názvů funkcí nebude mít žádný vliv na hledání plagiátů.

Přejmenování názvů proměnných je však již potřeba řešit. Jedním z možných řešení je označit všechny proměnné jednotným označením.

4.3 Změna datových typů

Jak již bylo zmíněno ve druhé kapitole, v jazyku C i C++ existuje několik datových typů. Základní rozdělení je na celočíselné a racionální. Jednotlivé typy celočíselných datových typů se liší pouze počtem bitů, či znaménkem. U racionálních čísel se také liší pouze počtem bitů. Znak (datový typ *char*) lze brát jako celočíselný datový typ, neboť se také liší pouze počtem bitů. Datový typ *bool* lze také nahradit jakýmkoliv jiným celočíselným datovým typem. Tudíž je vhodné převést všechny datové typy na jeden společný. Co se týče znaménka u celočíselných datových typů a znaků, někdy výsledný program ovlivňuje, zda je datový typ znaménkový či nikoliv. Avšak ve většině případů na tom nezáleží, a proto lze znaménko také ignorovat.

4.4 Transformace konstrukcí

V jazycích C a C++ existuje spousta konstrukcí, které mají různý syntaktický zápis, avšak stejnou sémantiku. Například inkrementaci proměnné lze také provést několika způsoby. Jsou jimi:

```
x++;
++x;
x=x+1;
x+=1;
```

Možným řešením je převést tyto způsoby zápisu na jeden. Nejjednodušší je převést na nejdelší tvar, tzn. x++; ++x; či x+=1; převést na x=x+1; Totéž můžeme provést s dekrementací proměnné.

Deklarace proměnných a následné přiřazení hodnoty lze rovněž provést mnoha způsoby. Například:

```
int x=5, y=1;
```

lze zapsat také jako:

```
int x=5;
int y=1;
```

nebo:

```
int x;
int y;
x=5;
y=1;
```

nebo:

```
int x, y;
x=5;
y=1;
```

I v tomto případě je nejlepším řešením zkrácené zápisy převést na delší. To znamená oddělit od sebe deklaraci a přiřazení na dva samostatné příkazy a také operátor čárky rozdělit na samostatné příkazy. Tzn. například příkaz:

```
int x=5, y=1;
```

převést na posloupnost příkazů:

```
int x;
x=5;
int y;
y=1;
```

Další častou transformací je převod cyklu for na while, či naopak. Proto cyklus for můžeme převést na while, např.:

```
for (int i=0; i<10; i++)
{
  //telo cyklu
}
```

převedeme na:

```
int i=0;
while (i<10)
{
  //telo cyklu
  i++;
}
```

Dalšími typy transformací jsou u operátorů. Týká se to hlavně relačních operátorů v podmínkách. Například příkazy:

```
if (a>b)
  prikaz1;
else
  prikaz2;
```

lze převést na ekvivalentní příkazy:

```
if (a<=b)
  prikaz2;
else
  prikaz1;
```

Řešením může být označení všech relačních operátorů jedním stejným tokenem a také nebrat v úvahu pořadí příkazů. Ostatní operátory je vhodné také sdružit dohromady – aritmetické, bitové posuny, ostatní bitové, logické. Dále je vhodné ignorovat unární plus, mínus či logickou negaci.

Dva různé způsoby přístupu do struktury lze také převést na jeden typ, tj.:

```
(*struktura).polozka=0;
```

převedeme na:

```
struktura->polozka=0;
```

Všechny výše uvedené transformace se používají velmi často a je snadné je převést na jeden způsob zápisu bez vytváření složitých dynamických struktur, ale pouze přímým zápisem do výstupního řetězce (viz kapitola 5.2).

4.5 Proházení příkazů

Proházení příkazů, částí příkazů či rozdělení jednoho dlouhého příkazu na několik menších je velkým problémem odhalit. Možným řešením je počítat statistiky jednotlivých konstrukcí a následně jednotlivé statistiky porovnat. Tato metoda je navíc i velice rychlá. Avšak pro delší funkce může být tato metoda nepřesná (viz také kapitola 6.2.2). Proto je vhodné tuto metodu použít pouze jako doplňkovou, například pro zjištění, jestli má vůbec cenu porovnávat dvě funkce nějakou přesnější, avšak pomalejší metodou.

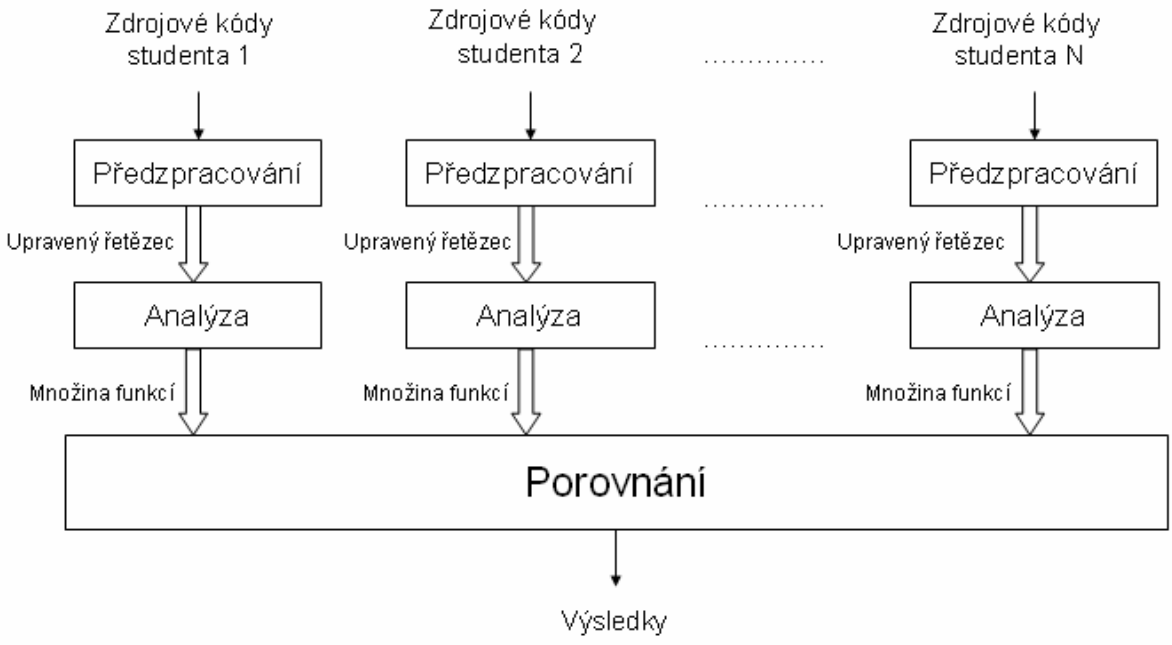
Pro přesné porovnání lze použít několik metod. Například obyčejné porovnání. Avšak to nám pouze řekne, zda se zdrojové kódy liší či nikoliv. Je však potřeba použít jiné metody, které nám řeknou, jak moc se liší. Pro tyto účely existují metody například Levenshteinova vzdálenost či Nejdelší společná podposloupnost. Avšak tyto metody nejsou moc odolné proti proházení jednotlivých příkazů.

Z toho důvodu je potřeba vytvořit vlastní porovnávací metodu, které nebude vadit proházení příkazů mezi sebou a také bude dostatečně přesná a rychlá. Navrženým řešením je rozdělit těla funkcí na jednotlivé příkazy a ty pak porovnávat mezi sebou. Tím se snadno vypořádáme s proházením příkazů mezi sebou. Podrobný návrh této porovnávací metody je popsán v kapitole 5.3.3.

5 Návrh řešení

V této kapitole se budeme zabývat návrhem řešení aplikace, která rozpozná velmi podobné programy napsané v programovacím jazyce C/C++.

Návrh aplikace je zobrazen na obrázku 5.1. Skládá se ze 3 hlavních částí – předzpracování, analýzy a porovnání. Všechny zdrojové kódy příslušející danému studentovi nejprve postupně projdou přes blok předzpracování, který z nich vytvoří jeden řetězec, jehož obsah tvoří upravené zdrojové kódy bez nadbytečných bílých znaků, s vyhodnocenými direktivami a s expandovanými makry. Tento řetězec se poté v dalším bloku analyzuje a vytvoří se z něj množina funkcí. Každá funkce bude obsahovat statistiky jednotlivých konstrukcí jazyka C/C++ v dané funkci, posloupnost těchto konstrukcí a také pole řetězců obsahující jednotlivé komentáře nacházející se v dané funkci. Toto se postupně provede pro zdrojové kódy všech studentů. Poté následuje porovnání, ve kterém se vzájemně porovnávají množiny funkcí mezi sebou. Na základě výsledků porovnání se rozhodne, které zdrojové kódy jsou plagiáty.



Obrázek 5.1: Schéma aplikace

5.1 Předzpracování

První částí aplikace je předzpracování zdrojových kódů, které je obdobou klasického předzpracování jazyka C pomocí preprocesoru. To znamená, že se odstraní všechny nadbytečné bílé znaky, načtou se a expandují makra, vloží se hlavičkové soubory, vyhodnotí se direktivy podmíněného překladu `#if`, `#elif`, `#ifdef`, `#ifndef` a `#else` a na základě toho se ignoruje kód, který se nebude provádět. Zdrojový kód se bude načítat přímo ze souborů a výsledek předzpracování se uloží do jednoho řetězce. Hlavním rozdílem oproti klasickému preprocesoru je ten, že bude možnost uchovávat komentáře.

Ve zdrojovém kódu můžeme narazit na tyto **bílé znaky:** mezera, nový řádek, návrat vozíku, tabelátor a nová stránka. Bílé znaky lze úplně v mnoha případech úplně vynechat. Ponechat se musí pouze mezi:

- dvěma identifikátory, např. `int x`;
- znaky `+` a `+`, např. `x=a+ +b`;
- znaky `-` a `-`, např. `x=a- -b`;
- znaky `&` a `&`, např. `x=a& &b`;
- znaky `/` a `*`, např. `x=*a/ *b`;
- znaky `>` a `>`, např. `vektor<vektor<int> >`.

Ve všech výše uvedených případech nahradíme všechny bílé znaky jednou mezerou.

Řádkový komentář rozpoznáme tak, že začíná posloupností znaků „//“. Jakmile narazíme na tuto posloupnost znaků, ignorujeme všechny následující znaky, dokud nenarazíme na konec řádku. Různé platformy však mají různé způsoby ukončení řádku. U platformy DOS a Windows je to CR+LF (\r\n), UNIX má pouze LF (\n) a MAC zase pouze CR (\r). Nutné je proto brát v úvahu všechny uvedené způsoby. Dále je nutné brát v úvahu i to, že řádkový komentář lze rozdělit na více řádků, pokud se vloží na konec řádku s komentářem zpětné lomítko. Překladač sice zahlásí varování, avšak zdrojový kód se normálně přeloží.

V případě uchovávání komentářů bude nutné převést řádkový komentář na blokový, neboť bílé znaky kromě potřebných mezer jsou ze zdrojového kódu odstraněny. Avšak uvnitř řádkového komentáře se může nacházet posloupnost znaků „*/“, která značí konec blokového komentáře. Proto je nutné rozdělit tyto znaky v komentáři mezerou. Posloupnosti bílých znaků uvnitř komentáře se nahradí pouze jednou mezerou.

Blokové komentáře zpracujeme podobně jako řádkové. Rozpoznají se tak, že začínají posloupností znaků „/*“. Následující znaky až do posloupnosti znaků „*/“ se buď ignorují nebo v případě uchovávání komentářů normálně uloží do výstupního řetězce. Při uchovávání komentářů se, stejně jako v případě řádkových komentářů, jednotlivé posloupnosti bílých znaků uvnitř komentářů nahradí jednou mezerou.

Direktivy procesoru se poznají dle toho, že začínají znakem „#“. Za ním následuje název direktivy. Je nutné uvažovat direktivy `#if`, `#elif`, `#else`, `#elif`, `#endif`, `#define`, `#ifdef`, `#ifndef`, `#undef` a `#include`. Zbývající direktivy `#error`, `#line` a `#pragma` můžeme ignorovat. Za názvem direktivy se nachází až do konce řádku další kód, který vykonává příslušná direktiva. Nachází-li se před koncem řádku zpětné lomítko, pokračuje kód vykonávaný direktivou na dalším řádku. V tomto kódu se mohou nacházet makra, která je nutné v případě direktiv `#if`, `#elif` a `#include` expandovat.

U direktiv `#if` a `#elif` je potřeba provést vyčíslení výrazu (viz kapitola 5.1.1), který se za touto direktivou nachází. Je-li výraz za direktivou `#if` pravdivý, tak následující kód ve zdrojovém kódu je normálně zpracováván, dokud se nenarazí na další direktivu podmíněného překladu (`#elif`, `#else`, `#endif`). Je potřeba také počítat s tím, že se direktivy `#if` mohou zanořovat, tudíž je nutné ukládat na zásobník, v jaké úrovni podmíněného překladu se nacházíme. V případě, že výraz za `#if` není pravdivý, následující kód se ignoruje, dokud se opět nenarazí na další direktivu podmíněného překladu. Další postup se provádí dle popisu v kapitole 2.2.3. U direktiv `#ifdef` a `#ifndef` se zjistí, zda je definováno makro, jehož název se nachází za direktivou. Je-li makro definováno, následující kód se bude normálně zpracovávat, dokud se nenarazí na další direktivu podmíněného překladu.

V případě, že makro není definováno, následující kód se ignoruje, dokud se nenarazí na další direktivu podmíněného překladu.

U direktivy `#define` se načte název makra, popřípadě parametry a obsah makra, a uloží se do seznamu maker. Při procházení zdrojového kódu pak musíme kontrolovat všechny identifikátory, zda nemají stejný název, jako některé z definovaných maker. V tom případě musíme tento identifikátor nahradit obsahem makra.

U direktivy `#include` se zjistí, zda soubor, který se bude vkládat, je systémový (název souboru se nachází v úhlových závorkách) či nikoliv (název souboru se nachází v uvozovkách). Systémový soubor se vynechá. V případě uživatelského souboru se kód nyní bude načítat z tohoto souboru, jakmile se dosáhne konce, bude se pokračovat v načítání předchozího souboru. Z důvodu nenačítání systémových souborů může nastat, že budou chybně vyčísleny výrazy u direktiv `#if` či `#elif`. A také to, že se neexpandují některá používaná makra – `true`, `false` apod. Z toho důvodu můžeme automaticky uložit do seznamu maker některá velmi často používaná makra. Například v případě `#include <stdbool.h>` se vloží do seznamu maker makra `true` (s nahrazující hodnotou 1) a `false` (s nahrazující hodnotou 0), v případě `#include <stdio.h>` se vloží do seznamu maker například makra `EXIT_SUCCESS` (s nahrazující hodnotou 0) a `EXIT_FAILURE` (s nahrazující hodnotou 1) apod.

Důležité je také zvlášť zpracovávat znaky, které se nacházejí mezi apostrofy či uvozovkami, protože tam se nezpracovávají žádné direktivy, komentáře ani bílé znaky. Všechny znaky mezi apostrofy či uvozovkami se normálně vkládají do výstupního řetězce.

Nyní máme popsaný preprocesor pro jazyk C. Preprocesor pro jazyk C++ bude vypadat stejně, akorát se před začátkem zpracovávání zdrojového kódu vloží do seznamu maker automaticky makra `true`, `false` a dále všechna makra, jejichž názvy a nahrazující hodnoty jsou uvedené v tabulce 3.1. Díky tomu se nahradí i alternativní reprezentace operátorů a analyzátor zdrojového kódu se již touto reprezentací nebude muset zabývat.

Zda se jedná o zdrojový kód jazyka C nebo C++ rozpoznáme podle přípony souboru, ze kterého budeme zdrojový kód načítat. Soubor pro jazyk C má nejčastěji příponu „`.c`“, soubor pro jazyk C++ má nejčastěji příponu „`.cpp`“, „`.c++`“ nebo „`.cc`“.

5.1.1 Vyčíslení výrazu

Ve výrazu se mohou nacházet pouze celočíselné konstanty, hodnoty v apostrofech, kulaté závorky a 23 druhů operátorů (všechny aritmetické, logické, relační a bitové operátory, podmíněný operátor). K těmto operátorům můžeme ještě přidat operátor čárka, který sice není povolen, avšak překladač u něj hlásí pouze varování a normálně jej provede. Proměnné se v direktivách berou jako hodnota nula. Pro provedení vyčíslení je nejprve potřeba převést převod výrazu z infixové notace do postfixové notace. **Postfixová notace**, nebo také obrácená polská notace, uvádí operátor dyadické operace za dvojici operandů.

Příklad postfixové notace: 3 5 7 + *

Postfixová notace má dvě významné vlastnosti:

- nepotřebuje závorky k potlačení vyšší priority operandů;
- pořadí operátorů vyjadřuje pořadí operací, kterými se výraz vyčísluje.

Převod lze provést například tímto způsobem [5, s. 62]:

1. Zpracovávej vstupní řetězec položku po položce zleva doprava a vytvářej postupně výstupní řetězec.
2. Je-li zpracovávanou položkou operand, přidej ho na konec vznikajícího výstupního řetězce.
3. Je-li zpracovávanou položkou levá závorka, vlož ji na vrchol zásobníku.
4. Je-li zpracovávanou položkou operátor, pak ho na vrchol zásobníku vlož v případě, že:
   - zásobník je prázdný,
   - na vrcholu zásobníku je levá závorka,
   - na vrcholu zásobníku je operátor s nižší prioritou.
5. Je-li na vrcholu zásobníku operátor s vyšší nebo shodnou prioritou, odstraň ho, vlož ho na konec výstupního řetězce a opakuj krok 4, až se ti podaří operátor vložit na vrchol.
6. Je-li zpracovávanou položkou pravá závorka, odebírej z vrcholu položky a dávej je na konec výstupního řetězce až narazíš na levou závorku. Tím je pár závorek zpracován.
7. Je-li zpracovávanou položkou omezovač ‚=‘, pak postupně odstraňuj prvky z vrcholu zásobníku a přidávej je na konec řetězce, až se zásobník zcela vyprázdní. Na konec se přidá rovnítko, jako omezovač výrazu.

Vyčíslení výrazu v postfixové notaci můžeme provést tímto způsobem [5, s. 61]:

1. Zpracovávej řetězec zleva doprava.
2. Je-li zpracovávaným prvkem operand, vlož ho do zásobníku.
3. Je-li zpracovávaným prvkem operátor, vyjmi ze zásobníku tolik operandů, kolikanární je operátor (pro dyadické operátory dva operandy), proveď danou operaci a výsledek ulož na vrchol zásobníku.
4. Je-li zpracovávaným prvkem omezovač ‚=‘, je výsledek na vrcholu zásobníku.

Uvedené dva algoritmy můžeme optimalizovat tak, že budeme mít dva zásobníky, jeden na operátory a závorky, druhý na operandy. Pokud se narazí ve vstupním výrazu na operand, vloží se na zásobník operandů. Narazí-li se na levou závorku, vloží se na druhý zásobník. Narazí-li se na operátor či pravou závorku, postupuje se stejně, jako v případě kroků 4, 5, 6 v algoritmu převodu do postfixové notace, avšak místo ukládání do výstupního řetězce se s konkrétním operátorem provede vyčíslení (viz třetí krok v algoritmu vyčíslování). Omezovač je v našem případě konec výrazu.

5.2 Analýza zdrojového kódu

Další část aplikace tvoří analýza zdrojového kódu. Jak již bylo uvedeno na začátku páté kapitoly, tato část má za úkol upravený zdrojový kód analyzovat a rozdělit na jednotlivé funkce, které budou později porovnávány. U každé funkce budeme uchovávat jméno funkce, tokenizovaný obsah funkce, statistiky konstrukcí, počet příkazů, počet tokenů a modul, ve kterém se daná funkce nachází. Pro porovnávání komentářů je nutné ještě ukládat komentáře (pole řetězců) a jejich počet. Struktura uchovávající tyto položky je vlastně porovnávací struktura každé funkce. Pro každého studenta bude počet těchto porovnávacích struktur roven počtu všech funkcí v celém studentově projektu. Množina těchto porovnávacích struktur u každého studenta bude implementována pomocí pole. Jednosměrně či dvousměrně vázaný seznam není potřeba, neboť po vytvoření se porovnávací struktury jednotlivých funkcí nemažou, paměť se uvolní až po porovnávání.

Analýza zdrojového kódu je částečně obdoba klasické syntaktické analýzy. Analýza se pro jazyk C bude lišit od programovacího jazyka C++, neboť jazyk C++ přináší několik zásadních změn.

Proto pro každý jazyk implementujeme vlastní funkci. Syntaktická analýza využívá lexikální analýzu, která vrací následující token ze zdrojového kódu. Lexikální analýzy budou využívat i obě naše funkce. Vzhledem k tomu, že lexikální analýza pro jazyk C je velmi podobná i pro jazyk C++, budeme ji implementovat do stejné funkce a pouze parametrem funkce rozlišíme, zda se jedná o lexikální analýzu pro jazyk C nebo C++.

5.2.1 Lexikální analýza

Lexikální analýza je činnost, kterou provádí tzv. lexikální analyzátor (scanner). Lexikální analyzátor rozdělí vstupní posloupnost znaků na lexémy – lexikální jednotky (např. identifikátory, čísla, klíčová slova, operátory, …). Tyto lexémy jsou reprezentovány ve formě tokenů. V praxi je lexikální analyzátor realizován pomocí konečného automatu [17].

V naší aplikaci budeme lexikální analyzátor implementovat podobně jako lexikální analyzátor jazyka C nebo C++, avšak lze jej zjednodušit, neboť nekonstruujeme překladač. Některé operátory můžeme sdružit a označit stejným tokenem. To samé můžeme udělat i pro některá klíčová slova, která se zpracovávají stejně (viz níže).

Při zpracování identifikátoru nejprve zjistíme, zda se jedná o klíčové slovo. **Klíčová slova** můžeme rozdělit podle toho, zda za daným klíčovým slovem následuje otevřená kulatá závorka nebo něco jiného. Tímto rozdělením klíčových slov na dvě skupiny urychlíme jejich vyhledávání. Existují však klíčová slova, která někdy mohou mít za sebou otevřenou kulatou závorku, a jindy zase nemusí. V jazyce C jsou to pouze klíčová slova `case` a `return`, u jazyka C++ ještě `asm` a `new`. U jazyka C++ můžeme rozdělit klíčová slova ještě na další kategorii a to klíčová slova mající za sebou otevřenou úhlovou závorku. Sem patří klíčová slova pro přetypování a klíčové slovo `template`. Na první pohled by se mohlo zdát, že urychlení moc velké nebude, neboť klíčových slov, za kterými musí vždy být pouze otevřená kulatá závorka je v jazyce C pouze 6. Avšak např. při volání funkce (za identifikátorem se vždy musí nacházet otevřená kulatá závorka) se projdou pouze klíčová slova, za kterými může být otevřená kulatá závorka, a už se nemusí procházet všechna ostatní klíčová slova, čímž dojde k urychlení. Musí se však počítat, že při zachovávání komentářů se může za identifikátorem nacházet komentář. Tento jev by neměl nastávat příliš často, neboť aby byl komentář čitelný, dává se většinou až za příkaz. Avšak nastat to může, proto pokud se za identifikátorem nachází komentář, přeskočíme tento komentář, podíváme se, jaký symbol se nachází za komentářem a poté se vrátíme před komentář, aby i tento komentář mohl být v dalším kroku případně zpracován. Modifikátory (`auto`, `extern`, `inline`, `register`, `static`, `volatile`) označíme jedním společným tokenem. Modifikátory nemají zásadní vliv při hledání plagiátů, a proto token označující modifikátor nebude brán u analýzy konstrukcí v potaz (viz další kapitola). Všechna klíčová slova označující datové typy spolu s klíčovým slovem `const` označíme jedním společným tokenem. Všechna ostatní klíčová slova označíme každé zvlášť svým vlastním tokenem (např. *token_if*, *token_while* apod). U klíčových slov jazyka C++ označíme klíčová slova `explicit`, `export`, `friend`, `mutable` a `virtual` jako modifikátory, neboť se budou ignorovat. Také sdružíme klíčová slova pro přetypování (`const_cast`, `dynamic_cast`, `reinterpret_cast`, `static_cast`) a klíčová slova pro přístup (`public`, `private`, `protected`). Pokud se nebude jednat o klíčové slovo a za identifikátorem bude otevřená kulatá závorka, jedná se zřejmě o deklaraci či volání funkce. Proto vrátíme token značící *identifikátor funkce*. V případě, že identifikátor nebude klíčové slovo a ani za ním nebude následovat otevřená kulatá závorka, vrátí se token značící *identifikátor*. Název identifikátoru (i identifikátoru funkce) bude uložen v pomocném řetězci, který se vrací společně s tokenem. V případě klíčového slova `operator` (klíčové slovo jazyka C++ pro přetěžování

operátorů) načteme ještě operátor nacházející se za tímto klíčovým slovem a uložíme jej do pomocného řetězce, který se vrací společně s tokenem. Díky tomu se budou moct hlavičky operátorových funkcí zpracovávat v analýze konstrukcí stejným způsobem, jako běžné funkce.

Všechna **čísla** označíme pouze jedním tokenem. A to jak osmičková, desítková a šestnáctková, tak i racionální. Znak(y) v apostrofech také označíme tokenem, který značí číslo. Je to z toho důvodu, že např. `0101` lze také zapsat jako `65` nebo `0x41` nebo `65.0` či `'A'`. Spolu s číslem se také načte exponent či příznaky za číslem.

Všechny **relační operátory** u jazyka C označíme stejným tokenem. Pro jazyk C++ je však potřeba otevřenou a uzavřenou úhlovou závorku označit jinými tokeny, neboť kromě operátorů menší než (resp. větší než) se používají u šablon a datových kontejnerů.

Bitové posuny (`<<`, `>>`) označíme stejným tokenem. Ampersand může kromě bitového and znamenat také referenci na proměnnou či funkci, proto jej označíme zvláštním tokenem. Zbylé **bitové operátory** již označíme společným tokenem. V případě složených bitových operátorů sdružíme složené operátory bitových posunů a ostatní složené bitové operátory.

Aritmetické operátory `+` a `−` označíme společným tokenem. V analýze konstrukcí pak u nich budeme rozlišovat, zda se jedná o unární či binární operátor, proto je nesdružujeme s ostatními aritmetickými operátory. Operátor `*` může kromě násobení znamenat také dereferenci, proto jej také označíme zvláštním tokenem. Zbylé dva aritmetické operátory dělení a zbytek po dělení označíme stejným tokenem. Složené aritmetické operátory `+=`, `−=`, `*=`, `/=` a `%=` již můžeme označit společným tokenem, neboť všechny tyto operátory označují binární aritmetickou operaci.

Logické and a **or** označíme stejným tokenem. Logickou negaci označíme jiným tokenem, který se však v analýze konstrukcí bude ignorovat, neboť negace se dá provést také prohozením příkazů, prohozením výrazů, změnou relačního operátoru apod.

Komentáře budeme ukládat do pomocného řetězce, který bude vracet spolu s tokenem lexikální analyzátor. Budou-li následovat dva komentáře bezprostředně za sebou, automaticky se spojí do jednoho. Neuzavřenost blokového komentáře již byla testována v předzpracování, tudíž můžeme předpokládat, že všechny komentáře budou správně uzavřeny.

Řetězec označíme jedním tokenem, obsah nacházející se uvnitř řetězce budeme ignorovat. Jednotlivé závorky (kulaté, hranaté, složené) se označí každá zvláštním tokenem, stejně tak středník, čárka, otazník a dvojtečka. Tečka a šipka (`->`) se označí jedním tokenem, značícím přístup ke členu struktury (resp. unionu, třídy). U jazyka C++ ještě označíme operátor `::` jako kvalifikátor.

5.2.2 Analýza konstrukcí pro jazyk C

Pro naši aplikaci není potřeba vytvářet syntaktický analyzátor, který se používá u jazyka C pro vytvoření syntaktického stromu. Stačí nám pouze detekovat jednotlivé konstrukce jazyka C a ty uložit do výstupního řetězce a zároveň vytvářet statistiky o počtu různých konstrukcí. Při ukládání těchto konstrukcí budeme zároveň provádět transformace popsané v kapitole 4.4. Jak již bylo zmíněno výše, při této analýze budeme využívat lexikální analýzy, která nám vždy na požádání vrátí následující token.

Analýzu konstrukcí lze rozdělit na dvě části. Jedna část bude zpracovávat globální konstrukce, druhá část bude zpracovávat konstrukce uvnitř funkcí. Mimo funkce se v jazyce C mohou nacházet deklarace globálních proměnných a jejich inicializace, definice nových typů, struktur a výčtových typů a také deklarace funkcí. Deklaraci proměnné poznáme podle po sobě jdoucích tokenů „`TYP ID`“. Protože nenačítáme systémové hlavičky, může nastat, že se v testovaném zdrojovém kódu bude nacházet datový typ, u kterého nepoznáme, že se jedná o typ. V tom případě dojde k posloupnosti

tokenů „`ID ID`“, kterou vyhodnotíme tak, že první `ID` je datový typ a jedná se tedy také o deklaraci proměnné. Na globální úrovni však není potřeba konstrukce zaznamenávat. Proto deklarace globálních proměnných, výčtových typů, struktur a nově definovaných typů budeme pouze detekovat, ale nebudeme je zaznamenávat. Důležitou konstrukcí je však deklarace funkce. Tu rozpoznáme tak, že budou po sobě následovat tokeny „`TYP ID(`“. V jazyce C není povinná návratová hodnota, tudíž jako deklaraci funkce můžeme považovat i posloupnost tokenů „`ID(`“. Název `ID` si uložíme do nějaké pomocné proměnné. Následně se budou načítat tokeny tak dlouho, dokud se nenarazí na odpovídající pravou kulatou závorku. Nyní pokud se za závorkou nachází středník nebo operátor čárka, jedná se pouze o prototyp funkce. V tom případě pokračujeme dál ve zpracovávání globálních konstrukcí. Pokud se však za parametry nachází otevřená složená závorka, jedná se o definici funkce. Proto přidáme do seznamu funkcí další položku, u které inicializujeme hodnoty. Poté můžeme přejít ke zpracovávání konstrukcí uvnitř funkce.

Konstrukce uvnitř funkcí budeme ukládat jak do statistik funkce, tak i do řetězce obsahu funkce. Zpracování obsahu funkce budeme reprezentovat lokálním stavovým automatem. Zpátky do zpracovávání globálních konstrukcí se přejde v případě, že narazíme na odpovídající ukončující složenou závorku funkce. Zpracovávání jednotlivých konstrukcí bude probíhat podle postupů, které jsou uvedeny ve čtvrté kapitole. To znamená, že deklarace několika proměnných pomocí operátoru čárky rozdělíme na samostatné příkazy deklarací jednotlivých proměnných, deklaraci a následné přiřazení proměnné nahradíme samostatnými dvěma ekvivalentními příkazy, to samé i pro jejich kombinace. Inkrementaci a dekrementaci proměnné převedeme na posloupnost tokenů: *přiřazení proměnné*, *použití proměnné*, *aritmetický operátor* a *číslo*. U operátorů `+`, `-`, `*` a `&` zjistíme, zda se jedná o unární či binární operátor. O binární operátor se jedná, pokud předchozí token byl identifikátor, číslo nebo pravá kulatá závorka. V ostatních případech je to unární operátor, který ignorujeme. V případě binárního operátoru vložíme do výstupního řetězce token značící *aritmetický operátor* resp. *bitový operátor*. Proměnnou, za kterou následuje složený operátor aritmetické operace a přiřazení (`+=`, `-=`, `*=`, `/=`, `%=`), nahradíme posloupností tokenů *přiřazení proměnné*, *použití proměnné* a *aritmetický operátor*. Podobně to provedeme i pro proměnnou, za kterou následuje složený operátor přiřazení s bitovou operací (`&=`, `|=`, `^=`) či bitovým posunem (`<<=`, `>>=`). Změnou bude pouze to, že místo tokenu označující *aritmetický operátor* vložíme do výsledného řetězce token označující *bitový operátor* nebo *bitový posun*. Středník ukládáme jako token označující *oddělovač*. V případě více středníků za sebou se vloží pouze jeden oddělovač, ostatní středníky se ignorují.

Přístup ke členu struktury `struktura->clen`, `struktura.clen` označíme pouze jedním tokenem, který značí *přístup ke členu struktury*. V případě vícenásobného přístupu k prvku struktury vložíme tento token tolikrát, kolikrát je ve výrazu zahrnut operátor *tečka* nebo *šipka*. V případě, že nějaký člen ve struktuře je pole (např. `struktura.pole[i].hodnota`), uložíme do řetězce obsahu funkce posloupnost tokenů *přístup k prvku struktury*, *pole*, *použití proměnné* a *přístup k prvku struktury*.

U příkazu `if` vložíme do řetězce obsahu funkce token označující *větvení if* a poté normálně zpracujeme výraz v závorkách příkazu. Následně vložíme oddělovač příkazů. To je z toho důvodu, abychom oddělili konstrukci `if` od příkazu, který se provede v případě splnění podmínky v `if`. Stejný postup provedeme i v případě cyklu `while`, s tím, že místo tokenu označující konstrukci *if*, vložíme do řetězce token označující *cyklus*. V případě cyklu `for` se nejprve zpracuje první výraz sloužící k deklaraci či přiřazení proměnných. Poté se vloží token označující *cyklus* a zpracuje se druhý výraz (podmínka). Nakonec se zpracuje třetí výraz v konstrukci cyklu `for`. Tímto převedeme cyklus `for` na cyklus `while`. Třetí výraz se však vloží na začátek těla cyklu a správně se by měl

vložit spíše až na konec. Nemusí to však platit vždy, navíc u Vlastní porovnávací metody nezáleží na pořadí příkazu. Přepínač `switch` zpracujeme tak, že do řetězce obsahu funkce vložíme token označující *přepínač* a poté normálně zpracujeme výraz v závorkách příkazu, následně vložíme oddělovač příkazů. U příkazu `case` vložíme do výstupního řetězce pouze token označující *návěští*, výraz za `case` až po dvojtečku ignorujeme.

Deklarace struktur, unionů, výčtových typů a nově definovaných typů ignorujeme, neukládáme je do obsahu funkce ani do statistik. Je to z toho důvodu, že tyto deklarace se většinou nedávají do těla funkce, ale globálně. A to i v tom případě, že se daná deklarace objevuje pouze v jedné funkci. Pokud by je jeden student přemístil do těla funkce, vzniklo by množství tokenů navíc. Díky ignorování se to neprojeví. Tyto deklarace navíc nemají příliš vypovídající hodnotu o zdrojovém kódu a často bývají podobné.

Přetypování v jazyce C poznáme podle toho, že je v kulatých závorkách uveden datový typ a za ním následuje identifikátor. Avšak pokud bude v kulatých závorkách uveden neznámý datový typ, může nastat při rozpoznávání přetypování problém. Například posloupnost tokenů `(id) *x` může znamenat přetypování obsahu na adrese `x` na datový typ `id` nebo také součin proměnných `id` a `x`. Z toho důvodu nebudeme přetypování označovat žádným vlastním tokenem a v případě přetypování na standardní datový typ vložíme do výstupního řetězce pouze token označující *použití proměnné*.

Komentáře se budou ukládat do pole řetězců. Všechny modifikátory můžeme ignorovat, neboť pouze určují, kam se mají proměnné uložit. Nemají tedy zásadní vliv na funkci programu. Kulaté a složené závorky se do výstupního řetězce také neukládají.

5.2.3 Analýza konstrukcí pro jazyk C++

Analýza konstrukcí pro jazyk C++ je z části podobná analýze konstrukcí pro jazyk C. Hlavním rozdílem je, že uvnitř každé funkce může být definována třída, ve které se mohou nacházet další funkce. Funkce se také mohou nacházet uvnitř struktur. Tudíž, pokud uvnitř funkce narazíme na třídu nebo strukturu, přejdeme zpracovávání globálních konstrukcí. Do zásobníku si však uložíme všechny informace, které budou potřebné pro pokračování ve zpracovávání příslušné funkce, jakmile se narazí na odpovídající uzavírající složenou závorku, která bude uzavírat strukturu či třídu.

Třídy mohou obsahovat několik konstruktorů. V nich se pouze inicializují některé proměnné uvnitř třídy. Z toho důvodu je nemá cenu porovnávat s ostatními funkcemi, proto konstruktory budeme ignorovat. To samé provedeme i pro destruktory, které pouze uvolňují paměť.

Rozdílem oproti jazyku C je také to, že funkce nemusí být jen globální, ale může také být členskou metodou dané třídy. Hlavička funkce pak vypadá: `int mojeTrida::vratCislo()`. Konstrukci `int mojeTrida` vyhodnotíme normálně jako deklaraci proměnné. Poté, když narazíme na kvalifikátor (`::`), přejdeme do stavu, ve kterém budeme tak dlouho, dokud nenarazíme na token typu `id(`. To z toho důvodu, že metoda může být ve třídě, která je podtřídou jiné třídy, tudíž hlavička je například ve tvaru: `int nadTrida::Trida::dalsiTrida::vratCislo()`. Důležité je také rozpoznat, zda se jedná o metodu nebo konstruktor, neboť konstruktory se ignorují. To poznáme snadno, neboť v jazyce C++ musí mít každá metoda návratový typ, ale konstruktor návratový typ nemá. Takže v případě posloupnosti tokenů `id::` přejdeme do stavu, ve kterém čekáme na token typu `id(`. A poté přejdeme do stavu zpracování konstruktoru.

V jazyce C++ lze použít přetypování známé z jazyka C nebo také nové způsoby přetypování (viz kapitola 3.4.6). Proto nové způsoby přetypování převedeme na klasický způsob přetypování v jazyce C. Tudíž načteme celou konstrukce nového způsobu přetypování a do řetězce obsahu funkce

vložíme pouze token označující *použití proměnné*. Tím dostaneme stejný výsledek jako v případě klasického přetypování.

V jazyce C++ také existují na rozdíl od jazyka C šablony. Velmi často se využívá různých datových kontejnerů, které jsou součástí standardních šablon jazyka C++ (viz kapitola 3.4.4). Pokud se za identifikátorem nachází levá úhlová závorka, může se jednat o *datový kontejner* nebo o operátor *menší než*. O použití šablony se jedná, pokud za levou úhlovou závorkou následuje datový typ (popřípadě více datových typů oddělených čárkou) a dále pravá úhlová závorka. Datovým typem může být například opět datový kontejner, takže i s tím je potřeba počítat. V ostatních případech se jedná o operátor *menší než*. Datový kontejner zpracujeme tak, že celý výraz až po ukončující pravou úhlovou závorku bereme jako datový typ. Do řetězce obsahu funkce tedy nic neukládáme. Po ukončující úhlové závorce následuje identifikátor, který se zpracuje tak, že do řetězce obsahu funkce vložíme token značící deklaraci proměnné.

5.3 Porovnání

Poslední část aplikace tvoří porovnání jednotlivých funkcí všech studentů navzájem mezi sebou. Tato fáze bude jedna z nejdelších, neboť například pro 500 zdrojových kódů se musí provést 124750 porovnání těchto zdrojových kódů (viz vzorec 5.1).

$$\binom{n}{k} = \binom{500}{2} = \frac{500 * 499}{2} = 124750$$

Vzorec 5.1: Výpočet počtu porovnání

Dále musíme vzít v úvahu, že se musí u každého zdrojového kódu porovnat navzájem každá funkce. Pokud by oba zdrojové kódy měly například 20 funkcí, musí se provést dalších 400 porovnání. Takže pro 500 zdrojových kódů, z nichž každý bude mít průměrně 20 funkcí, je potřeba provést 49900000 porovnání jednotlivých funkcí. Z toho důvodu je nutné, aby porovnávání funkcí bylo co možno nejrychlejší. Proto je nutné provést nějakou optimalizaci.

Nemá cenu porovnávat dvě funkce, z nichž například jedna obsahuje 10 tokenů a druhá 50 tokenů. Je jasné, že tyto funkce plagiáty určitě nebudou. Proto se nejprve vypočítá rozdíl počtu tokenů mezi jednotlivými funkcemi a pokud bude větší, než stanovená mez, porovnávání těchto funkcí se přeskočí. Protože každá funkce může být různě dlouhá, je nevhodné volit jako mez konkrétní číslo, které by bylo stejné pro všechny funkce. Lepší je si zvolit, o kolik procent se může lišit velikost jedné funkce od druhé. Jakého dosáhneme průměrného urychlení použitím této optimalizace, je uvedeno v kapitole 6.2.3.

Nyní může přijít na řadu porovnání funkcí pomocí statistické metody. Tato metoda je velmi rychlá, neboť se vždy porovnává pouze pole čísel, ve kterém jsou uloženy počty jednotlivých konstrukcí. Podle vzorce 5.2 vypočteme celkový rozdíl mezi statistikami. Dle vypočteného rozdílu rozhodneme, zda mohou být funkce považovány za plagiát. Jako mez zde opět zvolíme minimální procentuální shodu mezi statistikami funkcí.

$$\sum_{i=0}^{n} abs(pole1[i] - pole2[i])$$

Vzorec 5.2: Výpočet rozdílu statistik mezi dvěma funkcemi

Po rozhodnutí, zda mohou být funkce plagiáty, se může pokračovat v porovnávání dalších funkcí. Avšak statistická metoda nemusí být v některých případech příliš přesná a i zcela odlišné funkce může označit jako plagiát. Z toho důvodu je lepší ještě detailně porovnat těla jednotlivých funkcí. K tomuto účelu je možný výběr ze tří různých metod. Popisy jednotlivých metod se nachází v kapitolách 5.3.1, 5.3.2 a 5.3.3. Jednotlivé metody buď vrátí počet shodných tokenů (Nejdelší společná podposloupnost, Vlastní porovnávací metoda) nebo počet rozdílných tokenů (Levenshteinova vzdálenost). Z těchto hodnot vypočteme, z kolika procent jsou jednotlivé funkce shodné. Bude-li shoda větší, než je stanovená mez, označíme porovnávané funkce jako plagiáty.

Po kontrole všech funkcí se musí dále rozhodnout, zda dva zdrojové kódy lze celkově považovat za plagiát. To rozhodneme podle počtu shodných funkcí. Avšak každá funkce může být různě dlouhá, tudíž při rozhodování bude hrát roli i délka funkce.

Kromě testování všech projektů mezi sebou (N:N) se bude moct porovnávat i jeden projekt s ostatními (1:N). Toto porovnávání se může hodit v případě dodatečné kontroly plagiátů, kdy nějaký student odevzdá projekt dodatečně, popřípadě v původním odevzdaném projektu má nějakou chybu (neuzavřený blokový komentář, neuzavřené uvozovky apod.), kvůli které nepůjde daný projekt přeložit a také ani zkontrolovat touto aplikací.

Další možností je testovat pouze jeden konkrétní zdrojový kód v případě, že projekt se skládá z několika nezávislých zdrojových kódů (například domácí úlohy v předmětu Algoritmy).

5.3.1 Levenshteinova vzdálenost

Levenshteinova vzdálenost mezi dvěma řetězci je dána minimálním počtem operací potřebných k transformaci jednoho řetězce na druhý. Operacemi se rozumí vložení, smazání nebo nahrazení jednoho znaku. Například Levenshteinova vzdálenost mezi řetězcem „kitten“ a řetězcem „sitting“ je 3 (‘s‘ se nahradilo ‘k‘, ‘i‘ se nahradilo ‘e‘ a ‘g‘ bylo přidáno na konec). Levenshteinova vzdálenost tedy vyjadřuje podobnost (resp. rozdílnost) dvou řetězců. Občas se nazývá „změnová vzdálenost“ nebo také „editační vzdálenost“.

Pro výpočet Levenshteinovy vzdálenosti se používá matice velikosti (n+1) × (m+1), kde n je délka prvního řetězce a m je délka druhého řetězce. První řádek matice se inicializuje hodnotami 1, 2, 3,…m a první sloupec také hodnotami 1, 2, 3, … n. Poté se vyplní ostatní políčka matice tak, že se hodnota políčka matice:

```
d[i, j] = minimum (d[i-1, j] + 1, d[i, 1-1] + 1, d[i-1, j-1] + cost)
```

kde *cost* má hodnotu 0, pokud je znak v prvním řetězci na pozici *i* shodný se znakem ve druhém řetězci na pozici *j*, v případě neshody je *cost* rovna 1 [15].

Algoritmus má kvadratickou časovou složitost O(n×m). Je však možné jej částečně urychlit, pokud přeskočíme shodné znaky na začátku a na konci obou řetězců, čímž u shodných řetězců klesne kvadratická časová složitost na lineární.

5.3.2 Nejdelší společná podposloupnost

U této metody máme dvě posloupnosti čísel a chceme najít jejich Nejdelší společnou podposloupnost, tedy takovou posloupnost, kterou můžeme získat z A i B odstraněním některých prvků. Například pro posloupnosti:

A=2 3 3 1 2 3 2 2 3 1 1 2
B=3 2 2 1 3 1 2 2 3 3 1 2 2 3

je jednou z Nejdelších společných podposloupností tato posloupnost:

C=2 3 1 2 2 3 1 2

Více o Nejdelší společné podposloupnosti se můžete dočíst na [14] nebo v bakalářské práci Matěje Kačice [7], který tuto metodu používá ve své aplikaci pro porovnávání těl funkcí. Tuto metodu opět můžeme částečně urychlit přeskočením shodných znaků na začátku a na konci obou řetězců.

5.3.3 Vlastní porovnávací metoda

Porovnávací metoda by se měla vypořádat s proházením příkazů a zároveň by měla být rychlá. Na základě těchto požadavků jsem vymyslel metodu, jejíž princip je následující. Nejprve vytvoříme dva seznamy, které budou obsahovat příkazy, u kterých došlo při porovnávání k neshodě. Každá funkce bude mít svůj vlastní seznam, který bude na začátku prázdný. Poté postupně budeme brát příkazy z porovnávaných funkcí. Pokud budou příkazy shodné, posuneme se na další příkaz v obou funkcích. V opačném případě oba příkazy uložíme do příslušných seznamů příkazů obsahujících neshodné příkazy. Poté porovnáme další příkazy. Budou-li shodné, opět se přesuneme na další, v případě neshody se nejprve pokusíme porovnat příkaz jedné funkce s příkazy uloženými v seznamu neshodných příkazů druhé funkce. V případě shody se u první funkce posuneme na další příkaz a zrušíme tento příkaz ze seznamu neshodných příkazů u druhé funkce. V případě neshody u všech příkazů v seznamu provedeme stejný postup pro příkaz druhé porovnávané funkce, který se pokusíme porovnat se všemi uloženými příkazy v seznamu neshodných příkazů první funkce. Pokud opět nedojde ke shodě s žádným příkazem v seznamu, uložíme oba příkazy do příslušných seznamů a posuneme se na další příkaz v obou porovnávaných funkcích. V případě, že v jedné z funkcí dojdeme na konec a ve druhé budou stále příkazy, porovnáváme všechny tyto příkazy pouze s příkazy uloženými v seznamu neshodných příkazů první funkce. Po projití všech příkazů v obou porovnávaných funkcí nám v seznamech zůstanou příkazy, které jsou neshodné. Budou-li oba seznamy prázdné, jsou obě funkce shodné, maximálně měly proházené příkazy. Bude-li prázdný pouze jeden seznam, pravděpodobně to bude značit, že jedna z funkcí obsahuje pouze nějaké příkazy navíc. Příkazy však mohou být různě dlouhé, proto je vhodnější, aby metoda vracela počet shodných tokenů. Celkový počet shodných tokenů vypočteme tak, že při každé shodě příkazů přičteme k výsledku počet tokenů obsažených v tomto příkazu.

Na první pohled se může zdát, že tato metoda bude mít velkou časovou složitost. Při bližším pohledu však zjistíme, že pokud budou všechny příkazy shodné a ve stejném pořadí, časová složitost bude lineární, v případě prohození všech příkazů, které se budou lišit vždy až v posledním tokenu, bude složitost kvadratická $O(n \times m)$. Kvadratické časové složitosti se však ve většině případů ani zdaleka nedosáhne (viz kapitola 6.2.1).

Navržená metoda lze ještě implementačně optimalizovat. Místo kopírování celého příkazu do seznamu neshodných příkazů stačí do seznamu ukládat pouze číslo příkazu v poli řetězců. Tudíž seznam lze implementovat jako pole čísel. Při analýze zdrojového kódu zjistíme, kolik příkazů se v dané funkci nachází. Tím pádem i předem víme maximální velikost seznamu, která nikdy nepřesáhne počet příkazů. Díky tomu můžeme oběma seznamům přidělit paměť již na začátku porovnávání a vyhneme se tím pomalému dynamickému přidělování paměti. Formálně zapíšeme algoritmus této metody tímto způsobem:

```
int myCompare(char ** s1, char ** s2, int cnt_com_s1, int cnt_com_s2)
{
  int list_s1[cnt_com_s1]; //Seznam neshodnych prikazu z pole retezcu s1
  int list_s2[cnt_com_s2]; //Seznam neshodnych prikazu z pole retezcu s2
  int index_list_s1=0;     //Index seznamu s1
  int index_list_s2=0;     //Index seznamu s2
  int i=0;                 //Index prikazu v poli retezcu s1
  int j=0;                 //Index prikazu v poli retezcu s2
  int same=0;              //Pocet shodnych tokenu
  bool not_end=true;       //Priznak konce cyklu

  while (not_end)
  {
    bool equal=false;
    if (i<cnt_com_s1 && j<cnt_com_s2)
    {
      if (strcmp(s1[i], s2[j])==0)
      {
        equal=true;
        same+=strlen(s1[i]);
        i++;
        j++;
      }
    }
    if (!equal && i<cnt_com_s1 && index_list_s2>0)
    {
      for (int ii=0;ii<index_list_s2 && !equal; ii++)
      {
        if (strcmp(s1[i], s2[list_s2[ii]])==0)
        {
          if (index_list_s2>0)
             list_s2[ii]=list_s2[--index_list_s2];
          else
             --index_list_s2;
          equal=true;
          same+=strlen(s1[i]);
          i++;
        }
      }
    }
    if (!equal && j<cnt_com_s2 && index_list_s1>0)
    {
      for (int ii=0;ii<index_list_s1 && !equal; ii++)
      {
        if (strcmp(s1[list_s1[ii]], s2[j])==0)
        {
          if (index_list_s1>0)
            list_s1[ii]=list_s1[--index_list_s1];
          else
            --index_list_s1;
          equal=true;
          same+=strlen(s1[j]);
          j++;
        }
      }
    }
```

```
    if (!equal)
    {
      if(i<cnt_com_s1)
        list_s1[index_list_s1++]=i++;
      if(j<cnt_com_s2)
        list_s2[index_list_s2++]=j++;
    }
    if ((i>=cnt_com_s1)&&(j>=cnt_com_s2))
      not_end=false;
  }
  return same;
}
```

Předchozí dvě metody (Nejdelší společná podposloupnost a Levenshteinova vzdálenost) však pracují se dvěma řetězci, nikoliv se dvěma poli řetězců. Všechny příkazy jsou uloženy v jednom řetězci a navzájem jsou odděleny tokenem značícím oddělovač. Z toho důvodu provedeme úpravu tohoto algoritmu, která bude spočívat v tom, že pro porovnání dvou příkazů nepoužijeme vestavěnou funkci jazyka C (strcmp), ale postupně porovnáme jednotlivé tokeny v obou řetězcích od určité pozice (začátek daného příkazu) po token oddělovače (konec daného příkazu). Indexy i a j uvedené v algoritmu výše budou v tomto případě značit aktuální pozici v řetězcích a do seznamů neshodných příkazů se budou ukládat pozice začátku příkazů v řetězci, které jsou neshodné.

5.3.4 Porovnání komentářů a jejich výpis

Pro porovnávání a výpis podobných komentářů použijeme metodu, která bude založena na Vlastní porovnávací metodě kombinovanou s Nejdelší společnou podposloupností. Pomocí upravené Vlastní porovnávací metody postupně nalezneme úplně shodné komentáře a vypíšeme je. Poté nám v seznamech neshodných komentářů zůstanou ostatní komentáře. Z prvního seznamu postupně vezmeme každý komentář a ten postupně porovnáme se všemi ostatními komentáři ve druhém seznamu pomocí metody Nejdelší společné podposloupnosti. Vybereme komentář, který se nejvíce shoduje, a pokud se také shoduje víc, než je stanovená mez, vypíšeme tyto komentáře a odebereme je ze seznamu. Takto to provedeme i pro všechny ostatní komentáře prvního seznamu.

5.4 Výpis výsledků aplikace

Výsledky kontroly plagiátů se budou vypisovat do zvoleného souboru, popřípadě i na standardní výstup. Na začátku výstupního souboru bude vždy uvedeno, jaká metoda byla použita pro detailní porovnání těl funkcí a také použité konfigurační hodnoty pro minimální shodu délek funkcí, minimální shodu statistik, minimální shodu na detailní porovnání a také minimální celkovou shodu funkcí pro označení projektu jako plagiát.

Další část bude obsahovat chybové hlášky od preprocesoru či analýzy konstrukcí.V těchto hláškách bude vždy uvedeno, k jaké chybě došlo a u kterého projektu. Například:

```
xabcde00: Neuzavreny blokovy komentar.
```

Poté bude následovat výpis plagiátorských skupin. To se hodí v případě, že když například bude od sebe opisovat skupina pěti studentů, budeme znát celou skupinu a nebudeme ji muset již zjišťovat z nalezených dvojic.

Nakonec bude následovat podrobný výpis dvojic plagiátů. Tyto dvojice budou seřazeny sestupně podle celkové dosažené shody. Po výpisu loginu dvojice a celkové shody (statistické metody i metody pro detailní porovnání) bude následovat seznam shodných funkcí. U každé funkce bude uveden její název u obou studentů a shoda těchto funkcí (opět pro statistické porovnání i detailní porovnání). V případě výpisu podobných komentářů se pod každou funkcí ještě vypíší shodné komentáře. Výsledek tedy bude vypadat například takto:

```
xfghij00 – xvwxyz00: 95.87% | 87.70%
-----------------------------------
zapis – write: 88.89% | 84. 21%
matice_soucin – mmult: 100.00% | 100. 00%
  vynuluj mezisoucet pro prvek
main – main: 91.29% | 72.86%
  zpracovani parametru – zpracujeme parametry
  chybny pocet parametru
matice_spicky – mpeak: 100.00% | 100. 00%
```

6 Výsledky a zhodnocení

Aplikace byla navržena a implementována tak, aby byla co nejrychlejší, snadno konfigurovatelná, přenositelná a také správně odhalila co nejvíce plagiátů. V této kapitole budou uvedeny výsledky testování, rychlost aplikace a také možná rozšíření do budoucna.

6.1 Testování aplikace

Aplikace byla testována na třech různých typech zdrojových kódů:

1. projekty s předpřipravenou kostrou;
2. projekty bez kostry v jednom modulu;
3. velmi rozsáhlé projekty rozdělené do několika modulů.

V prvním případě byly použity domácí úlohy z předmětu Algoritmy (v tabulkách a grafech jsou označeny zkratkou IAL). Zdrojové kódy jsou velmi podobné, mají stejný počet funkcí, které dělají stejnou činnost, pouze se mírně implementačně liší. Proto je vhodné u tohoto typu zdrojových kódů volit konfigurační hodnoty pro porovnání velmi vysoké, optimální je zvolit hodnotu 0,9 (tzn. 90 %) pro všechny čtyři konfigurační hodnoty pro porovnávání – minimální shodu délek, statistik, detailního porovnání a minimální celkovou shodu funkcí.

Ve druhém případě byly použity čtyři různě dlouhé projekty z předmětu Základy programování (v tabulkách a grafech jsou označeny zkratkami IZP1–IZP4). Zde je již projekt rozdělen na různý počet funkcí a každá může dělat něco jiného. Zde je dobré volit již menší konfigurační hodnoty pro porovnání. Pro detailní test a celkovou shodu je vhodné nastavit hodnoty v rozmezí 0,6–0,8 v závislosti na délce projektu a použité metodě detailního porovnání. Hodnoty pro délku funkce a statistický test je vhodné volit v rozmezí 0,7–0,8. Při menších hodnotách bude porovnání trvat mnohem déle, přestože celkový výsledek bude ve většině případů stejný.

V posledním případě byly použity projekty z předmětu Formální jazyky a překladače (v tabulkách a grafech jsou označeny zkratkou IFJ). Zde se plagiáty téměř nevyskytují, lze nalézt pouze některé podobné součásti (například velmi podobný lexikální analyzátor). Konfigurační hodnoty pro minimální shodu délek funkcí a statistik je vhodné volit stejně, jako ve druhém případě (0,7–0,8). Pro detailní porovnání a celkovou shodu je vhodné již volit nižší hodnoty (0,5–0,6).

6.2 Výsledky testů

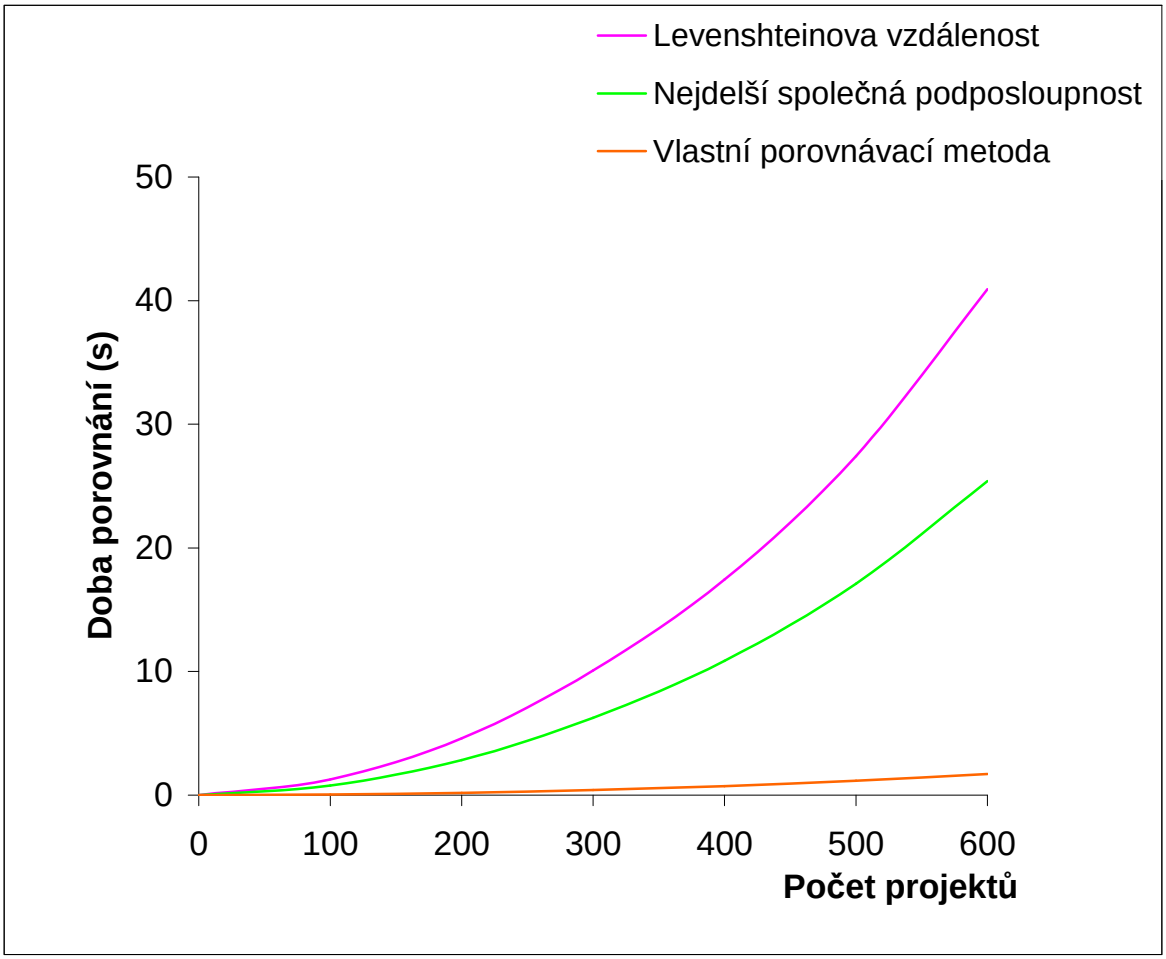
V této kapitole jsou uvedeny výsledky různých testů aplikace. Testování probíhalo na vzorcích projektů a konfiguračních hodnotách uvedených v kapitole 6.1. Všechny testy probíhaly na operačním systému Windows XP SP3 s konfigurací Intel Pentium IV 2,66 GHz a 512 MB RAM. Aplikace také byla vyzkoušena na operačním systémech Linux a FreeBSD.

6.2.1 Rychlost jednotlivých metod

První uvedený test porovnává celkové doby porovnání při použití jednotlivých metod pro detailní porovnání těl funkcí. Tento test byl prováděn na prvním projektu z předmětu Základy programování. Výsledky jsou uvedeny v tabulce 6.1 a grafu 6.1. Porovnávání má kvadratickou časovou složitost, tzn. zvýšíme-li počet porovnávaných prvků dvojnásobně, celková doba porovnávání se zvýší čtyřnásobně. Doba porovnávání se však pro jednotlivé metody značně liší.

Počet projektů		100	200	300	400	500	600
Doba porovnání v sekundách	Pouze statistická metoda	0,031	0,109	0,265	0,468	0,735	1,078
	Nejdelší společná podposloupnost	0,782	2,843	6,265	10,859	17,110	25,407
	Levenshteinova vzdálenost	1,265	4,609	10,093	17,437	27,422	40,922
	Vlastní porovnávací metoda	0,047	0,188	0,4065	0,734	1,171	1,703

Tabulka 6.1: Časová složitost jednotlivých metod



Graf 6.1: Doba porovnání u jednotlivých metod

Hned při prvním pohledu na graf 6.1 můžeme vidět, že Vlastní porovnávací metoda je mnohonásobně rychlejší, než Nejdelší společná podposloupnost či Levenshteinova vzdálenost. Celková doba porovnání s použitím Vlastní porovnávací metody je pouze 1,5× delší, než je celková doba porovnání s použitím pouze statistické metody. U Nejdelší společné podposloupnosti je celková doba porovnání asi 23,5× delší, u Levenshteinovy vzdálenosti je dokonce až 38× delší. Z toho důvodu jsou metody Nejdelší společná podposloupnost a Levenshteinova vzdálenost použitelné pouze u krátkých projektů, popřípadě je nutné alespoň snížit počet porovnání nastavením vysokých konfiguračních hodnot pro minimální shodu délek funkcí a minimální shodu statistik funkcí.

Nyní se ještě podíváme, kolik tokenů se v jednotlivých metodách porovnává mezi sebou u několika vybraných funkcí. Mezi vybranými funkcemi jsou dvě úplně shodné, dále pak funkce, které se liší pouze z části a pak funkce, které se liší značně. Všechny porovnávané funkce však mají

minimálně ze 70ti % shodnou délku a minimálně ze 70ti % shodné statistiky funkce. Výsledek je uveden v tabulce 6.2. První sloupec (n) značí počet tokenů první funkce, druhý sloupec (m) počet tokenů druhé funkce, třetí sloupec (m × n) maximální počet porovnání v případě, že se tokeny na začátku i na konci funkce vůbec neshodují. Ve čtvrtém sloupci je uvedeno počet porovnávání tokenů s přeskočením shodných tokenů na začátku a na konci řetězců obsahu funkcí. Toto množství porovnávání tokenů probíhá u metod Nejdelší společná podposloupnost a Levenshteinova vzdálenost. Poslední pátý sloupec uvádí počet porovnání tokenů v případě použití Vlastní porovnávací metody.

n	m	m × n	Přeskočení shodných tokenů	Vlastní porovnávací metoda
211	189	39879	39481	2045
55	72	3930	3472	219
122	136	16592	16336	805
51	51	2601	51	51
70	78	5460	4614	212
17	19	323	199	33
210	189	39690	39293	1536

Tabulka 6.2: Počet porovnávaných tokenů

Podle naměřených hodnot v tabulce 6.2 můžeme konstatovat, že u Vlastní porovnávací metody dochází k mnohonásobně menšímu počtu porovnávání tokenů. A to v některých případech až o 90 % méně.

Nejdelší společná podposloupnost je rychlejší, než Levenshteinova vzdálenost, přestože v obou případech se porovnává stejný počet tokenů. Důvodem je to, že u první zmíněné metody se v matici inicializuje pouze n+1 hodnot, u druhé n+m hodnot a také u Levenshteinovy vzdálenosti trvá déle provádění vnitřní smyčky.

Příklad počtu porovnání tokenů u Vlastní porovnávací metody

Nyní si ještě na jednoduchém příkladu dvou krátkých funkcí ukážeme, jakým způsobem probíhá porovnávání těl funkcí u Vlastní porovnávací metody a ke kolika porovnání tokenů dochází. Tím si také lépe objasníme princip této metody uvedený v kapitole 5.3.3.

Mějme tedy dvě funkce, první má 4 příkazy, druhá má 5 příkazů. Jednotlivé tokeny první funkce jsou:

```
20 23 21 50 27 59
1 21 16 21 59
4 21 59
4 21 59
```

Tokeny druhé funkce jsou:

```
19 59
20 23 21 50 27 59
1 21 16 21 59
20 22 59
4 21 59
```

1. krok

V prvním kroku se porovná první příkaz prvního řetězce (první funkce) a první příkaz druhého řetězce (druhé funkce). Již při porovnání prvního tokenu dojde k neshodě. Oba seznamy neshodných příkazů jsou prázdné, proto do seznamu neshodných příkazů prvního řetězce uložíme pozici začátku prvního příkazu v prvním řetězci a do seznamu neshodných příkazů druhého řetězce zase uložíme pozici začátku prvního příkazu v druhém řetězci.

První krok je hotový. Došlo v něm k jednomu porovnání tokenů a poté přeskočení osmi tokenů (při přechodu na další příkaz v obou řetězcích). Při přeskakování tokenů se jednotlivé tokeny porovnávají s tokenem oddělovače, který značí konec příkazu. Proto budeme i přeskočení na další příkaz započítávat do celkového počtu porovnání tokenů.

2. krok

Ve druhém kroku se porovná druhý příkaz prvního řetězce s druhým příkazem druhého řetězce. Zde se opět již v prvním tokenu liší. Protože seznamy již nejsou prázdné, pokusíme se porovnat příkaz z prvního řetězce s příkazem uloženým v seznamu neshodných příkazů druhého řetězce (`19 59`). Avšak opět se liší již v prvním tokenu. Dále zkusíme porovnat příkaz z druhého řetězce s příkazem uloženým v seznamu neshodných příkazů prvního řetězce (`20 23 21 50 27 59`). Zde se už všechny tokeny plně shodují. Ze seznamu neshodných příkazů prvního řetězce odstraníme tento příkaz.

V tomto kroku došlo celkem k porovnání osmi tokenů. Celkově tedy zatím došlo k porovnání sedmnácti tokenů.

3. krok

Ve třetím kroku se porovná druhý příkaz z prvního řetězce se třetím příkazem z druhého řetězce. Všechny tokeny jsou shodné, proto přistoupíme k dalšímu kroku.

V tomto kroku došlo rovnou ke shodě, proto bylo porovnáno pouze 5 tokenů. Celkově už došlo k porovnání 22 tokenů.

4. krok

Ve čtvrtém kroku se porovná třetí příkaz z prvního řetězce a čtvrtý příkaz z druhého řetězce. Neshoda je již na prvním tokenu. Seznam neshodných příkazů prvního řetězce je prázdný, seznam neshodných příkazů druhého řetězce obsahuje jeden příkaz (`19 59`), který se porovná se třetím příkazem prvního řetězce. Hned na prvním tokenu dojde k neshodě. Proto nezbývá, než do příslušných seznamů uložit pozice začátku jednotlivých příkazů.

V tomto kroku došlo ke dvěma porovnání a přeskočení šesti tokenů. Celkově tedy už došlo k porovnání 30 tokenů.

5. krok

V pátém kroku dojde k porovnání posledního příkazu prvního řetězce a posledního příkazu druhého řetězce. Všechny tokeny se shodují. V obou řetězcích to byly poslední příkazy, tudíž může být porovnávání těchto dvou funkcí ukončeno.

V posledním kroku došlo ke třem porovnání, celkově tedy došlo ke 33 porovnáním. U metod Nejdelší společná podposloupnost a Levenshteinova vzdálenost by pro porovnání těchto dvou řetězců bylo zapotřebí porovnat 199 tokenů (viz tabulka 6.2).

Pro doplnění zde ještě uvedeme, jakou nám Vlastní porovnávací metoda vrátí shodu. Byly nalezeny 3 shodné příkazy, které jsou tvořeny celkem 11 tokeny (nezapočítávají se oddělovače jednotlivých příkazů). Shodu vypočteme jako podíl počtu shodných tokenů a průměrného počtu tokenů obou funkcí. Výpočet výsledné shody pro zadané dvě funkce je uveden ve vzorci 6.1.

$$shoda = \frac{11}{\frac{13+14}{2}} = \frac{11}{13{,}5} = 0{,}8148 = 81{,}48\ \%$$

Vzorec 6.1: Výpočet shody dvou funkcí

6.2.2 Přesnost jednotlivých metod

Kromě rychlosti je důležité ještě porovnat přesnost výsledného porovnání u jednotlivých metod. Pokud porovnáváme dvě funkce, které jsou s velkou pravděpodobností plagiáty, měl by být celkový výsledek porovnání co nejvyšší, v případě, že nejsou plagiáty, měl by být výsledek co nejnižší.

1. funkce

```
double lnx(double vstup, double eps) {
  if (vstup == 0.0)
    return -INFINITY;
  if isinf(vstup)
    return INFINITY;
  if ((vstup < 0.0)||isnan(vstup))
    return NAN;
  double vystup;
  double citatel;
  unsigned long jmenovatel;
  double aktualni;
  double soucet = 0;
  long int stat = 0;
  const double od_eps=sqrt(IZP_E);
  const double od_eps_1=1.0/od_eps;
  while (vstup < od_eps_1) {
    vstup = vstup * od_eps;
    stat--;
  }
  while (vstup > od_eps) {
    vstup = vstup / od_eps;
    stat++;
  }
  vstup--;
  citatel = vstup;
  for(jmen=1; jmen; jmen++) {
    aktualni = citatel / jmen;
    soucet = soucet + aktualni;
    if(fabs(aktualni) <= eps)
      break;
    citatel = citatel - vstup;
  }
  vystup = soucet + (stat / 2.0);
  return vystup;
}
```

2. funkce

```
double lnx(double vstup, double eps)
{
  unsigned long jmenovatel;
  double citatel;
  long int ec=0;
  double actual;
  double soucet=0;
  const double SQRT_E=sqrt(IZP_E);
  const double SQRT_E_1=1.0/SQRT_E;
  if ((vstup<0.0)||isnan(vstup))
    return NAN;
  if (vstup==0.0)
    return -INFINITY;
  if isinf(vstup)
    return INFINITY;
  while(vstup<SQRT_E_1){
    vstup*=SQRT_E;
    ec--;
  }
  while(vstup>SQRT_E) {
    vstup/=SQRT_E;
    ec++;
  }
  vstup--;
  citatel=vstup;
  for(jmen=1; jmen; jmen++){
    actual=citatel/jmen;
    soucet+=actual;
    if(fabs(actual)<=eps)
      break;
    citatel*=-vstup;
  }

  return soucet+((double)ec/2.0);
}
```

Obrázek 6.1: Zdrojové kódy dvou podobných funkcí

Příklad si ukážeme na dvou funkcích uvedených na obrázku 6.1, které jsou převzaty z druhého projektu z předmětu Základy programování. Uvedené funkce slouží k výpočtu přirozeného logaritmu pomocí iterační metody.

Při bližším pohledu na funkce uvedené na obrázku 6.1 můžeme usoudit, že se s velkou pravděpodobností jedná o plagiáty. Jsou pouze přejmenovány některé proměnné, proházeny některé příkazy a transformovány nějaké konstrukce, například příkaz `soucet=soucet+aktualni` je transformován na `soucet+=actual`. Po porovnání všemi metodami, které aplikace obsahuje, získáváme údaje uvedené v tabulce 6.3.

Metoda	Výsledek porovnání
Statistická metoda	98,52 %
Nejdelší společná podposloupnost	76,75 %
Levenshteinova vzdálenost	67,16 %
Vlastní porovnávací metoda	92, 54 %

Tabulka 6.3: Výsledky porovnání dvou funkcí

V uvedených funkcích jsou navzájem přehozené dvě části programu – deklarace proměnných a ošetření vstupů. Statistická metoda i Vlastní porovnávací metoda si s proházením příkazů poradí, tudíž vrací shodu větší než 90 %. Problém nastává u zbylých dvou metod (Nejdelší společná podposloupnost a Levenshteinova vzdálenost), kdy v důsledku prohození dvou částí programu prudce klesne shoda funkcí.

Vlastní porovnávací metoda však má problém se změnami uvnitř příkazu. Příkazy se v této metodě berou jako jeden celek. Stačí malá změna v příkazu a celý příkaz je neshodný. Například příkaz:

```
if(NULL!=(x=malloc(sizeof(int)));
```

lze zapsat také jako:

```
if((x=malloc(sizeof(int))!=NULL);
```

Podle statistické metody dosahují tyto dva příkazy shodu 100 %, podle Nejdelší společné podposloupnosti 75 %, podle Levenshteinovy vzdálenosti 50 %, avšak podle Vlastní porovnávací metody 0 %. Avšak shoda se hodnotí v rámci celé funkce, takže změny uvnitř příkazu se neprojeví tak zásadně.

Jednotlivé metody byly zkoumány na dalších různých zdrojových kódech. V případě, že nebyly proházeny příkazy mezi sebou, dosahuje nejlepších výsledků Nejdelší společná podposloupnost. V případě proházení příkazů dosahuje nejlepších výsledků Vlastní porovnávací metoda. Tato metoda také nejspolehlivěji rozpozná rozdílné zdrojové kódy a tedy je u této metody nejmenší pravděpodobnost, že dva rozdílné zdrojové kódy vyhodnotí jako plagiáty (viz také příklad níže). Levenshteinova vzdálenost je nejpomalejší a také dosahuje nejhorších výsledků, proto není doporučeno ji používat.

V úvahu připadá ještě otázka, proč nepoužívat pouze statistickou metodu, neboť dosahuje nejvyšší rychlosti, nevadí ji ani proházení příkazů navzájem a ani proházení tokenů uvnitř příkazu.

Důvodem je to, že tato metoda někdy označí jako plagiáty funkce, které jsou zcela odlišné. Pokud si například vezmeme dvě náhodné funkce, které mají podobné konstrukce, ale dělají něco jiného, statistická metoda může vrátit velkou míru shody. Příklad si ukážeme na dvou funkcích (viz obrázky 6.2 a 6.3). První funkce vrací počet špiček v matici a druhá slouží pro přidělení paměti matici.

```
int spicky(TMatica *tmatica)
{
  int a, b, pom, spicky = 0;
  for(a = 0; a < tmatica->riadky; a++)
  {
    for(b = 0; b < tmatica->stlpce; b++)
    {
      if((pom = Osmiokolie(tmatica, a, b)) == 1)
        spicky++;
    }
  }
  return spicky;
}
```

Obrázek 6.2: Zdrojový kód funkce pro zjištění počtu špiček v matici

```
TMATRIX* alloc_matrix(int height, int width)
{
   int i=0;
   TMATRIX *st_matrix;
   if((st_matrix = malloc(sizeof(TMATRIX))) == NULL)
   {
     return NULL;
   }
   if((st_matrix->matrix = malloc(height*sizeof(void*))) == NULL)
   {
     return NULL;
   }
   while(i<height)
   {
     if((st_matrix->matrix[i] = malloc(width*sizeof(int))) == NULL)
     {
       return NULL;
     }
     i++;
   }
   st_matrix->width=width;
   st_matrix->height=height;
   return st_matrix;
}
```

Obrázek 6.3: Zdrojový kód funkce pro přidělení paměti matici

Na první pohled je vidět, že funkce jsou zcela rozdílné. Porovnáním těchto dvou funkcí všemi metodami získáme výsledky uvedené v tabulce 6.4. Statistická metoda dává shodu 81,63 %. Naproti tomu Vlastní porovnávací metoda dává hodnotu pouze 20,60 %. Z tabulky také můžeme vidět fakt, že Vlastní porovnávací metoda nejlépe rozpozná rozdílné funkce. Nejdelší společná podposloupnost také dosahuje vyšší hodnoty (62,4 %). To je způsobeno tím, že obsah funkce pro zjištění špiček je z větší části obsažený také ve funkci pro přidělení paměti matici (po transformacích a tokenizaci zdrojových kódů). Obecně použití Nejdelší společné podposloupnosti vede k výpisu největšího počtu plagiátů,

použití Vlastní porovnávací metody vede při stejných konfiguračních hodnotách mnohem menšího počtu plagiátů, proto je vhodné snížit minimální shodu detailního porovnání pro označení funkce jako plagiát.

Metoda	Výsledek porovnání
Statistická metoda	81,63 %
Nejdelší společná podposloupnost	62,40 %
Levenshteinova vzdálenost	48,00 %
Vlastní porovnávací metoda	20,60 %

Tabulka 6.4: Výsledky porovnání dvou rozdílných funkce

6.2.3 Dosažené zrychlení neporovnáváním různě dlouhých funkcí

Jak již bylo uvedeno v kapitole 5.3, nemá cenu porovnávat různě dlouhé funkce. V tabulce 6.5 je uveden počet porovnávání funkcí další metodou (statistickou), pokud vyřadíme ta porovnání, u nichž je procentuální shoda délek menší, než je stanovená mez (0–80 %).

Minimální shoda délek funkcí		0 %	20 %	40 %	60 %	80 %
Počet porovnání	IAL	30383194	26118131	19399617	12639536	7164126
	IZP1	7075647	5853521	4185320	2749859	1418782
	IZP2	11189639	8939959	6256592	4039714	2004448
	IZP3	18836149	14683650	10255840	6502559	3180515
	IZP4	4235200	3113812	2138563	1350098	660285
	IFJ	15796974	10679426	7257423	4619676	2266471

Tabulka 6.5: Závislost počtu porovnání na shodě délek funkcí

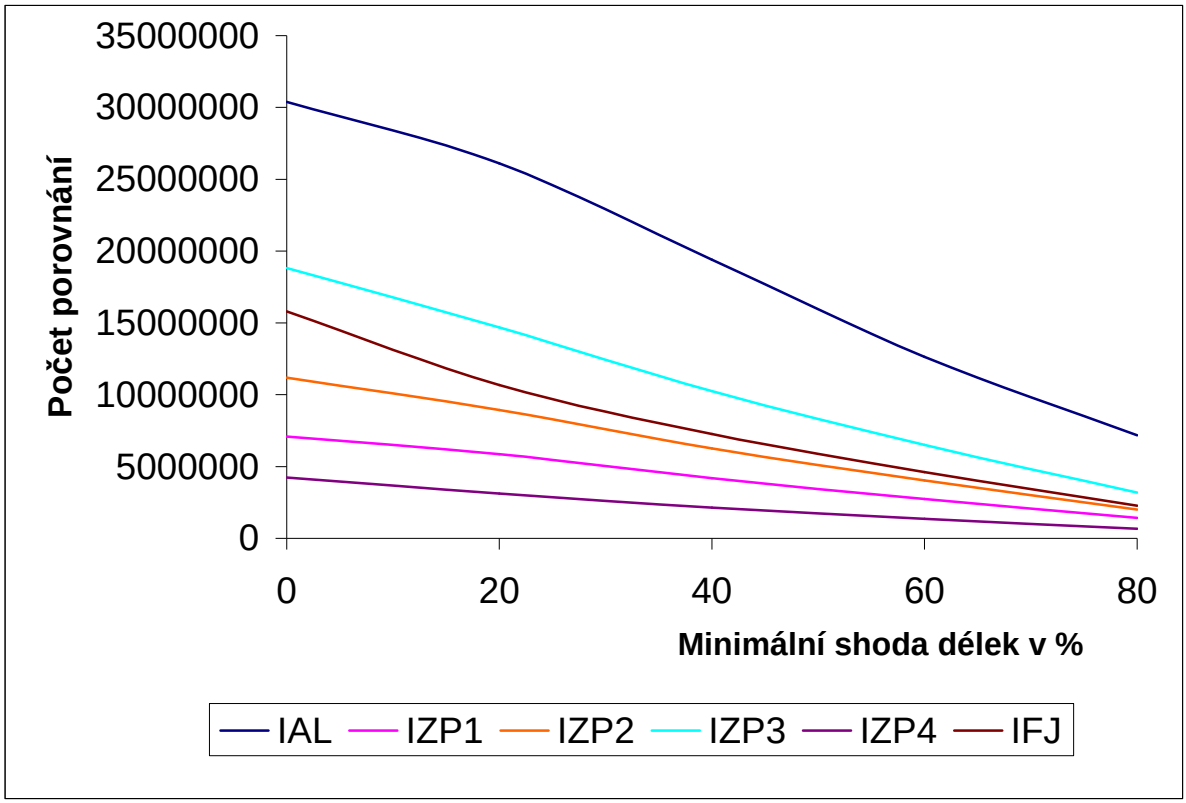
Z tabulky můžeme pozorovat, že zvyšováním minimální shody délek funkcí značně klesá množství dalších porovnání funkcí. Pro lepší názornost jsou naměřené hodnoty uvedené v grafu 6.2 a v tabulce 6.6 je uvedeno procentuální snížení počtu porovnání další metodou vyřazením porovnání funkcí, jejichž délky se shodují v méně než 20ti–80ti %.

Minimální shoda délek funkcí		20 %	40 %	60 %	80 %
Procentuální počet vyřazených funkcí	IAL	14,04	36,15	58,40	76,42
	IZP1	17,27	40,85	61,14	79,95
	IZP2	20,11	44,09	63,90	82,09
	IZP3	22,05	45,55	65,48	83,11
	IZP4	26,48	49,51	68,12	84,41
	IFJ	32,40	54,06	70,76	85,65

Tabulka 6.6: Procentuální snížení počtu porovnávání v závislosti na délce funkcí

Téměř pro většinu projektů můžeme vyřadit ta porovnávání, u kterých se délka funkcí neshoduje alespoň v 60ti %. Pro projekty s kostrou lze volit i vyšší čísla. Dle tabulky 6.6 můžeme vidět, že pokud vyřadíme z porovnávání funkce, které nemají alespoň z 60ti % stejnou délku, počet

dalších porovnání klesne o 58–70 %, přestože počet nalezených plagiátů se ve většině případů nesníží.



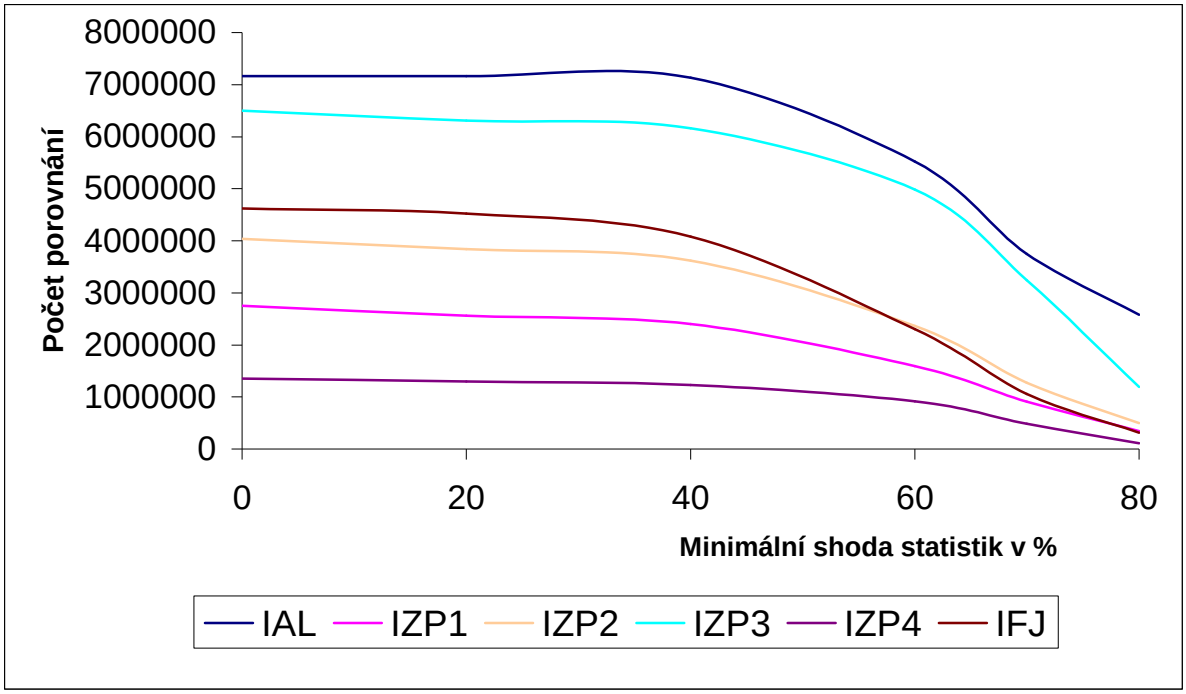
Graf 6.2: Závislost počtu porovnání na shodě délek funkcí

6.2.4 Dosažené urychlení použitím statistické metody

Nyní si uvedeme, jakého zrychlení jsme dosáhli díky tomu, že se před samotným detailním porovnávání nejprve provede statistický test, a na základě jeho výsledku nebudeme porovnávat ty funkce, které jsou podle statistické metody shodné méně, než je stanovená mez. Výsledky testu jsou uvedeny v tabulce 6.7 a grafu 6.3. V tomto testu bylo vyřazeno porovnání těch funkcí, které měly shodu délek menší než 60 %, u domácí úlohy z předmětu Algoritmy menší než 80 %. V tabulce můžeme vidět, že počet porovnání klesá již velmi pomalu, výraznější je to až po vyřazení porovnání funkcí, u nichž je výsledek statistické metody větší než 60 %. Je to zřejmě z toho důvodu, že velmi rozdílné funkce byly vyřazeny již při porovnávání délek funkcí. Avšak při volbě minimální shody statistických funkcí dle optimálních hodnot pro daný typ projektů, se sníží počet detailních porovnání přibližně o 40–60 %, přestože se celkový výsledek počtu plagiátů nebude příliš lišit.

Minimální shoda statistik		0 %	20 %	40 %	60 %	70 %	80 %
Počet porovnání	IAL	7163998	7163994	7131541	5522613	3750322	2581608
	IZP1	2749859	2561386	2401614	1590690	910505	345617
	IZP2	4039714	3842352	3617749	2367829	1269841	496291
	IZP3	6502584	6310977	6160870	4980375	3242518	1192161
	IZP4	1350098	1297031	1230144	913003	488387	110090
	IFJ	4619676	4523623	4082598	2303985	1054728	315182

Tabulka 6.7: Závislost počtu porovnání na shodě statistik funkcí



Graf 6.3: Závislost počtu porovnání na shodě statistik funkcí

Je však potřeba ještě uvažovat, jakého dosáhneme časového urychlení, neboť statistické porovnání také trvá určitou dobu. Příklad si ukážeme na 3. projektu z předmětu Základy programování (IZP3). Minimální shodu délek funkcí nastavíme na 60 %. Nejprve změříme dobu porovnání pro Vlastní porovnávací metodu s tím, že nepoužijeme statistickou metodu. Poté postupně budeme zvyšovat minimální shodu statistik až na 90 %.

Minimální shoda statistik	-	0 %	20 %	40 %	60 %	70 %	80 %	90%
Doba porovnání (s)	58,531	59,860	59,609	59,156	51,813	31,734	12,422	4,297
Počet nalezených dvojic plagiátů	4	4	4	4	4	4	4	3

Tabulka 6.8: Doba porovnání v závislosti na minimální shodě statistik funkcí

Z tabulky 6.8 můžeme vidět, že režie způsobená statistickou metodou je pouze 1,329 s. Tato režie se týká pouze doby porovnávání, neuvádí se zde režie vzniklá při analýze zdrojových kódů. Dále můžeme vidět, že až po shodu 60ti %, klesá doba porovnání velmi pomalu. Obrat nastává až od 60ti %, neboť funkce se shodou menší než 60 % jsou už vyřazeny při porovnávání délek funkcí. Při minimální shodě 80 % klesne doba porovnávání přibližně o 78,7 %, přestože počet nalezených plagiátů je stejný. Za úvahu ještě stojí zkusit změřit dobu porovnání bez použití statistické metody při vyřazení porovnání funkcí, které mají minimální shodu délek 70 % a 80 %. Zjišťujeme, že pro 70 % je doba porovnání 43,594 s, a pro 80 % je doba porovnání 29,343 s. Počet nalezených plagiátů je v obou případech opět 4. Pokud nastavíme minimální shodu délek na 80 % a minimální shodu statistik na 80 %, doba porovnání je 8,875 s, což je o 69,75 % rychlejší, než bez použití statistické metody.

6.2.5 Celková doba odhalování plagiátů

V tabulce 6.9 je uvedena doba zpracování zdrojových kódů, doba porovnání pro optimální konfigurační hodnoty s použitím Vlastní porovnávací metody a celková doba běhu aplikace.

Projekt	Doba zpracování zdrojových kódů (s)		Doba porovnání (s)	Doba celkem (s)
	První spuštění	Další spuštění		
IAL	10,969	3,370	4,500	15,469
IZP1	2,596	1,718	4,407	7,000
IZP2	5,210	2,531	18,395	23,605
IZP3	5,450	3,750	27,375	32,825
IZP4	3,125	2,312	11,281	14,406
IFJ	14,515	7,125	25,203	39,718

Tabulka 6.9: Celková doba běhu aplikace

Z tabulky 6.9 můžeme vidět, že i u velmi rozsáhlých projektů dokáže aplikace nalézt plagiáty v řádů desítek vteřin. Při dalších spuštění je v některých případech doba zpracování zdrojových kódů až třikrát rychlejší. Důvodem je zřejmě to, že zdrojové soubory zůstávají uloženy v operační paměti, tudíž není potřeba je znovu načítat z pevného disku. Další spuštění aplikace na stejné zdrojové kódy se může hodit v tom případě, že například poprvé byly zvoleny nevhodné konfigurační hodnoty, a z toho důvodu se vypsalo buď velké množství plagiátů nebo naopak žádné plagiáty. Dále můžeme z tabulky 6.9 vidět, že doba porovnání bývá obvykle větší, než doba zpracování zdrojových kódů.

6.3 Porovnání aplikace s již existujícími aplikacemi

Jak již bylo zmíněno v úvodu, tato práce volně navazuje na bakalářské práce [13] a [7], které se také zabývají tímto tématem. Aplikace, která je součástí první zmíněné práce, je určena pro kontrolu plagiátů v domácích úlohách v předmětu Algoritmy. Domácí úlohy mají již předpřipravenou kostru, zdrojové kódy se liší velmi málo. Pro tyto úlohy zmíněná aplikace dosahuje dobrých výsledků a také je velmi rychlá. Avšak pro programy, které nemají předpřipravenou kostru, není tato aplikace příliš vhodná, neboť aplikace nerozděluje projekty na jednotlivé funkce, ale bere projekty jako celek, který porovnává pomocí statistické metody.

Druhá aplikace, která je součástí druhé uvedené bakalářské práce [7], je již dělána pro projekty bez kostry. Používá preprocesor a automaticky také načítá zdrojové kódy ze zadaného adresáře. Pro porovnávání využívá statistické metody a Nejdelší společné podposloupnosti. Avšak hlavní nevýhodou této aplikace je příliš dlouhá celková doba hledání plagiátů. Zatímco naše aplikace zvládne vyhledat plagiáty v řádu desítek vteřin, aplikaci [7] to trvá více jak 11 minut. Je to z toho důvodu, že se porovnávají úplně všechny funkce mezi sebou vždy statistickou metodou a zároveň se i porovnávají těla funkcí pomocí metody Nejdelší společná podposloupnost. Nejsou vyřazena ta porovnání funkcí, kde se délky funkcí liší víc, než je stanovená mez. Také těla funkcí se tam porovnávají i v případě, že statistiky funkcí se liší víc než je stanovená mez. Dále se ani nepřeskakují shodné tokeny na začátku a na konci řetězců obsahující tokeny funkcí. Jak je vidět z testů uvedených výše, samotná metoda Nejdelší společná podposloupnost nedosahuje příliš velké rychlosti a není tedy vhodná pro rozsáhlé projekty.

Další nevýhodou aplikace [7] je používání externího preprocesoru. Ten se vždy spouští jako samostatný program pro všechny testované projekty a vytváří pro každý projekt nový soubor, ve kterém se nachází spojené zdrojové kódy daného projektu upravené preprocesorem. Naše aplikace

obsahuje vlastní preprocesor. Díky tomu se urychlí zpracování zdrojových kódů, neboť předzpracované zdrojové kódy se ukládají přímo do operační paměti. Není třeba je zapisovat do souboru na pevný disk a pak znovu načítat.

Dalšími vylepšeními naší aplikace oproti [7] jsou také lepší transformace konstrukcí, nalezení a výpis shodných či velmi podobných komentářů, seřazení plagiátů podle výsledků testů, možnost kontroly pouze jednoho konkrétního projektu s ostatními nebo možnost kontroly pouze jednoho zdrojového souboru konkrétního názvu.

6.4 Rozšíření do budoucna

Aplikace je navržena tak, aby ji šlo jednoduše dále rozšiřovat. Do budoucna by mohla obsahovat následující rozšíření:

- vylepšení kontroly komentářů, hledání pravopisných chyb v nich;
- převod složitějších konstrukcí, které lze zapsat více způsoby, na jeden tvar;
- uživatelské rozhraní;
- možnost kontrolovat plagiáty v programech psaných v dalších jazycích, které jsou podobné jazykům C a C++.

7 Závěr

Cílem diplomové práce bylo prostudovat veškeré konstrukce programovacích jazyků C a C++ a na základě získaných poznatků navrhnou a implementovat aplikaci, která rozpozná velmi podobné programy napsané v těchto programovacích jazycích.

Za svůj vlastní přínos této aplikace považuji vlastní preprocesor jazyků C a C++, který je funkčně téměř srovnatelný s běžným preprocesorem jazyka C, avšak je speciálně upravený pro naši aplikaci. Dalším přínosem jsou pokročilejší transformace konstrukcí, které lze zapsat několika způsoby, na jeden ekvivalentní tvar. Důležitým přínosem je také Vlastní porovnávací metoda, která se snadno vyrovná s proházením příkazů mezi sebou, a také je velmi rychlá. Díky tomu aplikace dokáže vyhledat plagiáty u velkého množství projektů během desítek vteřin. Také umí porovnat komentáře a poté shodné či velmi podobné vypsat, takže uživatel aplikace může u funkcí označených jako plagiát snadno hledat v komentářích stejné pravopisné chyby, stejné překlepy apod. Další možný vývoj této aplikace je uveden v kapitole 6.4.

Aplikace je implementována v programovacím jazyce C a použity byly pouze standardní knihovny, tudíž je zaručena přenositelnost na jiné platformy. Kromě operačního systému Windows XP, na kterém byla aplikace vyvíjena a testována, byla aplikace také zkoušena na Linuxu (Red Hat 9) a FreeBSD 7.0.

Literatura

[1] Bílek, P. *Programování v jazyku C/C++* [online]. [cit. 23. 11. 2008]. Dostupné na URL: <http://www.sallyx.org/sally/c/index.php>.

[2] Dostál, R. *Objektově orientované programování v C++* [online]. [cit. 23. 11. 2008]. Dostupné na URL: <http://www.builder.cz/serial24.html>.

[3] Harbison, S. P., Steele, G. L., *Referenční příručka jazyka C.* SCIENCE, Veletiny, 1996. ISBN 80-901475-50.

[4] Herout, P. *Učebnice jazyka C - 1. díl.* IV. přepracované vydání. KOOP, České Budějovice, 2006. ISBN 80-7232-220-6.

[5] Honzík, J. M. *Algoritmy: Studijní opora*. FIT VUT, Brno, 2007.

[6] Churý, L. *Kurz C++* [online]. [cit. 23. 11. 2008]. Dostupné na URL: <http://programujte.com/index.php?rubrika=26&sekce=84&kategorie=3>.

[7] Kačic, M. *Aplikace pro odhalování plagiátů u rozsáhlých projektů*. FIT VUT, Brno, 2008.

[8] Kračmar, S., Vogel, J. *Programovací jazyk C* [online]. [cit. 23. 11. 2008]. Dostupné na URL: <http://fs.cvut.cz/cz/U201/skrc.html>.

[9] Němec, J. *C/C++* [online]. [cit. 23. 11. 2008]. Dostupné na URL: <http://www.linuxsoft.cz/article_list.php?id_kategory=186>.

[10] Peringer, P. *Jazyk C*. [online]. [cit. 23. 11. 2008]. Dostupné na URL: <http://www.fit.vutbr.cz/study/courses/CPP/public/Prednasky/CPP/>.

[11] Růžička, D. *Učíme se jazyk C* [online]. [cit. 06.04.2009]. Dostupné na URL: <http://www.builder.cz/serial3.html>.

[12] Šaloun, P. *Programovací jazyk C++ pro zelenáče*. Neocortex s.r.o., Praha, 2005. ISBN 80-86330-18-4

[13] Šalplachta, P. *Aplikace pro odhalování plagiátů*. FIT VUT, Brno, 2007.

[14] Škoda, P., Mareš, M. *Recepty z programátorské kuchařky* [online]. [cit. 23. 11. 2008]. Dostupné na URL: <http://ksp.mff.cuni.cz/tasks/17/cook5.html>.

[15] Wikipedia: The Free Encyklopedia. *Levenshtein distance* [online]. [cit. 23. 11. 2008]. Dostupné na URL: <http://en.wikipedia.org/wiki/Levenshtein_distance>.

[16] Wikipedie: Otevřená encyklopedie. *C (programovací jazyk)* [online]. [cit. 23. 11. 2008]. Dostupné na URL: <http://cs.wikipedia.org/wiki/Jazyk_C>.

[17] Wikipedie: Otevřená encyklopedie. *Lexikální analýza* [online]. [cit. 06. 04. 2009]. Dostupné na URL: <http://cs.wikipedia.org/wiki/Lexikální_analýza>.

[18] Wikipedie: Otevřená encyklopedie. Plagiát [online]. [cit. 23 .11. 2008]. Dostupné na URL: <http://cs.wikipedia.org/wiki/Plagiát>.

[19] Wikipedie: Otevřená encyklopedie. *Preprocesor jazyka C* [online]. [cit. 23. 11. 2008]. Dostupné na URL: <http://cs.wikipedia.org/wiki/Preprocesor_jazyka_C>.

Seznam příloh

Příloha A – Manuál k aplikaci

Konfigurační hodnoty lze nastavit buď parametry příkazové řádky nebo v konfiguračním souboru. Pokud daná konfigurační hodnota nebude nastavená parametrem nebo konfiguračním souborem, bere se výchozí hodnota.

Parametry aplikace:

Parametr	Popis
-h, --help	Vypsání nápovědy.
-path *cesta*	Nastavení cesty k adresářům s projekty.
-out_file *nazev*	Nastavení názvu výstupního souboru pro výpis.
-test_file *nazev*	Nastavení názvu testovaného souboru.
-method *typ_metody*	Nastavení typu metody pro detailní porovnání funkcí.
-cmp_length *hodnota*	Nastavení minimální shody délek dvou funkcí pro další porovnávání.
-cmp_statistic *hodnota*	Nastavení minimální shody statistik dvou funkcí.
-cmp_detailed *hodnota*	Nastavení minimální shody detailního testu pro označení funkce jako plagiát.
-cmp_same_function hod*nota*	Nastavení minimální celkové shody funkcí pro označení celého projektu jako plagiát.
-test_1_N *nazev*	Testování pouze jednoho projektu z adresáře *nazev* s ostatními projekty.

Tabulka A.1: Parametry aplikace

Parametry začínající předponou *cmp* se nastavují v rozmení 0.0 (0 %) až 1.0 (100 %). Typ metody pro detailní porovnání se nastavuje čísly 0–3 (0 – pouze statistické porovnání, 1 – Nejdelší společná podposloupnost, 2 – Levenshteinova vzdálenost, 3 – Vlastní porovnávací metoda).

Příklad spuštění aplikace:

```
DP –path IZP_1 -out_file=izp1.txt –method=3 –cmp_same_function=0.7
```

Lepší je však nastavovat konfigurační hodnoty přímo v konfiguračním souboru *conf.txt*. Nastavení pomocí parametrů je vhodné použít pouze při dávkovém spouštění aplikace. V konfiguračním souboru mají jednotlivé položky stejný název, avšak pro přiřazení hodnoty se místo mezery vkládá rovná se. Například:

```
path=IZP_1
cmp_same_function=0.7
```

Kromě hodnot uvedených v tabulce lze ještě nastavit, zda vypisovat podobné komentáře u funkcí označených jako plagiát, minimální shodu komentářů pro vypsání, dále zda seřadit výpis plagiátů sestupně podle dosažené shody a minimální délku funkce pro porovnávání.

Příloha B – Obsah přiloženého CD

1. app – spustitelná aplikace
2. doc – textová část diplomové práce
3. src – zdrojové kódy aplikace
4. readme.txt – popis obsahu CD