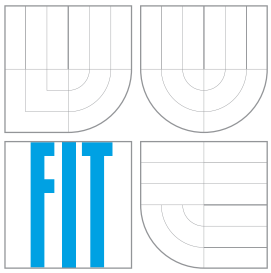
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

APLIKACE PRO ODHALOVÁNÍ PLAGIÁTŮ U ROZSÁHLÝCH PROJEKTŮ

APPLICATION FOR DETECTION OF PLAGIARISM

BAKALÁŘSKÁ PRÁCE

BACHELOR'S THESIS

AUTOR PRÁCE — MATEJ KAČIC

AUTHOR

VEDOUCÍ PRÁCE — Ing. ROMAN LUKÁŠ, Ph.D.

SUPERVISOR

BRNO 2008

Zadanie bakalárskej práce

1. Preštudujte všetky konštrukcie programovacieho jazyka C a C++.
2. Zamyslite sa nad spôsobmi detekcie dvoch veľmi podobných programov napísaných v jazykoch C a C++.
3. Navrhnite štruktúru aplikácie, ktorá rozpozná dva potencionálne rovnaké programy napísané v programovacích jazykoch C a C++.
4. Danú aplikáciu implementujte.
5. Funkčnosť aplikácie otestujte na vzorkách projektov odovzdaných študentmi z minulých rokov.
6. Zhodnoťte dosiahnuté výsledky, porovnajte vašu aplikáciu s už existujucími aplikáciami a navrhnite ďalšie možné rozšírenia do budúcnosti.

Licenčná zmluva

Licenčná zmluva je uložená v archíve Fakulty informačných technológií Vysokého učení technického v Brne.

Abstrakt

Cieľom práce je vytvoriť aplikáciu, ktorá rozpozná plagiáty v programovom kóde v projektoch bez kostry. Zaoberá sa konštrukciami v jazyku C a C++ a ich následným použitím pri detekcii plagiátov. Projekty prejdú fázami preprocesora, lexikálnej analýzy a tvorby porovnávacej štruktúry. Následne sa porovnávajú štatistickým testom a „Body“ testom založeným na Najdlhšej spoločnej podpostupnosti.

Kľúčové slová

jazyk C, jazyk C++, plagiát, odhaľovanie plagiátov, preprocesor, lexikálna analýza, syntaktická analýza, dynamické programovanie, najdlhšia spoločná podpostupnosť

Abstract

Main goal of this thesis is to create application, which can detect plagiarism in program code of projects without skeleton. It describes constructions of C/C++ language and their usage for detection of plagiarism. Projetcs are analysed by preprocesor, lexical analyse and phase of making structure of compare. After that they are compared one to each another by statistical test and Body test depends on Longest common subsequence.

Keywords

C language, C++ language, plagiarism, detection of plagiarism, preprocessor, lexical analyse, syntax analyse, dynamic programming, longest common subsequence, LCS

Citácia

Matej Kačic: Aplikace pro odhalování plagiátů u rozsáhlých projektů, bakalářska práce, Brno, FIT VUT v Brně, 2008

Aplikácia pre odhaľovanie plagiátov v rozsiahlych projektoch

Prehlásenie

Čestne prehlasujem, že som túto bakalársku prácu vypracoval samostatne pod vedením pána Ing. Romana Lukáša, Ph.D.

.....

Matej Kačic

6. mája 2008

Poďakovanie

Veľmi rád by som poďakoval vedúcemu mojej bakalárskej práce Ing. Romanovi Lukášovi, Ph.D. za jeho pripomienky, odborné rady a poskytnutie testovacích projektov z IFJ. Ďalej by som poďakoval RNDr. Jitke Kreslíkovej, CSc. za poskytnutie projektov z IZP pre účel testovania. Poďakovanie si určite zaslúži i moja rodina. Za ich trpezlivosť a pomoc. Ďakujem vám všetkým.

Obsah

Kapitola 1

Úvod

Keď práca niekoho iného je reprodukovaná bez znalostí zdroja, je to známe ako plagiát. Najviac prípadov sa nachádza v akademických inštitúciách, kde študenti kopírujú svoje práce medzi sebou, v presvedčení, že urobili všetko preto, aby zahladili po sebe stopy, pričom zabúdajú na jednu dôležitú vec, ktorou je vzdelanie.

Plagiát v softwarovom kóde môžme definovať ako program, ktorý bol vytvorený z iného programu s určitým počtom transformácií tak, aby na prvý pohľad nebolo poznať zdroj. Medzi základné transformácie patrí zmena komentárov, identifikátorov a riadiacich štruktúr (nahradenie cyklu `for` cyklom `while` apod.).

V rámci tejto bakalárskej práce sa pokúsim ukázať spôsob, ako odhaliť plagiáty a problémy, s ktorými som sa pri mojom výskume stretol, pričom sa zamerám na jazyk C resp. C++.

Práve základnými konštrukciami týchto jazykov sa zaoberá úvodná druhá kapitola. Pomerne krátka tretia kapitola analyzuje problémy detekcie plagiátov na základe koštrukcií jazykov. Návrhom celého systému sa zaoberá štvrtá kapitola. Na základe predspracovania zdrojových kódov pomocou preprocesora, lexikálnej analýzy a analýzy štruktúr môže začať porovnávanie tvorené dvomi testami. Štatistickým testom a „Body“ testom porovnávajúcim tokeny pomocou najdlhšej spoločnej podpostupnosti. Reálne výsledky implementovaného systému, dĺžky trvania a porovnania na rôznych prísnostiach sa objavia na stránkach piatej kapitoly.

Kapitola 2

Jazyk C/C++

Začiatkom 70. rokov Dennis Ritchie z Bell Laboratories potreboval pre implementáciu operačného systému jazyk, ktorý by bol stručný a výstižný, a zároveň by vytváral kompaktné rýchle programy s podporou riadenia hardwaru. Takto položil základy jazyka C, ktorý mal označenie K&R. Neskôr boli vydané ďalšie normy ako ANSI C a ISO/IEC 9899:1999, kde boli pridané rôzne rozširujúce vlastnosti.

C++ podobne ako C začína svoj život v laboratóriách Bellu, kde ho začiatkom 80. rokov vyvinul Bjarne Stroustrup, ktorého cieľom bolo urobiť písanie programov ešte jednoduchšími. Hľadisko objektovo orientovaného programovania bolo inšpirované simulačným jazykom Simula67. C++ je teda nadstavba jazyka C, čo znamená, že každý program preložiteľný v jazyku C je preložiteľným aj v C++.

2.1 Lexikálne prvky jazyka

Program pozostáva z postupnosti lexikálnych prvkov, ktoré na seba nadväzujú alebo sú oddelené oddeľovačmi. Za oddeľovač sa považuje neprázdna postupnosť bielych znakov (medzera, tabelátor, nový riadok, nová strana, návrat vozíka) alebo komentár, ktorý môže byť:

- riadkový - `// komentár`
- blokový - `/* komentár*/`

Vnorené komentáre nie sú povolené. Lexikálne prvky jazyka C ale aj C++ tvoria:

- identifikátory
- kľúčové slová
- konštanty
- interpunkčné znaky - `[ ] ( ) { } ...`

- reťazce
- operátory

2.1.1 Identifikátory a kľúčové slová

Je to skupina znakov používaná na identifikáciu alebo pomenovanie premenných, typov a funkcií. Identifikátor v programe musí byť jedinečný, pričom pozostáva z alfanumerických znakov a podtržítka, kde prvý symbol nemôže byť číslica. Oba jazyky sú *case-sensitive*. [1]

Kľúčové slová sú identifikátory, ktoré už majú vopred priradený význam. Podľa normy ANSI C [2] sú ako kľúčové slová definované: **asm, auto, break, case, char, const, continue, default, do, double, else, enum, extern, float, for, goto, if, inline, int, long, main, register, return, short, signed, sizeof, static, struct, switch, typedef, union, unsigned, void, volatile, wchar_t, while.**

Pre jazyk C++ podľa normy ISO/IEC 14882 [3] boli definové ako kľúčové slová: **asm, auto, bool, break, case, catch, char, class, const, const_cast, continue, default, delete, do, double, dynamic_cast, else, enum, explicit, export, extern, false, float, for, friend, goto, if, inline, int, long, main, mutable, namespace, new, operator, private, protected, public, register, reinterpret_cast, return, short, signed, sizeof, static, static_cast, struct, switch, template, this, throw, true, try, typedef, typeid, typename, union, unsigned, using, virtual, void, volatile, wchar_t, while.**[1]

2.1.2 Konštanty

Označujú konkrétnu hodnotu jedného dátového objektu daného typu. Preto sa u konštanty vždy rozlišuje jej typ, ktorý je daný typom jej konkrétnej hodnoty[4]. Konštanty sa podľa typu delia na:

- celočíselné - dekadické, oktálové (začína „0“) a hexadecimálne (začína „0x“). Je možné použiť explicitnú príponu „l“ resp. „L“ (0xFEL).
- znakové - sú vyjadrené medzi apostrofmi ’ ’, svojou oktálovou hodnotou za opačným lomítkom alebo *escpape* sekvenciou.
- v pohyblivej rádovej čiarke - bývajú zapísané v tvare celé číslo a desatinná časť oddelených desatinnou čiarkou (1.5) alebo v tvare mantisa a exponent (15e-1)
- enum - pomenúva konštantu typu int
- typu reťazec - postupnosť znakov medzi , môže obsahovať *escape* sekvencie, je ukončený binárnou nulou.

[1] Kľúčové slová **sizeof, typeid, new, delete** patria medzi operátory.

2.1.3 Operátory

Spolu s premennými a konštantami vytvárajú výrazy. Podľa počtu argumentov sa delia na operátory :

- Unárne operátory sa zapisujú pred argument a sú vyhodnocované zprava doľava. Operátor `++` a `--` sa môžu zapísať aj za argument.
  - `*` dereferencia (získanie obsahu miesta v pamäti podľa adresy)
  - `&` referencia (získanie adresy objektu)
  - `-` aritmetické mínus
  - `!` logická negácia
  - `~` bitový doplnok
  - `++` resp. `--` inkrementácia resp. dekrementácia hodnoty pred vyhodnotením nasledujúceho, resp. po vyhodnotení predchádzajúceho operandu
  - `(typ)` explicitný prevod na typ v zátvorke (pretypovanie)
  - `sizeof` získa veľkosť objektu alebo typu
- Binárne operátory majú dva operandy, podrobný prehľad je v tabuľke 2.1. Okrem nových kľúčových slov jazyk C++ pridáva aj nové lexikálne prvky a nové operátory[5]:
  - `::` kvalifikátor, sprístupňuje priestory mien
  - `->*` dereferencia ukazovateľa na člena triedy cez ukazovateľ na objekt
  - `.*` dereferencia ukazovateľa na člena triedy cez objekt
- Ternárny operátor bude podrobnejšie popísaný v nasledujúcej kapitole.

2.2 Syntaktické a sémantické prvky jazyka

Správnu postupnosť lexikálnych prvkov nazvime syntaxou. Práve syntaktická analýza overí správnu syntax a sémantická akcia pridá jednotlivým konštrukciám význam. V tejto časti ukážem základné konštrukcie jazyka C a C++.

2.2.1 Deklarácia premenných

Môžeme ju nájsť aj na globálnej, ale aj na lokálnej úrovni. Podľa toho majú premenné platnosť globálnu resp. lokálnu. Dôležité je poznamenať, že lokálna premenná dokáže zatieniť globálnu. Jazyk C++ pridáva možnosť usporiadať premenné do priestorov mien, čo vzniklo ako dôsledok rozrastania sa programov s možnosťou konfliktu mien. Obecný tvar deklarácie:

Aritmetické	Sčítanie	`+`
	Odčítanie	`-`
	Násobenie	`*`
	Delenie	`/`
	Modulo	`%`
Relačné	menší ako	`<`
	väčší ako	`>`
	menší alebo rovný	`<=`
	väčší alebo rovný	`>=`
	rovný	`==`
	nerovný	`!=`
Logické	logický súčin (AND)	`&&`
	logický súčet (OR)	`||`
Bitové	bitový súčin (AND)	`&`
	bit. exkluzívny súčet (XOR)	`^`
	bitový súčet (OR)	`|`
Priradenia	základné	`=`
	kombinované	`+= -= /= *= %=`
		`>>= <<= &= ^= |=`
Posuny	posun vľavo	`<<`
	posun vpravo	`>>`
Iné	operátor čiarka	`,`
	sprístupňuje prvky štruktúry	`.`
	sprístupňuje prvky štruktúry cez ukazovateľ	`->`

Tabuľka 2.1: Binárne operátory

```
typ   identifikátor  [=inicializácia];
typ   identifikátor  [=inicializácia], identifikátor  [=inicializácia];
```

Kľúčové slová `extern`, `auto`, `register`, `static` rozširujú deklaráciu o spôsob uloženia premennej a `const` vytvára symbolickú konštantu.

2.2.2 Dátové typy

- základný preddefinovaný typ (`char`, `int`, `float`, `double`, `bool`)
- odvodený typ (ukazateľ, pole, `struct`, `union`, `class`)
- vymenovaný typ `enum`
- užívateľom definovaný typ `typedef`

- prázdny typ `void`

Základné preddefinované typy môžu byť modifikované klasifikátorom `short`, `long`, `unsigned`, ktorý vyjadruje použitú podmnožinu alebo rozšírenie daného typu.

Štruktúra resp. union je dátový heterogénny typ zložený z prvkov, ktoré môžu byť rôzneho typu. Vnorené štruktúry, ako aj deklarácia v štruktúre je tiež prípustná. Union oproti štruktúre používa v jednom okamihu iba jeden prvkok.

```
struct {
  typ  identifikátor;
  typ2 identifikátor;
} premenná_1, premenná_2 ;

struct názov_štruktúry {
  typ  identifikátor;
  typ2 identifikátor;
};

struct názov_štruktúry premenná_1, premenná_2 ;

typedef struct {
  typ  identifikátor;
  typ2 identifikátor;
} nový_typ;
nový_typ premenná_1, premenná_2 ;
```

Pole na rozdiel od štruktúr je homogénny dátový typ, je jednorozmerné a indexované od nuly. Viacrozmerné pole sa vytvára ako pole jednorozmerných polí.

```
typ identifikátor[3], identifikátor_2[2][4];
```

Vymenovaný typ `enum` poskytuje alternatívny prostriedok ku `const` na vytváranie symbolických konštánt. Dovoľuje definovať nové typy, avšak obmedzeným spôsobom. Syntax je podobná so syntaxou štruktúry:

```
typedef enum {
identifikátor_1=1, identifikátor_2
} nový_typ;

enum  názov_enumu{
identifikátor_1=1, identifikátor_2
};
```

```
enum {
identifikátor_1=1, identifikátor_2
};
```

Pojem trieda resp. `class` bola pridaná až v jazyku C++ a je nástrojom, ktorý umožňuje vykonať abstrakciu do užívateľom definovaného typu. Kombináciou dát a metód(funkcií) vytvára elegantné a ľahko manipulujúce celky. Syntax triedy môžeme porovnať so štruktúrou bez metód. [6]

```
class T {
// dáta (i iné objekty)
int i;
// metódy
void m();
// špecifikácia prístupových práv
private:
// definícia vnorených typov, tried, konštánt, ...
typedef int typ;
};
```

Špecifikácia prístupových práv:

- `public` môže byť použitý ľubovoľnou funkciou
- `private` iba pre metódy a friend funkcie danej triedy
- `protected` ako private, ale na viac je dostupná v metódach a friend funkciách tried odvodených z tejto triedy

Preťažovanie operátorov je tiež vymoženosťou jazyka C++ a priraďuje viacej významov jednému symbolu. Rozlišuje operácie podľa kontextu a sprehľadňuje zápis programov. Preťažovať nemôžme operátory preprocesora `# ##` a operátory `. .* :: ?:`. Okrem operátorov je možné preťažovať aj funkcie s rovnakým meno, odlišujú sa len počtom a typom parametrov.

Kľúčovým konceptom objektovo orientovaného programovania, ktoré C++ podporuje, je dedičnosť. Umožňuje odvodiť novú triedu už z existujúcich tried, kde trieda dedí všetky dátové zložky bázovej triedy a metódy okrem konštruktorov a deštruktorov.
Ukážka zápisu dedenia:

```
class B {
public:
...
};
class C : public B {
```

```
public:
....
};
```

Šablóny tried poskytujú lepší spôsob generovania obecných deklarácií tried(generické programovanie). Typy sú parametrizované iným typom.
Príklad deklarácie šablón:

```
template<class R, class T> R funkce(T);
template<typename T> class Vector;
```

Príklad použitia šablón:

```
int i = funkce<int>(3.14);
double d = funkce<double,double>(3);
Vector<int> vec;
```

2.2.3 Riadiace štruktúry

Základnou jednotkou riadiacej štruktúry je príkaz. Riadiace štruktúry predpisujú poradie vykonávania jednotlivých výpočtov.

- výrazový príkaz `a=1+b;`
- blok, postupnosť príkazov uzavretých v { }
- podmienený príkaz umožňuje nám vetviť priebeh programu.
  Skrátený tvar: `if (výraz) príkaz;`

  Úplný tvar: `if (výraz) príkaz1; else príkaz2;`
- prepínač vetví program na viacero vetiev

  ```
  switch (výraz)
    {
    case hodn1 : prikaz1;break
      ...
    case hodnn : prikazn;break;
    default : prikaz;break;
    }
  ```
- príkazy cyklu

  ```
  while (podmienka) príkaz;

  do príkaz while (podmienka);

  for (inicializácia ; podmienka ; príkaz) prikaz;
  ```

- príkazy skoku: `break`, `continue`, `return`, `goto`
- Funkcie predstavujú základnú stavebnú jednotku jazyka C a C++. Obecný tvar funkcie je:

```
Typ_výsledku identifikátor(deklarácia parametrov); // deklarácia

Typ_výsledku identifikátor(deklarácia parametrov) // definícia
{
  //telo funkcie
}
```

2.2.4 Preprocesor

Pod týmto pojmom rozumieme množinu príkazov, ktoré sa vyhodnotia ešte pred samotnou kompiláciou. Umožňuje nám pracovať so symbolickými konštantami, textovými makrami, vkladať text a v neposlednom rade môžeme použiť podmienený preklad. Direktívy preprocesora sa začínajú znakom `#` hneď na začiatku riadku.

- Definícia symbolických konštánt a makier má tvar:

```
#define identifikátor (argumenty) [príkazy]
#define N 128
#define max(a,b) ((a>b)?a:b)
```

- vloženie systémových alebo užívateľských hlavičiek

```
#include <stdio.h>
#include "htable.h"
```

- podmienený preklad umožňuje vynechávať časti zdrojového kódu podľa podmienky

```
#if podmienka
#ifdef  symb_konštanta
#ifndef symb_konštanta
#else
#endif
```

- `#line` pomocná direktíva určujúca pôvod zdrojového kódu.

Kapitola 3

Analýza problému

Ako som už naznačil v úvode plagiát v softwarovom kóde môžeme definovať ako program, ktorý bol vytvorený z iného programu s určitým počtom transformácií, tak aby na prvý pohľad nebolo poznať zdroj. Dôležitou vlastnosťou aplikácie pre odhaľovanie plagiátov je správna detekcia podozrivého kódu. Inak povedané, musíme vedieť rozoznať plagiát od podobného projektu. V tejto kapitole sa budem venovať práve týmto transformáciám a spôsobmi detekcie.

Medzi základné transformácie, ktoré plagiátori používajú vo svojich prácach, patrí:

- zmena komentárov
- zmena dátových typov
- zmena identifikátorov
- pridanie redundantných príkazov alebo premenných
- zmena štruktúrnych prvkov jazyka
- kombinovanie originálu so svojím programom
- zmena príkazov za svoje ekvivalenty
- premiestnenie funkcie pri rozsiahlom projekte do iného modulu

Komentáre, ako už bolo spomenuté v sekcii 2.1, sa delia na riadkové a blokové. Väčšina plagiátorov mení komentáre, aj keď sú pre daný programový kód irelevantné, v dôsledku čoho považujeme komentáre z hľadiska rozsiahlych projektov za nepodstatné, čiže ich ignorujeme. Pre detekciu programov, ktoré majú už zadanú kostru však porovnanie komentárov môže byť prínosom.

Jazyky C a C++ obsahujú len pár základných typov a modifikátorov. Avšak kombináciou modifikátorov a typov môžeme dosiahnuť značné množstvo kombinácií odvodených typov, ktoré sú podobné bázovému typu. Transformácie typu `int - unsigned` alebo `int - long` sú v plagiátoch tak časté, že ich nemožno ignorovať. Táto zmena nemá skoro žiadny

vplyv na algoritmus, ale dokonale zmení deklaráciu premenných alebo definíciu nových typov. Ďalším spôsobom, ktorým možno zatieniť typ premennej, je použitie kľúčového slova `typedef`. Napríklad `typedef unsigned int uint;` nám vytvorí nový typ `uint`, ktorý bude zhodný s typom `unsigned int`. Jednoduché riešenie tohto problému je abstrahovať všetky deriváty základných typov na ich základy.

Zmena identifikátorov je ďalším častým prvkom z rady úkonov pri plagiátorstve. Zmeniť identifikátory premenných, nových typov, či prvkov štruktúrovaných typov je jednoduché a zároveň ľahko odhaliteľné. Jedno z riešení je urobiť abstrakciu v zmysle, že budeme uvažovať, či sa jedná len o premennú, typ alebo prvok štruktúrovaného typu resp. nebudeme uvažovať ich názov. Ďalším, o niečo náročnejším riešením je zavedenie tabuľky symbolov, používanú napríklad prekladačom pri preklade, do ktorej sa pri definícii a deklarácii pridá nový symbol a neskôr pri spracovaní tiel funkcií môžeme premenovať symboly jednotným spôsobom tak, aby bola zachovaná funkčnosť algoritmu. Toto riešenie má istú nevýhodu, pretože pri analýze by sme potrebovali poznať všetky definície použitých typov, čo by znamenalo spracovávať tisíce riadkov systémových hlavičiek. Ako bolo popísané v predchádzajúcej kapitole jazyky C a C++ majú pred vlastným prekladom takzvanú fázu predspracovania *preprocessing*. Musíme brať do úvahy, že symbolické konštanty a makrá preprocesora majú tiež svoje identifikátory, ktoré však nie sú platné po tejto fáze. Ako vhodné riešenie sa ponúka predspracovať si zdrojový kód ešte pred samotným vyhľadávaním plagiátov.

Medzi zmeny štruktúrnych prvkov jazyka považujeme také transformácie, ktoré zmenia na prvý pohľad zdrojový kód, ale myšlienka algoritmu zostane zachovaná. Najčastejším typom týchto transformácií sú zmeny cyklov `while` za `do-while` resp. `for`, a následné zmeny podmienok v cykloch alebo v podmienenom príkaze `if`, kde pomocou znalosti boolovej algebry sa veľmi jednoducho zapíše ekvivalentný výraz. Napríklad `a==1&&b!=0` môžeme prepísať pomocou De Morganovho zákona do tvaru `!(a!=1||b==0)`. Tieto zmeny sú pomerne ťažko odhaliteľné. Jednou z možností ako odhaliť tieto zmeny je štatistická metóda popísaná v nasledujúcej kapitole.

V každom jazyku existujú také stavby jazyka, ktoré sú úplne iné, ale význam majú rovnaký. Príkladom môže byť zmena operátora `++`: `i++;` = `i+=1;` = `i=i+1;` = `i=1+i`. Riešením by bolo transformovať všetky postupnosti tokenov rovnakého typu a významu na rovnakú postupnosť a následne začať porovnávať.

Pri rozsiahlych projektoch je použitie modularity samozrejmosťou. Tento prostriedok umožňuje plagiátorovi vhodne schovať opísaný kód tým, že poprehadzuje funkcie medzi jednotlivými modulmi. Väčšina podobných programov na odhaľovanie plagiátov porovnáva zdrojový kód len v rámci modulu, preto by bolo vhodné porovnávať projekty ako celky. Čo však prináša komplikácie ako konflikty identifikátorov a pod.

Pridávanie redundantných prvkov v programovom kóde, kombinovanie originálu so svojím programom a rozdelenie alebo spojenie kódu funkcií zo zdroja je veľmi ťažké odhaliť. Tieto prvky by mohli byť predmetom ďalšieho skúmania.

Kapitola 4

Návrh riešenia

Po analýze problému detekcie plagiátov sa v tejto kapitole budem venovať návrhu jeho riešenia. Podrobne bude popísaná každá fáza, ktorou zdrojové kódy jednotlivých projektov prejdú.

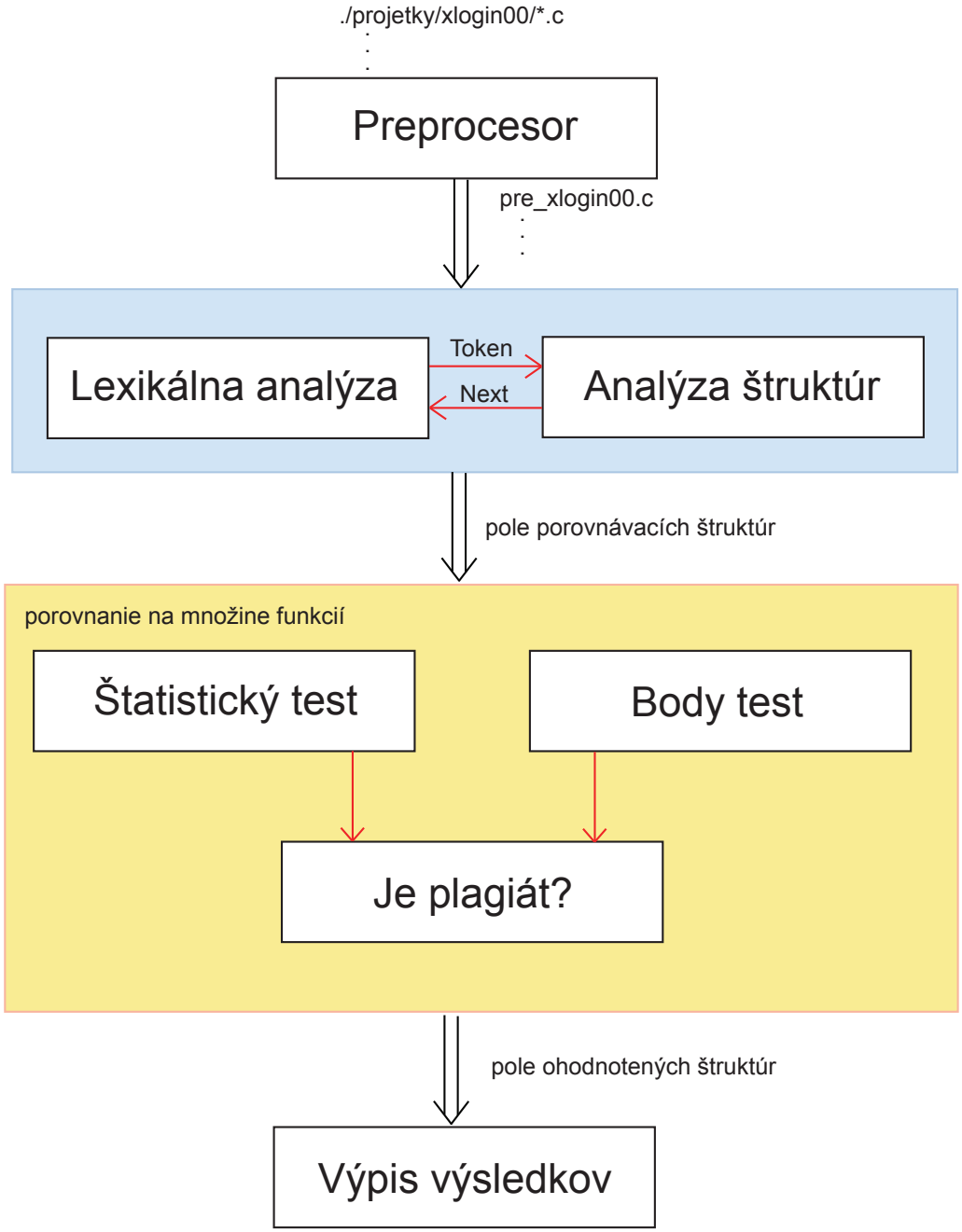
Schéma 4.1 popisuje základný koncept fungovania systému. Na vstupe sú adresáre so všetkými projektami, z ktorých preprocesor vezme všetky zdrojové súbory a predspracuje ich. Výstupom je jeden súbor s celým zdrojovým kódom projektu. Nasleduje ďalšia fáza, tvorba porovnávacej štruktúry obsahujúca dve základné časti. Lexikálnu analýzu a analýzu štruktúr, ktoré spolu komunikujú na princípe používanom v prekladačoch, kde lexikálna analýza sa spustí na základne volania analýzy štruktúr. Výstupom tejto fázy je pole štruktúr pripravených na ďalšiu fázu, porovnanie, ktoré pracuje na množine funkcií všetkých projektov. Porovnanie obsahuje dva testy, štatistický a „Body“ test, pričom kombinácia výsledkov testov rozhoduje, či daná dvojica funkcií je plagiát alebo nie. Poslednou častou je štruktúrovaný výpis výsledkov z poľa ohodnotených štruktúr.

Odlišnosti jazykov C a C++ spôsobili, že systém sa musí prepnúť medzi jednotlivými módmi. Každý mód zabezpečuje spracovanie jedného jazyka. Vďaka vhodnému návrhu celého systému sa prepnutie uskutočňuje len na úrovni tvorby porovnávacej štruktúry. Podrobnosti sú popísané v časti 4.2.3 o analýze štruktúr.

4.1 Preprocesor

Prvým blokom spracovania je preprocesing. Ako základ som použil open source preprocesor *mcpp* šírený pod BSD licenciou, ktorého autor je Kiyoshi Matsui. Tento preprocesor spĺňuje špecifikácie jazyka C podľa normy C99 a jazyka C++ podľa normy C++98. Je plne prenositeľný na všetky platformy. Podľa technickej správy [7] je *mcpp* najvýkonnejší a zároveň plne dodržujúci normy.

Preprocesor bolo treba poupraviť. Zdrojový text zo všetkých modulov sa bude spájať do jedného súboru, avšak preprocesing bude prebiehať nad každým modulom samostatne. Druhou úpravou je zakázanie vkladania systémových hlavičiek, pretože pre samotné porov-



Obrázok 4.1: Schéma ...

nanie nie sú potrebné. Problém však bude vznikať až pri tvorbe porovnávacej štruktúry, pretože nebudeme v tom momente poznať definíciu všetkých použitých typov. Riešenie tohto problému bude popísané v časti 4.2 venovanej tvorbe porovnávacej štruktúry. Užívateľské hlavičky budú vkladané iba raz, aby sme zabránili viacnásobnému vloženiu. Ďalšou úpravou prešlo aj vkladanie direktívy preprocesora `#line`, ktoré používam v tvorbe porovnávacej

štruktúry na zistenie pôvodného modulu danej funkcie.

4.2 Tvorba porovnávacej štruktúry

4.2.1 Návrh porovnávacej štruktúry

Dôležitým krokom bol vhodný návrh dátovej štruktúry, kde by sme mohli uschovať všetky potrebné dáta z analýzy štruktúr a výsledky z porovnávania. Mali by sme rozlišovať globálnu a lokálne úrovne projektov, pri ktorých musíme uschovať štatistické údaje, názvy funkcií a pod. Keďže porovnávanie funkcií je sekvenčné môžeme použiť jednosmerný lineárny zoznam pre uschovanie funkcií. V konečnom dôsledku som zvolil ako základ pole štruktúr pre globálnu úroveň, z ktorej sa sprístupňuje zoznam lokálnych funkcií.

4.2.2 Lexikálna analýza

Načítava znaky zo vstupu, rozpoznáva a klasifikuje lexémy, ktoré reprezentuje vo forme tokenov. Lexéma je postupnosť znakov v zdrojovom programe odpovedajúca vzoru pre nejaký token, pričom vzor je pravidlo, zvyčajne regulárny výraz, popisujúce množinu znakov pre daný token. Lexikálna analýza odstraňuje komentáre a prázdne miesta v zdrojovom kóde a detekuje kľúčové slová.

Pre implementáciu lexikálne analýzy som zvolil deterministický konečný automat, ktorý je postavený na základe noriem jazykov C a C++. Lexikálna časť jazykov je veľmi podobná, môžeme povedať, že jazyk C je podmnožinou jazyka C++, v dôsledku čoho automat pre spracovanie jazyka C++ môže spracovávať jazyk C. Výnimku musíme urobiť u tabuľky kľúčových slov, pretože mnohý užívatelia používajú kľúčové slová z jazyka C++ pre svoje identifikátory.

4.2.3 Analýza štruktúr

Pri preklade jazyka do vnútorného interpretačného kódu sa používa syntaktická analýza spolu so sémantickými pravidlami. Pre jazyk C a C++ sa používa LR syntaktický analyzátor, čo je rozšírený zásobníkový automat, ktorého činnosť je založená na LR tabuľke.

Syntaktická analýza nie je pre odhaľovanie plagiátov potrebná, pretože na rozdiel od prekladu, nepotrebujeme generovať podrobný vnútorný kód, ale stačí nám detekovať základné konštrukcie jazyka, ktoré sa dajú analyzovať konečným automatom. Nemožno takto zanalyzovať všetky konštrukcie jazyka napr. definícia štruktúry v štruktúre, pretože konečný automat je oveľa slabší ako LR syntaktická analýza. Súčasťou tejto analýzy je tabuľka symbolov, ktorá je implementovaná pomocou *hash* tabuľky, kde budeme ukladať všetky identifikátory. Umožňuje nám to detekciu nových typov, prípadne jednotné premenovanie premenných. Analýza zároveň počíta štatistiky, ktoré sa používajú pre štatistický test popísaný v časti 4.4.

Deklaráciu nových premenných, funkcií či parametrov zistíme podľa postupnosti tokenov `TYP ID`. Problém však môže nastať v dôsledku nevkladania systémových hlavičiek, pretože tabuľka symbolov nepozná typy definované v týchto hlavičkách, čo znamená, že analýza dostane postupnosť tokenov `ID ID`. Keďže toto je neplatná konštrukcia jazyka, môžeme tento problém vyriešiť tak, že keď nám príde postupnosť `ID ID` budeme to považovať za deklaráciu, pričom prvý identifikátor považujeme za typ a vložíme ho do tabuľky symbolov. Inicializačnú časť deklarácie ignorujeme. V zásade automat rozpoznáva len to, čo pozná. V prípade, že narazí na nejakú nezrovnalosť, má v sebe implementované zotavenie z chyby. Ak v automate nastane chyba, prejde do chybového stavu, z ktorého môže vyjsť len za podmienky:

```
if neprišiel žiadny token "{" a prišiel token ";"
   tak nasledujúci stav je start.
else if prišiel token "{"   {
       čakáme na token "}", aby sme uzavreli blok
       if prišiel ";"
         stav je start
}
```

Definíciu základného stavebného prvku jazyka, funkciu, detekujeme ako postupnosť tokenov `TYP ID(parametre) {` .Tým sa dostávame do lokálnej časti s vlastným konečným automatom. Telo funkcie okrem deklarácií sa ukladá a neskôr sa použije v „Body“ teste popísaného v sekcii 4.5. Koniec funkcie sa zisťuje podľa poslednej uzatváracej zátvorky bloku.

Pri definovaní nových typov vložíme do tabuľky symbolov symbol reprezentujúci tento typ. Definícia môže byť typu `struct` alebo `union` a `enum`. Pri definovaní nových typov treba brať do úvahy aj kľúčové slovo `typedef`, ktorá značne zmení význam identifikátorov pri definícii.

Po zlúčení zdrojového textu so všetkých modulov sa môže vyskytnúť redundantná deklarácia či definícia. Pri deklarácii premenných si môžeme dovoliť túto chybu ignorovať, pretože nám to nič neovplyvní. Avšak pri definícii funkcií by sme prišli o celé telo funkcie, tak vhodným riešením je zkonkatenovať názov modulu s názvom funkcie, čo nám zaručí jednoznažnosť identifikátora.

Funkcie o malom počte príkazov sú z porovnávania vyradené, pretože tak malý kúsok kódu sa veľakrát nedá inak zapísať.

Pre jazyk C++ je potrebné urobiť v automate určité zmeny. Budeme musieť dorobiť spracovanie konštrukcie tried. Ďalším problémom sú priestory mien a preťažovanie funkcií, pretože by vznikalo mnoho konfliktov v tabuľke symbolov, ale aj pri výpise funkcie by sa mohli zhodovať názvy funkcií. Optimálne riešenie je zkonkatenovať názov modulu, menný priestor s názvom funkcie resp. metódy prípadne pri preťažení funkcie pridať na koniec ešte počítadlo. Tvar pri výpise by potom vyzeral: `názov modulu + menný priestor + názov`

`metódy + počítadlo`.

4.3 Porovnanie

Po predpracovaní, tokenizácii a analýze štruktúr prichádza na rad porovnanie, ktoré pracuje na množine funkcií, čo nám umožní odhaliť plagiátorov na úrovni funkcií.

Vezmime však do úvahy počet porovnaní. Keby bolo 100 projektov a každý by mal v priemere 20 funkcií, tak množina všetkých funkcií má 2000 prvkov. Počet porovnaní na množine funkcií je 1 980 000. Ak by sme zohľadnili fakt, že funkcia označená za plagiát sa nemusí porovnávať, potom počet porovnaní výrazne klesne a je nepriamoúmerný miere plagiátorstva. Inak povedané, keby všetky funkcie boli vzájomné plagiáty, tak počet porovnaní je 2 000. Ďalej môžeme vylúčiť zo vzájomného porovnania funkcie nachádzajúce sa v jednom projekte.

Porovnanie dvoch funkcií vykonávajú dva testy. Štatistický test a Body test, ktorých kombinácia výsledkov rozhoduje o plagiáte funkcie, ktoré sú následne zaradené do skupín podľa typu plagiátu. Pri pozitívnom náleze si musíme vytvárať zoznam plagiátov, aby bolo možné spätne vypísať skupiny plagiátorov, a zároveň označiť funkcie pre vynechanie z ďalšieho porovnania.

4.4 Štatistický test

Je to test založený na počítaní kľúčových prvkov jazyka a následnom počítaní podobnosti. Pri návrhu tohto testu som vychádzal z kapitoly o základných konštrukciách jazyka, ktorá mi poskytla ucelený prehľad o základných a pritom špecifických znakoch algoritmu. V analýze štruktúr sa počítajú počty:

- parametrov
- lokálnych premenných
- podmienok (`if`)
- cyklov (`while`, `do`, `for`)
- kľúčových slov
- priradení (základné + kombinované)
- aritmetických operácií (`+ - *`)
- logických operácií (`&& ||`)
- relačných operácií (`< > >= <=`)
- bitových operácií (`& | ^`)

- dynamických alokácií pamäte (`malloc`, `realoc`, `new`)
- dealokácie pamäte (`free`, `delete`)
- operátorov `sizeof`
- indexácií mimo deklaráciu
- blokov
- bitových posunov (`<< >>`)
- konštánt

Výsledok porovnania sa počíta ako vážený priemer, pričom každý znak má pridelenú svoju váhu, ktorú je možné meniť pomocou konfiguračného súboru.

4.5 „Body“ test

V dôsledku tokenizácie sa veľkosť algoritmu značne zmenšila a pripravila sa pôda pre test, pri ktorom sa navzájom porovnávajú tokeny. Problémom však je určenie miery podobnosti dvoch sekvencií tokenov. Najzákladnejšou mierou podobnosti je určenie rovnakých tokenov na rovnakých pozíciách, ktorej algoritmus by mohol vyzerať takto:

```
iMax=Max(strlen(s1), strlen(s2));
    iMin=Min(strlen(s1), strlen(s2));
    for(int i=0;i<iMax;i++) {
      if(i==iMin)
        break;
      if(s1[i]==s2[i])
        count++;
    }
    if(count>0)
      return ((double)count/iMax);
  return 0.0;
```

Ďalšou metódou, ako zistiť podobnosť sekvencií je určenie Levenshtein-ovej vzdialenosti, nazývanou tiež editačná vzdialenosť, ktorej výsledok je minimálny počet operácií potrebných na transformovanie jedného reťazca na druhý, kde operácie sú vkladanie, vymazanie alebo nahradenie jedného znaku. Napríklad Levenshtein-ová vzdialenosť medzi „kitten“ and „sitting“ je 3. Body test však na určenie podobnosti používa princíp Najdlhšej spoločnej podpostupnosti *Longest common subsequence*. Pre pochopenie princípu implementácie tohto algoritmu je potrebné uviesť problematiku dynamického programovania.

4.5.1 Dynamické programovanie

Tento termín bol po prvý krát použitý v 40. rokoch Richardom Bellmanom na popísanie procesu riešenia problémov, kde je potreba nájsť najlepšiu sekvenciu rozhodnutí.[8] Neskôr v 50. rokoch bol tento termín predefinovaný. Dynamické programovanie rieši problémy kombinovaním riešení podproblémov. Nie je to programovanie ako tvorba počítačového kódu, ale skôr implementácia algoritmu tabuľkovou metódou. Používa sa pre riešenie rekurzívnych problémov nerekurzívne, to znamená, že každý problém vyrieši iba raz a výsledok uloží do tabuľky, čo zabraňuje znovu počítaniu problémov. Dynamické programovanie je väčšinou aplikované na optimalizačné problémy, inak povedané na problémy, ktoré majú veľa možných riešení.

Vývoj algoritmov pomocou dynamického programovania môžeme rozčleniť na tri časti:

- Charakteristika štruktúry optimálneho riešenia
- Rekurzívne definovať optimálne riešenie, rozdeliť problémy na podprobémy s optimálnym riešením
- Použiť optimálne riešenia podproblémov k zostaveniu optimálne riešenia problému

Môžeme teda povedať, že problém pozostáva z podproblémov, ktoré sa navzájom prekrývajú. Napríklad pre výpočet *Fibonacciho* postupnosti používame tento rekurzívny algoritmus:

```
function fib(n)
       if n = 0
           return 0
       else if n = 1
           return 1
       return fib(n - 1) + fib(n - 2)
```

Výpočet `fib(4)` môžeme rozložiť takto:

```
1.  fib(4)
2.  fib(3)+fib(2)
3. (fib(2)+fib(1))+(fib(1)+fib(0))
4. ((fib(1)+fib(0))+fib(1))+(fib(1)+fib(0))
```

Ako môžeme vidieť niektoré podproblémy napr. `fib(2)` sú počítané viac krát, čo v konečnom dôsledku vedie k exponenciálnemu času riešenia. V tomto prípade môžeme použiť pole, kde budeme ukladať všetky čísla, ktoré sme vypočítali. Funkcia bude na výpočet potrebovať $\Theta(n)$ čas namiesto exponenciálneho. Táto technika pre ukladanie už vypočítaných podproblémov sa nazýva *memoization*. Nerekurzívny algoritmus pomocou dynamického programovania:

```
int fib(int n) {
  int a[n];
  a[0]=1;
  a[1]=1;
  for(int i=2;i<n;i++) {
    a[i]=a[i-1]+a[i-2];
  }
  return a[n-1];
}
```

4.5.2 Najdlhšia spoločná podpostupnosť

Máme postupnosť X a Y. K vyriešeniu problému najdlhšej spoločnej podpostupnosti (Longest common subsequence) by sme potrebovali vypočítať všetky podpostupnosti X a porovnať ich s podpostupnosťami Y. Ak postupnosť X má $1..n$ prvkov, tak obsahuje 2^n podpostupností. V konečnom dôsledku potrebujeme na výpočet exponenciálny čas. Dynamické programovanie ponúka vhodnú alternatívu na riešenie tohto problému pričom má zložitosť $\Theta(mn)$. Problém najdlhšej spoločnej podpostupnosti má optimálne riešenie, kde podproblém korešponduje s pármi prefixov dvoch sekvencií. Máme postupnosť $X = \{x_1, x_2, \ldots, x_n\}$ a definujeme $i-$tý prefix postupnosti X, pre $i = \{1, 2, \ldots, n\}$ ako $X_i = \{x_1, x_2, \ldots, x_i\}$.[1]

Optimálnu štruktúru vyjadruje teorém[2], prevzatý z knihy „Introduction to Algorithms“[9]: Nech $X = \{x_1, x_2, \ldots, x_n\}$ a $Y = \{y_1, y_2, \ldots, y_m\}$ sú postupnosti a nech $Z = \{z_1, z_2, \ldots, z_k\}$ je nejaká najdlhšia spoločná podpostupnosť postupností X a Y.

- Keď $x_n = y_m$, potom $z_k = x_n = y_m$ a Z_{k-1} je najväčšia spoločná postupnosť X_{n-1} a Y_{m-1}
- Keď $x_n \neq y_m$, potom $z_k \neq x_n$ z toho vyplýva, že Z je najväčšia spoločná postupnosť X_{m-1} a Y
- Keď $x_n \neq y_m$, potom $z_k \neq y_m$ z toho vyplýva, že Z je najväčšia spoločná postupnosť X a Y_{m-1}

Nech $c[i, j]$ je dĺžka najdlhšej spoločnej podpostupnosti postupností X a Y potom optimálne riešenie vyjadruje rekurzívny zápis:

$$c[i,j] = \begin{cases} 0 & \text{keď } i = 0 \text{ alebo } j = 0, \\ c[i-1, j-1] + 1 & \text{keď } i, j > 0 \text{ a } x_i = y_j, \\ max(c[i, j-1], c[i-1, j]) & \text{keď } i, j > 0 \text{ a } x_i \neq y_j. \end{cases}$$

Z ktorého je jednoduché navrhnúť algoritmus pomocou dynamického programovania:

[1] x_0 je prázdna postupnosť

[2] dôkaz teorému je uvedený v literatúre

```
double LCS_length (const char* x, const char *y) {
  unsigned m=strlen(x);
  unsigned n=strlen(y);
  if(m==0||n==0)
    return 0.0;
  unsigned i, j;
  int** C=alloc_matrix(m+1, n+1);
  for(i=1;i<=m;i++)
    C[i][0]=0;
  for(j=0;j<=n;j++)
    C[0][j]=0;
  for(i=1;i<=m;i++) {
    for(j=1;j<=n;j++)
      if(x[i-1]==y[j-1])
        C[i][j]=C[i-1][j-1]+1;
      else
        C[i][j]=Max(C[i][j-1], C[i-1][j]);
  }
  int ret=C[m][n];
  free_matrix(C, m);
  return (double)ret;
}
```

Algoritmus alokuje maticu o veľkosti $m \times n$, čo je pre porovnanie plagiátov neprípustné. Matica takých rozmerov slúži len na spätný výpis podpostupnosti a pre výpočet podobnosti potrebujeme poznať len dĺžku podpostupnosti a dĺžku najdlhšej sekvencie. Po optimalizácii algoritmus alokuje len maticu $2 \times n$, čo je prípustné pre porovnanie.

4.6 Obnovenie porovnávania pri páde aplikácie

Odhaľovanie plagiátov spotrebuje veľa procesorového času, pri rozsiahlych prácach aj niekoľko hodín, na porovnanie jednotlivých prác. Veľmi nepríjemným zistením môže byť, že sa aplikácia neplánovane ukončila. Zväčša sa to stáva na serveroch, kde je obmedzený procesorový čas. Práve preto som navrhol spôsob, akým je možné pokračovať v porovnávaní po skončení aplikácie.

Prvým krokom bolo uložiť informácie, ktoré sa vytvorili pri tvorbe porovnávacej štruktúry. Neskôr samotné porovnanie si ukladá už porovnané práce do súborov. Vychádzal som z princípu porovnávania v časti 4.3, kde sa vezme funkcia prvej práce a porovná sa so všetkými neporovnanými prácami. Ak sa nájde podobnosť funkcií uložia sa medzi výsledky druhej funkcie do súboru. Výsledky prvej funkcie, rozdelenie plagiátov do skupín a číslo poslednej porovnanej práce sa uložia až po skončení porovnania projektu. Pri obnove porovnania sa

začne porovnávať od poslednej úplne porovnanej práce, čím sa program dostane do stavu pred svojim neplánovaným ukončením.

Každý projekt má svoj vlastný súbor s porovnávacou štruktúrou. Skupiny plagiátov spolu s indexom poslednej porovnanej práce sú ukladané do samostatného súboru. Týmto sa zabránilo ukladaniu veľkého množstva informácií a zrýchlilo sa porovnanie.

4.7 Výpis výsledkov

S ohľadom na prehľadnosť som navrhol štruktúrovaný výpis výsledkov, ktorý poskytuje informácie potrebné k dokázaniu plagiátorstva.

Ak sa v danej práci nachádza aspoň jedna funkcia, považovaná za plagiát, tak sa práca vypíše do súboru `!plagiator.txt`, kde spolu s loginom študenta sa vypíše aj zoznam skupín plagiátorov. Každá skupina obsahuje funkcie veľmi podobné. Potom nasleduje podrobný výpis jednotlivých skupín, ktorý obsahuje názov funkcie, výsledky testov v percentách a názov modulu s číslami riadkov, kde sa daná funkcia nachádza. Práce, neobsahujúce ani jednu funkciu označenú za plagiát, sa vypisujú do súboru `!nonplagiator.txt`. Príklad `!plagiator.txt`:

```
xlogin06 - PLAGIATOR!
------------------------
    LPG: 1    choice()
    LPG: 1    error()

xlogin00 - PLAGIATOR!
------------------------
    LPG: 1    vyber()
    LPG: 2    error()

ID: 1  Login    Function             BASE    BODY          LINE
-------------------------------------------------------------------------
->   xlogin06    choice                                 server.c(173 - 224)
     xlogin00    vyber                 91%    100%      tcpserver.c(63 - 123)

ID: 2  Login    Function             BASE    BODY          LINE
-------------------------------------------------------------------------
->   xlogin06    error                                  error.c(8 - 16)
     xlogin00    error                100%    100%      error.c(8 - 16)
```

Príklad `!nonplagiator.txt`:

```
xlogin05 - OK
```

Kapitola 5

Výsledky a zhodnotenie práce

5.1 Testovanie a štatistiky

Aplikácia bola testovaná na projektoch z Formálnych jazykov a prekladačov (IFJ) a zo Základov programovania (IZP). Všetky testy prebiehali na operačnom systéme FreeBSD 7.0 s konfiguráciou Intel Centrino 1,73 GHz a 1GB ram. Pre úplnú prenositeľnosť bola aplikácia vyskúšaná aj na serveroch *eva* (FreeBSD 6.3) a *merlin* (CentOS 64bit Linux). Pomocou nástroja *valgrind* boli odstránené všetky problémy s dynamickou pamäťou.

5.1.1 IZP

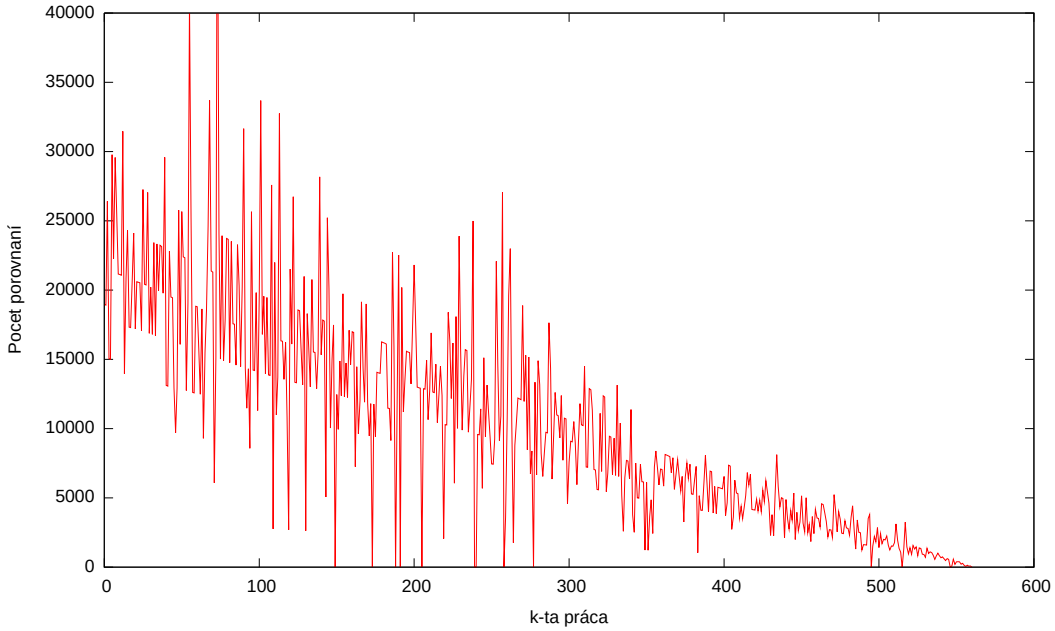
Predmet Základy programovania ponúka vhodné projekty na otestovanie aplikácie. Predmet sa vyučuje v prvom ročníku, študenti nie sú skúsení a často kopírujú svoje práce. Projekty sú krátke a jednoduché. Testoval som na nich rôzne úrovne prísnosti a pozoroval koľko sa odhalilo plagiátorov.

Tabuľka 5.1 ukazuje koľko, ktorá časť porovnávania trvá. Môžeme tiež pozorovať rozdiel medzi počtom porovnaní, kedy by sa porovnávali už porovnané funkcie, a počtom skutočných porovnaní. Práve graf 5.1 ukazuje priebeh počtu porovnania, kde je dobre vidieť ako počet porovnaní klesá v závislosti na počte porovnaných prác. Výkyvy poukazujú na to, ako sa mení počet porovnaní pri jednotlivých projektoch. V prípade, že v projekte boli všetky funkcie označené ako plagiát, tak počet porovnaní je nula a naopak.

Testy prebehli na rôznych úrovniach prísnosti, postupne 80%, 85%, 90%, 95% a 99%, po porovnaní sa odstránili lokálne skupiny plagiátorov obsahujúce okopírované funckie z kostry prvého projektu. Počet plagiátorov hneď po porovnaní a po následnom ručnom odstránení lokálnych skupín zobrazuje graf 5.2. Po námatkovom ručnom porovnaní toho, čo našla aplikácia, zisťujem, že aplikácia našla podobnosť správne. O tom, či sa skutočne jedná o plagiát alebo nie, musí rozhodnúť kompetentná osoba.

	Čas [s]	[%]
Preprocesor	0,13	0,02
Tvorba porov. štruktúry	9,41	1,46
Porovanie (Base + Body)	635,18 (22,24 + 567,62)	98,52
Spolu	644,71 (11 min)	100
Teoretický počet porovnaní: **6 450 927**		
Reálny počet porovnaní: **5 785 840**		

Tabuľka 5.1: Porovnanie IZP, base: 80% a body: 80%



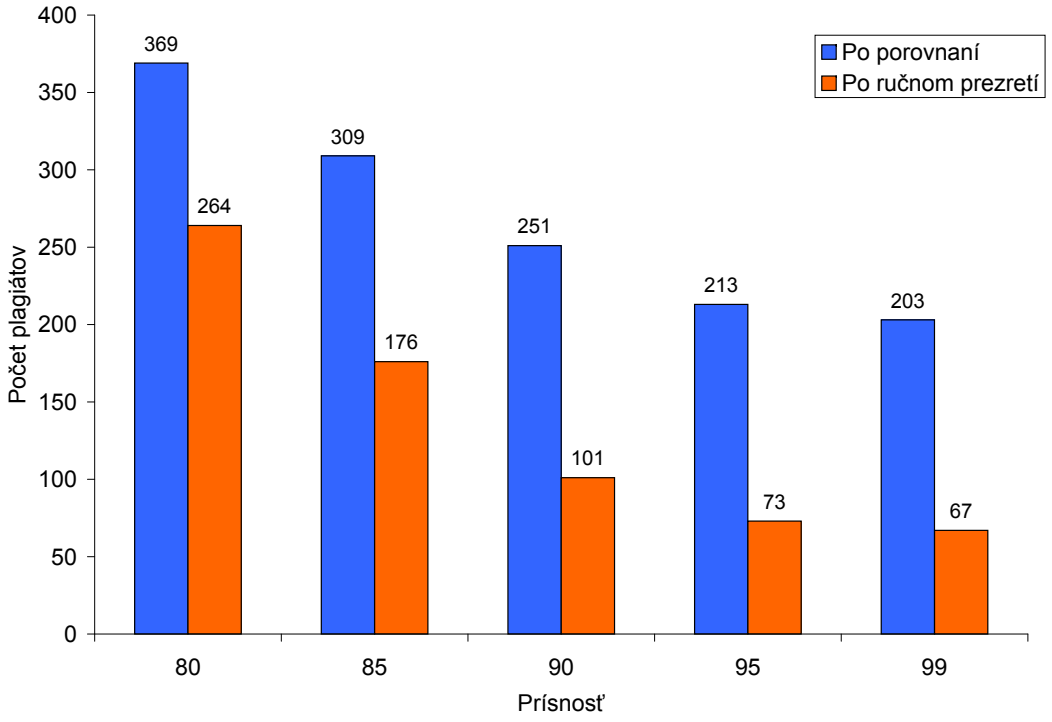
Obrázok 5.1: Graf priebehu počtu porovnaní pri IZP(80%,80%)

5.1.2 IFJ

Projekty z IFJ sú veľmi rozsiahle, väčšina sa pohybuje okolo 5000 riadkov. Nájdenie plagiátora v týchto projektoch je takmer nemožné, pretože funkcie implementujú konečný automat, čo znamená funkciu aj na tisíc riadkov. Našli sa zhody napríklad pri implementácii zásobníka, lineárneho zoznamu, *hash* tabuľky a pri algoritmoch triedenia. Projekty veľmi dobre poslúžili na odstránenie chýb pri tvorbe porovnávacích štruktúr.

5.2 Rozšírenia do budúcnosti

Aplikácia je implementovaná tak, aby bola ľahko rozšíriteľná. Do budúcna by som navrhoval tieto rozšírenia:



Obrázok 5.2: Graf počtu plagiátorov na rôznych úrovniach prísnosti

- Tvorba porovnávacích štruktúr by mohla byť založená na gramatike jazyka C/C++, čo by umožňovalo spracovať všetky dostupné konštrukcie týchto jazykov. Pre spracovanie gramatík by bolo vhodné použiť nástroje *lex* a *yacc*.
- Paralelizovať algoritmus porovnávania
- Štatisticky určiť mieru celkového plagiátorstva, napríklad pomocou prieniku
- Navrhnúť vhodné užívateľské rozhranie
- Ukladať výsledky aj porovnávacie štruktúry do databázy a navrhnúť informačný systém pre spracovanie plagiátov

Kapitola 6

Záver

Cieľom bakalárskej práce bolo navrhnúť spôsob, akým odhaliť plagiátorov v programovom kóde. Bolo treba prekonať mnoho prekážok, no napriek tomu môžem povedať, že cieľ sa podarilo splniť. Detekcia však nie je stopercentná, ale pomocou možných vylepšení je možné v budúcnosti sa k tejto hranici aspoň priblížiť.

Po podrobnej analýze konštrukcií jazykov C a C++ som navrhol systém odhaľovania plagiátov na základe dvoch testov, ktoré spolu tvoria porovnávacie jadro systému. Štatistický test sa vyhodnocuje váženým priemerom a „Body“ test je založený na hľadaní najdlhšej spoločnej podpostupnosti. Pred samotným porovnaním bolo potrebné zdrojový kód pripraviť pomocou preprocesora, lexikálnej analýzy a tvorby porovnávacej štruktúry. Tokenizácia razantne zmenšila veľkosť kódu, čo prispelo k urýchleniu porovnania.

V konečnom dôsledku, aj keď bola vykonaná automatická detekcia plagiátov, stále je potrebný zásah človeka k dokázaniu viny, či vyneseniu posledného verdiktu nad plagiátorom.

Literatúra

[1] ŠALPLACHTA, Pavel. *Aplikace pro odhalování plagiátů*. [s.l.], 2007. 39 s. FIT VUT v Brne. Vedúci bakalárskej práce Ing. Roman Lukáš, Ph.D.

[2] *ISO/IEC 9899: Programming languages - C* [online]. 2005 [cit. 2008-05-03]. Dostupné z WWW: <http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1124.pdf>

[3] *ISO/IEC 14882: Programming languages - C++* [online]. 2003 [cit. 2008-05-03]. Dostupné z WWW: <http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2007/n2461.pdf>

[4] PODLUBNÝ, Igor, HOROVČÁK, Pavel. *Úvod do programovania v jazyku C* [online]. 1998 [cit. 2008-05-03]. Dostupné z WWW: <http://people.tuke.sk/igor.podlubny/C/>.

[5] PERINGER, Peter. *Seminář C++* [online]. 2004 [cit. 2008-05-03]. Dostupné z WWW: <http://www.fit.vutbr.cz/study/courses/ICP/public/Prednasky/ICP.pdf>.

[6] PRATA, Stephen. *Mistrovství v C++*. [s.l.]: Computer Press, 2001. 1028 s. ISBN 80-7226-339-0.

[7] MATSUI, Kiyoshi. *High quality C preprocessor mcpp* [online]. 2008 [cit. 2008-05-03]. Dostupné z WWW: <http://garr.dl.sourceforge.net/sourceforge/mcpp/mcpp-summary-27.pdf>.

[8] DREYFUS, Stuart. *Richard bellman on the birth of dynamic programming* [online]. 2002 [cit. 2008-05-03]. Dostupné z WWW: <http://www.wu-wien.ac.at/usr/h99c/h9951826/bellman_dynprog.pdf>.

[9] CORMEN, Thomas H., et al. *Introduction to Algorithms*. 2nd edition. [s.l.]: MIT Press, 2002. 1143 s. ISBN 0-262-03293-7.

Prílohy

Manuálová stránka programu

NÁZOV

sim-code — detekcia plagiátov v programovom kóde

POUŽITIE

`sim-code -dir [adr] [možnosti]`

POPIS

Program **sim-code** detekuje plagiáty v programovom kóde. Projekty sa nachádzajú v adresári *adr*, každý projekt má svoj adresár, podľa svojho loginu, so zdrojovými textami. Výstupom sú súbory `!plagiator.txt` a `!nonplagiator.txt`.

MOŽNOSTI

- -dir *adresár*

  Adresár s projektmi.

- -recovery

  Obnova aplikácie pri jej neplánovanom ukončení.

- -deleteG *skupiny*

  Vymaže všetky lokálne skupiny plagiátorov. Čísla skupín v úvodzovkách oddelené medzerou. (max. 32 skupín)

- -cppmode

  Prepne porovnávanie na jazyk C++

- -h, –help

  Výpíše nápovedu.

PRÍKLAD POUŽITIA

```
sim-code -dir ./test

sim-code -dir ./test -cppmode

sim-code -dir ./test -recovery

sim-code -dir ./test -deleteG "1 2 3"

sim-code --help
```

AUTOR

Kačic Matej <xkacic00@stud.fit.vutbr.cz>