VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

NÁSTROJ NA VIZUALIZACI PLAGIÁTŮ V RŮZNÝCH PROGRAMOVACÍCH JAZYCÍCH

TOOL FOR VISUALIZATION OF PLAGIARISM IN SEVERAL PROGRAMMING LANGUAGES

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. MICHAL BANČÁK**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. ZBYNĚK KŘIVKA, Ph.D.**
SUPERVISOR

BRNO 2019

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav informačních systémů (UIFS) Akademický rok 2018/2019

Zadání diplomové práce

Student: **Bančák Michal, Bc.**
Program: Informační technologie Obor: Informační systémy
Název: **Nástroj na vizualizaci plagiátů v různých programovacích jazycích**
Tool for Visualization of Plagiarism in Several Programming Languages
Kategorie: Překladače

Zadání:

1. Seznamte se s ANTLR pro generování abstraktního syntaktického stromu (AST) pro daný zdrojový kód.
2. Identifikujte a popište různé plagiátorské techniky v různých programovacích jazycích.
3. Podle pokynů vedoucího navrhněte detekci podobných oblastí na základě AST dvou zadaných projektů a navrhněte vhodný způsob vizualizace těchto podobností uživateli. V návrhu uvažujte také různé možnosti označení částí bez provádění detekce podobnosti (např. vypuštění zadaných částí).
4. Navržený nástroj implementujte a uživatelsky testujte pro alespoň tři jazyky (např. C, PHP a Python 3) a všechny identifikované plagiátorské techniky.
5. Implementovaný nástroj spolu se samotnou vizualizací porovnejte s existujícími nástroji pro zobrazení rozdílů ve zdrojových textech a nastiňte další možný vývoj nástroje.

Literatura:

- Ondřej Krpec: Rozpoznání plagiátů zdrojového kódu v jazyce PHP, bakalářská práce, Brno, Fakulta informačních technologií VUT v Brně, 2015
- Terence Parr: *The Definitive ANTLR 4 Reference*. 2nd Edition, Pragmatic Bookshelf, 2013
- dle pokynů vedoucího

Při obhajobě semestrální části projektu je požadováno:

- Body 1 až 3.

Podrobné závazné pokyny pro vypracování práce viz http://www.fit.vutbr.cz/info/szz/

Vedoucí práce: **Křivka Zbyněk, Ing., Ph.D.**
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2018
Datum odevzdání: 22. května 2019
Datum schválení: 28. října 2018

Abstrakt

Práca sa zaoberá návrhom a implementáciou nástroja na detekciu plagiátov v programovacích jazykoch C, Python a PHP. Popisuje techniky, ktoré slúžia na zakrytie plagiátorstva. Cieľom práce je vytvoriť návrh nástroja na detekciu a vizualizáciu plagiátov, ktoré využívajú tieto techniky a jeho následnú implementáciu. Nástroj vykonáva detekciu transformáciou zadaných vstupných projektov do formy abstraktného syntaktického stromu, ktorý je získaný pomocou lexikálnej a syntaktickej analýzy, ktoré bude následne porovnávať navrhnutým algoritmom, ktorý využíva ohodnocovanie uzlov a podstromov pomocou *hash* funkcie. Nástroj taktiež nájdené časti kódu, u ktorých mohlo potenciálne prísť k plagiátorstvu, vizualizuje vo forme podstromu abstraktného syntaktického stromu, príslušnému danej nájdenej časti kódu. Práca ďalej popisuje testovanie tohto nástroja na identifikovaných plagiátorských technikách a špecifikuje, ktoré dokáže pri detekcii obísť. Ďalej práca načrtáva možný ďalší vývoj nástroja.

Abstract

The thesis describes the design and implementation of a plagiarism tool for programming languages C, Python and PHP. It describes techniques that are used to cover a plagiarism. The aim of this work is to create a tool for detection and visualization of plagiarisms covered up using these techniques. The tool performs detection by transforming input projects into an abstract syntactic tree, which is obtained by lexical and syntactic analysis. These trees will be compared by a proposed algorithm that uses node and subtree valuation using the hash function. The found parts of the code that could potentially lead to plagiarism are visualized in the form of a subtree of an abstract syntactic tree that represents the parts of the code found by the tool. Further, the work describes testing of this tool on identified plagiarism techniques and specifies which of them it can eliminate. In its conclusion, the work describes the possible further development of the tool.

Kľúčové slová

Detekcia plagiátov, vizualizácia plagiátov, hash funkcia, ANTLR, stromový prístup, abstraktný syntaktický strom.

Keywords

Plagiarism detection, plagiarism visualization, hash function, ANTLR, tree approach, abstract syntax tree.

Citácia

BANČÁK, Michal. *Nástroj na vizualizaci plagiátů v různých programovacích jazycích.* Brno, 2019. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Zbyněk Křivka, Ph.D.

Nástroj na vizualizaci plagiátů v různých programovacích jazycích

Prehlásenie

Prehlasujem, že som túto prácu vypracoval samostatne pod vedením pána Ing. Zbyňka Křivky Ph.D., uviedol som všetky literárne pramene a publikácie, z ktorých som čerpal.

.....
Michal Bančák
22. mája 2019

Poďakovanie

Rád by som poďakoval pánovi Ing. Zbyňkovi Křivkovi Ph.D. za veľkú trpezlivosť a veľa užitočných rád a pripomienok k mojej práci.

Obsah

Kapitola 1

Úvod

Pojem plagiát je čoraz viac používaný a nie len na akademickej pôde. Plagiát je podľa slovenského slovníka [15] definovaný ako umelecké, literárne alebo vedecké dielo, na ktorého vytvorenie bola použitá časť, alebo celé dielo iného autora, bez uvedenie zdroja a teda v snahe túto časť, alebo celé dielo, vydávať za svoje a zatajiť pôvod. Plagiátorstvo patrí k najhorším priestupkom na akademickej pôde.

Odhalenie plagiátu je veľmi náročný proces. Je nutné nájsť zhodu alebo podobnosť, ktorá ešte nie nutne musí značiť plagiát. V niektorých prípadoch je dokonca podobnosť očakávaná. Následne aj po nájdení zhody je nutné preukázať, že k plagiátorstvu prišlo. Avšak pri jednoduchých zmenách v diele je pre človeka ešte náročnejšie podobnosť nájsť. Preto je snaha o vytvorenie nástroja, ktorý by na základe istých metód, dokázal zjednodušiť odhalenie plagiátu.

V tejto práci je zameriavané plagiátorstvo v programovacích jazykoch, teda odhaľovanie plagiátov v zdrojových kódoch. Jedná sa o náročnejšiu úlohu v porovnaní s písaným textom. Zdrojový kód je pre ľudské oko menej prirodzený a teda pri nie veľmi veľkých zmenách v kóde je podobnosť čoraz menej pre človeka viditeľná.

Táto práca je organizovaná nasledovne. V kapitole 2 sú popísané možné transformácie kódu na zastieranie skutočnosti, že došlo k plagiátorstvu. Tiež táto kapitola popisuje obecný proces a rozdelenie spôsobov samotnej detekcie. Na konci tejto kapitoly sa zhrnú existujúce nástroje. Kapitola 3 potom bude popisovať návrh aplikácie na vizualizáciu a detekciu plagiátov v rôznych programovacích jazykoch. V kapitole 4 sa nachádza popis následnej implementácie navrhnutého nástroja, teda mapovanie návrhu do zdrojového kódu. Táto kapitola popisuje implementáciu prednej (front-end) a zadnej (back-end) časti. V kapitole 5 sa popisuje testovanie implementovaného nástroja, ktoré bolo interné, ale aj užívateľské. To zároveň definuje schopnosti implementovaného nástroja. Táto kapitola taktiež obsahuje porovnanie nástroja s nástrojmi zobrazujúcimi rozdiely v zdrojových kódoch a možný následný vývoj nástroja.

Kapitola 2

Plagiátorstvo v obore informačných technológií

V informatike sa pojem plagiát najčastejšie používa pri použití cudzieho zdrojového kódu, alebo jeho časti, s tým, že je vydávaný ako vlastný výtvor. Avšak v tomto obore často podobnosť, alebo úplná zhoda, nie je považovaná za plagiát. Podľa Zeydmana [21] existuje 6 dôvodov pre podobnosť zdrojových kódov a to:

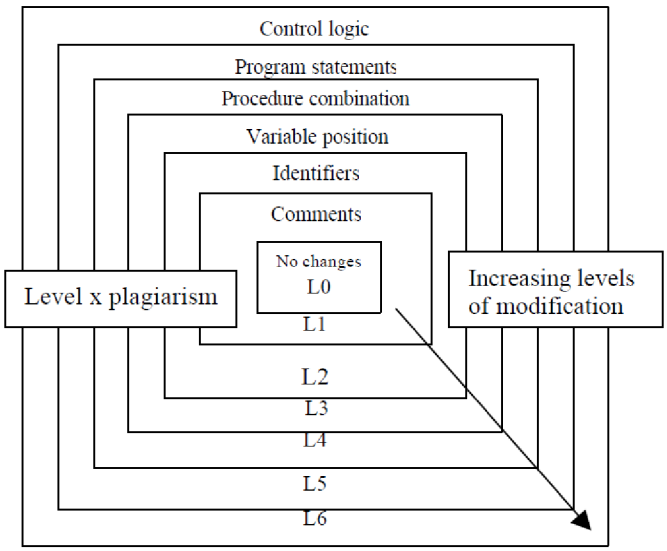
- **Zdrojový kód tretej strany** – Použitie kódu alebo knižnice tretej strany (často s voľnou licenciou).
- **Nástroje generujúce kód** – Použitie generátorov kódu často speje k podobnosti, keďže každý generátor používa rovnaké elementy.
- **Bežne používané názvy premenných** – Názvy, ktoré sa používajú v školách a bežne aj v praxi, napr. "tmp", "result", atď.
- **Všeobecne známe algoritmy** – Niektoré algoritmy sú k verejne k dispozícii, a preto ich využitie vedie k väčšej podobnosti.
- **Jeden autor** – Každý programátor má svoj štýl programovania a teda pri vytvorení rôznych programov vedie k podobnosti kódov, aj keď sa jedná o odlišné programy s odlišnou funkcionalitou. Pre vyhnutiu sa nepríjemnostiam je vhodné použitie vlastného kódu z iného programu tiež citovať. Je tak možné predísť seba-plagiátorstvu (angl. self-plagiarism).
- **Skopírovanie kódu** – Použitie cudzieho kódu zvyšuje podobnosť. Rozlišuje sa však použitie autorizované a neautorizované. Autorizované použitie je také, kde je uvedený zdroj, prípadne odkaz a autor.

Zo všetkých týchto možností je práve posledná tá, ktorú považujeme za plagiátorstvo. Pri neautorizovanom použití sebou nevytvoreného kódu, dochádza prakticky ku krádeži duševného vlastníctva (angl. intellectual property).

2.1 Transformácie pri plagiátorstve

V plagiátorstve sa zväčša nejedná len o skopírovanie kódu. Keďže v drvivej väčšine ide o úmyselné plagiátorstvo, je nutné skopírovaný kód upraviť, pretože čistú kópiu kódu je

v dnešnej dobe priveľmi jednoduché odhaliť, napr. pomocou nástroja na porovnanie súborov na zmeny. Takéto úpravy potom vytvárajú dojem odlišnosti a zväčšujú náročnosť odhalenia. Transformácie môžu byť jednoduché (zmena komentárov, názvov premenných), ale môžu byť aj zložité (zmena *while* cyklu na *for*, zmena postupnosti v príkaze *if*). Takéto rozdelenie do vrstiev zaviedol Fadhi v [5], ako ilustruje obrázok 2.1.



Obr. 2.1: Rozdelenie vrstiev modifikácií plagiátov. [5]

Whale [19] vo svojej práci popísal šesť metód zastieracích modifikácii plagiátov:

- Zmena komentárov (zhodné s vrstvou **L1**)
- Zmena dátových typov
- Zmena identifikátorov (zhodné s vrstvou **L2**)
- Pridanie opakujúcich sa a nepotrebných príkazov alebo premenných
- Zmena štruktúry podmienených príkazov
- Skombinovanie skopírovaných a vlastných príkazov

Je možné si všimnúť, že tento zoznam metód je podobný rozdeleniu do vrstiev na obrázku 2.1. Tieto zastieracie metódy sa môžu v rôznych programovacích jazykoch mierne líšiť, no v obecnom hľadisku sa dá povedať, že tieto techniky sú identifikované a univerzálne pre jazyk pre obecné použitie (angl. general purpose).

2.2 Obecný proces detekcie plagiátov

Plagiát je určitým spôsobom skopírovaný kód. Preto sa v niektorých literatúrach často používa pojem klon, ktorý značí, že určité časti kódu sú si podobné, čo znamená, že sa jedná o možný plagiát. Existujú 4 typy klonov. Pre ich popis je však nutné zadefinovať základné pojmy [16].

Kódový fragment (KF) – Je akákoľvek sekvencia riadkov kódu. Môže obsahovať komentáre a môže sa skladať z rôznych typov príkazov napr. telo funkcie, jej definícia alebo sa môže skladať len zo sekvencie príkazov nachádzajúcej sa na rôznom mieste v kóde. Je definovaná názvom súboru, začiatočným a koncovým riadkom.

Kódový klon – Fragment KF1 je klonom fragmentu KF2, ak pre nejakú funkciu podobnosti sú podobné, teda pre funkciu f (popísané nižšie) platí $f(KF1) = f(KF2)$. Dva kódové fragmenty, ktoré sú si navzájom podobné, vytvárajú tzv. *klonový pár (CF1,CF2)*. Keď je veľa fragmentov vzájomne podobných, vytvárajú *množinu klonov*.

Typy klonov – Existujú dva hlavné typy podobnosti medzi kódovými fragmentami a to fragmenty, ktorých podobnosť spočíva v zdrojovom texte kódu, alebo fragmenty, ktorých podobnosť spočíva vo funkcionalite, čiže samotný kód je textovo odlišný. Textová podobnosť klonov je veľmi často výsledkom skopírovania fragmentu a vloženia ho na iné miesto. Nasledujúce typy klonov sú založené ako na funkcionálnej tak aj na textovej podobnosti.

- **Typ 1** – Identické kódové fragmenty, s výnimkou bielych znakov, usporiadania kódu a komentárov
- **Typ 2** – Syntakticky identické kódové fragmenty, s výnimkou identifikátorov, reťazcov, dátových typov, bielych znakov, usporiadania kódu a komentárov
- **Typ 3** – Skopírované fragmenty so zložitejšími úpravami ako zmena, pridanie alebo odstránenie príkazov, ale bez zmien identifikátorov, reťazcov, dátových typov, bielych znakov, usporiadania kódu a komentárov
- **Typ 4** – Dva alebo viac kódových fragmentov, ktoré vykonávajú rovnaký výpočet, ale sú implementované rôznymi syntaktickými variantami.

Tieto typy klonov sú teda zhrnutím predchádzajúcich metód modifikácií na začiatku kapitoly.

Proces detekcie klonov sa teda musí pokúsiť nájsť kúsky kódov, ktoré majú vysokú podobnosť. Problém nastáva v tom, že nie je možné dopredu zistiť, ktoré kódové fragmenty sa môžu opakovať. Tým pádom by detekcia mala prebiehať tak, že sa porovnáva každý možný kódový fragment s každým iným možným fragmentom, čo je z hľadiska časovej náročnosti veľmi neefektívne riešenie. Niekoľkými opatreniami je snaha o zníženie oblasti porovnávania ešte predtým, než sa samotné porovnávanie vykonáva. Na to slúžia základné kroky detekcie [16], ktoré sú vyobrazené na obrázku 2.2. Je dôležité, že sa jedná o náčrt/návrh takého nástroju, avšak nie je nevyhnutné, aby všetky tieto fáze detekcie boli nutne zahrnuté v každom nástroji. Tak isto ich poradie nie je nevyhnutne rovnaké a často dochádza k modifikácii.

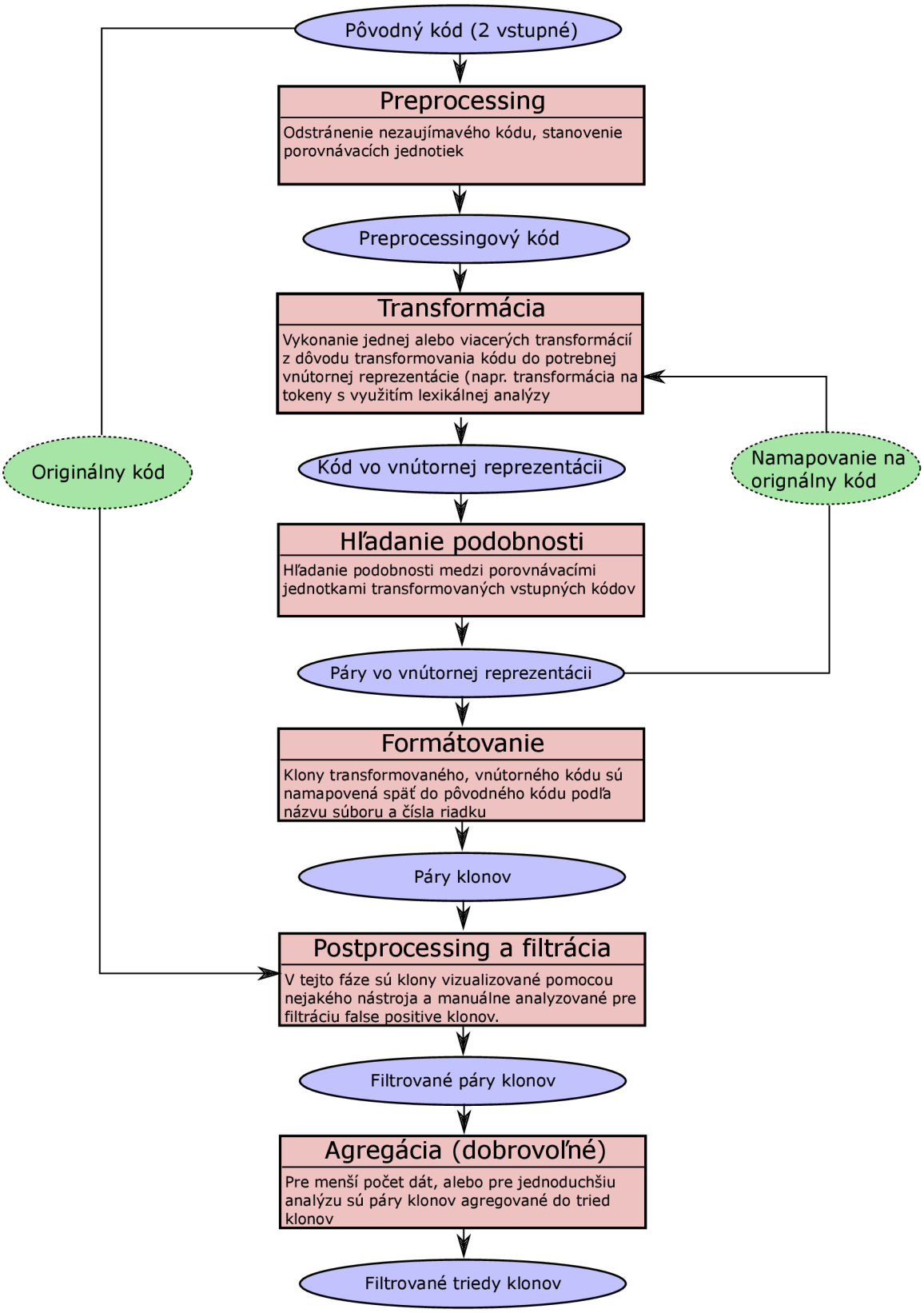
2.2.1 Predspracovanie

Na začiatku každej detekcie sú nutné počiatočné úpravy (angl. preprocessing). Táto časť sa skladá z troch hlavných úloh:

Odstránenie nezaujímavých častí – Každý zdrojový kód, alebo jeho časť, nezaujímavá pre porovnávanie je odstránená. Napríklad, ak sa program skladá z viacero jazykov (napr. SQL a Java) a detekcia je závislá na jazyku. Odstránia sa tiež kódy vygenerované generátormi, ktoré by spôsobovali priveľa *false positive* klonov. V prípade potreby je možné odstrániť nezaujímavé časti kódu aj neskôr.

Určenie zdrojových jednotiek – Po odstránení nezaujímavých častí kódov, je zostávajúci kód rozdelený na disjunktné kódové fragmenty, ktoré sa tak stávajú zdrojovými jednotkami.

Určenie porovnávacích jednotiek – Môže byť nutné, aby zdrojové jednotky boli ďalej rozdelené do ešte menších jednotiek v závislosti na použitej technike porovnávania. Napríklad môžu byť zdrojové jednotky rozdelené na riadky alebo aj na tokeny na porovnanie.



Obr. 2.2: Obecný proces na detekciu plagiátov.

Porovnávacie jednotky môžu byť získané aj zo syntaktickej štruktúry zdrojových jednotiek (*if* rozdelený na podmienku, blok *then* a blok *else*). V niektorých prípadoch môžu byť priamo zdrojové jednotky porovnávacími.

2.2.2 Transformácia

Ak sa jedná o inú ako textovú techniku porovnávania, sú často porovnávacie jednotky transformované do príslušnej vnútornej reprezentácie pre porovnávanie. Táto transformácia sa nazýva *extrakcia*.

V niektorých prípadoch môže po extrakcii nasledovať ešte normalizačná transformácia, ktorá slúži na odhalenie jednoduchých klonov. Normalizácia sa môže skladať z jednoduchých transformácií (odstránenie bielych znakov), ale aj komplexnejších (reorganizácia zdrojového kódu). Normalizácia môže predchádzať aj nasledovať extrakciu vnútornej reprezentácie.

Extrakcia

Extrakcia prevádza zdrojový kód do vhodnej formy, ktorá je kompatibilná so vstupom používaného algoritmu na porovnávanie. Extrakcia je teda plne závislá na používanej technike detekcie. Väčšinou nadobúda jednu z týchto foriem:

Tokenizácia – V prípade tokenovo zameraných techník detekcie, každý riadok zdrojového kódu je premenený na reťazec tokenov podľa príslušných lexikálnych pravidiel zdrojového jazyka. Týmto sú odstránené všetky biele znaky a komentáre.

Syntaktická analýza – Jedná sa o typ, ktorý sa využíva v syntakticky založených prístupoch. Celý zdrojový kód je prehnaný cez syntaktický analyzátor, kde výstupom je derivačný strom (parse tree), poprípade, po miernych úpravách, aj abstraktný syntaktický strom (AST). Zdrojové jednotky potom predstavujú podstromy derivačného stromu, alebo AST. Porovnávanie spočíva v hľadaní podobných podstromov. Prístupy založené na metrikách tiež môžu používať derivačný strom ako reprezentáciu na nájdene klonov podľa metriky pre podstromy. Pre túto prácu používaný typ extrakcie je práve syntaktická analýza.

Analýza toku riadenia a dát – Sémanticky založené prístupy generujú graf závislostí programu (PDGs) zo zdrojového kódu. Uzly tohto grafu reprezentujú príkazy a podmienky programu, zatiaľ čo hrany riadiace a dátové závislosti. Zdrojové jednotky na porovnávanie sú potom podgrafy.

Normalizácia

Ako už bolo spomenuté vyššie, normalizácia je voliteľný krok. Jedná sa o krok, ktorý zaisťuje odstránenie primitívnych zmien v bielych znakoch, komentároch a identifikátoroch, ktoré majú za úlohu zakryť plagiátorstvo.

Odstránenie bielych znakov – Takmer všetky prístupy ignorujú biele znaky. Môžu sa však vyskytnúť také, ktoré potrebujú niektoré biele znaky. Napríklad niektoré prístupy založené na metrikách môžu používať formátovanie ako časť ich porovnávania.

Odstránenie komentárov – Väčšina bežných prístupov ignoruje komentáre. Avšak existujú koncepty, ktoré využívajú aj komentáre na hľadanie klonov [10].

Normalizácia identifikátorov – Väčšina prístupov normalizuje identifikátory pred porovnávaní z dôvodu identifikovania klonov **Typu 2**. Obecne platí, že všetky identifikátory sú v zdrojovom kóde nahradené jedným univerzálnym.

Ďalšie možné transformácie – Ďalšie transformácie môžu zahŕňať nejaký typ reorganizácie zdrojového súboru, napríklad samotnú štruktúru programu.

2.2.3 Hľadanie podobnosti a formátovanie

V tejto časti sa kód vo vnútornej forme vloží na vstup porovnávaciemu algoritmu, kde sú navzájom porovnávané porovnávacie jednotky. Výstupom porovnávania je zoznam množín kandidátnych párov klonov. Konkrétne porovnávanie záleží na prístupe a voľbe algoritmu.

Vo formátovacej fáze sa potom jednotlivé páry klonov v transformovanom kóde získané porovnávaním konvertujú na páry klonov v pôvodnom kóde. V niektorých prípadoch sa páry získané porovnávaním mapujú na ich pozíciu v zdrojovom súbore.

2.2.4 Následné spracovanie, filtrácia a agregácia

Následne sú klony ponechané alebo odstránené na základe manuálnej analýzy alebo automatických heuristík.

- **Manuálna analýza** slúži na filtráciu kandidátnych párov mapovaných v zdrojovom kóde, ktorá rozlíši podozrivé klony a *false positive* klony. Manuálnu analýzu vykonáva vždy človek. Vhodná vizualizácia môže pomôcť urýchliť a zjednodušiť analýzu.
- **Automatické heuristiky** slúžia na automatickú filtráciu klonov. Heuristiky sú založené na dĺžke, frekvencii, atď.

Výstupom niektorých nástrojov sú triedy klonov. No niektoré nástroje ponúkajú len páry klonov. V snahe redukovať veľkosť dát, sa používa dodatočná analýza alebo štatistiky na agregáciu klonov do tried.

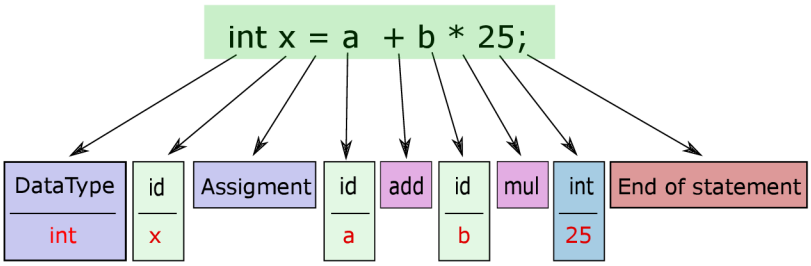
2.3 Prístupy pri detekcii klonov

Existujú rôzne prístupy na detekciu klonov. Na základe analýzy, ktorú vykonávajú na zdrojovom kóde, môžu byť rozdelené do štyroch základných kategórií a to *textové, lexikálne, syntaktické* a *sémantické* [16]. Rozlišujú sa teda tým, aký typ informácie a techník ich analýza používa.

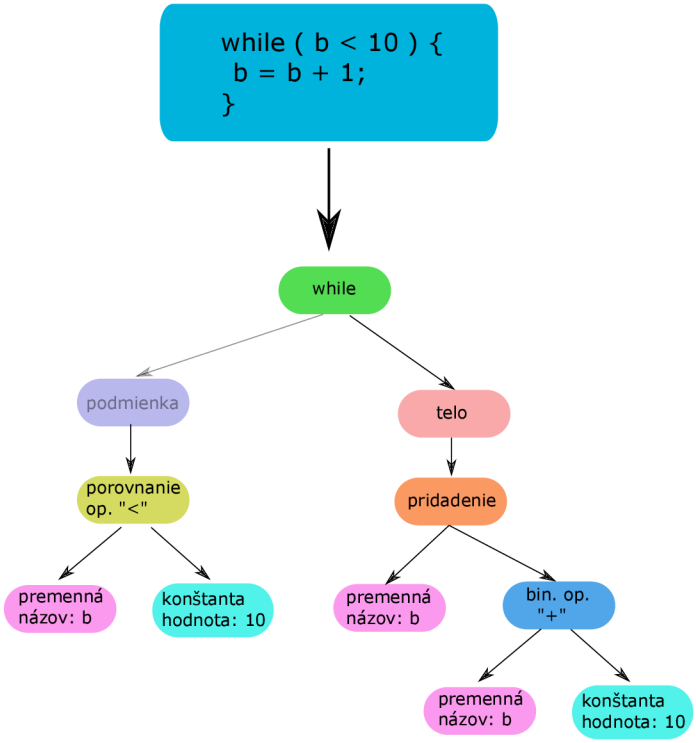
Textové prístupy alebo techniky nepoužívajú, alebo len veľmi málo, transformácie (normalizácie) zdrojového kód. Porovnávanie vykonávajú priamo na zdrojovom kóde. Používajú napríklad tzv. odtlačky (fingerprints) na časti zdrojového kódu. Porovnávané sú potom tieto odtlačky. Príkladom takéhoto prístupu je nástroj *The NiCad Clone Detector*, ktorý využíva metódu *NiCad* [4].

Lexikálne prístupy využívajú reťazce tokenov (obrázok 2.3) ako porovnávacie jednotky. Na začiatku je nutné zdrojový kód podrobiť lexikálnej analýze, ktorá vytvorí tok tokenov. Tento tok je následne rozdelený na sekvencie. Porovnávanie sa potom vykonáva tak, že sa hľadajú rovnaké sekvencie tokenov. Výstupom sú potom časti zdrojového kódu korešpondujúce práve týmto sekvenciám tokenov. Lexikálne prístupy sú všeobecne silnejšie než textové pri malých zmenách v kóde.

Syntaktické prístupy používajú syntaktickú analýzu na zisk derivačných stromov (DT) alebo, po miernej úprave, abstraktných syntaktických stromov (ďalej len AST), ktorý je vyobrazený na obrázku 2.4. Stromy sú následne porovnávacími jednotkami. Syntaktické prístupy sa globálne rozdeľujú do ďalších dvoch kategórií, a to stromové prístupy a prístupy založené na metrikách.



Obr. 2.3: Zdrojový kód transformovaný na reťazec tokenov.

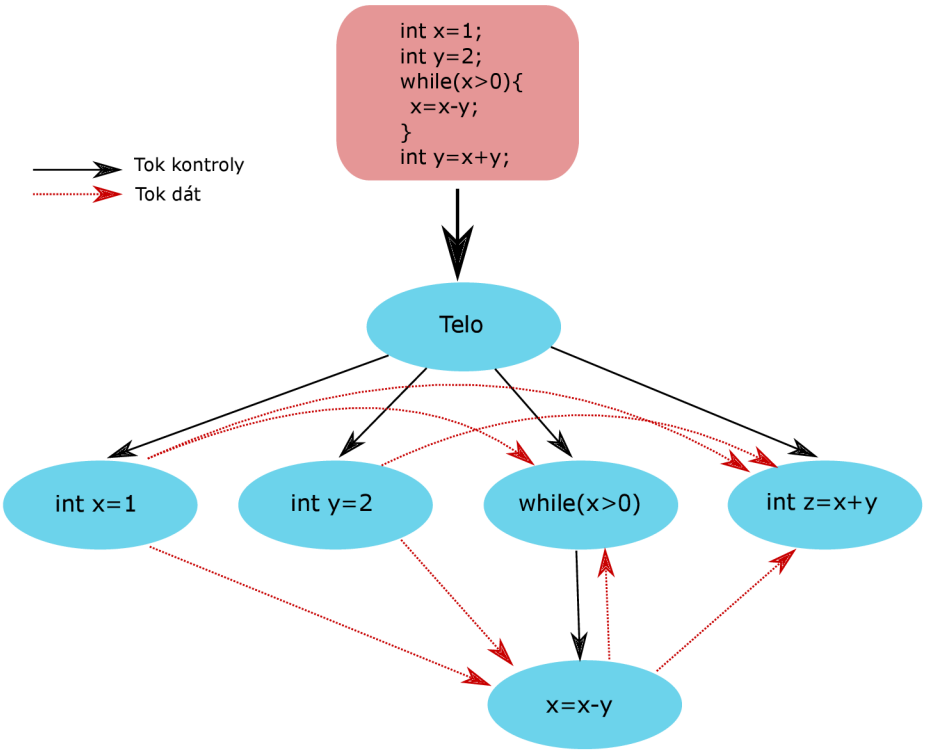


Obr. 2.4: Transformácia zdrojového kódu na abstraktný syntaktický strom pomocou syntaktickej analýzy.

- **Stromové prístupy** hľadajú klony tým, že hľadajú podobné podstromy. Názvy premenných, reťazce a ostatné sú zahrnuté v stromovej reprezentácii, čo umožňuje sofistikovanejšiu detekciu. V niektorých prípadoch sa však tieto prvky zámerne neberú v úvahu pri porovnávaní. Porovnávanie každého podstromu s každým podstromom je veľmi neefektívne pri pohľade na časovú zložitosť. Využívajú sa preto rôzne algoritmy na skrátenie tohto porovnávania. Jedným z najčastejšie využívaných metód je hashovanie.
- **Prístupy založené na metrikách** zhromažďujú metriky pre kódové fragmenty a následne porovnávajú vektory metrík a nie priamo AST alebo kód. Metriky sú však vytvorené na základe stromu. Často sa teda vytvoria metriky pre syntaktické časti ako funkcie, triedy, metódy.

Sémantické prístupy používajú statickú programovú analýzu na získanie viac presných informácii než jednoduchú syntaktickú podobnosť. Často je v týchto prístupoch

program reprezentovaný ako programový graf závislostí (obrázok 2.5). Uzly grafu vyjadrujú výrazy a príkazy, zatiaľ čo hrany reprezentujú kontrolné a dátové závislosti. V takejto reprezentácii sú výrazy a príkazy sémanticky nezávislé. Vyhľadávanie potom prebieha hľadaním izomorfných podgrafov.



Obr. 2.5: Programový graf závislostí (PDG) vytvorený zo zdrojového kódu.

Posledným spôsobom môžu byť hybridné prístupy zložené z viacerých prístupov. Najčastejším hybridným prístupom je taký, ktorý používa lexikálny prístup, teda tokenizáciu na získanie tokenov, ktoré sú následne porovnávané iným prístupom.

2.4 Prehľad existujúcich nástrojov

Na detekciu plagiátov existuje veľké množstvo algoritmov a nástrojov. Zatiaľ čo niektoré sú len na teoretickej úrovni, iné sú používané a účinné. Každý sa so zmenami v zdrojových kódoch na zahalenie plagiátu vyrovnáva inak. I keď v globálnom hľadisku sa radia medzi vyššie spomínané prístupy, zväčša používajú svoj algoritmus na detekciu. V tejto kapitole sú uvedené niektoré najviac používané (známe) alebo podobné cieľu tejto práci.

2.4.1 MOSS

MOSS [12], v preklade meranie podobnosti programov (Measure Of Software Similarity), je známy nástroj na detekciu plagiátov využívaný rôznymi univerzitami po celom svete. Bol vyvinutý v roku 1994. Využíva lexikálny prístup na získanie toku tokenov, ktoré následne porovnáva pomocou odtlačkov. Na to je využívaný algoritmus **Winnowing** [17]. MOSS podporuje veľkú škálu jazykov. MOSS nebol donedávna verejne prístupný a slúžil iba pre

inštruktorov a vedúcich kurzov programovania. Každopádne dnes je to už verejný nástroj a po získaní účtu je možné si tento nástroj stiahnuť.

2.4.2 CodeMatch

CodeMatch [3] je nástroj, ktorý sa sústreďuje na porovnávanie veľkého počtu zdrojových súborov. Užívateľ si môže zvoliť, aký typ porovnávania sa má vykonať. Umožňuje porovnávať príkazy, komentáre, identifikátory a postupnosť inštrukcií. Má taktiež podporu pre veľký počet jazykov.

CodeMatch produkuje výstup ako databázu, ktorá môže byť exportovaná do HTML. Výstupom sú hodnoty, ktoré zobrazujú stupeň korelácie. Nezobrazujú tak dôvod prečo sú si zdrojové kódy podobné, ale len ako sú si podobné. Každopádne to umožňuje pri hľadaní plagiátov vylúčiť nepodobné súbory a sústrediť sa len na tie, u ktorých je podobnosť podozrivá.

Veľkou výhodou je, že autori vyzývajú užívateľov, že v prípade, ak užívateľ potrebuje podporu pre jazyk, ktorý aktuálne nie je podporovaný, je jednoduché podporu doplniť za niekoľko dní. Nevýhodou je, že sa jedná o platený produkt.

2.4.3 AC2

AC2 [1] je nástroj, ktorý je v určitom ohľade podobný nástroju *CodeMatch*. Jeho autorom je Manuel Freire a bol vytvorený v Madride.

Dokáže porovnávať veľké množstvo súborov a výsledkom je hodnota podobnosti. Avšak podobnosť jednotlivých súborov zobrazuje aj graficky. Dokáže tak užívateľovi zobraziť, ktorá skupina zdrojových súborov má podozrivú podobnosť [6]. Tak isto umožňuje zobraziť dvojicu súborov a podobnosť ukázať vyznačenú priamo v zdrojovom kóde.

Nástroj AC2 na získanie podobnosti využíva metódu normalizovanej kompresnej vzdialenosti [2]. Táto metóda hľadá podobnosť podľa toho, ako zložité je previesť jeden program na druhý.

AC2 využíva nástroj na generovanie lexikálneho a syntaktického analyzátora nazývaný ANTLR (popísaný v kapitole 3), ktorý analyzátor generuje na základe gramatiky vo formáte *.g4*, čiže, tak ako autor popisuje, je jednoduché pridať podporu pre rôzne jazyky. Aktuálne AC2 má podporu pre jazyk C, C++ a Java s rôznymi štandardami týchto jazykov. Tento nástroj okrem spomínanej metódy ponúka dodatočne aj iné metódy porovnávania, napríklad porovnávanie sekvencie tokenov.

AC2 je pod licenciou *GPLv3*, čo znamená, že nástroj je verejne k dispozícii, je možné šíriť ľubovoľne jeho kópie. Taktiež je možná modifikácia, ale pod podmienkou, že aj modifikovaná/upravená/vylepšená verzia bude pod licenciou *GPLv3*, aby mohli aj túto novú verziu využívať ostatný.

Kapitola 3

Návrh nástroja na detekciu a vizualizáciu plagiátov

Návrh nástroja v tejto práci sa rozdeľuje na zadnú (angl. back-end) a prednú (front-end) časť. V zadnej časti sa rozoberá vnútorná reprezentácia, algoritmus na detekciu plagiátov a ich čiastočnú filtráciu. V prednej časti sa potom rozoberá návrh vizualizácie a grafického užívateľského rozhrania (GUI).

Nástroj bude implementovaný v jazyku Java za pomoci technológie *JavaFX*, ktorá slúži na vytvorenie GUI. Podporovanými jazykmi pre detekciu bude jazyk C, Python 3 a PHP. Návrh sa však nezameriava na konkrétny jazyk, ale ide o obecný popis, ako nástroj vyzerá.

3.1 Vnútorná reprezentácia a algoritmus detekcie

Tak ako bolo spomínané v kapitole 2, zdrojový kód je na začiatku nutné previesť do vnútornej reprezentácie. Nato, aby tento krok bol vykonaný, je nutné vykonať nevyhnutné predspracovanie, a to mať jeden zdrojový súbor. Tento nástroj však podporuje aj detekciu balíku súborov, teda projekty, ktoré je nutné spojiť. Preto prvotnou operáciou je spojenie obsahu súborov do jedného súboru.

Po získaní jedného zdrojového súboru pre každý projekt, je ďalším krokom z tohto zdrojového kódu vytvoriť abstraktný syntaktický strom (AST), ktorý bude slúžiť ako zdrojová jednotka porovnávania.

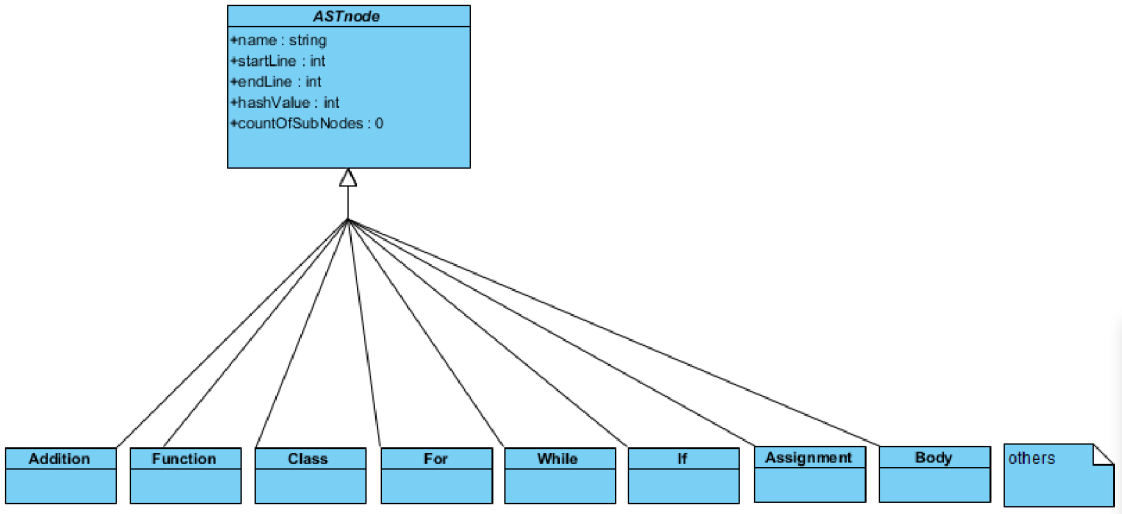
3.1.1 Abstraktný syntaktický strom ako zdrojová jednotka

V tomto prípade nástroj bude využívať syntaktický prístup, konkrétne stromový prístup. Zdrojovou jednotkou teda bude abstraktný syntaktický strom. Ako zdrojová jednotka by mohol slúžiť aj derivačný strom (priamy výstup syntaktickej analýzy), ale keďže derivačný strom obsahuje veľa neterminálov nepodstatných pre účel porovnávania, je neprehľadný a zbytočné zvyšuje náročnosť a zložitosť výpočtu. Avšak na získanie AST je nutné najprv získať derivačný strom. Na získanie tohto stromu je nutné zdrojový kód podrobiť syntaktickej analýze.

ANTLR [14], celým názvom *Ďalší nástroj na rozpoznávanie jazyka* (ANother Tool for Language Recognition) je nástroj pre jazyk Java. Jeho využitie vygeneruje zdrojový kód pre lexikálny a syntaktický analyzátor, ktorý rozpoznáva a kontroluje jazyk zadaný vstupnou gramatikou. To znamená, že tento nástroj dokáže vykonať lexikálnu a syntaktickú analýzu pre akýkoľvek jazyk zadaný gramatikou.

Jedným výstupom tohto nástroju je teda niekoľko zdrojových súborov, ktoré obsahujú zdrojový kód v jazyku Java. Následne, po preložení, slúžia na analýzu vstupného programu v jazyku danom gramatikou, podľa ktorej boli analyzátory vygenerované. Výstupom tejto analýzy je potom derivačný strom. Uzly derivačného stromu sú neterminály a terminály z gramatiky.

Samotný derivačný strom je však pre ďalšie použitie celkovo neprehľadný. Preto je vhodné, aj keď nie nutné, strom upraviť. Nato slúžia tzv. *visitore*, podľa návrhového vzoru *Visitor* (návštevník), ktoré umožňujú postupne prechádzať strom a rozlišovať jednotlivé uzly stromu. Je teda možné vytvoriť z derivačného stromu, získaného z vygenerovanej lexikálnej a syntaktickej analýzy nástrojom ANTLR, AST (príklad AST je na obrázku 2.4).



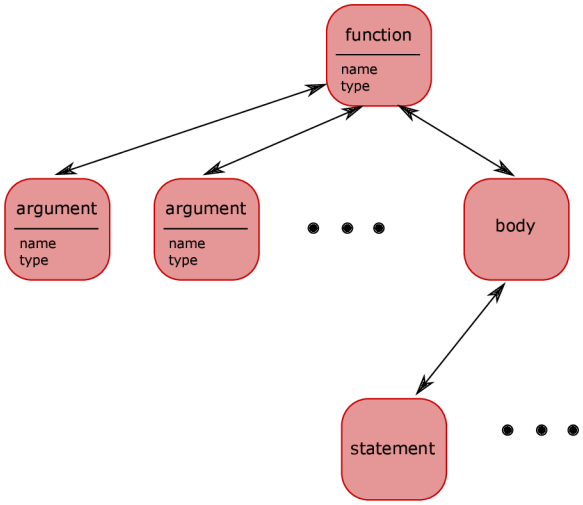
Obr. 3.1: Reprezentácia AST v diagrame tried. Každý uzol AST je objekt, ktorý reprezentuje jeho typ. Všetky tieto objekty dedia z abstraktnej triedy, čo znamená že vo výsledku je AST reprezentovaný ako kolekcia objektov rozširujúcich abstraktnú triedu.

AST je teda získaný postupným prechádzaním derivačného stromu. Nepodstatné uzly, ktoré slúžia na syntaktickú kontrolu sú ignorované a uzly podstatné pre hľadanie podobnosti sú reprezentované objektom, ktorý predstavuje typ uzlu. Základným typom je abstraktná trieda *ASTnode*, z ktorej tieto objekty dedia a strom je reprezentovaný týmito objektami, ktoré majú atribúty reprezentujúce potomkov a zároveň atribút referencie na rodiča. Takto je zaistená hierarchia stromovej štruktúry. Model tohto návrhu je vyobrazený na obrázku 3.1.

Každý spojený zdrojový kód projektu vloženého ako vstup na porovnávanie je teda transformovaný na AST. Nástroj však umožňuje nejakú časť kódu ignorovať, a to zadaním kódu do tomu pripraveného poľa. Užívateľ, ktorý chce nejakú časť kódov nezahŕňať do porovnávania, musí teda takýto kód pridať vo forme kódu. Tento kód je tak isto transformovaný do formy AST. Výslednú vnútornú reprezentáciu na abstraktnej úrovni zobrazuje obrázok 3.2.

Výsledný AST je zložený z funkcií, tried a rozhraní. Ak teda vezmeme do úvahy tieto tri typy podstromov, bez koreňového uzla, ktorý tieto stromy spája, dostaneme niekoľko stromov pre tieto tri typy. Potom množina týchto podstromov je TT pre prvý vstupný

zdrojový kód a ST pre druhý vstupný zdrojový kód. Množina všetkých uzlov stromov v TT je potom TN, pre ST je to SN. Počet uzlov stromu s najväčším počtom uzlov je TMC a SMC.



Obr. 3.2: Zdrojový kód transformovaný do AST. Každý objekt (uzol AST) reprezentuje typ uzlu v derivačnom strome (získaného zo syntaktickej analýzy). Všetky tieto objekty si držia odkazy na rodiča a potomkov pomocou ich atribútov, ktoré sú podtriedami abstraktnej triedy.

3.1.2 Úprava AST na získanie porovnávacích jednotiek

Po získaní AST je možné vykonať porovnávanie. To by prebiehalo štýlom porovnávania každý podstrom s každým podstromom. Takýto výpočet je veľmi neefektívny z hľadiska časovej zložitosti. Napríklad pre dva AST skladajúcich sa z N podstromov, je zložitosť porovnávania $O(n^2)$. Preto je nutná ďalšia úprava [18], ktorá výpočet zefektívni. Tým je myslené, získať nejaké ďalšie vlastnosti každého stromu (a podstromu), a následne porovnávať vždy také podstromy, ktoré majú tieto vlastnosti podobné. Je teda možné povedať, že každý strom je reprezentovaný koreňovým uzlom, ktorý si tieto novo získané informácie o vlastnostiach udržuje. Tieto vlastnosti je teda nutné získať pre každý uzol.

Ohodnotenie uzlov

Každý uzol je ohodnotený hodnotou, ktorú získame pomocou *hash* funkcie. Tá pracuje nasledovne. Majme uzol X. Jeho poduzle budú $C_1, C_2, .., C_i$ kde $i \geq 0$. Potom hodnota uzlu X je

$$hash(X) = \begin{cases} x & \text{if } i = 0 \\ x + \sum_1^i hash(C_i) & \text{if } i > 0 \end{cases} \tag{3.1}$$

kde x je hodnota z *hash* tabuľky, ktorá je určená typom uzlu X.

Porovnávacie jednotky ako informačný vektor

Následne vytvoríme informačný vektor, ktorý ponesie všetky dôležité informácie o uzle. Takýto vektor vytvoríme pre každý uzol. Vektor pre uzol teda je:

$$V_i = (T, S, H, start, end) \tag{3.2}$$

kde T je typ uzlu, S je počet poduzlov, H je hodnota získaná *hash* funkciou, *start* a *end* sú čísla, ktoré značia začiatočný a koncový riadok začiatku bloku príkazu, kde sa príkaz nachádza (napr. telo funkcie, telo while príkazu, atď.). Takéto vektory vytvoríme pre oba vstupné projekty.

Následne všetky tieto vektory priradíme do množín, podľa počtu poduzlov, teda $V_x \in \{V_n | N \in TN, S_n = S_x\}$. Táto množina je získaná pre každý zo zadaných projektov zvlášť, čiže pre všetky uzly v ST je výpočet analogický. Reprezentácia vektoru zastrešuje abstraktná trieda *ASTnode*, ktorá obsahuje atribúty predstavujúce tento vektor.

3.1.3 Porovnávanie hodnôt uzlov

Porovnávanie prebieha cez všetky uzly podľa ich počtu poduzlov. Presnejšie porovnávame vektory, ktoré spĺňajú podmienku $k \leq i \leq min(TMC, SMC)$, kde k je prah, $\{V_n | N \in TN, S = i\}$ s vektormi $\{V_n | N \in SN, S = i\}$. Týmto znížime časovú zložitosť na hodnotu medzi $O(n)^2/MC$ a $O(n^2)$, kde $MC = min(TMC, SMC)$. Na nastavenie *hash* funkcie a prahu porovnávania slúži konfiguračný súbor. Tieto hodnoty teda bude možné pomocou tohto súboru meniť.

Celý tento postup, vrátane ohodnotenia *hash* funkciou, sa vykoná aj pre zadané kódy na ignorovanie, pre ďalšiu možnosť vylúčiť podozrivú podobnosť v prípade, že sa jedná o práve vstupné kódy na ignorovanie.

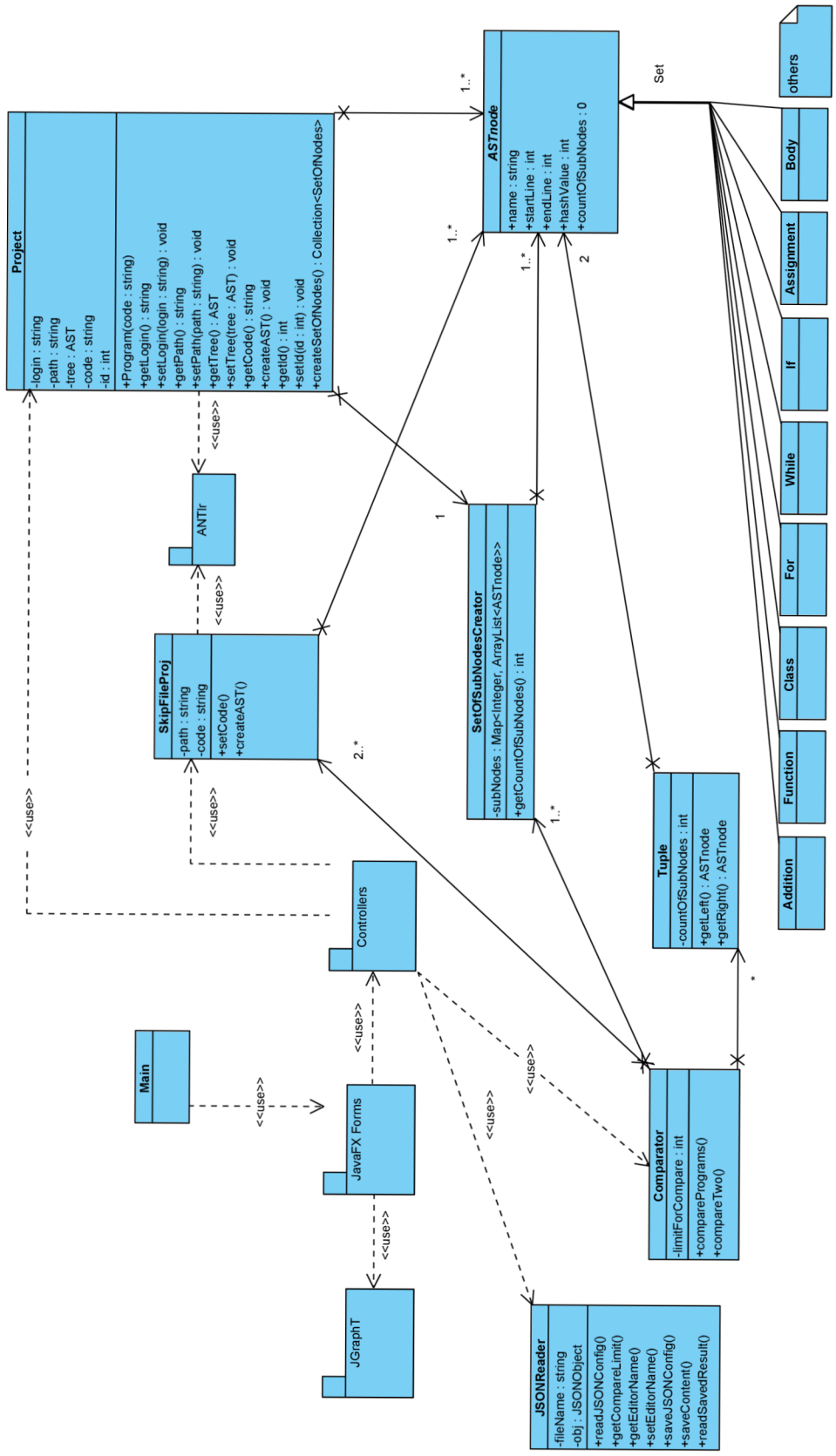
Porovnávanie prebieha cez všetky vektory s rovnakým počtom poduzlov. Ak nájdeme také dva vektory V_x a V_y, ktorých hodnota H sa rovná, čiže $H_x = H_y$, považujeme ich za podobné a uložíme ich do objektu *Tuple*, kde budú tvoriť tieto vektory dvojicu (V_x, V_y). No v prípade, že sa rovnajú zadanému kódu na ignorovanie, je ich zhoda ignorovaná. Tým, že porovnávanie je vykonávané od najvyššieho počtu uzlov, vždy po nájdení dvojice vektorov odstránime z množiny vektory s rovnakým, alebo vyšším začiatočným riadkom, a menším, alebo rovnakým koncovým riadkom, a tým predídeme duplicitným výpočtom. Tak isto tieto vektory s rovnakým začiatočným riadkom odstránime aj keď sa jedná o zhodu s kódom na ignorovanie. Následne dvojice, ktoré ostali, značia uzly, a teda časti kódov, ktoré sú podozrivo podobné. Tieto dvojice po namapovaní na pôvodný kód sú pripravené na vizualizáciu.

Takýto postup sa opakuje pre každý vstupný projekt s každým iným vstupným projektom. Je teda vytvorených niekoľko množín podozrivých dvojíc, pre každú dvojicu projektov zvlášť.

Uloženie výsledku

Nástroj umožňuje uložiť získané výsledky pre znovu zobrazenie, alebo pre prenos do iného zariadenia. Pri využití tejto možnosti sa neukladajú priamo objekty, ale len reprezentácia pre grafické zobrazenie. Nato sa využije súbor vo formáte *JSON*. Je teda nutné uložiť všetky dvojice AST (kódov). Dvojice budú ukladané ako prvky zoznamu, a tým pádom sa zachová príslušnosť dvojíc. Pre zobrazenie je nutné uložiť AST, ale aj kód. Uloženie celého zdrojového kódu je neefektívne, preto nástroj ukladá len fragmenty, ktoré sú považované za potenciálny klon. Tieto časti kódu je však nutné kompletne zachovať, preto budú uložené

k príslušnému AST. Taktiež je nutné reprezentovať samotný AST, a to tak, že každému uzlu sa uloží názov, a následne sú v zozname uložený jeho potomkovia. Tak je zachovaná stromová hierarchia a tým pádom vzťahy medzi uzlami. Všetky nájdené fragmenty pre dva projekty sa uložia, ako vo forme AST, tak aj priamo kód. Pre uloženie viacerých dvojíc projektov je nutné uložiť každé porovnávanie projektov zvlášť. Uloženie sa teda vykoná vždy pre jednu dvojicu projektov. Výsledný diagram tried, zahrňujúci všetky spomínané možnosti zadnej časti, je zobrazený na obrázku 3.3.

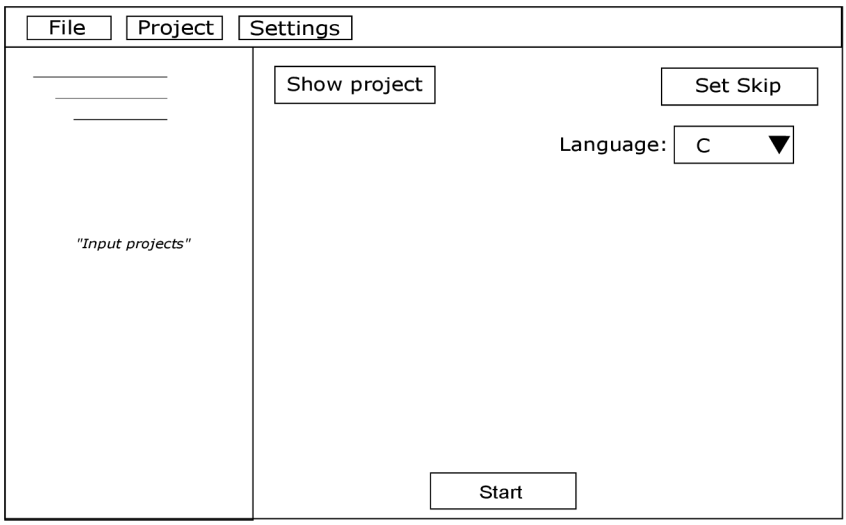


Obr. 3.3: Úplný návrhový diagram tried.

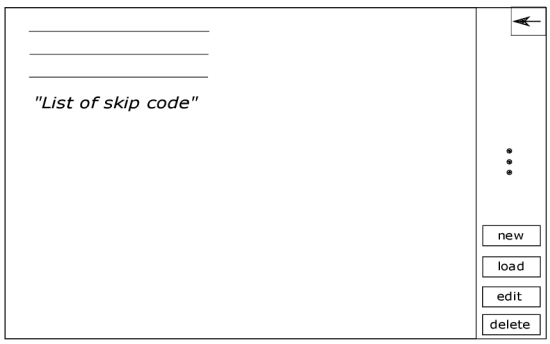
3.2 Návrh grafického užívateľského rozhrania a vizualizácia plagiátov

Pri návrhu užívateľského rozhrania ide vždy hlavne o to, ponúknuť čo najintuitívnejšie a najjednoduchšie prostredie pre užívateľa. Je teda dôležité, aby užívateľa program nezaťažoval s veľkým množstvom možností a nastavení.

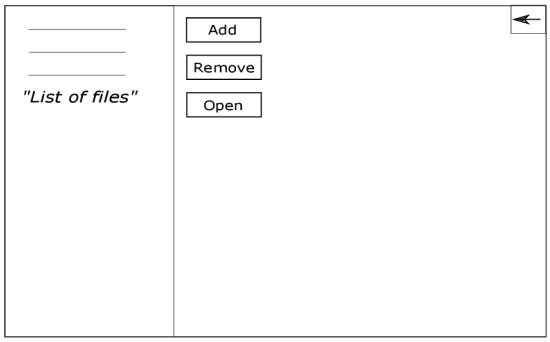
Na hlavnej obrazovke (obrázok 3.4) užívateľ bude mať možnosť načítať projekty, ktoré sa môžu skladať aj z viacero zdrojových súborov, ako už bolo popísané vyššie. Taktiež mu bude umožnené v záložke *Settings* otvoriť konfiguračný súbor v externom textovom editore.



Obr. 3.4: Hlavná obrazovka programu.



(a) Obrazovka zoznamu kódov na ignorovanie. Umožňuje pridať nový, editovať alebo zmazať existujúci.



(b) Obrazovka informácií o konkrétnom projekte. Zobrazuje jednotlivé zdrojové súbory, s možnosťou pridať, odstrániť, alebo zobraziť konkrétny súbor.

Obr. 3.5: Obrazovky možností pred porovnávaním.

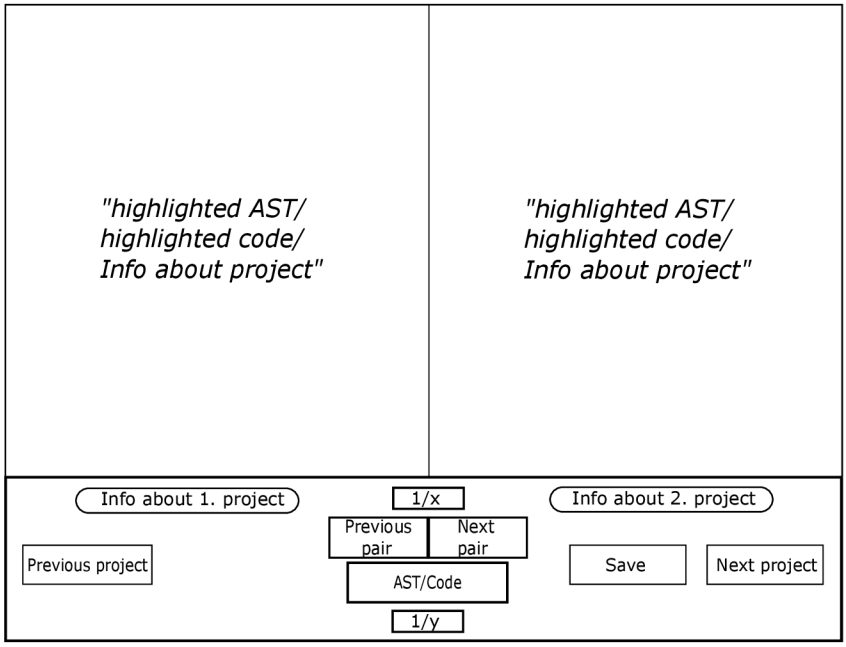
Medzi dôležité časti sa radí možnosť nastaviť množinu kódov, ktoré budú pri porovnávaní ignorované. Pre takýto zoznam ignorovaných kódov slúži obrazovka na obrázku 3.5a. Tá ponúka možnosť v externom editore kód na ignorovanie vytvoriť. Po zvolení možnosti na vytvorenie nového takého kódu, užívateľ do dialógu zadá názov. Následne môže kód vytvoriť a uložiť. Pridať nový kód je možné aj nahratím zo súboru. Existujúce kódy je možné

editovať a odstrániť. Na hlavnej obrazovke je nutné zadať jazyk, v ktorom zdrojové kódy budú.

V záložke *Project* je taktiež možné nahrať súbor s uloženým výsledkom z výpočtu, ktorý užívateľ vykonal inokedy, alebo preniesol na iné zariadenie. Zobrazujú sa tak dvojice, ktoré boli získané bez porovnávania v aktuálnom spustení.

Na obrázku 3.5b je vyobrazená obrazovka detailov projektu. Na túto obrazovku sa je možné dostať z hlavnej obrazovky a zobrazuje informácie o projekte. Na tejto obrazovke je taktiež umožnené pridať, odstrániť alebo otvoriť zdrojový súbor (v externom editore).

Po zvolení možnosti *start*, je vykonané porovnanie. Porovnáva sa každý projekt s každým iným projektom a výsledkom sú dvojice podobných častí zdrojových kódov. To zachytáva obrazovka na obrázku 3.6. Zobrazujú sa tak vždy dvojice kódových fragmentov podozrivých z plagiátorstva vo forme AST (obrázok 3.7b), alebo priamo v kóde (obrázok 3.7a).

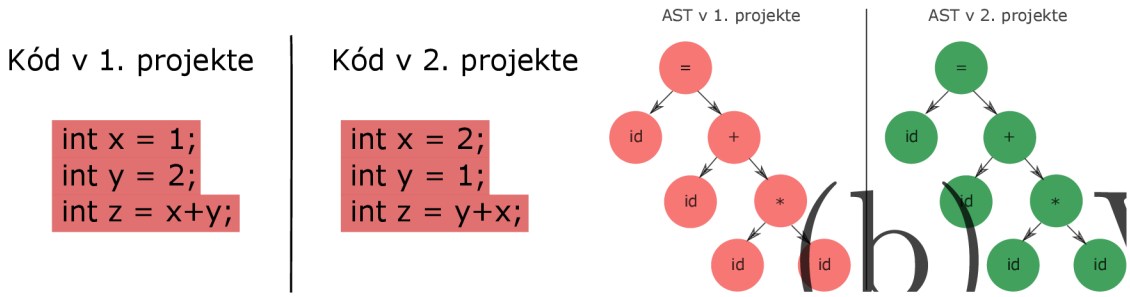


Obr. 3.6: Obrazovka zobrazujúca výsledky porovnávania. Zobrazuje dvojice kódových fragmentov vo formáte AST alebo priamo v kóde, a dodatočné informácie, ako cesty k projektom.

Taktiež je možné vidieť informácie o jednotlivých projektoch. Užívateľ môže prepínať medzi dvojicami kódových fragmentov, a tak isto medzi dvojicami projektov. Obidve dvojice sú taktiež počítané a je možné vidieť ich index v rámci všetkých dvojíc.

Táto obrazovka tiež ponúka možnosť uložiť výsledok pre neskoršie zobrazenie bez výpočtu, alebo pre prenos na iný počítač.

Jednotlivý prechod medzi obrazovkami, vrátané operácií, ktoré jednotlivé obrazovky prepínajú, zachytáva diagram na obrázku 3.8.



(a) Vizualizácia podozrivej dvojice v kóde.

Kapitola 4

Implementácia navrhnutého nástroja

Implementácia nástroja pozostáva, tak, ako bola navrhnutá, z dvoch častí. Prvou časťou je zadná (back-end) časť, ktorá zaobstaráva všetky algoritmické prvky, ktoré nepozostávajú len zo samotnej detekcie, ale aj nutných úprav, či už pred detekciou, alebo po nej. Druhou časťou je potom predná časť (front-end), ktorú vo veľkej miere zastupuje grafické užívateľské rozhranie (GUI), ktoré predstavuje most medzi užívateľom a zadnou časťou. Zaobstaráva teda komunikáciu medzi nástrojom a užívateľom. Zadná časť sa vykonáva až na základe akcií užívateľa. Preto sa tieto dve časti vo veľkej miere ovplyvňujú.

Nástroj bol vyvíjaný vo vývojovom prostredí *IntelliJ Idea* pod študentskou licenciou, ktorá podlieha nekomerčnému použitiu. Ako jazyk bol použitý programovací jazyk Java s vývojárskymi nástrojmi verzie 1.8.0-181.

4.1 Implementácia zadnej časti

Ako už bolo spomenuté, nástroj používa algoritmus detekcie založený na abstraktnom syntaktickom strome (ďalej len AST), ktorý je vytvorený na základe zadaného kódu. Preto je nutné vstupný kód do formy AST dostať. Následne je vykonaná detekcia. Výsledok detekcie je taktiež nutné upraviť tak, aby bol pripravený na zobrazenie v prednej časti. Práve preto je možné rozdeliť zadnú časť na tri logické časti, ktoré na seba nadväzujú, a to, počiatočné úpravy, algoritmus detekcie a úpravy na zobrazenie do grafu.

4.1.1 Počiatočné úpravy

Počiatočné úpravy pozostávajú s niekoľkých krokov. Na začiatku, v prípade, že vstupný kód pozostáva s viacero súborov, je nutné tieto súbory spojiť. Ako už bolo spomenuté, tieto súbory sú prečítané ako dátový typ *String*, a následne skonkatenované do jednej premennej a v prípade, že je to nutné, sú odstránené časti kódu, ktoré by pri takomto spájaní mohli viesť k nesprávnej syntaxi. Každý takýto kód predstavuje jeden projekt, ktorý má svoje špecifikácie, ako napríklad cestu k súboru. V prípade, že sa jedná o projekt s jedným zdrojovým súborom je to cesta s názvom súboru, avšak v prípade, že projekt predstavuje viacero súborov v adresári, je to cesta s názvom adresára. Táto cesta je zároveň jedinečný identifikátor projektu. Takýto projekt potom predstavuje trieda *Project*, ktorá je vyobrazená na obrázku 4.1.

Keďže nástroj podporuje aj zadanie súboru, ktorý obsahuje kód, ktorý značí, akú zhodu v detekcii je možné ignorovať, je nutné, aj takýto súbor spracovať. Avšak v tomto prípade sa jedná vždy o samostatné súbory, čiže jeden súbor pre jeden kód, a teda nie je nutné vykonávať spájanie. Takéto súbory sú reprezentované ako projekt na ignorovanie a v implementácii reprezentované triedou *SkipFileProj*. Jedná sa o podobnú triedu, ako trieda pre vstupný projekt, ale obsahuje informácie len o ceste a priamo kód.

```
public class Project {
//main file name and path to projects files
private String name;
private String path;
//clean code
private String code;
//AST
private ASTnode ast;
//Sets of subnodes
private SetOfSubNodesCreator creator;
//getters and setters and AST creator
}
```

Výpis 4.1: Trieda predstavujúca jeden projekt, obsahujúca kód, názov a cestu k súboru.

Počas vytváraní inštancií týchto tried, čiže objektov reprezentujúcich priamo vstupné projekty, je zároveň vykonaná transformácia do formy AST. Tá je zaistená taktiež priamo triedou *Project*, ktorá nato implementuje metódu. Vytvorenie AST je ďalším krokom počiatočnej úpravy. Je nutné ju vykonať pre všetky vstupné projekty. Každý objekt triedy *Project* si tak vytvorí svoj AST. Táto operácia využíva výstup nástroja *ANTLR*, ktorým je lexikálny a syntaktický analyzátor. Tie boli vygenerované z gramatiky vo formáte *.g4* a pozostáva z dvoch hlavných tried, a to *Lexer* a *Parser*. *Lexer* predstavuje lexikálny analyzátor, ktorého konštruktor vyžaduje prúd znakov (*CharStream* v Jave), ktorý po transformácii predstavuje vstupný kód. Následne sa vytvára prúd tokenov z tohto lexikálneho analyzátora, ktorý sa využíva na vytvorenie inštancie triedy *Parser*. Tým získame derivačný strom vo forme tzv kontextov. Nástroj obsahuje tri typy analyzátorov, čiže jeden pre každý jazyk.

Po vytvorení derivačného stromu nasleduje jeho transformácia do takej reprezentácie, ktorá je vyžadovaná pri porovnávaní. Takúto reprezentáciu predstavuje abstraktná trieda *ASTnode* (výpis 4.2), ktorá je rodičovskou triedou tried, ktoré už predstavujú konkrétny uzol stromu.

```
public abstract class ASTnode {
public String name;
public ASTnode parent;
public int hashValue;
public int startLine,endLine;
public int countOfSubnodes;

//getters and setters
}
```

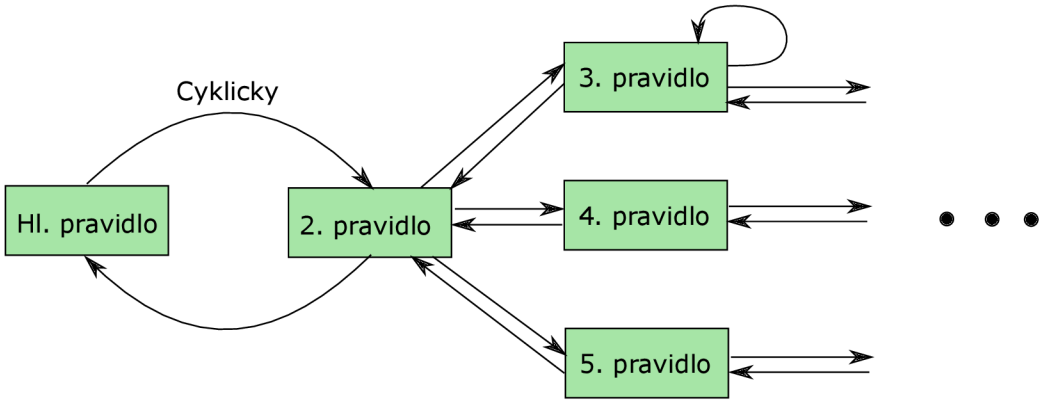
Výpis 4.2: Abstraktná trieda ASTnode, ktorá je využívaná ako rodičovská trieda pre všetky ostatné konkrétne uzly.

```
public class FuncDeclaration
    extends ASTnode {
private ASTnode type;
private ASTnode id;
private ArrayList<ASTnode>
    arguments;
private ASTnode body;
//getters and setters
}
```

Výpis 4.3: Príklad triedy, ktorá dedí od abstraktnej triedy ASTnode, a je uzlom predstavujúcim definíciu funkcie.

Táto trieda zároveň obsahuje atribúty, ktoré majú všetky uzly spoločné a sú dôležité pre ďalšie použitie. Tieto atribúty zachytávajú napríklad rodičovský uzol, začiatočný a koncový riadok v kóde, a ďalšie informácie, ktoré sú dôležité pre porovnávanie, napríklad hodnota získaná z hash funkcie, alebo počet poduzlov. Triedy predstavujúce konkrétne uzly sú teda potomkami abstraktnej triedy. Obsahujú už atribúty dôležité pred daný uzol stromu, napríklad výpis 4.3 zobrazuje triedu, predstavujúcu uzol definície funkcie. Atribút predstavujúci návratový typ je využívaný len v prípade jazyka C. Tento uzol nie je listom, preto sú jeho atribútmi ďalšie uzly. Listom je napríklad trieda predstavujúca identifikátor.

Samotná transformácia prebieha so vstupom, ktorým je derivačný strom. Výstupom je strom s uzlami v spomínanej forme. Na túto transformáciu sa využívajú tzv. visitore, ktoré umožňujú prechádzať derivačný strom. Sú to metódy, kde každá z týchto metód predstavuje jedno pravidlo v gramatike (obrázok 4.1). Je nutné mať pokryté každé pravidlo, ktoré sa môže vyskytnúť pri derivovaní stromu. Avšak v tejto implementácii nie sú pokryté všetky pravidla, keďže nástroj nepodporuje úplne využitie spomínaných jazykov, ale len základné bežné konštrukcie jazykov. Často sa v týchto metódach využíva rekurzia, keďže gramatika obsahuje zväčša ľavú, ale aj pravú rekurziu. Na správny prechod je nutné presne a správne rozlíšiť, ktorý kontext (pravidlo, uzol) nasleduje a následne vyvolať príslušnú metódu. Taktiež je dôležité si uvedomiť, že sa jedná o syntaktickú analýzu, čiže sémantiku nástroj nerozlišuje, napríklad, ak v jazyku C je zadaný neznámi typ, ktorý nie je explicitne predom definovaný, je aj tak v tomto nástroji braný ako by bol. V prípade, že sa nachádzame v metóde predstavujúcej pravidlo (uzol), ktoré má predstavovať uzol aj vo výslednom AST, vytvára sa príslušná inštancia triedy, ktorá predstavuje tento uzol. Po vytvorení sa zároveň uloží do objektu aj rodičovský uzol a taktiež začiatočný a koncový riadok daného pravidla (uzlu). Meno uzlu, ktoré je zdedeným atribútom z rodičovskej triedy *ASTnode* je vyplnené automaticky v konštruktore triedy predstavujúcej daný typ uzlu. Je tak zaistená konzistencia v názvoch uzlov.

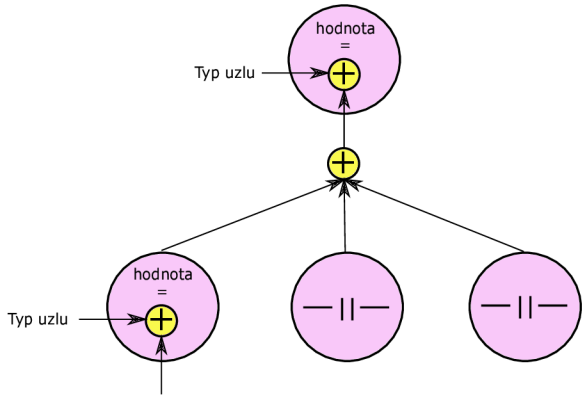


Obr. 4.1: Zobrazenie prechodov visitorov podľa pravidiel, ktoré predstavujú. Sú prepojené a využívajú rekurziu. Počas zanárania rastie strom, ktorý zastrešuje koreňový uzol aktuálneho podstromu.

Takýmto prechodom je pre každý vstupný projekt vytvorený príslušný AST vo vnútornej reprezentácii. Celý strom je zastrešený jedným hlavným uzlom, ktorý má referencie na všetky podstromy. Tento hlavný uzol je potom atribútom objektu typu *Project*. Každý takýto objekt si teda udržuje všetky dôležité dáta v podobe svojich atribútov. Avšak, aj keď už je AST vo vnútornej reprezentácii vytvorený, ešte mu chýbajú kľúčové vlastnosti, čiže hodnota uzlu a počet poduzlov, ktoré je možné získať pomocou niekoľkých prechodov

týmto stromom. Prvým prechodom musíme zaistiť ohodnotenie každého uzlu podľa *hash* funkcie. Keďže *hash* funkcia pracuje tak, že v prípade, že uzol ma synovské uzly je jeho hodnota odvodená podľa jeho typu, ale aj so súčtu jeho potomkov, je nutné ohodnocovať uzly od listov. Preto sa toto ohodnocovanie vykonáva rekurzívne volanou metódou, ktorá na základe typu uzlu získa hodnotu s konfiguračného súboru. Následne v prípade, že uzol má potomkov, cyklicky, pre každého potomka, je rekurzívne vyvolaná rovnaká metóda, ktorá takto rovnako pracuje s potomkom. Následne sa hodnoty potomkov pripočítavajú k hodnote uzlu a táto hodnota je navrátená rodičovskému uzlu (obrázok 4.2). Po navrátení hlavnému, koreňovému uzlu programu všetkých hodnôt poduzlov, získavame hodnotu hlavného uzlu, ktorá predstavuje hodnotu celého vstupného zdrojového kódu.

Druhým prechodom daného stromu získame počet poduzlov daného uzlu. Tento prechod je veľmi podobný prechodu na získanie hodnoty. Taktiež je nutné prejsť strom k listom a následne si vracať počet poduzlov. Atribút počtu poduzlov v hlavnom koreňovom uzle potom predstavuje presný počet uzlov, ktoré sa nachádzajú v celom AST získaného zo zdrojového kódu. Tieto prechody stromom a získanie hodnôt, či už uzlov, alebo počtu poduzlov, vykonáva objekt typu *Valuator*. Tento objekt na ohodnotenie prijíma na vstupe hlavný koreňový uzol, cez ktorý postupne vykoná ohodnotenie, a následne si všetky, už ohodnotené uzly, ukladá do zoznamu. Je to z dôvodu, že hierarchia stromu na následné operácia nie je nutná, a implementačne je to efektívnejšie z hľadiska veľkosti a zároveň prehľadnosti kódu.



Obr. 4.2: Diagram prechodu medzi obrazovkami.

Na rozdiel od ohodnotenia uzlov, je získanie počtu poduzlov len medzikrokom na využitie tejto hodnoty. Pre efektívnejšie porovnávanie, aby nebolo nutné porovnávať každý uzol s každým, sú už ohodnotené uzle, zo zisteným počtom poduzlov, rozdelené do množín, kde prvkami množiny sú uzly, ktoré majú rovnaký počet poduzlov. Túto funkcionalitu zaobstaráva trieda *SetOfSubNodesCreator*. Táto trieda predstavuje množinu, ktorá obsahuje všetky rozdelené množiny. Inštancia tejto triedy si taktiež udržuje cestu k zdrojovému súboru, ktorá je dôležitá pre neskoršie výsledky porovnávania. Ako vstup na rozdelenie do množín je zoznam získaný z objektu, ktorý vykonal ohodnotenie uzlov. Výsledkom sú množiny, ktoré sú udržiavané ako zoznamy v objekte typu *HashMap*, kde kľúčom je hodnota, ktorá predstavuje počet poduzlov a hodnota je priamo zoznam uzlov. Každý objekt typu *Project* si v atribúte udržiava objekt typu *SetOfSubNodesCreator*, čiže si udržiava aj dané množiny.

V prípade súborov na ignorovanie sa vykonáva ohodnotenie uzlov podľa hash funkcie a taktiež získanie počtu poduzlov. Avšak rozdelenie do množín podľa počtu poduzlov sa

nevykonáva, keďže sa predpokladá, že takéto súbory nebudú obsahovať dlhé kódy, ale len nejaké kódové fragmenty, napríklad nejaký algoritmus, ktorý je všeobecne známy a používaný. Po vykonaní získania týchto hodnôt, metóda na vytvorenie AST navráti ohodnotené uzly AST ako zoznam. Je to z toho dôvodu, že pre porovnanie, či nájdená zhoda nie je definovaná ako ignorovaná, nám stačí hodnota uzlu a počet pod uzlov, a nie celá stromová hierarchia.

Ohodnocovanie podľa typu uzlu, čiže hodnota získaná na základe typu uzlu, je vykonávané s využitím konfiguračného súboru, ktorý je vo formáte *JSON*, keďže práca s týmto formátom je v Jave vďaka knižniciam jednoduchá. Nejedná sa teda o binárny súbor a je možné ho editovať bez spúšťania aplikácia. Avšak pri chybnej úprave môže nastať neregulárne chovanie nástroja. Konfiguračný súbor sa nachádza v špecifickom adresári a jeho umiestnenie by nemalo byť menené. Konfiguračný súbor je teda nevyhnutnou súčasťou nástroja, bez ktorého by nástroj nepracoval regulárne. Správu konfiguračného zabezpečuje samostatná trieda *JSONReader*, ktorá okrem inej funkcionality využívanej v komunikácii s užívateľom, ponúka aj na základe názvu uzlu získať hodnotu. Taktiež je možné, aby užívateľ ovplyvnil tieto hodnoty v konfiguračnom súbore. Ale o tom viac až v neskorších sekciách.

4.1.2 Porovnávanie uzlov

Po príslušných počiatočných úpravách je všetko pripravené na vykonanie porovnávania. Priebeh tohto porovnávania zaisťuje trieda *Comparator*. Táto trieda má niekoľko kľúčových atribútov. Ako vstupné parametre slúži atribút zoznam ohodnotených uzlov kódov na ignorovanie a atribút predstavujúci minimálny počet poduzlov, ktorý musí uzol obsahovať, aby jeho prípadná zhoda s iným uzlom iného projektu bola vyhodnotená ako podozrivá zhoda. Tento limit sa taktiež nachádza v už spomínanom konfiguračnom súbore. Tieto dva vstupné parametre je nutné vložiť do inštancie objektu na porovnávanie skôr, než je porovnávanie vykonané.

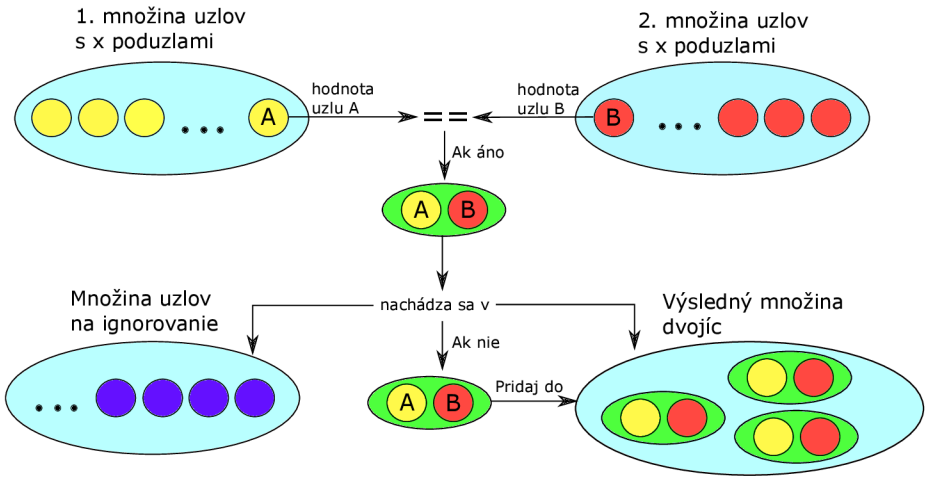
Ako zdroj porovnávania je parameter, ktorý predstavuje zoznam objektov typu triedy *SetOfSubNodesCreator*, ktoré obsahujú množiny uzlov s rovnakým počtom poduzlov. Každý takýto objekt predstavuje jeden projekt, čiže množinu uzlov pre daný projekt. Porovnávanie prebieha vždy pre dva projekty, čiže pre dve množiny množín s rovnakým počtom pod uzlov. Postupným prechádzaním sa porovná každý projekt s každým.

Pri vybraní dvojice projektov sa vždy berie jeden ako hlavný. Jeho zoznam množín je zoradený od najväčšieho. Porovnávanie sa teda vykonáva od uzlov z najväčším počtom poduzlov. Na začiatku porovnávania sa najprv skontroluje, či daná množina už nie je pod minimálnym limitom počtu pod uzlov. V prípade, že áno, je porovnávanie aktuálnej dvojice projektov ukončené. Následne sa prechádza cez všetky množiny hlavného projektu. V prípade, že aj sekundárny projekt obsahuje množinu s rovnakou hodnotou počtu poduzlov, je porovnávaný každý uzol s každým medzi týmito množinami. Keďže je rovnosť počtu poduzlov zaistená rozdelením do množín, porovnávanie vyhľadáva medzi týmito množinami uzly s rovnakou hodnotou získanou *hash* funkciou.

Nájdenie takýchto dvoch uzlov ešte neznamená, že tieto dva uzly budú zaradené do výstupu, ale je nutné skontrolovať, či sa dané uzly už ako dvojica nenachádzajú vo výsledku ako časť iných väčších podstromov, alebo či tieto uzly nie sú súčasťou uzlov získaných zo súborov na ignorovanie. V prípade kontroly, či to nie sú uzly, ktoré majú byť ignorované, sa jedná len o prechádzanie týmito uzlami a kontrolou, či sa hodnota nájdenej dvojice uzlov nerovná niektorému s uzlov na ignorovanie. V druhom prípade, kedy je nutné zistiť, či už

nájdené uzly nie sú zahrnuté vo výsledku, je využitý fakt, že porovnávanie prebieha od množín uzlov s najväčším počtom poduzlov. Keďže každý uzol obsahuje informácie o tom kde v rámci kódu začína, čiže začiatočný a koncový riadok daného uzlu, prechádzajú sa dvojice, ktoré už boli zaradené do výsledku a porovnávajú sa s aktuálnou vyhodnocovanou dvojicou podľa týchto informácií. Ak by nejaký uzol bol súčasťou iného uzlu, čiže väčšieho podstromu, jeho začiatočný riadok by bol minimálne rovnaký ako tohto podstromu a koncový riadok maximálne rovnaký. Na základe toho je možné predísť redundantným dátam.

Ak teda nájdená dvojica nie je zaradená ako ignorovaná a nenachádza sa už v inej dvojici, je táto dvojica zaradená do výsledku. Celý postup klasifikovania dvojice uzlov ako možný plagiát je vyobrazený na obrázku 4.3.



Obr. 4.3: Vyobrazenie postupu nájdenia dvojice uzlov, ktoré majú rovnakú hodnotu a kontroly, či sa nenachádzajú medzi uzlami na ignorovanie, alebo či už nie sú vo výslednej množine zahrnuté naduzlami.

Výsledok je vo forme zoznamu objektov typu *Tuple*, ktorá obsahuje dva prvky rovnakého typu, čiže dvojicu. Ako prvky tejto dvojice sú uložené nájdené uzly. Pre každé dva projekty teda vzniká zoznam dvojíc uzlov. V prípade, že takýto zoznam nie je prázdny je vytvorená ďalšia dvojica, tentoraz s prvkami typu *String*, ktorá obsahuje identifikátory projektov, čiže cestu k ich hlavným súborom. Následne sú zoznam s dvojicami a dvojica s identifikátormi projektov vložené do atribútov objektu typu *Comparator*, ktorý celé porovnávanie vykonáva. Týmito atribútmi sú teda zoznam zoznamov dvojíc uzlov stromu, čiže jeden zoznam pre dvojicu projektov a zoznam dvojíc s identifikátormi projektov, čiže dvojica pre dvojicu projekty.

4.1.3 Finálne úpravy do formy pre zobrazenie

Po vykonaní porovnávania sú získané výsledky detekcie v už spomínanej forme. Táto forma predstavuje internú reprezentáciu a na to aby mohla byť predaná prednej časti, pre zobrazenie užívateľovi, je nutné ju transformovať do takej formy, s ktorou pracuje predná časť. Tieto finálne úpravy je síce možné zaradiť do zadnej časti nástroja, avšak silne súvisia a spolupracujú s prednou časťou. Finálne úpravy tak zastrešuje trieda *ResultFormController*, ktorá už je kontrolórom pre časť grafického užívateľského rozhrania. Avšak samotné úpravy sú vykonávane bez vedomosti užívateľa, preto ich je možné zaradiť do zadnej časti.

Finálne úpravy sú prvotne vykonávané len pre prvú dvojicu projektov a ich prvú dvojicu uzlov, čiže podstromov. Pre ostatné dvojice sú teda finálne úpravy pre zobrazenie vykonávané až keď je to vyžadované (tzv. lazy evaluation).

Keďže nástroj umožňuje prehliadať nájdené podobné kódové fragmenty vo forme AST, ale aj vo forme priamo časti kódu, prvým krokom finálnych úprav je namapovanie každého z dvojice uzlov späť na zdrojový kód. Vytvára sa tak nový objekt typu triedy *Tuple*, ktorý obsahuje prvky typu *String*, predstavujúce časti kódu, ktorých interpretácie v AST sú stromy, ktorých koreňové uzly sú uzly z dvojice. Keďže každý uzol obsahuje číslo začiatočného a koncového riadku v kóde, je možné, po rozdelení príslušného kódu, uloženého v objekte triedy *Project*, na riadky, získať jednoducho. Vďaka tomu, že sa udržiava dvojica informácií o projektoch na rovnakom indexe, ako samotný zoznam dvojíc uzlov, je možné nájsť, na základe identifikátora, ktorým je cesta k súboru, príslušný objekt.

Ďalším krokom finálnych úprav je transformácia dvojice uzlov (predstavujúcich koreňový uzol podstromu) do formy, ktorá je nutná pre vizualizáciu. Táto operácia taktiež úzko súvisí s prednou časťou (front-end), keďže počas transformácie na objekty, ktoré sú uzlami zobrazovaného grafu, sa zároveň vykonáva generovanie uzlov a hrán grafu. Operáciou sa teda myslí transformácia objektov rôznych tried (tak ako bolo spomínané skôr), ktoré dedia z abstraktnej triedy *ASTnode* do objektov triedy *Vertex*. Tento objekt je pri deklarácii grafu zadaný ako typ uzlu.

Trieda *Vertex* tak predstavuje uzol a jej atribútmi sú názov uzlu, ktorý má každý objekt triedy, ktorá dedí z triedy *ASTnode*, zadaný pri vytváraní v konštruktore, čiže nemôže nastať nekonzistentný stav, kedy by nejaký takýto objekt nemal tento atribút vyplnený.

Ďalším atribútom triedy *Vertex* je zoznam objektov rovnakej triedy, ktorý predstavuje zoznam synovských uzlov. V prípade, že je takýto zoznam prázdny, jedná sa o listový uzol. Taktiež táto trieda prepisuje metódy *toString()*, *hashCode()* a *equal()*. Je to z toho dôvodu, že využívaná knižnica, ktorá je používaná na tvorbu grafu, práve pracuje s týmito metódami. Konkrétne metódu *toString()* na získanie názvu uzlu pre výpis v GUI a metódy *hashCode()* a *equal()* na rozlišovanie uzlov. Je to z toho dôvodu, že v prípade, že chceme vytvoriť hranu, musí byť jasné medzi ktorými dvomi objektami táto hrana vzniká. Z toho plynie, že každý takýto objekt musí obsahovať jedinečný identifikátor, ktorý je v métode *equal()* porovnávaný. V prípade, že by spomínané tri metódy neboli prepísané, ako unikátny identifikátor bol považovaný výstup metódy *toString()*. Avšak v takom prípade by nebolo možné mať uzly, ktoré majú rovnaký názov, pretože by boli tieto uzly považované za jeden, identický uzol. Preto trieda *Vertex* obsahuje ešte jeden atribút, unikátny pre každý uzol.

```
if (nd instanceof ArrayDec){
//chovanie
}
else if (nd instanceof EnumDec){
//chovanie
}
else if (nd instanceof FuncDeclaration){
//chovanie
}
// ... pre vsetky triedy
```

Výpis 4.4: Transformácia objektov, predstavujúcich uzly, do grafovej reprezentácie.

Keďže objekty, ktoré predstavujú uzly vo vnútornej reprezentácii, čiže nie uzly vykresľovaného grafu, sú síce objektami tried, ktoré sú podtriedami triedy *ASTnode*, ale priamo

ich štruktúra je rôzna. Každý takýto objekt má iné atribúty. Preto pri prechádzaní podstromov, pre transformáciu do grafovej reprezentácie, je dôležité definovať, ako sa má každý takýto objekt pretransformovať, čiže pre každú triedu definovať samostatné chovanie (výpis 4.4).

Prejdením celých podstromov, ktorých koreňové uzly sú v aktuálne zobrazovanej dvojici, sa vytvorí grafová reprezentácia. Takáto transformácia je vykonávaná vždy, až keď je to vyžadované, čiže táto operácia sa cyklicky opakuje, pokiaľ to užívateľ vyžaduje.

Posledným krokom, ktorý sa vykonáva len pri prvom využití porovnania, je definícia grafu. Na definíciu grafu a jeho tvorby sa využíva voľná knižnica pre Javu *JGraphT* [11]. Táto knižnica je pod licenciou LGPL 2.1. Umožňuje definovať akékoľvek objekty ako uzly, alebo hrany. V tomto nástroji, ako už bolo spomínané, sú uzlami objekty triedy *Vertex* a hrany sú štandardné (default). Každý nový objekt je nutné pridať do grafu ako uzol a následne, v prípade, že je to vyžadované, špecifikovať medzi vytvorenými uzlami hrany. Vytvorenie grafu, jeho uzlov a hrán vyobrazuje výpis 4.5. V nástroji je graf objektom triedy *DefaultListenableGraph*. Tá implementuje aj štandardný načúvač (listener) na udalosti.

```
Graph<Vertex, DefaultEdge> g = new DefaultDirectedGraph<>(DefaultEdge.class);
Vertex v1 = new Vertex(id, name1);
id++;
Vertex v2 = new Vertex(id, name2);
id++;
g.addVertex(v1);
g.addVertex(v2);
g.addEdge(v1,v2);
```

Výpis 4.5: Tvorba grafu s využitím knižnice JgraphT. Graf má definované uzly typu *Vertex* a hrany definované ako štandardné. Vytvorené su dva nové uzly, ktoré sú následne pridané do grafu a spojené hranou. Premenná *id* predstavuje unikátny identifikátor uzlu. Premenné *nameX* značia mená uzlov.

4.2 Implementácia prednej časti

Implementáciou prednej časti (front-end) sa rozumie implementácia grafického užívateľského rozhrania (GUI) a zároveň kontrolórov, ktorý spájajú GUI a zadnú časť. V tomto nástroji sa na tvorbu prednej časti používa open source vývojársky nástroj *JavaFX* [7] s využitím nástroja *JavaFX Scene Builder* [8], ktorý uľahčuje dizajn GUI. Vďaka tomuto nástroju vývojár nepíše priamo kód, ale len zostavuje GUI z rôznych *JavaFX* komponentov. Vo vývoji v *JavaFX* sa GUI pre desktopovú aplikáciu vyvíja podobne, ako pri webových aplikáciách.

Hlavným zdrojom GUI je súbor vo formáte *.fxml*. Jedná sa teda o typ značkovacieho jazyka *XML*. Každému takémuto kódu sa zadá kontrolór, ktorý danej obrazovke patrí. Tento kontrolór už je priamo trieda v Jave. Kontrolór slúži potom na spracovanie udalostí.

Predná časť sa dá na základe obrazoviek rozdeliť na štyri časti. Boli implementované podľa vytvoreného návrhu. Správu prechodu medzi týmito obrazovkami zaobstaráva objekt triedy *SceneManager*. Ten je v tomto prípade singleton a je tak prístupný pre akýkoľvek kontrolór. Tento manažér na základe požadovanej obrazovky vytvára inštanciu triedy *FXMLLoader* s cestou k danému *.fxml* súboru ako parametrom. Následne sú nahrané všetky komponenty, resp. je nahratý rodičovský komponent, ktorý obsahuje ostatné. Následne, vďaka objektu na nahrávanie *.fxml* obsahu, je možné získať tiež inštanciu príslušného kon-

trolóra k danej obrazovke. Túto inštanciu teda nie je vytváraná priamo, ale jej vznik je zapuzdrený vo vytváraní *FXMLLoader* objektu. Následne je možné, podľa potreby, vložiť potrebné atribúty kontrolóra a spustiť scénu.

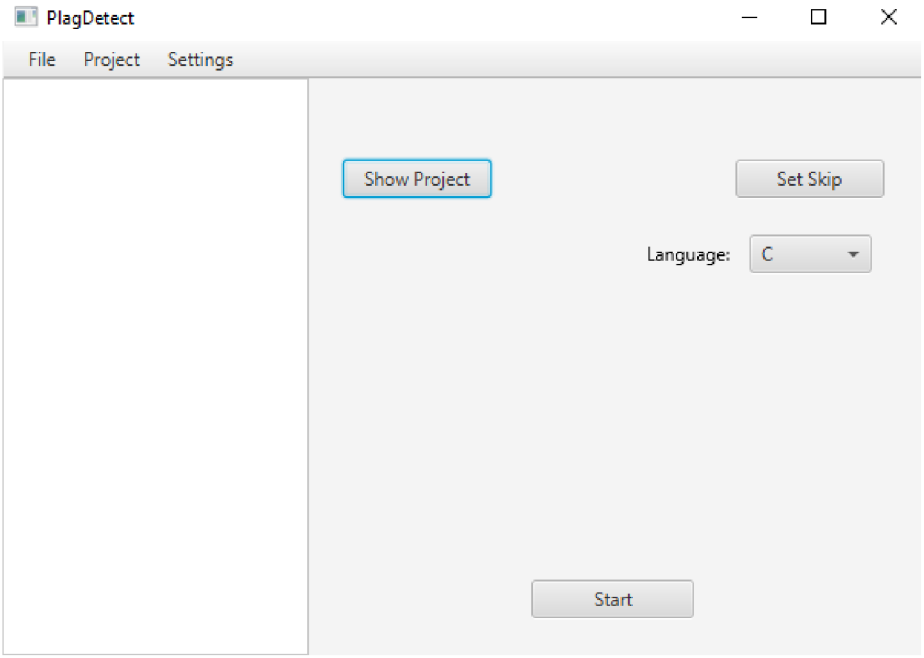
Prepínanie medzi obrazovkami manažér zaisťuje zmenou scény. Stage teda zostáva rovnaký, poprípade vzniká nový, ktorý značí nové okno, ktorý je potomkom hlavného.

Objekt typu *SceneManager* vzniká hneď po spustení nástroja priamo v *main* metóde. Jeho prvotným úkonom je nahratie hlavnej obrazovky a nahratie potrebných atribútov.

4.2.1 Hlavná obrazovka

Hlavná obrazovka (obrázok 4.4) je viditeľná teda hneď po spustení nástroja. Táto obrazovka zoskupuje všetky vstupné požiadavky užívateľa pre detekcie plagiátov a je možné z tejto obrazovky dostať sa na každú inú. Práve preto je považovaná za hlavnú.

Túto hlavnú obrazovku predstavuje obsah súboru *MainForm.fxml* a kontrolórom pre túto obrazovku je inštancia objektu triedy *MainFormController*.



Obr. 4.4: Hlavná obrazovka nástroja.

Dôležitým aspektom tejto obrazovky je práca so vstupnými súbormi. Práve na tejto obrazovke sa vstupné súbory dajú pridať. Existuje niekoľko možností, ako vstupný súbor pridať. Menu lišta v záložke *File* ponúka možnosti pridať súbor (súbory) cez dialógové okno typické pre daný operačný systém. Okrem priamo vstupných súborov, ktoré sú tým pádom brané ako náprotivky, je možné cez dialógové okno pridať aj adresár. Samozrejme tento adresár je potom považovaný za projekt, skladajúci sa z viacerých súborov. Ďalšou, rýchlejšou možnosťou na pridanie vstupných súborov je jednoduché pretiahnutie vybraných súborov priamo na okno aplikácie.

Správu súborov zaisťuje manažér, ktorým je inštancia objektu triedy *FileManager*. Tento objekt vytvára manažér scén, teda už spomínaná inštancia triedy *SceneManager*, ktorá následne tento manažér súborov ponúka všetkým kontrolórom, ktorý sa nejakým spôsobom podieľajú na práci so vstupnými súbormi.

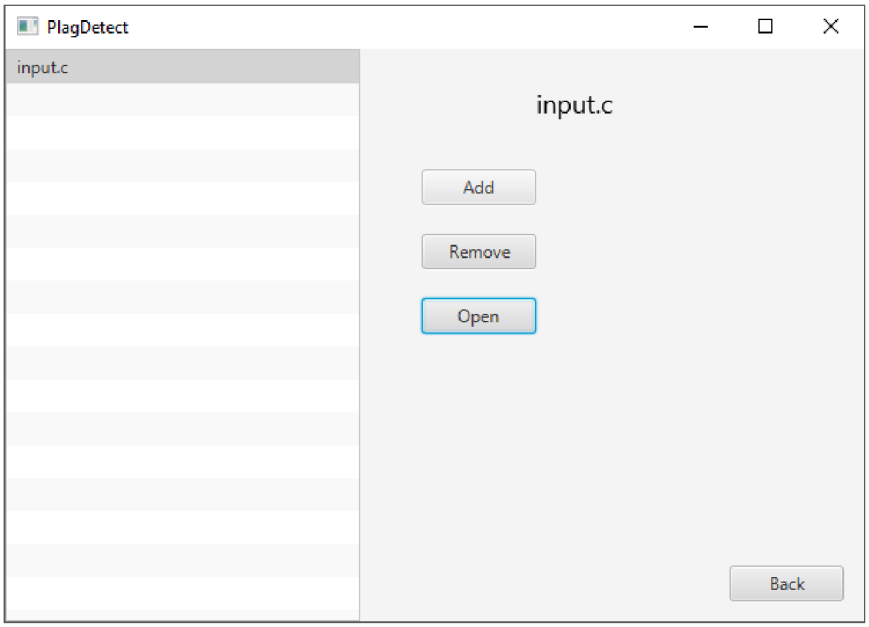
Na hlavnej obrazovke kontrolór teda pracuje s týmto manažérom kvôli vstupným súborom. Manažér si udržuje súbory priamo vo forme objektu triedy *File*. Avšak, keďže pri vstupných súboroch musí byť zahrnutá možnosť viacero súborov ako jeden projekt, je nutné si uchovávať aj informáciu o vzťahu súborov. Práve preto si manažér súborov ukladá vstupné súbory v inštancii triedy *HashMap*, kde atribúty sú zložené tak, že kľúčom je cesta k hlavnému súboru (tým je myslený buďto cesta k samostatnému súboru, alebo cesta k adresáru v prípade viacero projektov), a hodnotou je potom zoznam vstupných súborov v už spomínanej forme.

Hlavná obrazovka v záložke *Settings* tiež umožňuje určité nastavenia, napríklad otvoriť konfiguračný súbor spomínaný v popise implementácie zadnej časti. Na akékoľvek otvorenie textového súboru cez GUI je nutné mať nastavený externý textový editor. V prípade, že nastane pokus o otvorenie súboru (konfiguračného, alebo zdrojového) bez definovaného externého editora, dialógové okno nato upozorní a akcia sa zruší. Nastavenie externého editora tiež ponúka práve táto záložka v menu. Pod príslušnou možnosťou sa otvorí dialóg, ktorý umožňuje definovať cestu ku spustiteľnému súboru externého editora. Následne je táto cesta uložená do konfiguračného súboru. Keďže celú prácu s konfiguračným súborom zaisťuje trieda *JSONReader*, aj v tomto prípade tomu tak je. Táto trieda teda vykonáva uloženie a následné načítanie, v prípade potreby.

Taktiež je možné nastaviť minimálny počet poduzlov uzla, aby bolo vykonané porovnávanie tohoto uzla a možnosť nahrať uložený výsledok.

4.2.2 Obrazovka detailu projektu

Hlavná obrazovka zobrazuje vložené vstupné súbory ako zoznam. Po vybraní možnosti detailu konkrétneho súboru, alebo adresára, je pristúpené na detail vybraného projektu na obrazovke detailu projektu 4.5. Tú zaisťuje obsah súboru *ProjectDetail.fxml*, ktorý pracuje s kontrolórom triedy *ProjectDetailController*.



Obr. 4.5: Obrazovka zobrazujúca detail projektu.

Pred spustením tejto scény je kontrolórovi nahratý názov hlavného súboru a od manažéra scén taktiež získa manažéra súborov. Z tohto manažéra sú získané všetky súbory patriace do aktuálne zobrazovaného projektu.

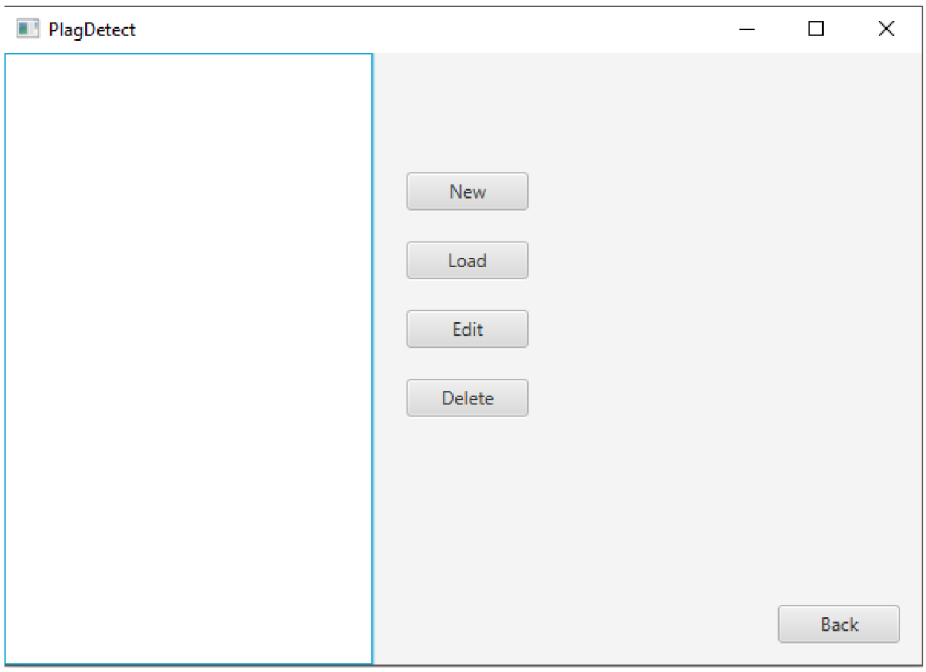
Užívateľ takto vidí celý balík projektu. Na tejto obrazovke môže pomocou dialógového výberu pridať ďalší súbor, ktorý tým pádom bude taktiež patriť do tohto vybraného projektu. Užívateľ tiež môže odstrániť súbor z projektu. V prípade, že odstráni všetky súbory, bude odstránený celý projekt. Užívateľ tiež môže otvoriť súbor v spomínanom externom editore.

4.2.3 Obrazovka správy súborov na ignorovanie

Ako už bolo spomenuté v implementácii zadnej časti, nástroj umožňuje zadať kód, ktorý bude v prípade zhody medzi vstupnými projektami ignorovaný. Na toto slúži práve obrazovka, ktorá zaobstaráva správu súborov 4.6, ktorý takýto kód obsahujú. Táto obrazovka teda zobrazuje takéto súbory ako zoznam súborov.

Tieto súbory, podobne ako vstupné, sú spravované inštanciou triedy *FileManager*, ktorá je prístupná všetkým. V tomto prípade sú takéto súbory držané ako zoznam, keďže nezáleží na mene súborov. Súbory sú potom ukladané na špecifickom mieste a je možné ich vytvárať priamo cez túto obrazovku. Pri využití tejto možnosti je nutné v dialógu zadať meno a následne sa prázdny súbor otvorí v už predom špecifikovanom externom editore. Tak isto je možné vložiť súbory cez tlačidlo *Load*, ktoré ponúka klasický dialógový výber. Taktiež je možné už pridaný súbor editovať v externom editore, alebo súbory vymazať.

Po vytvorení, alebo špecifikovaní takejto množiny súborov je možné sa vrátiť späť na hlavnú obrazovku.



Obr. 4.6: Obrazovka správy súborov na ignorovanie.

4.2.4 Obrazovka výsledku

Druhou hlavnou obrazovkou je výsledková obrazovka (obrázok 4.7 a 4.8). Tá predstavuje výsledok porovnávania. Na túto obrazovku je možné sa dostať z hlavnej dvomi spôsobmi, stlačením tlačidla *Start* a nahratím už vykonávaného výsledku. Po vybratí možnosti *Start* sa okrem výsledkovej obrazovky vytvára inštancia triedy *BackendClass*, ktorá zastrešuje vykonávanie zadnej časti postupne tak, ako bola popísaná v príslušnej sekcii tejto kapitoly. Cez manažéra súborov prístupného v manažérovi scén sú získané súbory na ignorovanie a vstupné súbory.

Po finálnych úpravách je vytvorený graf pomocou knižnice *JGraphT*. Tento graf však neslúži priamo na vizualizáciu, ale je transformovaný na *JGraphX* [9]. Ten umožňuje vizualizáciu grafu, ako *Swing* komponentu. V podstate sú teda vytvorené dva také grafy (každý pre jeden kódový fragment v dvojici), ktoré sú takto transformované. Keďže však GUI v tomto nástroji je vytvárané v technológii *JavaFX*, a nie *Swing*, je nutné siahnuť po možnosti, ktorá je podporovaná od *JavaFX* verzie 8, a to *Swing* komponent. Je teda možné nahrať akýkoľvek takýto komponent do tomu určeného miesta v *JavaFX*. V tomto prípade graf nahrávame do komponentu *StackPane*. Dôležitou vlastnosťou tohoto grafu je, že jeho nadtriedou je komponent *JScrollPane*, čo znamená, že je zaistená možnosť posúvať obsah s inak skrytým posuvníkom, v prípade, že je obsah väčší, než jemu určená plocha.

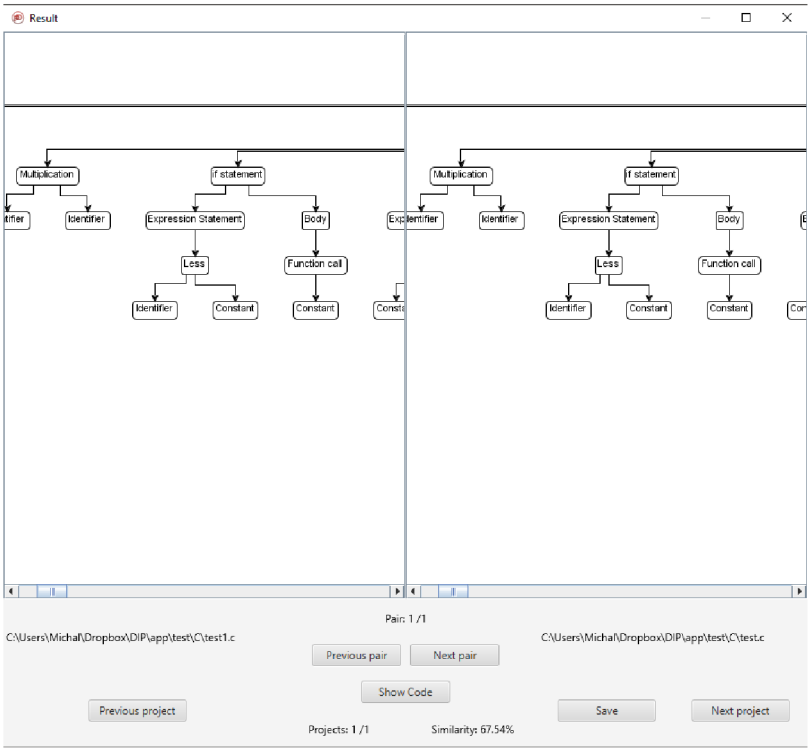
V prípade, že je zvolená možnosť načítania výsledku z porovnania vykonaného inokedy, je graf vytvorený iným spôsobom. Načítanie obsahu súboru zaisťuje trieda *JSONReader*, ktorá prečíta súbor vo formáte *JSON*. Avšak pri vytváraní grafu sa neprechádza strom v rovnakej podobe, ale je nutné vytvoriť nové objekty symbolizujúce uzly z existujúcich objektov v uloženom súbore. Vytvára sa tak nový strom v podobe objektov triedy *VertexJson*, ktorých atribúty sú meno uzlu a zoznam potomkov. Takto sú prevedené všetky dvojice stromov. Následne sú vyberané dvojice podľa udalostí užívateľa. Taktiež sa získavajú aj časti kódu, ktoré reprezentujú príslušné stromy, ale tie sa načítajú, ako reťazec znakov.

K načítaniu výsledku získaného z inokedy vykonávaného porovnávania je nutné aj takéto vykonanie uložiť. Práve to ponúka výsledková obrazovka. Ukladá sa vždy aktuálne prehliadaná dvojica projektov, čiže nájdené dvojice AST a im príslušné kódy. Uloženie, tak ako načítanie, zaobstaráva trieda *JSONReader*. Tá vytvára inštanciu triedy *Saver*, ktorá slúži práve nato, aby vykonala transformáciu. Tá postupným prechádzaním stromov dvojíc vytvára *JSON* objekty. Tie obsahujú meno, ako reťazec znakov a následne zoznam ďalších objektov, symbolizujúcich synovské uzly. Takéto objekty teda vytvoria stromovú štruktúru v podobe *JSON* objektov. Takáto hierarchia je následne uložená do špecifikovaného súboru. Kód sa potom ukladá ako čistý reťazec znakov.

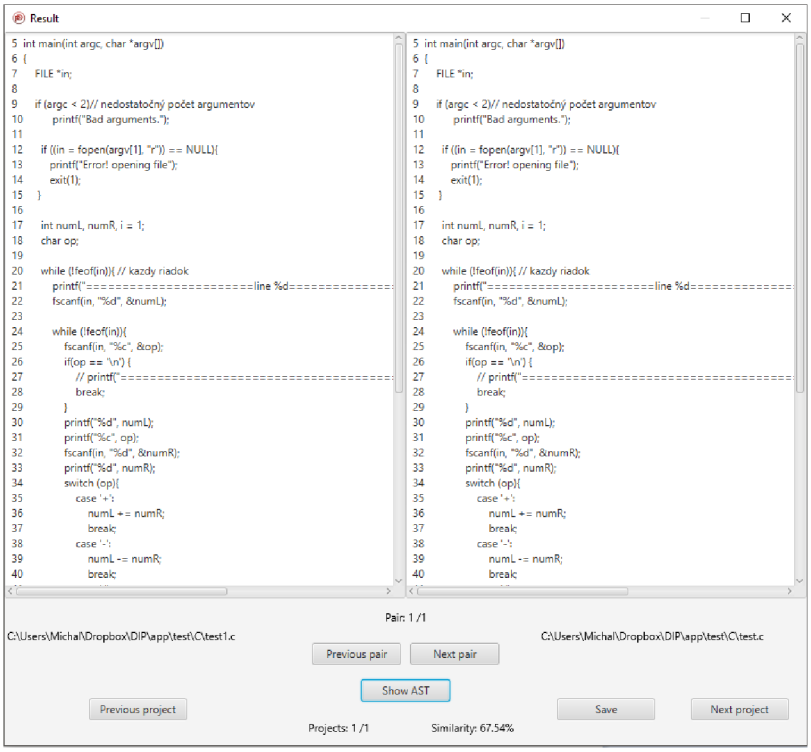
Ďalšími možnosťami sú listovanie medzi dvojicami nájdených kódových fragmentov v rámci dvoch projektov a zmena zobrazovaného kontextu z AST na kód. Čo sa listovania medzi dvojicami projektov týka, to je možné len, ak výpočet prebehol aktuálne. Keďže načítanie výsledku zo súboru zobrazuje vždy len jednu dvojicu projektov, je možnosť prepínať zrušená. Prepínať medzi dvojicami nájdených kódových fragmentov je však možná.

Výsledná obrazovka taktiež zobrazuje cestu k hlavnému súboru aktuálne prehliadanej dvojice projektov, index dvojice v fragmentov v rámci dvoch projektov a taktiež index dvojice projektov v rámci všetkých dvojíc projektov.

Zobrazenie výsledkov obrazovky vo forme AST sa nachádza na obrázku 4.7 a vo forme kódu na obrázku 4.8.



Obr. 4.7: Výsledková obrazovka zobrazujúce dvojicu kódových fragmentov vo forme AST.



Obr. 4.8: Výsledková obrazovka zobrazujúce dvojicu kódových fragmentov vo forme kódu.

Kapitola 5

Testovanie implementovaného nástroja

Po implementácii všetkých spomínaných častí bolo vykonané testovanie, ktoré prebiehalo v niekoľkých formách. Prvou formou je interné testovanie, ktoré prebiehalo bez testerov, ktoré overovalo funkčnosť jednotlivých prvkov nástroja, či už zadnej, alebo aj prednej časti.

Druhou formou je užívateľské testovanie, ktorého hlavným cieľom bolo získať spätnú väzbu od vzorky užívateľov, ktorá sa týka nie len funkčnosti, ale aj ako nástroj vplýva na nezainteresovaného užívateľa.

Testovaním, hlavne užívateľským, bol ovplyvnený výsledný nástroj, či už zo vzhľadového, alebo funkcionálneho hľadiska. Tým pádom, niektoré prvky, ktoré sú vo výslednom nástroji, sa neobjavili v pôvodnom návrhu, ale vznikli na základe výsledkov testovania.

5.1 Interné testovanie

Ako prvé prebiehalo interné testovanie, ktoré zahŕňalo všetky prvky zadnej časti nástroja. Zámerom interného testovania bolo zistiť správnosť fungovania nástroja, čiže overiť efektivitu identifikovania kódových fragmentov, ktoré sa zhodujú vo vlastnostiach, ktoré boli popísané v predchádzajúcich kapitolách, po transformácii na abstraktný syntaktický strom (AST).

Interné testovanie teda prebiehalo v niekoľkých častiach, kde prvotnou časťou bolo zistenie, ako vplývajú hodnoty získané z *hash* funkcie a prah minimálneho počtu poduzlov, na porovnávanie. Následnými testami bolo zistenie sily nástroja, teda aké techniky, využívané pri snahe zakryť plagiátorstvo, dokáže nástroj ignorovať, a teda považovať dané kódové fragmenty za identické.

5.1.1 Testovanie vplyvu hodnôt súvisiacich s detekciou

Okrem samotného porovnávania je nutné otestovať, aký vplyv, a či vôbec, majú dva interné atribúty na výsledok porovnávania. Týmito atribútmi sú prah, ktorý predstavuje minimálny počet poduzlov pri porovnávaní, a hodnoty typov uzlov. V prípade atribútu minimálneho počtu poduzlov sa nejedná o priame ovplyvnenie samotného porovnávania, keďže doň nijako nezasahuje. Tento atribút určuje, len aké uzly budú porovnávané. Ovplyvňuje teda až to, čo je vo výstupe. Na získanie vhodnej prednastavenej hodnoty, keďže túto hodnotu môže užívateľ zmeniť, bol vykonaný test. Vstupom tohto testu boli dve dvojice kódov, s rôznym počtom rovnakých kódových fragmentov. Keďže už z algoritmu plynie, že tento atribút je

	Projekt 1	**Projekt 2**
Počet riadkov	500	50

Tabuľka 5.1: Počet riadkov kódu dvojíc vstupných kódov na test hodnoty minimálneho počtu poduzlov.

veľmi subjektívny, jeho vhodná hodnota závisí na vstupe, boli vstupné súbory rôzne veľké. Tabuľka č. 5.1 zobrazuje počet riadkov vstupných kódov.

Pre takéto dva projekty teda postupne bola zvyšovaná hodnota minimálneho počtu poduzlov a hľadaným výstupom bol počet nájdených dvojíc kódových fragmentov. Výsledok zobrazuje tabuľka č. 5.2, ktorá pre dané dva projekty s počtom riadkov vyobrazeným v tabuľke č. 5.1 ukazuje, že táto hodnota nemôže byť pevne prednastavená pre rôzne vstupy. Pre krátke kódy s nízkym počtom riadkov, a teda aj s nízkym počtom príkazov, nemôže vzniknúť veľký strom, z čoho plynie, že táto hodnota by mala byť menená na základe dĺžky a veľkosti projektov.

Minimálny počet poduzlov	**Dvojice Projekt 1**	**Dvojice Projekt 2**
5	3472	10
10	1152	6
15	76	1
20	62	0

Tabuľka 5.2: Počet nájdených dvojíc pre menenú minimálnu hodnotu počtu poduzlov pre dvojicu vstupov s rôznym počtom riadkov.

Jedným z možných prístupov pri menení tejto hodnoty je nastaviť ju na nižšiu hodnotu, nájsť potenciálne dvojice projektov s veľkým počtom nájdených podobných kódových fragmentov, a následne skúsiť pre tieto dva projekty hodnotu zvýšiť a nájsť najväčšie dvojice fragmentov. Tým by sa mal odstrániť šum a zaistiť vyhľadanie potencionálneho plagiátorstva.

V prípade, hodnôt, z ktorými pracuje *hash* funkcia bol vykonaný test, ktorý má za úlohu zistiť, aké hodnoty sú najviac odolné voči falošne pozitívnymi (*false positive*) vyhodnoteniami. Vstupom tohto testu boli vybrané dva projekty v jazyku *Python*, ktorých počet riadkov sa blíži k hodnote 1000 a hodnota minimálneho počtu poduzlov bola nastavená na 12. V tomto teste teda boli vykonané tri pokusy s rôznymi hodnotami typov uzlov. Prvým typom boli hodnoty, ktoré sú postupnosťou a nie sú vysoké, teda hodnoty v intervale $< 1, 119 >$. Druhým typom boli hodnoty, ktoré sú taktiež postupnosťou, ale sú vyššie, a to v intervale $< 101, 219 >$, tretiu možnosť zastupovali náhodnejšie hodnoty, kde logicky súvisiace uzly mali hodnoty bližšie, ale medzi týmito logickými celkami boli hodnoty rôzne vzdialené. Pri poslednom type hodnôt sa využil pseudonáhodný generátor, ktorý pre každý uzol vygeneroval unikátnu hodnotu. Výsledkom tohto testu je potom počet nájdených dvojíc kódových fragmentov, ktoré sú vyhodnotené ako falošne pozitívne. Takéto výsledky zobrazuje tabuľka č. 5.3, kde je možné vidieť, že aj nesprávne zvolenie takýchto hodnôt môže výrazne ovplyvniť výstup nástroja. Prednastavené hodnoty využívajú poslednú spomínanú možnosť, a to náhodne získané hodnoty.

	$< 1 - 119 >$	$< 101 - 219 >$	**Logicky súvisiace**	**Náhodné**
Počet falošne pozitívnych hodnôt	18	18	0	0

Tabuľka 5.3: Zobrazenie počtu nájdených falošne pozitívnych dvojíc s využitím rôznych možností hodnôt podľa typu uzlov.

5.1.2 Testovanie algoritmu detekcie

Ďalším krokom interného testovania zadnej časti bolo otestovať, aké modifikácie, ktoré boli v úvodných kapitolách identifikované, ako plagiátorské techniky, je schopný nástroj eliminovať, a teda nájsť zhodné kódové fragmenty aj v takto modifikovaných zdrojových kódoch.

Pre potreby tohto testovania bola vytvorená testovacia sada, ktorá obsahuje balíky zdrojových súborov pre každý jazyk, teda tri balíky. Hlavný zdrojový súbor v každom balíku predstavuje terminálovú aplikáciu, ktorej funkcionalitou je primitívna kalkulačka. Tento kód obsahuje rôzne konštrukcie pre príslušný jazyk a jeho zámerom nie je, aby výsledná aplikácia bola naprogramovaná efektívne. Takýto kód pre jazyk C je vyobrazený vo výpise 5.1 (pre ďalšie dva jazyky sú tieto hlavné súbory vyobrazené v prílohe, konkrétne Python A.1, PHP A.2).

```
/*
* Funkcia, ktora precita subor a~vykona operacie z~kazdeho riadka
*
*/
int main(int argc, char *argv[])
{
   FILE *in;

   if (argc < 2)// nedostatocny pocet argumentov
      printf("Bad arguments.");

   if ((in = fopen(argv[1], "r")) == NULL){
      printf("Error! opening file");
      exit(1);
   }

   int numL, numR, i = 1;
   char op;

   while (!feof(in)){ // kazdy riadok
      printf("=======================line %d============================\n", i);
          // spracovanie noveho riadku
      fscanf(in, "%d", &numL);

      while (!feof(in)){
         fscanf(in, "%c", &op);
         if(op == '\n') {
            //
                printf("============================================================\n");
                // koniec riadku
```

```
            break;
         }
         printf("%d", numL);
         printf("%c", op);
         fscanf(in, "%d", &numR);
         printf("%d", numR);
         switch (op){
            case '+':
               numL += numR;
               break;
            case '-':
               numL -= numR;
               break;
            case '/':
               numL /= numR;
               break;
            case '*':
               numL *= numR;
               break;
            default:
               printf("Wrong operator?");
               exit(1);
         }
         printf("=%d\n", numL);
      }
      i++;
   }
   printf("============================================================\n");

   fclose(in); // riadkovy koemntar #3
}
```

Výpis 5.1: Hlavný zdrojový súbor pre testovaciu sadu jazyka C

Každý balík ďalej obsahuje niekoľko ďalších zdrojových súborov. Tieto súbory sú potom rôznou modifikáciou hlavného súboru. Modifikáciami sú teda zmeny hlavného súboru, ktoré predstavujú identifikované plagiátorské techniky. Toto testovanie sily nástroja sa potom skladá z jednotlivých testov, ktoré testujú práve porovnanie hlavného súboru s modifikovanými súbormi.

Väčšina výstupov testov je vyobrazených ako výstup jazyka C, avšak boli vykonané pre všetky jazyky a výsledky vo všetkých testoch, ktoré boli spoločné, čiže vykonané pre všetky jazyky, sa výsledok zhodoval. Taktiež je dôležité uviesť, že pre jednotlivé testy bola menená hodnota minimálneho počtu poduzlov tak, aby vo výsledku bola vždy len jedna dvojica najväčších kódových fragmentov.

Test č. 1 – Čistá kópia zdrojového kódu bez zmien

V prvom teste išlo o najjednoduchšiu zmenu, teda žiadnu zmenu. Tento test umožňuje potvrdiť, že nástroj nájde zhodné kódy na tej najminimálnejšej úrovni. Zároveň tento test kontroluje správnosť odstránenia redundancie, teda vynechanie podstromov z výstupu, keď sa už nachádzajú vo väčšom strome, ktorý už do výstupu zaradený bol. Keďže sa jedná o čistú kópiu je jasné, že každý podstrom má svoju kópiu v druhom kóde a teda je považovaný za kandidáta na pridanie do výstupu.

Výsledok zachytáva obrázok A.1, ktorý je vo forme kódu, keďže pri takých veľkých kódových fragmentoch, ktoré sú vlastne celým pôvodným zdrojovým kódom, je výsledný AST priveľký na ukážku v obrázku. Vo výsledku je možné vidieť, že bol identifikovaný celý zdrojový kód. Ďalšie dvojice boli teda na základe toho odfiltrované. Plagiát bezo zmeny je teda možné nástrojom odhaliť. Obrázok zobrazuje len časť výstupu, keďže sa jedná o najjednoduchší test. V tomto prípade je možné výsledok testu zdôvodniť tak, že sa jedná o čistú kópiu a tým pádom hlavný uzol AST, predstavujúci celý program, má všetky uzly rovnaké.

Test č. 2 a 3 – Kópia s odstránenými, alebo zmenenými komentármi

V druhom a treťom teste išlo o podobné testovanie. V prvom prípade bolo modifikáciou odstránenie komentárov a v druhom prípade zmena komentárov. Cieľom týchto dvoch testov je dokázať, že komentáre, ako také, nemajú žiaden vplyv na výsledky detekcie. Tieto modifikácie často patria k používaným technikám, ktoré sa zväčša snažia zakryť plagiát pred prípadnou kontrolou človekom. Avšak pre zistenie, ktoré techniky nástroj dokáže obísť, boli vykonané práve tieto testy.

Keďže tieto testy spolu úzko súvisia je na obrázku A.2 zobrazený výsledok, kedy je modifikáciou odstránenie komentárov. Ako je možné si na obrázku všimnúť, kód je opäť vyhodnotený ako rovnaký, nezávisle na zmene komentárov, a je teda možné potvrdiť, že nástroj nie je ovplyvniteľný akoukoľvek zmenou komentárov. Opäť je vo výslednom obrázku zobrazená len časť výstupu, v ktorom ale je možné si všimnúť, že v jednom kóde chýbajú komentáre. Výsledok opäť nezobrazuje výstup vo forme AST.

Výsledok tohto testu je možné zdôvodniť. Keďže sa jedná o stromový prístup, ktorý využíva syntaktickú analýzu, tak práve tá je dôvodom, prečo komentáre nie sú vo vyhodnocovaní vzaté do úvahy. Tým pádom akákoľvek manipulácia s komentármi nemá žiaden vplyv na detekciu.

Test č. 4 – Zmena identifikátorov

Ďalší test pozostával z modifikácie identifikátorov. Všetky identifikátory boli zmenené, okrem názvu hlavnej funkcie, ktorej sa identifikátor v jazyku C meniť nemôže. Jedná sa už o mierne zložitejšiu modifikáciu, ktorá by mohla byť využívaná v prípade, že tvorca plagiátu nevie, ako program pracuje.

Výsledok testu popisuje obrázok A.3, ktorý zobrazuje taktiež len časť výstupu, bez formy AST. Dôvodom tohto čiastočného vyobrazenia je fakt, že opäť po zmene identifikátorov sú výstupom celé vstupné súbory. Na obrázku je však možné si všimnúť, že sú identifikátory zmenené. Výsledok tohto testu dokazuje, že ani zmena identifikátorov neovplyvní výsledok detekcie.

Dôvod, ktorý je zodpovedný za výsledky tohto testu je spôsobený typom *hash* funkcie. Tá každý uzol typu identifikátor hodnotí rovnako, bez ohľadu nato, akú hodnotu tento identifikátor má.

Test č. 5 – Zmena dátových typov

Piaty test testoval ďalšiu identifikovanú plagiátorskú techniku, tentoraz zmenu dátových typov. Tento test pre jazyky Python a PHP nemá význam, keďže tieto dva jazyky sú dynamicky typované. Avšak, keďže jazyk C je staticky typovaným jazykom, je zmena dátových typov, ktorá neovplyvní beh programu, možnou modifikáciou pre zakrytie plagiátorstva.

Výsledok zachytáva obrázok A.4, ktorý obsahuje v hornej časti výstup vo forme kódu a v dolnej časť AST. Tento výstup už nepredstavuje celé vstupné zdrojové kódy, ale len kódový fragment. V tomto prípade bolo nájdené len telo príkazu *switch*. Z výsledku by mohlo vyplývať, že zmena dátových typov už ovplyvní detekciu. Avšak v tomto prípade, je nutné podotknúť, že záleží na forme modifikácie dátových typov. V tomto teste prišlo nie len k zmene primitívnych dátových typov, ale aj k zmene premennej na štruktúru poľa. To spôsobilo obmedzený výsledok, keďže nie len deklarácia, ale aj použitie samotného poľa vyprodukuje iné uzly AST. V prípade, že príde len ku zmene dátových typov, chová sa nástroj rovnako ako pri identifikátoroch, čiže berie do úvahy len uzol pre dátový typ, a nie konkrétny typ.

Test č. 6 – Vysunutie časti kódu do inej funkcie

Ďalším testom bola testovaná jednoduchá zmena štruktúry vystrihnutím časti kódu, a následným vložením do novo vytvorenej funkcie. Takáto zmena štruktúry tiež patrí medzi identifikované zastieracie techniky a pri reorganizácii zdrojového kódu môže byť pre ľudské oko zložitejšie odhaliť podobnosť.

Výsledkom tohto testu je objavenie rovnakého kódového fragmentu, ktorý sa zhoduje práve z vystrihnutou časťou kódu, ktorá bola premiestnená na iné miesto v kóde. Výsledok je viditeľný na obrázku A.5. V hornej časti je časť kódu zobrazujúca príkaz *while*, ktorý bol vystrihnutý. V dolnej časti to potom potvrdzuje časť AST.

Dôvodom výsledku tohto testu v pohľade na implementáciu nástroja je to, že akákoľvek časť kódu, ktorá obsahuje aspoň toľko uzlov, koľko špecifikuje minimálny počet poduzlov, je identifikovateľná na akomkoľvek mieste v kóde. Táto časť kódu predstavuje ten istý podstrom bez ohľadu nato, akému väčšiemu stromu patrí.

Test č. 7 – Pridanie a volanie zbytočnej funkcie

Následným cieľom ďalšieho testu bolo overiť, ako sa nástroj vysporiada s pridaním zbytočnej a bezvýznamnej funkcie a jej následným volaním. Jedná sa teda o kombináciu identifikovaných techník, a to zmenu štruktúry a pridanie bezvýznamného príkazu.

Výsledkom je opäť nájdenie zhodných kódových fragmentov, ale ešte na nižšej úrovni, čiže s menším počtom poduzlov. Obrázok A.6 zobrazuje výsledky tohoto testu. V hornej časti je opäť časť kódu a v dolnej vyobrazený koreňový uzol *While* nájdeného podstromu. Opäť je teda objavený príkaz *while*, no v tomto prípade už sa jedná o vnútorný.

Taktiež tento test je možné odôvodniť spôsobom implementácie. Definícia novej bezvýznamnej funkcie nie je v tomto prípade zámienkou horšej detekcie, ale paradoxne ňou je volanie tejto funkcie. Keďže v tomto teste sa opäť vyberala najhoršia možná možnosť, je táto nová funkcia volaná na rôznych miestach v kóde. To spôsobuje, že každý podstrom, ktorý by inak bol zhodný obsahuje nový podstrom, ktorý predstavuje práve volanie funkcie. Tento test je ale možné považovať za úspešný, keďže nástroj dokázal identifikovať zhodné kódové fragmenty aspoň v časti kódu, ktorá neobsahuje spomínané volanie novej funkcie.

Test č. 8 – Zmena podmienených príkazov

Medzi identifikované zastieracie techniky tiež patrí zmena podmienených výrazov. V tomto teste sa využil hlavný zdrojový súbor pre Python (A.1). Ten neobsahuje *switch-case* príkazy, ale priamo *if-else* príkazy. V tomto teste boli teda zmenené práve tieto podmienky tak, že logický výsledok majú rovnaký, ale obsahujú iné uzly.

Výsledkom tohto testu je prázdna množina zhôd, a teda takáto zmena môže znemožniť vyhľadanie podobných kódových fragmentov. Avšak, aj v tomto prípade, pri takýchto zmenách, je veľmi rozličné o aké vstupné zdrojové kódy sa jedná. V tomto prípade, je možné vidieť, že samotné telá podmienených výrazov obsahujú maximálne dva príkazy. Čo spôsobuje, že pre tieto bloky príkazov porovnávanie ani nie je vykonané. Ak by sme si však dostatočne rozšírili nejaké telo podmieneného príkazu (výpis 5.2 a 5.3), je možné si všimnúť na obrázku A.7, že pri takomto modifikovanom vstupe je možné podobné fragmenty odhaliť.

```
if (op == "+"):

   numL += numR
   print()
   print()
   print()
   print()
   print()
   print()
   print()
```

Výpis 5.2: Modifikácia vstupného súboru pre ukážku vplyvu štruktúry vstupného súboru na detekciu.

```
if (op != "-" and op != "*" and
    op != "/" ):
   numL += numR
   print()
   print()
   print()
   print()
   print()
   print()
   print()
```

Výpis 5.3: Modifikácia druhého súboru pre ukážku vplyvu štruktúry vstupného súboru na detekciu.

5.2 Užívateľské testovanie

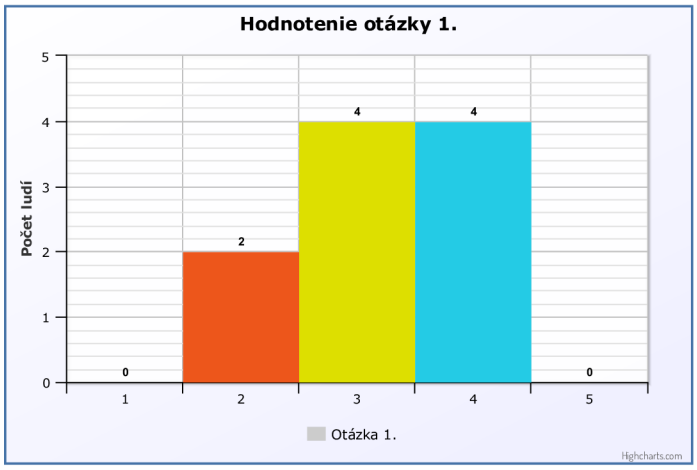
Druhou formou testovania bolo užívateľské testovanie. To malo za úlohu zistiť, aký je prvý kontakt s nezainteresovanými užívateľmi, avšak, ktorý vedia načo nástroj slúži. Tiež bolo zámerom užívateľského testovania overenie funkčnosti detekcie z kódov, ktoré boli vytvorené priamo testujúcimi užívateľmi.

Užívateľského testovania sa zúčastnilo 10 osôb z oboru informačných technológii, ktoré dostali len základné minimálne informácie o tom, načo nástroj pracuje. Každá z týchto osôb obdržala testovaciu verziu nástroja, hlavný zdrojový súbor z interného testovania a dotazník (príloha A.4). Užívateľské testovanie je teda rozdelené, tak ako vyplýva z dotazníka, na dve časti. Prvá časť sa zameriava na intuitívnosť, zložitosť práce s vizuálnou stránkou nástroja a pochopiteľnosť formy výstupu. Pre tento účel odpovedali užívatelia na otázky formou hodnotenia od 1 do 5.

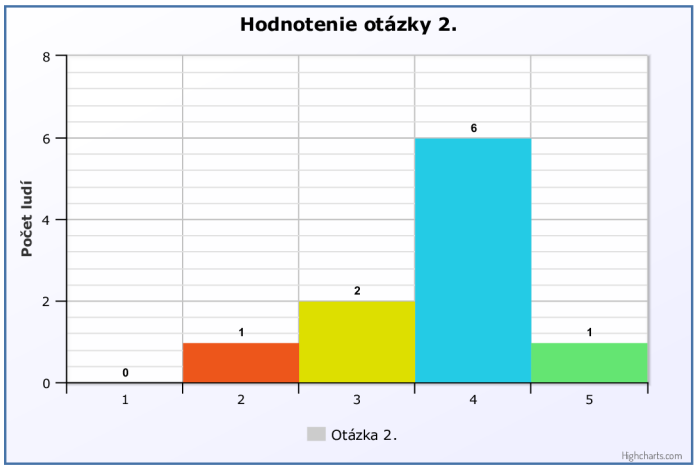
5.2.1 Prvotné otázky v užívateľskom testovaní.

Prvou otázkou sa zisťoval prvý dojem po spustení nástroja. Jednalo sa teda o test hlavne vzhľadový a či tento vzhľad nevplýva negatívne na užívateľa. Výsledky hodnotenia zobrazuje graf 5.1. Z výsledku je možné odvodiť, že nástroj je priemerný, čo sa týka vzhľadu. Užívatelia uviedli v pripomienkach zväčša protichodné pripomienky, a to, že nástroj má jednoduchý vzhľad a nie odpudivý, ale zároveň bolo uvedené, že vzhľad je až moc jednoduchý.

V druhej otázke sa vyžadovalo hodnotenie intuitívnosti nástroja, teda získať informácie o tom, či aj nový užívateľ, ktorý začne nástroj používať, dokáže s týmto nástrojom čo najviac pracovať bez toho, aby musel prejsť návod. Výsledky tejto otázky je zobrazený v grafe 5.2. Podľa tohto hodnotenie užívateľmi je nástroj nadpriemerne intuitívny, čo je možné odôvodniť aj tým, že nástroj neponúka veľa zbytočných dodatkových funkcionalít.



Obr. 5.1: Výsledky hodnotenie prvej otázky, ktorá zisťuje prvý dojem z nástroja po spustení.

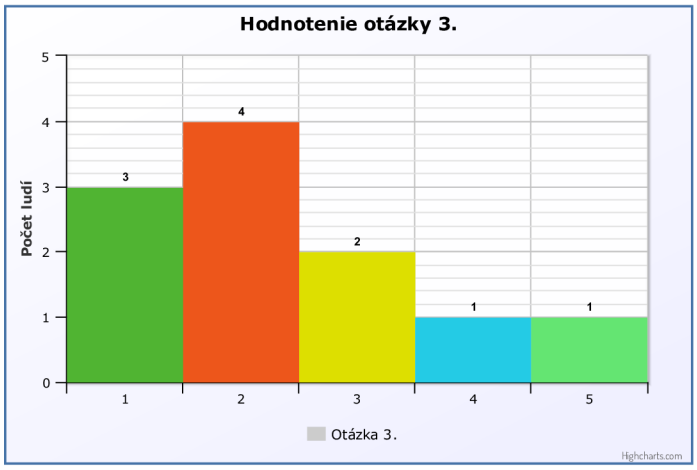


Obr. 5.2: Výsledky hodnotenie druhej otázky, ktorá zisťuje intuitívnosť nástroja.

Z pripomienok užívateľov vyplýva, že jedinou funkcionalitou, ktorá nebola bez nahliadnutia do manuálu pochopená, bola možnosť zadať kódy na ignorovanie.

Tretia otázka sa zaoberala potom už priamo formou, akou je zobrazovaný výstup. V tejto otázke bola snaha zistiť, či je výstup čitateľný a či je pochopiteľné, čo je výstupom. Hodnotenie tejto otázky je popisované grafom 5.3, na ktorom je možné si všimnúť, že hodnotenie je rôznorodé. Avšak väčšina hodnotení bola podpriemerná. Ako dôvod užívatelia v pripomienkach uvádzali, že výstup v kóde je prijateľný, keďže zobrazuje aj číslo riadku, a teda je ľahké tento kódový fragment nájsť v pôvodnom súbore. Problémovým bol až výstup vo forme stromu. Uviedli, že ten je náročné pochopiť v prípade nájdenie dlhého zhodného kódového fragmentu.

Nezávisle na týchto otázkach užívatelia pridali niekoľko návrhov na zmenu. Vo väčšine vyplnených dotazníkov sa vyskytovala pripomienka, ktorá vyžadovala nejakú číselnú hodnotu, predstavujúcu podobnosť. Taktiež jednou z frekventovaných pripomienok bolo nejakým spôsobom pracovať s grafom tak, aby bol viac čitateľnejší, napr. rozbaľovanie a schovávanie podstromov.



Obr. 5.3: Výsledky hodnotenie tretej otázky, ktorá zisťuje formu výstupu porovnávania.

5.2.2 Užívateľské testovanie detekcie

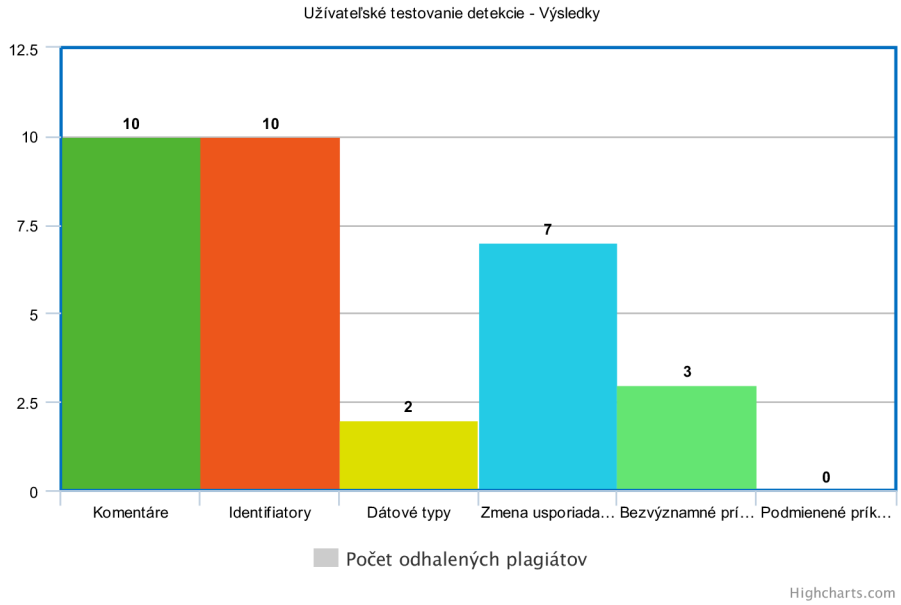
Druhá fáza užívateľského testovania spočívala v podobnom postupe, ako interné testovanie. Ide teda predovšetkým o testovanie detekcie, pri využívaní rôznych identifikovaných zastieracích techník. V tomto prípade užívatelia obdržali hlavný zdrojový súbor (výpis 5.1 pre C, a príloha A.1, alebo A.2) v jazyku, ktorý si mohli sami vybrať z ponúkaných troch jazykov. Následne dostali za úlohu vytvoriť nový zdrojový kód, práve z predlohy, ktorou je hlavný zdrojový kód, za použitia definovaných zastieracích techník. Išlo teda o 6 techník, ktoré postupne testujúci použili a vytvorili tak nový zdrojový kód, ktorý bol následne, spolu s hlavným súborom, vstupom detekcie. Výsledkom týchto testov potom bola úspešnosť detekcie plagiátu, čiže vykonané detekcie, pre danú modifikáciu, ktoré odhalili minimálne jednu dvojicu zhodných kódových fragmentov. Tak ako v internom testovaní, aj v tomto prípade išlo o modifikácie:

1. Zmena komentárov
2. Zmena identifikátorov
3. Zmena dátových typov (iba pre jazyk C)
4. Zmena usporiadania kódu
5. Pridanie zbytočných a bezvýznamných príkazov
6. Zmena podmienených príkazov

Výsledky tohto testovania potom zobrazuje histogram 5.4. Tak, ako bolo zistené už v internom testovaní, aj v tomto prípade prvé dve modifikácie neovplyvnili žiadnym spôsobom detekciu a všetkých 10 vzoriek od testujúcich osôb bolo úspešne odhalených.

Pri modifikácii, ktorá sa týkala zmeny dátových typov, išlo, opäť ako v internom testovaní, iba o jazyk C. Užívatelia, ktorý si tento jazyk nevybrali, tento test nevykonávali. Celkovo teda zmenu dátových typov vykonali 4 užívatelia, u ktorých bola detekcia úspešná v 2 prípadoch. Neúspešnosť ostatných spočívala v rovnakých zmenách, aké boli popísané v internom testovaní, a teda tým, že boli zamenené primitívne dátové typy, ale aj zmena premennej na ukazovateľ alebo pole.

V ďalšom prípade, čiže v zmene usporiadania, boli niektoré vzorky neodhalené. Dôvodom bol štýl zmeny usporiadanie. Premiestnenie časti kódu, ktorá obsahuje menej ako minimálny počet poduzlov a tým pádom nebude odhalená, zmení svojou absenciou na pôvodnom mieste a svojou prítomnosťou na novom mieste AST. Práve preto takáto zmena nebude odhalená. V prípade pridania bezvýznamných príkazov záležalo v akom množstve



Obr. 5.4: Výsledky užívateľského testovania detekcie plagiátov, ktoré využívajú identifikované techniky na zakrytie plagiátorstva.

a na akých miestach tieto príkazy boli pridané a v poslednej modifikácii, čiže zmene podmienených príkazov prišlo práve k tomu, čo bolo spomínané v internom testovaní. Celkovo tak užívateľské testovanie potvrdilo výsledky interného testovania. Taktiež potvrdilo to, že schopnosť detekcie je silne závislá na štýle vstupných súborov.

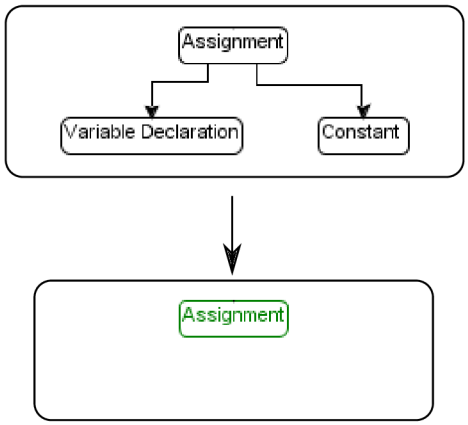
5.2.3 Zmeny implementované na základe výsledkov testovania

Na základe prvej časti užívateľského testovania, ktorá sa týkala hlavne vizuálnej stránky nástroja boli implementované dodatočné prvky. Prvým takýmto doplnkom je číselná hodnota podobnosti kódových fragmentov [18]. Tú definuje rovnica:

$$P = \frac{\sum f(X)}{AN} \tag{5.1}$$

kde X je nájdený podstrom, funkcia f vracia počet poduzlov v danom strome X a AN je počet všetkých poduzlov celého vstupného projektu. Jedná sa teda o sumu všetkých poduzlov nájdených podstromov, vydelenú počtom všetkých uzlov. Avšak tento výpočet sa vykonáva len pre jeden projekt.

Druhým pridaným prvkom je možnosť zmenšovať graf na základe výberu uzlu, na ktorý je následne kliknuté. Takémuto uzlu sú tým všetky jeho poduzly schované. Tým sa docieli zmenšenie grafu a pri veľkom strome k lepšej čitateľnosti. Takéto schovanie poduzlov zobrazuje obrázok 5.5.



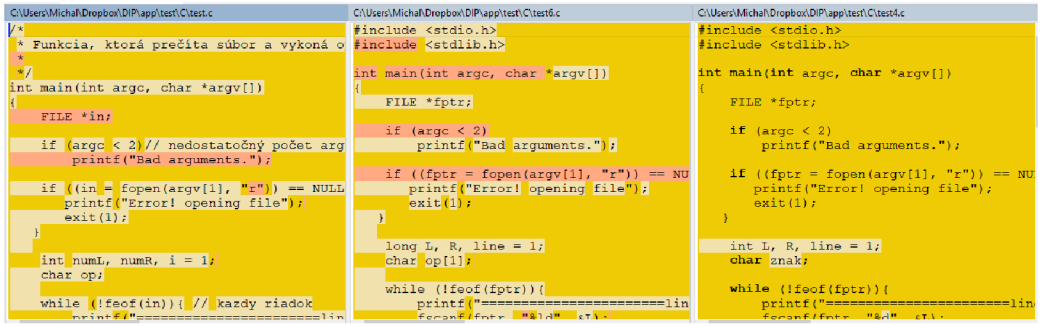
Obr. 5.5: Pridaný prvok manipulácie s výstupným zobrazením AST pomocou zabalení poduzlov vybraného poduzla. Pri takomto zabalení je uzol zvýraznený zelenou farbou.

5.3 Porovnanie s existujúcimi nástrojmi na zobrazovanie rozdielov v zdrojových kódoch

Nástroj sám o sebe nepatrí do rodiny nástrojov, ktoré zobrazujú rozdiel, ale práve naopak zobrazuje zhodné časti. Avšak keďže sa jedná o nástroj, ktorý sa snaží nie len o čo najefektívnejšiu detekciu, ale aj o čo najvhodnejšiu vizualizáciu výstupu detekcie, ktorá okrem nájdenej zhody, zobrazí aj v čom príslušný AST je rovnaký, je vhodné porovnať tento nástroj s nástrojmi, ktoré graficky znázorňujú rozdiely v zdrojových súboroch.

5.3.1 WinMerge

WinMerge [20] je nástroj pre operačný systém *Windows*, ktorý slúži na vizuálne zobrazovanie rozdielov a v súboroch, alebo adresároch, a ich spájanie. Najčastejšie sa používa na rozlišovanie, čo sa zmenilo v rôznych verziách súborov a spájanie týchto zmien. Taktiež podporuje *Unicode*, editor s rôznymi farbami, *Windows Shell* integráciu. Rozdiel súborov zobrazuje zvýrazňovaním odlišností na riadkoch. Identifikátorom je tak, ako aj v prípade



Obr. 5.6: *Výstup nástroja WinMerge, ktorý slúžia na zobrazovanie rozdielov v akýchkoľvek súboroch. Naraz podporuje zobrazovanie rozdielov v až troch súboroch.*

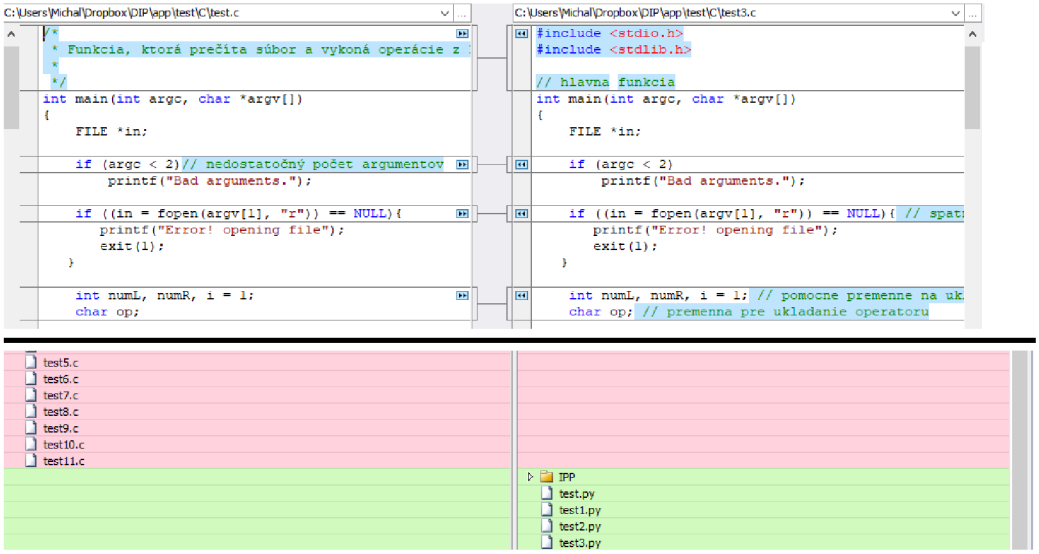
tuná vytvoreného nástroja, cesta k súboru. Naraz umožňuje zobrazovať rozdiely medzi tromi súbormi. Jeho výhodou je aj preklad do rôznych jazykov. Výstup pri porovnávaní hlavného súboru v jazyku C, popísaného v internom testovaní a jedného z modifikovaných súborov

v tomto nástroji zobrazuje obrázok 5.6. Odlišnosti označuje rôznymi farbami (štandardne zlatou).

Pri porovnaní s tu popisovaným nástrojom, *WinMerge* zobrazuje rozdiely v celom vstupnom kóde, teda nezobrazuje len jednotlivé časti. Taktiež neumožňuje iný, než textový výstup. Avšak, ako bolo spomenuté naraz dokáže porovnávať až tri súbory. Podobne, ako aktuálny nástroj, nezvýrazňuje syntax programovacích jazykov.

5.3.2 Code Compare

Na rozdiel od *WinMerge*, ktorý slúži na zobrazenie rozdielov v akýchkoľvek súboroch, *Code Compare* je zameraný na programovacie jazyky, a tým pádom podporuje zvýraznenie syntaxe. Taktiež tento nástroj obsahuje integráciu *VisualStudio*. Avšak opäť sa jedná o nástroj využívaný hlavne na spájanie súborov medzi rôznymi verziami. Okrem zdrojových kódov, *Code Compare* umožňuje nájsť rozdiely aj medzi adresármi. Avšak zobrazuje už len rozdiel medzi stromovou hierarchiou súborov, a teda nie priamo obsah. Nástroj popisovaný v tejto práci tiež umožňuje ako vstup zadať adresár, ale jeho obsah je spojený a je teda považovaný za jeden súbor. Výstup tohto nástroja pre obe možnosti zachytáva obrázok 5.7.



Obr. 5.7: Výstup nástroj *Code Compare* pre dva súbory z testovacieho balíka pre jazyk C v hornej časti, a rozdiel medzi balíkmi pre jazyk C a Python v dolnej časti.

Existuje veľa takýchto nástrojov, ktoré zobrazujú rozdiely, či už obecne v súboroch, alebo v adresároch. Zväčša sú ich výstupy textové, s prípadným zvýraznením syntaxe. Rozdiel medzi týmito nástrojmi a nástrojom predstaveným v tejto práci je aj vtom, že nástroj nezobrazuje celý vstupný text so zvýraznením rovnakej časti, ale zobrazuje len danú časť, s príslušnými číslami riadkov. Taktiež neponúkajú stromovú formu nájdených rozdielov a možnosť prepínať medzi týmito formami výstupu. Vo všeobecnosti však tieto nástroje majú oveľa menšiu časovú zložitosť, keďže na detekciu sa v tomto nástroji využíva syntaktická analýza, transformácia derivačného stromu do AST a následné porovnávanie všetkých vstupných projektov.

5.4 Ďalší možný vývoj nástroja

Existuje veľa možných rozšírení tohoto nástroja, napr. rozšírenie podpory pre nový jazyk. Nato, aby sa táto podpora pre akýkoľvek iný jazyk pridala, je nutne implementovať nové triedy. Prvým krokom by bolo získanie gramatiky daného jazyka vo formáte *.g4*. Tie, pre veľa jazykov, už sú pripravené priamo v repozitári využívaného nástroja *ANTLR* [13]. Následne je možné priamo v prostredí *IntelliJ IDEA* vygenerovať nové triedy, predstavujúce lexikálnu a syntaktickú analýzu.

Po získaní analyzátorov, je možné zo vstupného zdrojového súboru v danom jazyku, získať derivačný strom. Pre podporu chceného jazyka je teda nutné vytvoriť triedu obsahujúcu návštevníkov (*visitore*) pre každé pravidlo v gramatike, teda pre každý uzol derivačného stromu, a pretransformovať tento strom do AST, ktorý pozostáva z už vytvorených tried, ktorých nadtriedou je abstraktná trieda *ASTnode*. Je možné vytvoriť úplne nové triedy, ale nadtrieda musí byť rovnaká. Pokiaľ teda budú vytvorené nové triedy predstavujúce uzly, je taktiež nutné pridať ich ohodnotenie a získanie počtu poduzlov do triedy *Valuator*. Takéto nové triedy, pokiaľ vzniknú, je taktiež nutné pridať do triedy *VertexCreator*, ktorá slúži na vyplnenie grafu. A ak je v záujme aj podpora ukladania výsledkov s novými triedami pre uzly, je taktiež nutné pridať tieto uzly do triedy *Saver*, ktorá zaobstaráva transformácie nájdených stromov do *JSON* formátu. V prípade, že nové triedy reprezentujúce uzly AST nebudú vytvorené a sú využité len existujúce, stačí vytvoriť návštevníkov derivačného stromu.

Ďalším možným vývojom sú rôzne heuristiky na zlepšenie detekcie. Jednou z takýchto heuristík je transformácia podmienok v podmienených príkazoch. Keďže sa často, pri snahe zakryť plagiátorstvo, siahne na zmenu podmienok s logicky rovnakým významom, za použitia iných, často opozičných operácii, by takáto heuristika pretransformovala podmienky do jednotnej formy. Napríklad každú nerovnosť transformovať na rovnosť a zameniť blok príkazov, ktoré sa vykonávajú v prípade, že je podmienka splnená, s blokom, ktorý sa vykoná, keď je podmienka nesplnená. Tým by sa dosiahla lepšia schopnosť odhaliť zhodné kódové fragmenty, na ktoré bola aplikovaná plagiátorská technika, týkajúca sa zmeny podmienených príkazov.

Kapitola 6

Záver

V úvode práce bola rozoberaná všeobecná problematika plagiátorstva a možné obecné popisy prístupov, ako je možné postupovať, v prípade implementácie algoritmu na detekciu plagiátov. V práci sú spomenuté taktiež existujúce nástroje na vyhľadávanie zhôd v zdrojových kódoch.

V tejto práci bol teda navrhnutý nástroj na detekciu plagiátov, ktorý využíva stromový prístup, teda pomocou syntaktickej analýzy získa zo vstupného zdrojového súboru derivačný strom, ktorý je potom transformovaný do príslušnej formy abstraktného syntaktického stromu (AST). Následne sú rozdelené podstromy AST do množín tak, že v jednej množine sa nachádzajú podstromy s rovnakým počtom poduzlov. Takéto množiny vzniknú pre všetky vstupné projekty a tieto množiny sú medzi všetkými projektami porovnávané na hodnotu uzlu. Hodnotu uzlu získame *hash* funkciou, ktorá ohodnocuje uzly podľa ich typu a typov ich potomkov. Taktiež bolo navrhnuté grafické užívateľské rozhranie, ktoré umožní užívateľovi definovať vstupné súbory a poprípade súbory, ktoré udržujú kód, ktorý by mal byť ignorovaný, a zobrazí výsledok detekcie vo forme častí kódov a AST.

Nástroj bol následne implementovaný podľa toho, ako bol navrhnutý a riadne testovaný, či už interne, alebo užívateľsky.

Zhodnotenie schopnosti odhaliť zhody v zdrojových kódoch je možné na základe testovania. Nástroj dokáže eliminovať niektoré techniky na zakrytie skutočnosti, že prišlo k plagiátorstvu, avšak, tak ako ukázalo testovanie, jeho schopnosť záleží na vstupných kódoch. Všeobecne je možné povedať, že čím dlhšie sú vstupné zdrojové kódy, tým je vyššia pravdepodobnosť lepšej detekcie.

Výstupy detekcie sú zobrazované užívateľovi ako AST, alebo priamo príslušných častí kódu. Taktiež, na základe testovania, bola doplnená funkcionalita pre vizualizáciu AST, z dôvodu lepšej čitateľnosti. Na základe spomínaných možností ďalšieho vývoja, je možné nástroj zefektívniť, alebo ho spraviť viac robustným, pridaním podpory pre ďalšie jazyky.

Literatúra

[1] *AC 2*. https://github.com/manuel-freire/ac2, [Online; navštíveno 4.1.2019].

[2] Cebrián, M.; Alfonseca, M.; Ortega, A.: The normalized compression distance is resistant to noise. *IEEE Transactions on Information Theory*, ročník 53, č. 5, 2007: s. 1895–1900.

[3] *CodeMatch*. https://www.safe-corp.com/products_codematch.htm, [Online; navštíveno 4.1.2019].

[4] Cordy, J. R.; Roy, C. K.: The NiCad clone detector. In *2011 IEEE 19th International Conference on Program Comprehension*, IEEE, 2011, s. 219–220.

[5] Faidhi, J. A.; Robinson, S. K.: An empirical approach for detecting program similarity and plagiarism within a university programming environment. *Computers & Education*, ročník 11, č. 1, 1987: s. 11–19.

[6] Freire, M.; Sopan, A.: Gene similarity uncovers mutation path vast 2010 mini challenge 3 award: Innovative tool adaptation. In *Visual Analytics Science and Technology (VAST), 2010 IEEE Symposium on*, IEEE, 2010, s. 287–288.

[7] *JavaFX*. https://openjfx.io/, [Online; navštíveno 10.5.2019].

[8] *JavaFX Scene Builder*. https://www.oracle.com/technetwork/java/javase/downloads/javafxscenebuilder-info-2157684.html, [Online; navštíveno 10.5.2019].

[9] *Jgraph*. https://www.jgraph.com/index.html, [Online; navštíveno 10.5.2019].

[10] Marcus, A.; Maletic, J. I.: Identification of high-level concept clones in source code. In *Automated Software Engineering, 2001.(ASE 2001). Proceedings. 16th Annual International Conference on*, IEEE, 2001, s. 107–114.

[11] Michail, D.; Kinable, J.; Naveh, B.; aj.: JGraphT–A Java library for graph data structures and algorithms. *arXiv preprint arXiv:1904.08355*, 2019.

[12] *Moss - Measure Of Software Similarity*. https://theory.stanford.edu/~aiken/moss/, [Online; navštíveno 4.1.2019].

[13] Parr, T.: *ANTLR*. https://github.com/antlr, [Online; navštíveno 12.5.2019].

[14] Parr, T.: *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013.

[15] Peciar, Š.: *Slovník slovenského jazyka*. Vydavatel'stvo SAV, 1959.

[16] Roy, C. K.; Cordy, J. R.; Koschke, R.: Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of computer programming*, ročník 74, č. 7, 2009: s. 470–495.

[17] Schleimer, S.; Wilkerson, D. S.; Aiken, A.: Winnowing: local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, ACM, 2003, s. 76–85.

[18] Tao, G.; Guowei, D.; Hu, Q.; aj.: Improved plagiarism detection algorithm based on abstract syntax tree. In *Emerging Intelligent Data and Web Technologies (EIDWT), 2013 Fourth International Conference on*, IEEE, 2013, s. 714–719.

[19] Whale, G.: Identification of program similarity in large populations. *The Computer Journal*, ročník 33, č. 2, 1990: s. 140–146.

[20] *WinMerge*. http://winmerge.org/?lang=en, [Online; navštíveno 19.5.2019].

[21] Zeidman, B.: What, Exactly, Is Software Plagiarism? *Intellectual property today*, 2007.

Príloha A

Prílohy pre testovanie

A.1 Hlavný súbor interného testovania pre jazyk Python

```
#toto je nejaky kod pre tuto funkciu
def main(argc,argv):
   #a :)
   a = []
   if (argc < 2): #malo argc aj ked zbytocne
      print("Bad arguments.")

   if (len(argv) < 1):
      print("Too few arguments")
      exit(1)
   a = argv
   i = 0
   numL = 0
   numR = 0
   op = ""
   while (i < len(argv)): #kazdy list v~liste
      print("========line " + str(i+1) + " ==========")
      numL = argv[i][0]
      j = 1
      while (j < len(argv[i])): #list sa spocitava ziadna precendica :D
         op = argv[i][j]
         j += 1
         numR = argv[i][j]
         j += 1
         print(numL)
         print(op)
         print(numR)
         if (op == "+"):
            numL += numR
         elif (op == "-"):
            numL -= numR
         elif (op == "/"):
            numL /= numR
         elif (op == "*"):
            numL *= numR
         else:
```

```
            print("Wrong operator?")
            exit(1)
         print("  " + str(numL)) #print odsadeny vysledok
      i += 1
   print("========================================")

main(3,[[1,"+",2,"*",6],[2,"*",2,"/",2]]) #volanie main
```

Výpis A.1: Hlavný súbor testovacej sady pre jazyk Python. Jedná sa o kalkulačku, ktorá ako vstup berie zoznam zoznamov matematických operácii a čísiel, kde každý vnútrný zoznam reprezentuje jeden výpočet. Každý medzi výpočet vypisuje.

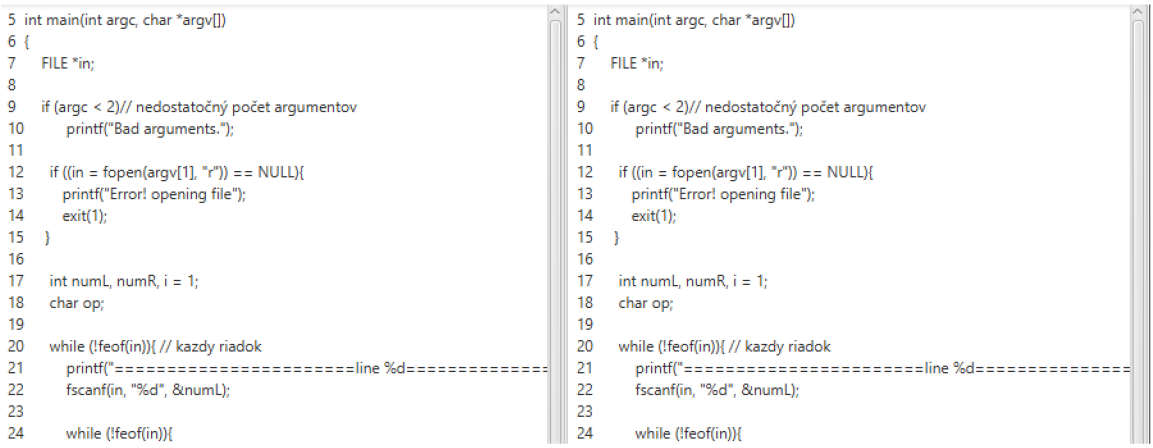
A.2 Hlavný súbor interného testovania pre PHP

```
<?php
   if ($argc < 2){ # malo argc aj ked zbytocne
      echo "Bad arguments.";
      exit(1);
   }
   if (!($in = fopen($argv[1], 'r'))){ # opening output file
      echo "Error! opening file";
      exit(1);
   }
   $i = 1;
   $numL = 0;
   $numR = 0;
   $op = '+';
   while (!feof($in)) { # each line
      echo "========line ", $i, " ==========\n";
      $numL = (int)fgetc($in);
      while (!feof($in)) { # each operator
         $op = fgetc($in);
         if($op == "\n") {
            break;
         }
         if(feof($in)) {
            break;
         }
         echo $numL;
         echo $op;
         $numR = (int)fgetc($in);
         echo $numR;
         switch ($op) { # select operator
            case '+':
               $numL = $numL + $numR;
               break;
            case '-':
               $numL = $numL - $numR;
               break;
            case '/':
               $numL = $numL / $numR;
               break;
```

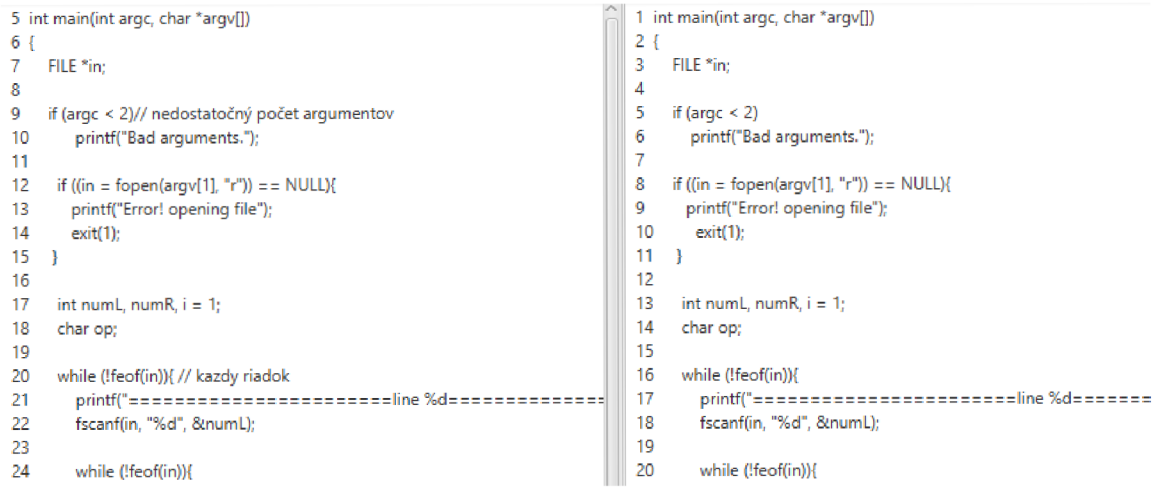
```
            case '*':
               $numL = $numL * $numR;
               break;
            default:
               echo "Wrong operator?";
               exit(1);
               break;
         }
         echo "=", $numL, "\n"; # result
      }
      $i = $i + 1; # new line
   }
   fclose($in);
?>
```

Výpis A.2: Hlavný súbor testovacej sady pre jazyk PHP. Jedná sa o kalkulačku, ktorá ako vstup berie zoznam zoznamov matematických operácii a čísiel, kde každý vnútrný zoznam reprezentuje jeden výpočet. Každý medzi výpočet vypisuje.

A.3 Výsledky testov



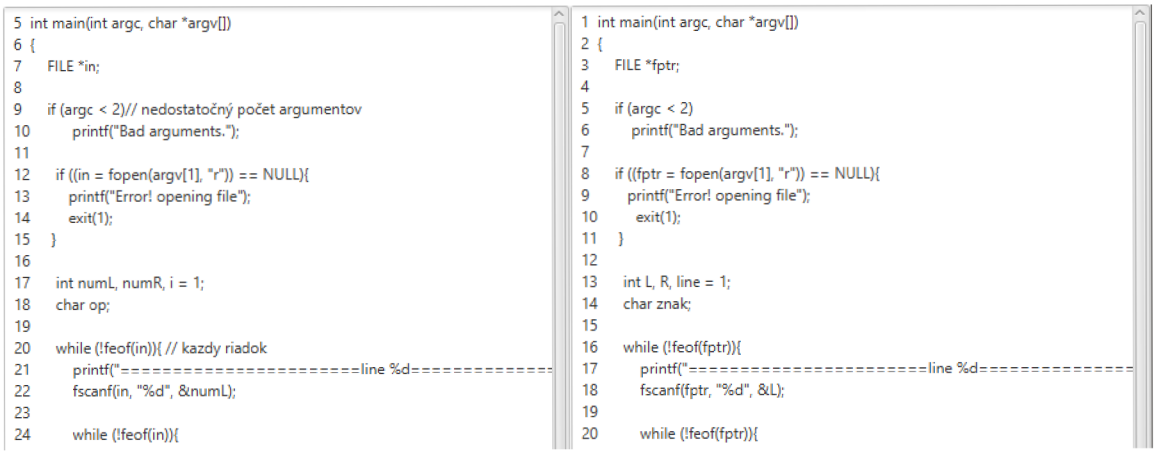
Obr. A.1: Výsledok testu č. 1, ktorý pozostáva z dvoch súborov, kedy druhý je čistou kópiou prvého. Výsledok je zobrazený vo forme kódu. Zobrazovaná je len časť výstupu.



```
5  int main(int argc, char *argv[])
6  {
7     FILE *in;
8
9     if (argc < 2)// nedostatočný počet argumentov
10       printf("Bad arguments.");
11
12    if ((in = fopen(argv[1], "r")) == NULL){
13       printf("Error! opening file");
14       exit(1);
15    }
16
17    int numL, numR, i = 1;
18    char op;
19
20    while (!feof(in)){ // kazdy riadok
21       printf("=======================line %d==============
22       fscanf(in, "%d", &numL);
23
24       while (!feof(in)){
```

```
1  int main(int argc, char *argv[])
2  {
3     FILE *in;
4
5     if (argc < 2)
6        printf("Bad arguments.");
7
8     if ((in = fopen(argv[1], "r")) == NULL){
9        printf("Error! opening file");
10       exit(1);
11    }
12
13    int numL, numR, i = 1;
14    char op;
15
16    while (!feof(in)){
17       printf("=======================line %d=======
18       fscanf(in, "%d", &numL);
19
20       while (!feof(in)){
```

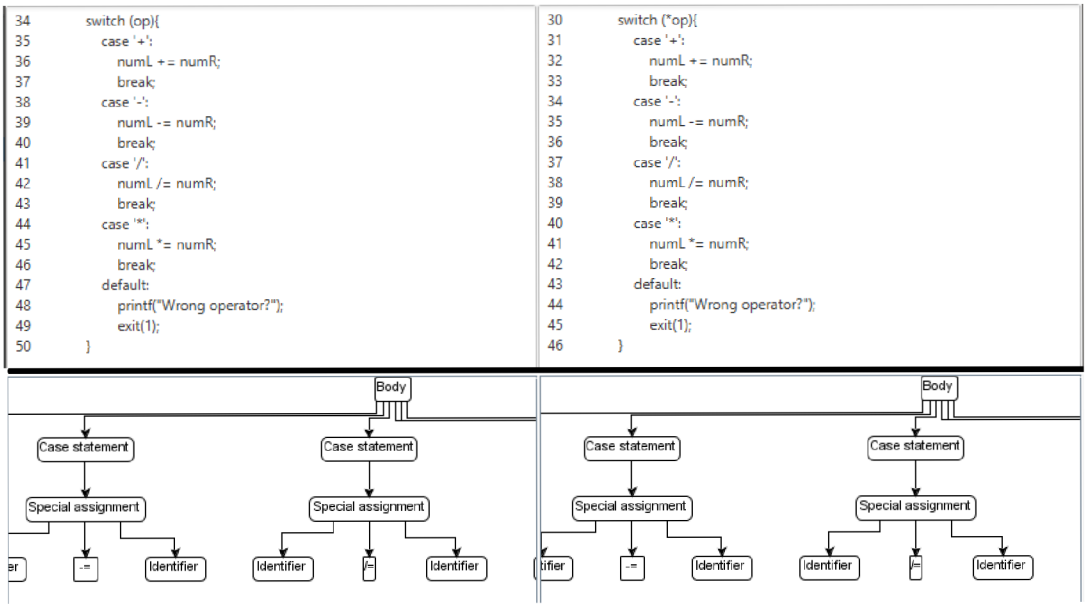
Obr. A.2: Výsledok testu č. 2, ktorý zobrazuje výsledok porovnávania dvoch vstupných súborov, kedy druhým súborom je modifikácia hlavného odstránením komentárov. Výsledok testu 3, kedy sú komentáre pozmenené je obdobný. Zobrazovaná je len časť výstupu.



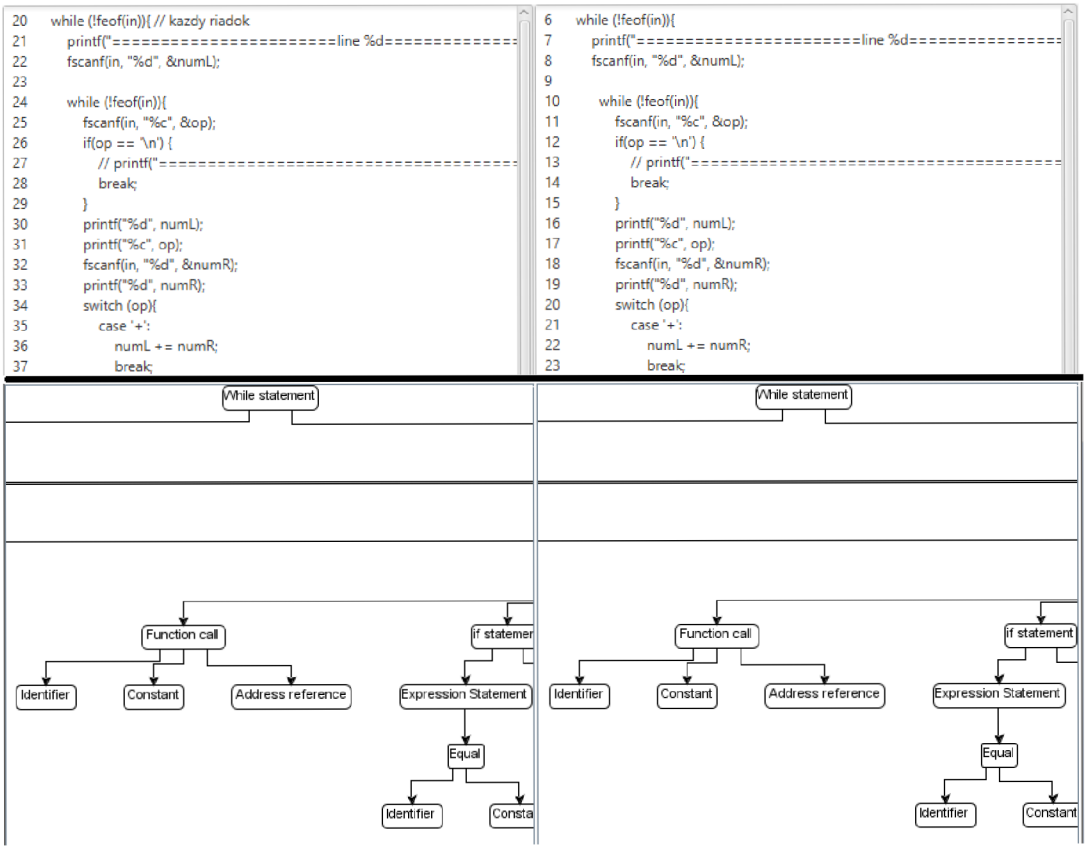
```
5  int main(int argc, char *argv[])
6  {
7     FILE *in;
8
9     if (argc < 2)// nedostatočný počet argumentov
10       printf("Bad arguments.");
11
12    if ((in = fopen(argv[1], "r")) == NULL){
13       printf("Error! opening file");
14       exit(1);
15    }
16
17    int numL, numR, i = 1;
18    char op;
19
20    while (!feof(in)){ // kazdy riadok
21       printf("=======================line %d==============
22       fscanf(in, "%d", &numL);
23
24       while (!feof(in)){
```

```
1  int main(int argc, char *argv[])
2  {
3     FILE *fptr;
4
5     if (argc < 2)
6        printf("Bad arguments.");
7
8     if ((fptr = fopen(argv[1], "r")) == NULL){
9        printf("Error! opening file");
10       exit(1);
11    }
12
13    int L, R, line = 1;
14    char znak;
15
16    while (!feof(fptr)){
17       printf("=======================line %d==============
18       fscanf(fptr, "%d", &L);
19
20       while (!feof(fptr)){
```

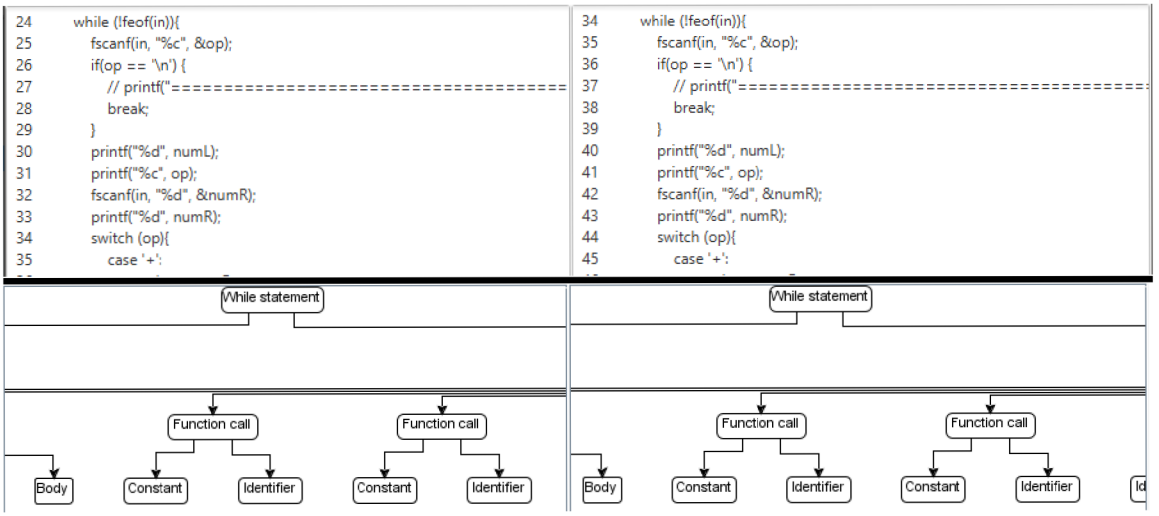
Obr. A.3: Výsledok testu č. 4, ktorý zobrazuje výsledok pri porovnaní zdrojových kódov so zmenou identifikátorov. Zobrazovaný výstup je len časť.



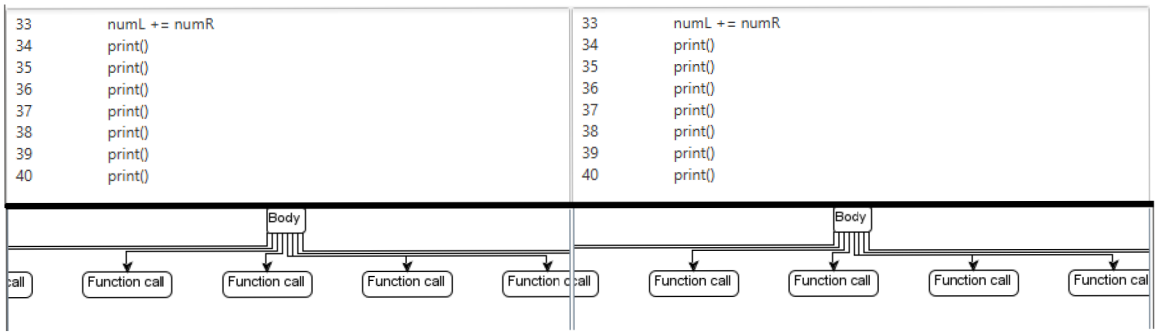
Obr. A.4: Výsledok testu č. 5, ktorý zobrazuje výsledok pri zmene dátových typov v zdrojových kódoch. V hornej časti je možné vidieť výsledok formou kódu, v dolnej časti formou AST.



Obr. A.5: Výsledok testu č. 6, ktorý zobrazuje vystrihnutie časti kódu do inej funkcie. V hornej časti je možné vidieť kód, v dolnej časť AST.



Obr. A.6: Výsledok testu č. 7, ktorý zobrazuje výsledok porovnavania pri vytvorení a použití bezvýznamnej funkcie.



Obr. A.7: Výsledok testu č. 8 po modifikácii, ktorý zobrazuje nájdené zväčšené telo *if* príkazu. V hornej časti je kód a v dolnej časť AST.

A.4 Dotazník pre užívateľské testovania

Dotazník

Testovanie nástroja PlagDetect

Meno: _____

Pri odpovedaní zakrúžkuje číselnú hodnotu kde, 1 znamená najhoršie a 5 najlepšie.

1.

1.	Ako na vás nástroj pôsobí pri prvom pohľade?	*1*	*2*	*3*	*4*	*5*
2.	Príde vám práca s nástrojom intuitívna?	*1*	*2*	*3*	*4*	*5*
3.	Je pre vás zobrazenie výsledku čitateľné?	*1*	*2*	*3*	*4*	*5*

Prípadné pripomienky píšte tu:

_____

_____

_____

_____

_____

_____

_____

2.

Podľa vzorového zdrojového súboru vytvorte nový kód, v ktorom zmeníte:

1. Komentáre
2. Identifikátory
3. Zmeníte dátové typy (iba pre C)
4. Usporiadajte kód inak (napr. vystrihnúť časť kódu)
5. Pridajte nepotrebné a bezvýznamné príkazy
6. Podmienečné príkazy

Príloha B

Obsah priloženého pamäťového média

- app – Zdrojového kódy aplikácie, spustiteľný *jar* a dokumentácia
- README.txt – obsahuje širší popis štruktúry súborov
- text – zdrojové súbory textu správy a výsledný text vo formáte PDF