VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

DIFF PRO RŮZNÉ TYPY DOKUMENTŮ

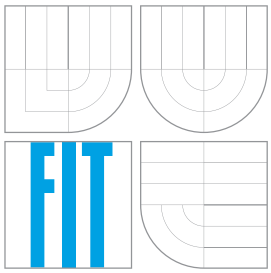
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. MICHAL ZEMKO

BRNO 2011

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

DIFF PRO RŮZNÉ TYPY DOKUMENTŮ

MULTIPLE DOCUMENT TYPE DIFF

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. MICHAL ZEMKO

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. PETR CHMELAŘ

BRNO 2011

Abstrakt

Tato diplomová práce se zabývá porovnáním různych typů dokumentů, především zdrojových kódů. Popisuje problematiku porovnání zdrojových kódů a různé způsoby jejího řešení, od jednoduchého řádkového srovnání, až po srovnání AST. Zvolenou metodou bylo srovnání na základě lexikální analýzy. Ta je v práci popsána i s nástroji na její automatizaci. Cílem bylo navrhnout a implementovat modulární aplikaci porovnávající různé typy dokumentů. Implementovaný modul porovnává zdrojové kódy v programovacích jazycích C/C++, Java a Python. Tento modul je snadno rozšiřitelný o srovnávání dalších jazyků.

Abstract

This thesis deals with comparing different types of files, especially source codes. It describes the problem of comparing source code and different ways of solving this problem, from simple line comparison, to AST comparison. Chosen method was comparison based on lexical analysis. This is also described in the work with instruments of its automation. The goal of this thesis is to design and implement modular application, which compares different types of files. The implemented module compares source code in programming languages C/C++, Java a Python. This module is easily extendable for comparisons with other languages.

Klíčová slova

diff, porovnávaní, sémantické porovnávaní, zdrojový kód, lexikální analýza

Keywords

diff, comparation, semantic comparation, source code, lexical analysis

Citace

Michal Zemko: Diff pro různé typy dokumentů, diplomová práce, Brno, FIT VUT v Brně, 2011

Diff pro různé typy dokumentů

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Petra Chmelaře.

.....
Michal Zemko
23. května 2011

Poděkování

Děkuji vedoucímu práce Ing. Petrovi Chmelařovi za odbornou pomoc a pedagogické vedení při řešení mé diplomové práce.

Obsah

Kapitola 1

Úvod

Pri práci na počítači často potrebujeme porovnať dva dokumenty. Dôvody môžu byť rôzne.

Napríklad pri práci na dôležitom projekte užívateľ priebežne ukladá čiastkové výsledky svojej práce a chce porovnať zmeny medzi dvoma verziami. Vhodné je to hlavne vtedy, ak sa so zmenou zaniesla do projektu chyba. Zmeny nemusí kontrolovať sám autor, ale aj iné osoby ako vedúci projektu, aby zhodnotil vynaložené úsilie alebo tímový kolega, ak potrebuje porozumieť vývoju projektu.

Iný dôvod pre porovnávanie dokumentov, je získať zoznam zmien medzi dokumentami. Toto je užitočné pri verzovaní dokumentov, pretože namiesto uloženia celého dokumentu stačí uložiť len prevedené zmeny voči originálu. Takto ušetríme miesto na disku. V prípade prenosu po sieti, je táto úspora výrazne citeľná.

V súčastnosti existuje množstvo porovnávacích programov, ale väčšina z nich podporuje len jednoduché textové porovnávanie, prípadne porovnanie zložiek. Druhou skupinou sú úzko špecializované porovnávacie programy, ktoré sa zameriavajú na porovnávanie konkrétnych typov dokumentov, ako sú súbory kancelárskych balíkov Microsoft Office alebo Open Office, pdf, zdrojové kódy niektorých jazykov, xml a iné typy súborov.

Každý typ dokumentu je špecifický a vyžaduje si samostatný prístup. Textové dokumenty sa porovnávajú riadok po riadku s niektorými možnosťami ako ignorovanie bielych znakov alebo veľkosti písmen. Dokumenty kancelárskych balíkov a pdf dokumenty obsahujú tiež text, avšak inak interne reprezentovaný. XML dokumenty sú zas špecifické svojou stromovou štruktúrou. Archívy zas množstvom typov komprimácie.

Samotnou kapitolou sú zdrojové kódy. Tu existujú rôzne prístupy porovnávania, od textového cez porovnávanie sekvencií tokenov a abstraktných syntaktických stromov až po porovnávanie grafov toku riadenia a dát. Práca so zdrojovými kódmi na vyššej úrovni abstrakcie si vyžaduje znalosti lexikálnej a syntaktickej analýzy, resp. znalosti nástrojov na generovanie takýchto analyzátorov (lex, yacc). Veľmi dôležité a potrebné sú aj rôzne porovnávacie algoritmy.

Cieľom tejto práce je vyvinúť modulárnu aplikáciu – `mediadiff`, ktorá porovnáva rôzne typy súborov. Od obyčajných textových súborov, cez multimediálne súbory, zdrojové kódy programovacích jazykov, až po aplikačne a obsahovo špecifické typy súborov. Výsledkom práce by mala byť jedna aplikácia, ktorá porovná rôzne typy dokumentov a zobrazí ich rozdiel. Ako modul, implementujúci porovnávanie konkrétneho typu súborov, bolo zvolené porovnávanie zdrojových kódov rôznych programovacích jazykov. Porovnávanie prebieha v dvoch krokoch, pomocou lexikálnej analýzy rozdelíme zdrojový kód daného programovacieho jazyka na zoznam tokenov, a v druhej fáze takto získané zoznamy porovnáme s možnosťou ignorovať zmeny v určitých špecifických typoch tokenov, ako napríklad názvy

identifikátorov, literály a iné.

Práca rozoberá v druhej kapitole (2) algoritmy na porovnávanie dokumentov. Popísané sú algoritmy na porovnávanie neštruktúrovaných dát, stromových štruktúr a sémantické porovnávanie zdrojových kódov. V tretej kapitole (3) je detailne popísaná lexikálna analýza zrojových kódov a nástroje, ktoré takúto analýzu dokážu automatizovane vykonávať. Štvrtá kapitola (4) popisuje súčasný stav v oblasti aplikácií a definuje štandardne používané formáty výstupu. Ďalšia kapitola (5) sa venuje analýze problému a návrhu konkrétnej aplikácia, pričom využíva teoretické poznatky popísané v predošlých kapitolách. Šiesta kapitola (6) popisuje implementáciu navrhnutého riešenia a v druhej časti sú uvedené testy a ich výsledky. V predposlednej kapitole 7 sme uviedli možné pokračovanie práce. A nakoniec posledná kapitola, záver (8), popisuje dosiahnuté výsledky a zhodnocuje celkovú prácu.

Táto práca nadväzuje na predchádzajúcu bakalársku prácu [60], v ktorej bolo implementované porovnávanie textu, konfiguračných súborov, súborov systému LaTeX a obrázkov. Práca taktiež vychádza zo semestrálneho projektu [61], v ktorom boli spracované úvodné kapitoly 2 a 4.

Kapitola 2

Algoritmy

V tejto kapitole si popíšeme základné porovnávacie algoritmy. Prvá časť sa venuje porovnávaniu neštruktúrovaných dát, najmä textov. Táto časť vychádza z práce [60]. V druhej časti sú stručne popísané základné algoritmy na porovnávanie stromových štruktúr. V poslednej časti sa venujeme sémantickému porovnávaniu zdrojových kódov.

Vzhľadom na rozsah práce nie je možné uvádzať všetky detailné informácie o danej problematike, preto sú pri niektorých textoch uvedené odkazy na literatúru pre prípadné rozšírenie znalostí.

2.1 Porovnávanie neštruktúrovaných dát

V programoch na porovnávanie textových súborov sa využívajú dva základné algoritmy:

- Algoritmus, ktorý nájde najdlhší spoločný podreťazec oboch súborov. 2.1.1
- Algoritmus, ktorý určí, v akej miere sú dva reťazce podobné. 2.1.3

Postupne si popíšeme základný algoritmus na nájdenie najdlhšieho podreťazca, ktorý sa nachádza v oboch reťazcoch. V anglickej literatúre sa môžeme stretnúť s týmto algoritmom pod názvom "Longest common subsequence"(LCS). Tento algoritmus je klasický informatický problém a je základom pre program `diff` a jemu podobné. Taktiež má veľké uplatnenie v bioinformatike.[41]

Algoritmus LCS je dôležitý pri porovnávaní dvoch textov a hľadaní rozdielov medzi nimi. Ak nájdeme časti súborov, ktoré sú spoločné pre oba súbory, dokážeme veľmi ľahko získať zoznam zmien medzi oboma súbormi. Tento problém si vysvetlíme na nasledujúcom príklade:

Máme dve sekvencie znakov: `a b c d f g h j q z` a `a b c d e f g i j k r x y z`. Chceme nájsť najdlhšiu možnú postupnosť znakov, obsiahnutých v oboch sekvenciách. Táto spoločná sekvencia znakov musí mať rovnaké poradie znakov ako v oboch sekvenciách. V našom prípade je spoločná sekvencia `a b c d f g j z`. Ak porovnáme túto sekvenciu s prvou sekvenciou, zistíme, ktoré znaky sa nenachádzajú v druhej sekvencii, čiže boli zmazané. Keď porovnáme sekvenciu spoločných znakov s druhou sekvenciou, zistíme, ktoré znaky nie sú obsiahnuté v prvej sekvencii, čiže boli do druhej sekvencie pridané. Na tomto princípe pracujú programy na porovnávanie súborov. Po porovnaní spoločnej sekvencie s oboma pôvodnými sekvenciami znakov získame rozdiel oboch súborov. Ak označíme znaky,

ktoré sú jedinečné len v prvom súbore znakom - a znaky jedinečné v druhom súbore znakom +, dostaneme následovný výstup: [39]

```
e   h i   q   k r x y
+   - +   -   + + + +
```

Ak však pre nás nie je podstatné, ktoré časti reťazcov sú odlišné a zaujíma nás iba miera odlišnosti vyjadrená číslom, dokážeme túto mieru odlišnosti vyjadriť pomocou Hammingovej vzdialenosti, respektíve pomocou Levenshteinovej vzdialenosti.

V následujúcich dvoch podkapitolách si vysvetlíme jednotlivé algoritmy. Zameriame sa predovšetkým na algoritmus nájdenia najdlhšieho spoločného podreťazca, ktorý je z hľadiska porovnávania súborov najdôležitejší.

2.1.1 Najdlhší spoločný podreťazec

V tejto kapitole bude algoritmus LCS popísaný podľa pána Cormena [19].

V bioinformatických aplikáciach potrebujeme často porovnávať DNA dvoch (alebo viac) odlišných organizmov. Vzor DNA obsahuje reťazec molekúl nazývaných bázy. Možné bázy sú adenín, guanín, cytosín a thymín. Reprezentujúc každú z týchto báz ich počiatočným písmenom, môže byť vzor DNA vyjadrený ako reťazec nad konečnou množinou $\{A, C, G, T\}$. Napríklad, DNA jedného organizmu môže byť $S_1 = ACCGGTCGAGTGCGCGGAAGCC GGCCGAA$, zatiaľ čo DNA iného organizmu môže byť $S_2 = GTCGTTCGGAATGCCGTT GCTCTGTAAA$. Jedným z cieľov porovnávania dvoch vzorov DNA je určiť, ako "podobné"si dva vzory DNA sú, ako určité meradlo, ako veľmi príbuzné tie dva organizmy sú. Podobnosť môže byť a je definovaná rôznymi spôsobmi. Napríklad, môžeme povedať, že dva vzory DNA sú si podobné, ak jeden je podreťazec druhého. V našom prípade, ani jeden z S_1 a S_2 nie je podreťazcom druhého. Alternatívne by sme mohli povedať, že dva vzory DNA sú si podobné, ak počet zmien potrebných k transformácii jedného do druhého je malý. Avšak iný spôsob merania podobnosti vzorov S_1 a S_2 je nájdenie tretieho vzoru S_3, ktorého bázy sa objavujú v každom zo vzorov S_1 a S_2. Tieto bázy sa musia vyskytovať v rovnakom poradí, ale nie nezbytne postupne. Dlhší vzor S_3 môžeme nájsť, ak sú si S_1 a S_2 viac podobné. V našom prípade, najdlhší vzor S_3 je $GTCGTCGGAAGCCGGCCGAA$.

Formalizujme problém nájdenia najdlhšieho spoločného podreťazca. Podreťazec daného reťazca je iba daný reťazec s nula alebo viac vynechanými elementami. Formálne, je daná sekvencia $X = \langle x_1, x_2, \ldots, x_m \rangle$, iná sekvencia $Z = \langle z_1, z_2, \ldots, z_k \rangle$ je **subsekvencia** X, ak existuje stúpajúca sekvencia $\langle i_1, i_2, \ldots, i_k \rangle$ indexov X taká, že pre všetky $j = 1, 2, \ldots, k$ platí $x_{i_j} = z_j$. Napríklad, $Z = \langle B, C, D, B \rangle$ je subsekvencia sekvencie $X = \langle A, B, C, B, D, A, B \rangle$ s korešpondujúcou postupnosťou indexov $\langle 2, 3, 5, 7 \rangle$.

Pre dané dve sekvencie X a Y môžeme povedať, že sekvencia Z je **spoločná subsekvencia** sekvencií X a Y, ak je Z podsekvencia X aj Y. Napríklad ak $X = \langle A, B, C, B, D, A, B \rangle$ a $Y = \langle B, D, C, A, B, A \rangle$, sekvencia $\langle B, C, A \rangle$ je spoločná podsekvencia oboch sekvencií X aj Y. Sekvencia $\langle B, C, A \rangle$ nie je najdlhšia spoločná podsekvencia sekvencií X a Y, pretože má dĺžku 3 a sekvencia $\langle B, C, B, A \rangle$, ktorá je spoločná v oboch sekvenciách X aj Y má dĺžku 4. Sekvencia $\langle B, C, B, A \rangle$ je najdlhšia spoločná subsekvencia sekvencií X a Y, takisto ako aj sekvencia $\langle B, D, B, A \rangle$. Ďalej už neexistuje spoločná subsekvencia dĺžky 5 alebo dlhšej.

Pri hľadaní **najdlhšej spoločnej subsekvencie** máme dané dve sekvencie $X = \langle x_1, x_2, \ldots, x_m \rangle$ a $Y = \langle y_1, y_2, \ldots, y_n \rangle$ a prajeme si nájsť, čo najdlhšiu spoločnú subsekvenciu sekvencií X a Y. V tejto časti dokážeme, že problém nájdenia najdlhšej spoločnej subsekvencie môže byť vyriešený efektívne použitím dynamického programovania.

Charakteristika najdlhšej spoločnej subsekvencie

Prístup k riešeniu LCS problému hrubou silou znamená vytvoriť všetky subsekvencie sekvencie X a skontrolovať každú takúto subsekvenciu, či nie je tiež subsekvencia sekvencie Y. Každá subsekvencia sekvencie X odpovedá podmnožine indexov $\{1, 2, \ldots, m\}$ sekvencie X. Takto dostaneme 2^m subsekvencií sekvencie X, takže takýto prístup vyžaduje exponenciálny čas $\Theta(n2^m)$, ktorý je nepraktický pre dlhé sekvencie. LCS problém má "optimálnu štruktúru", ktorú dokazuje následujúca veta 2.1.1. Optimálna štruktúra znamená, že problém môžeme rozdeliť na menšie jednoduchšie podproblémy, a tak ďalej, až nakoniec sa riešenie stane triviálne. Ako môžeme vidieť, prirodzené triedy podproblémov zodpovedajú párom "prefixov"dvoch vstupných sekvencií. Ak máme byť dôsledný, je daná sekvencia $X = \langle x_1, x_2, \ldots, x_m \rangle$, definujeme i-tý **prefix** sekvencie X, pre $i = 0, 1, \ldots, m$ ako $X_i = \langle x_1, x_2, \ldots, x_i \rangle$. Napríklad, ak $X = \langle A, B, C, B, D, A, B \rangle$, potom $X_4 = \langle A, B, C, B \rangle$ a X_0 je prázdna sekvencia.

Veta 2.1.1 *Optimálna štruktúra LCS*

Nech $X = \langle x_1, x_2, \ldots, x_m \rangle$ a $Y = \langle y_1, y_2, \ldots, y_n \rangle$ sú sekvencie, a nech $Z = \langle z_1, z_2, \ldots, z_k \rangle$ je nejaká LCS sekvencií X a Y.

1. Ak $x_m = y_n$, potom $z_k = x_m = y_n$ a Z_{k-1} je LCS sekvencií X_{m-1} a Y_{n-1}.
2. Ak $x_m \neq y_n$, potom $z_k \neq x_m$ značí, že Z je LCS sekvencie X_{m-1} a Y.
3. Ak $x_m \neq y_n$, potom $z_k \neq y_n$ značí, že Z je LCS sekvencie X a Y_{n-1}.

Dôkaz 2.1.1

(1) Ak $z_k \neq x_m$, potom by sme mohli pridať $x_m = y_n$ do Z, aby sme dostali spoločnú subsekvenciu sekvencií X a Y v dĺžke $k + 1$. Toto odporuje predpokladu, že Z je najdlhšia spoločná subsekvencia sekvencií X a Y. Takže musíme mať $z_k = x_m = y_n$. Teraz, prefix Z_{k-1} je $k - 1$ dlhá spoločná subsekvencia sekvencií X_{m-1} a Y_{n-1}. Prajeme si dokázať, že je to LCS. Predpokladajme zámerne rozpor, že máme spoločnú subsekvenciu W sekvencií X_{m-1} a Y_{n-1} s dĺžkou lepšou než $k - 1$. Potom pripojením $x_m = y_n$ do W vytoríme spoločnú subsekvenciu sekvencí X a Y, ktorých dĺžka je väčšia než k, čo je v protiklade. (2) Ak $z_k \neq x_m$, potom Z je spoločná subsekvencia sekvencií X_{m-1} a Y. Ak by bola spoločná subsekvencia W sekvencií X_{m-1} a Y s dĺžkou väčšou než k, potom by bola W tiež spoločná subsekvencia sekvencií X_m a Y, čo je v rozpore s predpokladom, že Z je najdlhšia spoločná subsekvencia sekvencií X a Y. (3) Dôkaz je symetrický s (2).

Formulácia vety 2.1.1 ukazuje, že LCS dvoch sekvencií obsahuje v sebe LCS prefixov dvoch sekvencií. Takže ako sme mohli vidieť, LCS problém má optimálnu štruktúru.

Rekurzívne riešenie

Veta 2.1.1 naznačuje, že keď hľadáme LCS sekvencií $X = \langle x_1, x_2, \ldots, x_m \rangle$ a $Y = \langle y_1, y_2, \ldots, y_n \rangle$, overíme buď jeden, alebo dva podproblémy. Ak $x_m = y_n$, musíme nájsť LCS sekvencií X_{m-1} a Y_{n-1}. Pripojením $x_m = y_n$ do tejto LCS získame LCS sekvencií X a Y. Ak $x_m \neq y_n$, potom musíme riešiť dva podproblémy: hľadať LCS sekvencií X_{m-1} a Y a LCS sekvencií X a Y_{n-1}. Tá sekvencia, ktorá bude z týchto dvoch sekvencií dlhšia, bude LCS sekvencií

X a Y. Pretože tieto prípady vyčerpali všetky možnosti, vieme, že jeden z optimálnych podproblémov riešenia musí byť použitý v LCS sekvencií X a Y.

Môžeme ľahko vidieť prekrývajúce sa podproblémy pri hľadaní LCS. K nájdeniu LCS sekvencií X a Y možno budeme potrebovať nájsť LCS sekvencií X a Y_{n-1} a sekvencií X_{m-1} a Y. Ale každý z týchto podproblémov obsahuje podproblém nájsť LCS sekvencií X_{m-1} a Y_{n-1}.

Tak ako pri maticovom násobení, naše rekurzívne riešenie LCS problému zahrňuje stanovenie opakovania pre hodnotu optimálneho riešenia. Definujme $c[i,j]$ ako dĺžku LCS sekvencií X_i a Y_j. Ak $i = 0$ alebo $j = 0$, jedna zo sekvencií má nulovú dĺžku, takže LCS má dĺžku 0. Optimálnu subštruktúru LCS problému vracia následujúci rekurzívny vzorec.

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j, \\ \max(c[i, j-1], (c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j, \end{cases} \tag{2.1}$$

Všimnime si, že v tejto rekurzívnej formulácii, podmienka v probléme obmedzuje, nad ktorými podproblémami môžeme brať ohľad. Keď $x_i = y_j$ môžeme a mali by sme uvažovať nad podproblémom hľadania LCS sekvencií X_{i-1} a Y_{j-1}. Inak namiesto toho uvažujeme dva podproblémy hľadania LCS sekvencií X_i a Y_{j-1} a sekvencií X_{i-1} a Y_j. V predchádzajúcom dynamicky programovanom algoritme sme skúmali, že žiaden podproblém nebol vylúčený kvôli podmienkam v probléme. Hľadanie LCS nie je iba dynamicky programovaný algoritmus, ktorý vylúčil podproblémy založené na podmienkach problému.

Výpočet dĺžky najdlhšej spoločnej subsekvencie

Na základe rovnice 2.1 by sme mohli ľahko napísať rekurzívny algoritmus pracujúci v exponenciálnom čase, ktorý vypočíta dĺžku LCS dvoch sekvencií. Avšak na vyriešenie tohoto problému môžeme použiť dynamické programovanie a výrazne znížiť zložitosť na $\Theta(mn)$.

```
LCS–Length(X,Y)
    m ← len[X]
    n ← len[Y]
    for i ← 1 to m
        do c[i,0] ← 0
    for j ← 0 to n
         do c[0,j] ← 0
    for i ← 1 to m
        do for j ← 1 to n
             do if x_i = y_i
                    then c[i,j] ← c[i−1,j−1] + 1
                         b[i,j] ← "↖"
                    else if c[i−1,j] ≥ c[i,j−1]
                              then c[i,j] ← c[i−1,j]
                                   b[i,j] ← "↑"
                              else c[i,j] ← c[i,j−1]
                                   b[i,j] ← "←"
    return b and c
```

Algoritmus 2.1: Algoritmus na zistenie LCS

Funkcia `LCS-length` prijíma ako argumenty dve sekvencie: $X = \langle x_1, x_2, \ldots, x_m \rangle$ a $Y = \langle y_1, y_2, \ldots, y_n \rangle$. Funkcia ukladá hodnoty $c[i, j]$ do tabuľky $c[0..m, 0..n]$, ktorej položky sú počítané po riadkoch. Funkcia udržuje tiež tabuľku $b[1..m, 1..n]$, kvôli jednoduchému zostrojeniu optimálneho riešenia. Intuitívne, $b[i, j]$ ukazuje na položku tabuľky zodpovedajúcu optimálnemu riešeniu podproblému vybraného počas počítania $c[i, j]$. Funkcia vracia tabuľky b a c. Položka $c[m, n]$ obsahuje dĺžku LCS sekvencií X a Y. Obrázok 2.1 zobrazuje tabuľky b a c, vypočítané pomocou algoritmu LCS, na sekvenciách $X = \langle A, B, C, B, D, A, B \rangle$ a $Y = \langle B, D, C, A, B, A \rangle$.

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

Obrázek 2.1: Tabuľky b a c vypočítané pomocou algoritmu LCS

Čas behu funkcie na výpočet LCS je $\Theta(mn)$, pre každý údaj v tabuľke sa počíta so zložitosťou $\Theta(1)$. Políčko v riadku i a v stĺpci j obsahuje hodnotu $c[i, j]$ a príslušnú šipku hodnoty $b[i, j]$. Údaj 4 v políčku $c[7, 6]$ – pravý spodný roh tabuľky – je dĺžka LCS $\langle B, C, B, A \rangle$ sekvencií X a Y. Pre každé $i, j > 0$, údaj $c[i, j]$ závisí iba na $x_i = y_j$ a hodnotách v políčkach $c[i - 1, j]$, $c[i, j - 1]$ a $c[i - 1, j - 1]$, ktoré sú vypočítané pred $c[i, j]$. Na rekonštrukciu elementov LCS nasledujeme šipky v $b[i, j]$ od pravého dolného rohu. Cesta je na obrázku zvýraznená. Každá šipka $\nwarrow$ na ceste zodpovedá údaju (zvýraznenému), pre ktorý platí $x_i = y_j$, ktorý je členom LCS.

Vytvorenie najdlhšej spoločnej subsekvencie

Tabuľka b vrátená algoritmom hľadajúcim LCS môže byť použitá pre rýchle zostavenie LCS sekvencií $X = \langle x_1, x_2, \ldots, x_m \rangle$ a $Y = \langle y_1, y_2, \ldots, y_n \rangle$. My jednoducho začneme na políčku $b[m, n]$ a prechádzame cez tabuľku následujúc šipky. Vždy, keď narazíme na $\nwarrow$ v políčku $b[i, j]$, vieme, že $x_i = y_j$ a tak získame ďalší element LCS. Elementy LCS získané touto metódou sú v obrátenom poradí.

Nasledujúca rekurzívna funkcia vytlačí LCS sekvencií X a Y v správnom poradí. Prvé volanie funkcie je `print_LCS(b,X,[dĺžka[X],[dĺžka[Y]])`.

```
1  def print_LCS(X, Y):
2      if i = 0 or j = 0:
3              return
4      if b[i][j] == "↖":
5              print_lcs(b, X, i-1, j-1)
6              print x_i
7      elif b[i][j] == "↑":
8              print_lcs(b, X, i-1, j)
9      else:
10             print_lcs(b, X, i, j-1)
```

Algoritmus 2.2: Algoritmus na vytlačenie LCS

Pre tabuľku b z obrázku 2.1 vytlačí táto funkcia "BCBA". Funkcia pracuje so zložitosťou $\Theta(m+n)$. Od posledného i a j sa dekrementuje každý ďalší stupeň rekurzie.

2.1.2 Ratcliff/Obershelp algoritmus

V roku 1983 bol uverejnený algoritmus, na vyhľadanie najdlhšej spoločnej postupnosti, v článku s názvom „Pattern matching: the gestalt approach". V tejto časti si algoritmus popíšeme podľa [68].

Ľudia rozpoznávajú vzory ako funkčný celok, ktorý má určité vlastnosti. Napríklad človek dokáže rozpoznať obraz v predlohe bodov, ktoré sa majú spojiť (connect-the-dots puzzle) ešte pred dokončením, alebo dokonca ešte pred začatím spájania týchto bodov. Tento proces dopĺňania chýbajúcich častí porovnávaním s tým čo už poznáme z predchádzajúceho pozorovania nazývame „gestalt".

Ratcliff/Obershelp algoritmus používa rovnaký proces k rozhodnutiu ako veľmi podobné si dva jednorozmerné vzory sú. Keďže text je jednorozmerná postupnosť znakov, algoritmus vráti hodnotu, ktorá vyjadruje zhodu dvoch reťazcov, respektíve, môže určiť najdlhší spoločný podreťazec.

Najlepší spôsob ako popísať algoritmus, pomocou bežnej počítačovej terminológie, je pomocou vyhľadávania s „wildcard"[1], ktoré nepotrebuje wildcard. Namiesto toho si algoritmus vytvorí vlastné wildcards na základe najlepšej zhody nájdenej medzi dvoma reťazcami. Presnejšie, algoritmus nájde najdlhšiu skupinu znakov, ktoré sú v oboch reťazcoch. Túto skupinu znakov použije ako zarážku rozdeľujúcu oba reťazce na dve polovice. Obe tieto polovice následne umiestni na zásobník pre ďalšie spracovanie. Táto procedúra je opakovaná pre všetky podreťazce na zásobníku pokiaľ nie je zásobník prázdny. Algoritmus nakoniec spočíta hodnotu, do akej miery sa dva reťazce zhodujú, ktorá reprezentuje dvojnásobok počtu znakov spoločných v oboch reťazcoch podelenú počtom znakov v reťazcoch. Výsledok reflektuje percentuálnu zhodu dvoch reťazcov. Ďalším výstupom algoritmu môže byť zoznam indexov spoločných podreťazcov, ktorý sa môže použiť pri porovnávaní dvoch reťazcov a zobrazení zmien medzi nimi.

[1] Znak/reťazec, ktorý môže byť zamenený za iný znak/reťazec. Najčastejšie za takýto znak označuje pomocou „?".

Prácu algoritmu si ukážeme na príklade. Predpokladajme, že chceme zistiť podobnosť medzi slovom „Pennsylvania"a „Pencilvaneya". Najväčšia skupina znakov, spoločná v oboch reťazcoch je „lvan"a dva zvyšné podreťazce, naľavo „Pennsy"a „Penci", napravo „ia"a „eya". Obe tieto skupiny reťazcov umiestníme na zásobník kvôli ďalšiemu spracovaniu, a nastavíme zhodu na osem – dvojnásobok dĺžky spoločného podreťazca. V ďalšom kole vyberieme zo zásobníku dvojicu reťazcov „ia"a „eya", kde algoritmus nájde spoločný podreťazec „a". Zhoda je teraz desať. Podreťazce „i"a „ey"sú umiestnené na zásobník, ale v ďalšom kole ihneď vyjmuté zo zásobníku. V týchto reťazcoch nie je žiadny spoločný podreťazec. Následne algoritmus vyjme zo zásobníku reťazce „Pennsy"a „Penci". Najdlhší spoločný podreťazec majú „Pen". Algoritmus povýši zhodu na 16. Zvyšné podreťazce „nsy"a „ci"sú umiestnené na zásobník. Keď algoritmus vyjme posledné reťazce „nsy"a „ci"zo zásobníku, nenájde spoločný podreťazec. Zásobník je prázdny a algoritmus môže spočítať hodnoty zhody ktorú našiel. Výsledok je zhoda 67 %. Najdlhší spoločný podreťazec je v tomto prípade „Penlvana".

Pôvodný článok obsahoval algoritmus implementovaný v jazyku symbolických inštrukcií. Algoritmus je implementovaný v knižnici `difflib`, ktorá je použitá v implementácii.

2.1.3 Porovnávanie reťazcov

Ak nepotrebujeme vedieť aké zmeny nastali medzi dvoma reťazcami, ale stačí nám iba číslo vyjadrujúce ako veľmi sú dva reťazce podobné, môžeme použiť jeden z nasledujúcich algoritmov.

Hammingova vzdialenosť

Hammingova vzdialenosť medzi dvoma reťazcami je rovná počtu zmenených znakov na korešpondujúcich pozíciách v rovnako dlhých reťazcoch. Toto číslo vyjadruje minimálny počet zmien, pomocou ktorých dostaneme z prvého reťazca druhý. V inom význame je toto číslo rovné počtu chýb, ktoré transformovali prvý reťazec do druhého. Hammingovu vzdialenosť si ukážeme na príklade. Máme reťazce `akre538sk4558sl` a `akrq53rsk4118sl`. Pre tieto reťazce je Hammingova vzdialenosť 4. Presnú definíciu Hammingovej vzdialenosti nájdeme v [31].

Levenshteinova vzdialenosť

Levenshteinova vzdialenosť, tiež známa pod názvom "transformačná vzdialenosť", medzi dvoma reťazcami je daná minimálnym počtom zmien, potrebných k transformácii jedného reťazca na druhý. Transformácie znamenajú vloženie, zmazanie alebo zámenu jedného znaku. Levenshteinovu vzdialenosť si popíšeme podľa [40].

Napríklad Levenshteinova vzdialenosť medzi slovami "kitten"a "sitting"je 3, pretože následujúce tri zmeny prvého slova vytvoria druhé slovo. Neexistuje iný spôsob s menej než tromi zmenami.

1. kitten → sitten (zámena "k"za "s")
2. sitten → sittin (zámena "e"za "i")
3. sittin → sitting (vloženie "g"na koniec slova)

```
int LevenshteinDistance(char s[], char t[])
    // d je tabulka s ~m+1 riadkami a n+1 stlpcami
    int d[m][n];

    for (i = 0; i < m; i++)
        d[i][0] = i;
    for (j= 0; j < n; j++)
        d[0][j] = j;
    for (i = 1; i < m; i++)
        for (j= 1; j < n; j++)
            {
             if (s[i] == t[j])
                 cost = 0;
             else
                 cost = 1;

             d[i][j] = minimum(
                                d[i-1][j] + 1, // vlozenie
                                d[i][j-1] + 1, // zmazanie
                                d[i-1][j-1] + cost // zamena
                              );
            }
    return d[m][n]
```

Algoritmus 2.3: Funkcia na výpočet Levenshteinovej vzdialenosti dvoch reťazcov[40].

Levenshteinova vzdialenosť sa považuje za zobecnenú Hammingovu vzdialenosť, ktorá sa používa pre reťazce rovnakej dĺžky a používa iba zámenu znakov.

Bežne používaný dynamicky programovaný algoritmus na vypočítanie Levenshteinovej vzdialenosti vyžaduje použitie matice o veľkosti $(n + 1) \times (m + 1)$, kde m a n sú dĺžky dvoch reťazcov. Tento algoritmus je založený na Wagner-Fischerovom algoritme. Nasledujúca funkcia `LevenshteinDistance` 2.3 berie dva reťazce, s dĺžky m a t dĺžky n a vypočíta Levenshteinovu vzdialenosť medzi nimi.

Pri použití funkcie na slová uvedené v príklade, dostaneme výpis uvedený v tabuľke 2.1. Kroky, ktoré vedú k transformácii reťazca sú podtrhnuté. V políčku v pravom spodnom rohu je číslo, zodpovedajúce Levenshteinovej vzdialenosti daných reťazcov.

2.2 Porovnávanie stromových štruktúr

Serializované stromové štruktúry, XML dokumenty, môžeme považovať za text. Potom môžeme použiť LCS algoritmus na zistenie zmien. To je vhodné pri strojovom (raw) ukladaní zmien vo verzovacom systéme. Avšak za účelom podpory lepšej služby, je vhodné použiť špecifické algoritmy pre stromové dátové štruktúry.

Existuje niekoľko algoritmov pre zistenie rozdielu medzi dvoma štruktúrovanými dokumentami. Líšia sa v časovej a priestorovej zložitosti, presnosti (minimálnosti) výsledku a rýchlosti. V tejto podkapitole si popíšeme niekoľko z nich.

	0	*1*	*2*	*3*	*4*	*5*	*6*
0		k	i	t	t	e	n
1	s	1	2	3	4	5	6
2	i	2	1	2	3	4	5
3	t	3	2	1	2	3	4
4	t	4	3	2	1	2	3
5	i	5	4	3	2	2	3
6	n	6	5	4	3	3	2
7	g	7	6	5	4	4	3

Tabulka 2.1: Výstup funkcie na výpočet Levenshteinovej vzdialenosti [40].

2.2.1 Prvé algoritmy

Prvý algoritmus na porovnávanie stromových štruktúr bol uverejnený v roku 1977 profesorom Selkowom [70]. Pre jednoduchosť definoval, že editačné operácie sú povolené len na listoch oboch stromov.

Toto obmedzenie prekonáva algoritmus pána Tai-a [76] a popisuje dynamicky programovaný algoritmus, ktorý určuje rozdiel medzi dvomi koreňovými, označenými, zoradenými stromami. Rozdiel je chápaný ako minimum úprav potrebných na transformáciu jedného stromu na druhý, pričom povolené úpravy sú zámena uzlov, zmazanie uzlu a vloženie uzlu. Algoritmus má zložitosť $O(V.V'.L^2.L'^2)$, kde V a V' je počet uzlov a L a L' je maximálna hĺbka stromov T a T'.

2.2.2 Rozšírený Zhang a Shasha algoritmus

Zhang a Shasha [90] navrhli podobný algoritmus ako Tai, používajúci postorder[2] číslovanie zľava do prava. Ďalej uviedli pojem rozdiel medzi dvoma usporiadanými lesmi, čo znamená minimálny počet úprav potrebných na transformáciu jedného lesu na druhý. Algoritmus používa rovnaké úpravy, ale jeho časová zložitosť je $O(V.V'.min(L,K).min(L',K'))$, kde V a V' je počet uzlov, K a K' je počet listov a L a L' je maximálna hĺbka stromov T a T'. Priestorvá zložitosť algoritmu je $O(V.V')$.

V práci [71] publikovali ďalšie algoritmy zaoberajúce sa problematikou porovnávania stromových štruktúr. Niektoré z týchto algoritmov sú riešené paralelne.

Tento algoritmus je použitý v programoch *Logilab XML Diff* a *Microsoft XML Diff*.

2.2.3 Chawatheho algoritmy

Chawathe publikoval viac algoritmov pre porovnávanie stromovo-štruktúrovaných dát. Najdôležitejšie z nich si popíšeme nižšie.

Detekcia zmeny v hierarchicky štruktúrovaných informáciách

V článku [18] je popísaný algoritmus, ktorý porovná dve verzie stromovo-štruktúrovaných dát. Identifikátory objektov sa neuvažujú, pretože nie vždy sú prítomné. Úpravy sú definované ako vloženie, zmazanie alebo zmena uzlu a presun podstromu, pričom posledná z úprav sa nenachádza v predchádzajúcich algoritmoch. Hlavná úloha je v tomto prístupe

[2] Priechod stromom v tvare ľavý podstrom, pravý podstrom, koreň.

je rozdelená na dve: „Dobrú zhodu"a „Minimálny vyhovujúci editačný skript". Prvú úlohu môžme popísať ako nájdenie zhody medzi objektami v dvoch verziách, druhou úlohou je nájsť minimálny počet úprav potrebných k transformácii prvého objektu na druhý.

Intuitívne, algoritmus zostrojí maticu v duchu LCS, ale niektoré hrany sú odstránené, čo vynúti zmazanie (resp. vloženie) uzlu a celého podstromu tohto uzlu. Detailnejšie, diagonálne hrany existujú iba ak korešpondujúce uzly majú rovnakú hĺbku podstromu, horizontálne (resp. vertikálne) hrany z (x, y) do $(x+1, y)$ existujú iba ak hĺbka uzlu s prefixom označeným $x+1$ v T je menšia, než hĺbka uzlu $y+1$ v T'.

Algoritmus má zložitosť $O(ne+e^2)$, kde n je počet listov stromu a e je „vážená editačná vzdialenosť"(typicky $e \ll n$) a je rýchlejší než predchádzajúci algoritmus, ktorý má zložitosť $O(n^2 \log^2 n)$.

Porovnávanie hierarchických dát v externej pamäti

Ďalšou prácou bol algoritmus na porovnávanie hierarchických usporiadaných dát v externej pamäti [17]. Tento algoritmus je vhodný v prípade, že porovnávané súbory sú príliš veľké. Pri porovnávaní sú minimalizované I/O operácie s diskom. Povolené úpravy sú vloženie, zmazanie alebo zmena a sú aplikovateľné iba na listy stromu.

2.2.4 Porovnávanie neusporiadaných stromových štruktúr

Porovnávanie neusporiadaných stromových štruktúr je NP-úplný problém [91], ale obmedzením možných mapovaní medzi dvomi dokumentami, Zhang [89] navrhol algoritmus s kvázi kvadratickou časovou zložitosťou.

2.3 Sémantické porovnávanie

Pri porovnávaní textových súborov sme text chápali ako zoznam riadkov. Pri stromových štruktúrach bola situácia o niečo zložitejšia. Dáta, narozdiel od textu, mali viac úrovní a na ich porovnanie sme použili špecifickejšie algoritmy. Ale ani v jednom z predchádzajúcich prípadov sme porovnávané dáta neinterpretovali. V následujúcom texte sa zameriame na sémantické porovnávanie predovšetkým zdrojových kódov rôznych programovacích jazykov.

Pri porovnávaní dokumentov so zdrojovými kódmi nájdeme niektoré časti kódu, ktoré sú syntakticky rozdielne avšak sémanticky môžu byť rovnaké. Pri takýchto častiach kódu budeme určovať jeho podobnosť, v anglickej literatúre pod názvom „code similarity". Existuje niekoľko prác, ktoré sa zaoberajú skúmaním podobnosti kódu, avšak za účelom detekcie duplicitného kódu, resp. plagiátorstva. Tieto rôzne prístupy si popíšeme podľa [69].

2.3.1 Základné definície

Fragment kódu, ktorý je identický alebo podobný s iným fragmentom v zdrojovom kóde budeme nazývať klon kódu. Žiaľ, nie je jednotná alebo všeobecná definícia pre klon kódu a všetky navrhnuté detekčné metódy používajú svoju vlastnú definíciu klonu kódu [49]. Ďalej si popíšeme definície klonov z rôznej literatúry.

Baxter a spol [12] definujú klon kódu ako segment kódu, ktorý je podobný podľa určitej definície podobnosti. Zatiaľ čo určuje definíciu založenú na prahu podobnosti stromov pre blízke klony, nie je špecifická definícia detekcie podobnosti nezávislých klonov. Vágnejšiu definíciu poskytol Kamiya a spol [34]. Definuje klony ako časti zdrojového kódu, ktoré sú „identické"alebo „podobné"navzájom. Pokiaľ termín „identický"znamená „presnú kópiu",

potom nie je formálna definícia termínu „podobný". Podobná nejasná definícia je predstavená a používaná Burdom a spol. [16] v ich práci, kde segment kódu je označený ako klon, ak je jeden alebo viac výskytov tohto segmentu v zdrojovom kóde s alebo bez „menších"úprav. Nie je špecifikované, čo je myslené pod pojmom „menší". Tak ako Baxter a spol [12], detekcia je závislá na prahovo-orientovanej definícii pojmu „podobný", uvažujú aj iný autori [44, 51, 35]. Ďalšie pokusy sú považované za prekonanie týchto problémov podobnosti [15, 64].

Aby sa predišlo takýmto nejasnostiam pojmov „podobný"a „minoritný", pokúsime sa klasifikovať kategórie definícií klonov kódu. Napríklad Marcus a spol [54] poskytli ordinálnu stupnicu ôsmich rôznych typov klonov, z ktorých niektoré majú presnú definíciu. Napríklad kategória „Odlišný Názov"odkazuje na klony, ktoré sa líšia len v názve identifikátorov medzi dvoma segmentami kódu. A však stupnica nestačí na riadnu definíciu klonov. Napríklad, je definovaná kategória „Podobný Výraz"na identifikáciu klonov, ktorých výrazy sú rozdielne, ale stále „podobné". Blazinska a spol [10] predstavila 18 rôznych kategórií klonov založených na tom, ktoré syntaktické elementy boli zmenené, a koľko metód bolo duplikovaných. Zatiaľ čo väčšina kategórií je špecifická jednou zmenou v kóde, stále máme kategórie „Jeden veľký rozdiel", ktorý znamená jednu sekvenciu rozdielnych tokenov vo výraze alebo v ostatných častiach tela funkcie. „Dva veľké rozdiely"znamenajú zmeny v dvoch jednotkách a „Viac veľkých rozdielov"znamená zmeny v troch alebo viac jednotkách. Všetky tieto spomínané typy sú neurčité.

Problém minimálnej veľkosti klonu je otázny. Niektoré štúdie ukazujú, že technike založenej na tokenoch, napríklad *CCFinder*, je hranica 30 tokenov určená ako rozumné minimum veľkosti klonu [34, 36, 38]. Ďalšie štúdie argumentujú, že meranie veľkosti klonu s ohľadom na počet riadkov by mohla byť lepšia voľba. Avšak niektorí ľudia s týmto výrokom nesúhlasia. Napríklad, v Bellonovom nástroji [15], minimálna veľkosť klonu bola nastavená na 6 nespracovaných riadkov. Na druhú stranu, Baker [9] nastavil minimálny prah na 15 nekorešpondujúcich riadkov, zatiaľ čo Johnson [33] na 50 riadkov. Niektoré štúdie považovali počet AST/PGD uzlov ako mieru veľkosti klonu a považovali túto mierku ako prah [51, 43]. Niektoré štúdie pracovali len s funcionálnymi klonmi s akoukoľvek veľkosťou [57, 48]

2.3.2 Typy podobnosti kódu

Máme dva základné typy podobnosti medzi dvoma fragmentami kódu. Dva fragmenty kódu môžu byť podobné na základe podobnosti textu zdrojového kódu alebo môže byť podobná ich funkcionalita bez textovej podobnosti. Prvý typ klonov je často výsledkom kopírovania kódu a vloženia na iné miesto. V následujúcom texte popíšeme typy klonov založených na type podobnosti dvoch fragmentov podľa:

- Textová podobnosť je založená na podobnosti textu. Rozlišujeme následujúce typy klonov [14, 46]:

  Typ I: Identické fragmenty kódu, až na zmeny bielych znakov a komentárov.

  Typ II: Štrukturálne/syntakticky identické fragmenty, až na zmeny identifikátorov, literálov, typov, usporiadania a komentárov.

  Typ III: Skopírované fragmenty so zmenami. Zámena príkazov, pridanie alebo zmazanie a naviac zmeny identifikátorov, literálov, typov, usporiadania a komentárov

- Funkčná podobnosť: Ak funkcionalita dvoch fragmentov kódu je identická alebo podobná, t.j. majú podobné pre a post contitions, nazývame ich sémantické klony [43, 47, 63, 22] a označujeme ich ako *Typ IV*:

  Typ IV: Dva alebo viac fragmentov kódu, ktoré vykonávajú rovnaký výpočet, ale sú implementované pomocou rôznej syntaxe.

Toto rozdelenie striktne nedefinuje rastúcu postupnosť úrovní od *Typu I* po *Typ IV*, ale analytická komplexnosť a zložitosť v detekovaní rastie od *Typu I* po *Typ IV*. Detekcia klonov *Typu IV* je najťažšia, aj keď máme veľkú znalosť o konštrukcii programu a návrhu softvéru. V následujúcom texte si popíšeme každý typ aj s príkladmi. Príklady sú prebrané z [69].

Podobnosť kódu typu I

Pri podobnosti typu I je nový fragment kódu rovnaký ako originálny. Avšak môžu sa tam vyskytovať určité úpravy v bielych znakoch (whitespace), komentároch a usporiadaní.

Originálny fragment kódu:

```
if (a >= b) {
   c = d + b; // Comment1
   d = d + 1;}
else
   c = d - a; //Comment2
```

A nový pozmenený fragment zdrojového kódu:

```
if (a>=b) {
    // Comment1'
   c=d+b;
   d=d+1;}
else // Comment2'
   c=d-a;
```

Môžme vidieť, že tieto dva fragmenty kódu sú textovo rovnaké, ak odstránime biele znaky a komentáre.

Podobnosť kódu typu II

Pri podobnosti typu II je nový fragment kódu rovnaký ako originálny okrem niektorých možných úprav, ako sú zmena názvov korešpondujúcich identifikátorov (názvy premenných, konštánt, tried, metód,...), typov, usporiadania a komentárov. Kľúčové slová a štruktúra kódu sú v podstate rovnaké ako originál. Originálny fragment kódu:

```
if (a >= b) {
   c = d + b; // Comment1
   d = d + 1;}
else
   c = d - a; //Comment2
```

A nový pozmenený fragment zdrojového kódu:

```
if (m >= n)
   { // Comment1'
   y = x + n;
   x = x + 5; //Comment3
   }
else
   y = x - m; //Comment2'
```

Ako vidíme, v novom fragmente kódu sa zmenil tvar, názvy premenných a hodnoty konštánt. Avšak syntaktická štruktúra kódu je rovnaká v oboch fragmentoch.

Podobnosť kódu typu III

Pri podobnosti typu III nový fragment kódu je modifikovaný zmenou, pridaním alebo zmazaním príkazov. Originálny fragment kódu:

```
if (a >= b) {
   c = d + b; // Comment1
   d = d + 1;}
else
   c = d - a; //Comment2
```

Ak tento kód rozšírime pomocou pridania príkazu `e = 1`, dostaneme nový fragment zdrojového kódu:

```
if (a>=b) {
   c = d + b; // Comment1
   e = 1; // This statement is added
   d = d + 1;}
else // Comment2'
   c = d - a;
```

Podobnosť kódu typu IV

Pri podobnosti typu IV je nový fragment kódu sémanticky podobný ako originály fragment. Pri tomto type podobnosti kódu nemusel nový fragment vzniknúť kópiou originálu, ale oba fragmenty mohli byť vytvorené dvomi rôznymi programátormi, ktorí implementovali rovnakú funkcionalitu. V následujúcom fragmente kódu sa počíta hodnota faktoriálu:

```
int i, j=1;
for (i=1; i<=VALUE; i++)
   j=j*i;
```

Ďalší fragment kódu je rekurzívna funkcia, ktorá tiež počíta faktoriál:

```
int factorial(int n)
{
   if (n == 0)
       return 1 ;
   else
       return n * factorial(n-1) ;
}
```

Zo sémantického hľadiska sú oba fragmenty rovnaké a patria pod typ IV, aj keď prvý fragment je jednoduchý kód a druhý fragment je rekurzívna funkcia bez lexikálnej, syntaktickej alebo štrukturálnej podobnosti s prvým fragmentom.

2.3.3 Všeobecný algoritmus porovnávania zdrojových kódov

Následujúci algoritmus popisuje všeobecné kroky pri porovnávaní dvoch dokumentov so zdrojovými kódmi. Tento algoritmus bol prebraný z [69] a upravený pre potreby porovnávania dvoch dokumentov. Pôvodný algoritmus hľadal duplicitné časti kódu v zdrojových súboroch, čo je principiálne ten istý problém ako porovnávanie dvoch dokumentov.

1. Preprocessing

Na začiatku procesu porovnávania sú zdrojové kódy rozdelené a sú určené domény porovnávania. Táto fáza obsahuje tri hlavné procedúry:

Odstránenie nepodstatných častí. Všetky časti nepodstatného kódu sú v tejto fáze odstránené. Napríklad odstránenie vloženého cudzieho kódu (napr. SQL v Jave, Assembler v C) za účelom rozdeliť rôzne programovacie jazyky.

Určenie zdrojových jednotiek. Zdrojový kód je rozdelený na disjunktné množiny, nazývané zdrojové jednotky, ktoré sa budú porovnávať navzájom. Zdrojové jednotky môžu obsahovať rôzne úrovne granularity. Napríklad súbory, triedy, funkcie, resp. metódy, bloky kódu, výrazy alebo sekvencia riadkov zdrojového kódu.

Určenie jednotiek porovnávania – granularita. Zdrojové jednotky ešte môžu byť delené na menšie časti v závislosti na porovnávacej metóde. Napríklad zdrojová jednotka môže byť rozdelené na riadky alebo dokonca tokeny. Avšak zdrojové jednotky môžu byť použité na porovnávanie ako celé. Napríklad metricky-orientované porovnávacie metódy.

2. Transformácia

Jednotky porovnávania zdrojového kódu sú transformované do inej prostrednej internej reprezentácie za účelom jednoduchého porovnávania alebo extrakcie porovnávaných vlastností. Táto transformácia sa môže veľmi líšiť, od veľmi jednoduchej, ktorá iba odstráni biele znaky a komentáre [8], až po veľmi komplexnú, ktorá generuje program dependency graph (PDG) [42, 47]. V následujúcich bodoch si popíšeme niektoré transformácie. Jedna alebo viac následujúcich transformácií môže byť čiastočne zahrnutá v porovnávacom algoritme.

Formátovanie zdrojového kódu. Formátovanie zdrojového kódu je jednoduchý spôsob reorganizácie kódu do štandardnej formy. Po aplikácii formátovania zdrojového kódu z rôznych formátovaní (napr. pre jazyk C ANSI, K&R, GNU) do spoločného štandardného formátu. Formátovanie kódu sa používa pri textovo-orientovanom porovnávaní, čo redukuje falošné rozdiely, spôsobené rozdielnym usporiadaním rovnakých segmentov kódu.

Odstránenie komentárov. Väčšina riešení (okrem Mercusa a spol [54] a Marylanda a spol [57]) ignoruje, resp. odstraňuje komentáre zo zdrojových kódov pred porovnávaním.

Odstránenie bielych znakov. Skoro všetky riešenia (okrem riadkovo-orientovaných) neuvažujú biele znaky. Riadkovo-orientované riešenia odstraňujú všetky biele znaky okrem koncov riadkov.

Tokenizácia. V prípade tokenovo-orientovaných riešení, je jednotka porovnávania rozdelená na tokeny korešpondujúce s lexikálnymi pravidlami daného programovacieho jazyka. *CCFinder* [34] a *Dup* [8] sú vedúcimi nástrojmi používajúcimi tokenizáciu zdrojového kódu.

Parsovanie. V prípade stromovo-orientovaného prístupu je celý zdrojový kód parsovaný a je zostavený (anotovaný) abstraktný syntaktický strom (AST). V takomto prípade je zdrojová jednotka reprezentovaná stromom a jednotky porovnávania sú jej podstromy. Porovnávacie algoritmy používajú tieto stromy [12, 79, 88].

Generovanie grafu závislostí programu. Sémanticky-orientované prístupy generujú graf závislostí programu (program dependency graph – PDG) zo zdrojového kódu. Jednotky porovnávania sú podgrafy celého PDG. Detekčný algoritmus potom hľadá izomorfické podgrafy [42, 47]

Normalizácia identifikátorov. Väčšina riešení normalizuje identifikátory pred porovnávacou fázou. Všetky identifikátory v zdrojových kódoch sú prepísané jediným tokenom. Avšak Baker [8] aplikuje systematickú normalizáciu identifikátorov, aby našiel parametrizované rovnaké fragmenty kódu.

Transformácia elementov programu. Okrem normalizácie identifikátorov, môžeme aplikovať niekoľko ďalších transformácií na elementy zdrojového kódu.

Počítanie metrík. Metricky-založené prístupy počítajú niekoľko metrík z pôvodného a/alebo transformovaného (AST,PDG,...) zdrojového kódu. Tieto metriky sú použité pri porovnávaní [57, 45].

Vyššie uvedené transformácie len poskytujú prehľad súčasne používaných transformačných techník. Niektoré ďalšie transformácie rôznych úrovní sú aplikované na zdrojovom kóde pred fázou *detekcie zhody*.

3. Porovnávanie

Transformovaný kód je vstupom do porovnávacieho algoritmu. Existuje niekoľko rôznych druhov algoritmov, ktoré porovnávajú dva zdrojové kódu. Niektoré populárne sú založené na suffix-tree [67, 58] algoritme [8, 34], dynamickom hľadaní vzorov (dynamic pattern matching – DPM) [25, 45] a porovnávaní hashovaných hodnôt [12, 57]. Tieto algoritmy si popíšeme v ďalšej časti.

Výstup z tejto fáze je zoznam zodpovedajúcich si fragmentov kódu, resp. zoznam rozdielnych fragmentov kódu. Každé dva korešpondujúce fragmenty kódu sú lokalizované vzhľadom na transformovaný kód. Napríklad pri tokenovo-orientovanom prístupe to bude štvorica, začiatok a koniec v prvej sekvencii tokenov a začiatok a koniec subsekvencie v druhej sekvencii tokenov.

4. Formátovanie

V tejto fáze je zoznam rozdielnych fragmentov kódu, lokalizovaných vzhľadom na transformovaný kód, konvertovaný na zoznam lokalizované vzhľadom na originálny kód. Jednoducho povedané, každá pozícia dvojice fragmentov kódu získaná z predchádzajúcej fáze je konvertovaná na čísla riadkov v originálnych zdrojových súboroch. Jednotný formát pozostáva z dvoch n-tíce, {(Názov prvého súboru, Počiatočný riadok, Koncový riadok),(Názov druhého súboru, Počiatočný riadok, Koncový riadok)}.

5. Postprocessing

V tejto fáze sú filtrované falošné zhody pomocou manuálnej analýzy a/alebo pomocou vizualizácie.

Manuálna analýza. Po extrahovaní zhodných (rozdielnych) fragmentov zdrojového kódu sa tieto fragmenty manuálne kontrolujú. Touto kontrolou vylúčime chybné výsledky.

Vizualizácia. Získané zoznamy zmien môžeme vizualizovať a tak urýchliť proces manuálnej analýzy.

6. Agregácia

Za účelom zredukovania veľkosti dát alebo pre urýchlenie niektorých analýz môžeme agregovať niektoré zhodné fragmenty kódu.

Jednotlivé fázy popísané vyššie sú veľmi všeobecné a jednu alebo viac z nich môžeme v danom porovnávacom procese vynechať.

2.3.4 Porovnávanie zdrojových kódov

V literatúre je prezentovaných niekoľko techník na porovnávanie dvoch zdrojových kódov. Väčšina techník detekuje rôzne typy rozdielov a rôznu úroveň granularity výsledkov.

Porovnávanie prebieha prevažne v dvoch fázach pozostávajúcich z transformácie a porovnávania. V prvej fáze je zdrojový kód transformovaný do internej formy, ktorá umožňuje použiť účinnejšie porovnávacie algoritmy. Počas nasledujúcej porovnávacej fáze sú detekované zhodné časti kódu. Pretože hlavnú úlohu hrá práve interný formát, je vhodné klasifikovať porovnávacie algoritmy práva podľa neho. V následujúcom texte si popíšeme základné techniky porovnávania.

Textovo-orientované porovnávanie

V tomto prípade je zdrojový kód považovaný za zoznam riadkov (reťazcov). Dva fragmenty kódu sa porovnávajú riadok po riadku za účelom nájsť zoznam rovnakých riadkov. Na porovnávanie môžeme použiť algoritmus LCS 2.1.1.

Pre lepšie výsledky môžeme previezť transformácie:

- odstránenie komentárov,
- odstránenie bielych znakov (okrem koncov riadkov),
- normalizácia kódu,
- normátovanie kódu.

Pri normalizácii meníme názvy identifikátorov, konštánt, reťazcov, literálov a typov v kóde. Toto môžeme vykonať globálne, nahradiť všetky identifikátory rovnakým názvom alebo parametrizovane, t.j. nájsť vhodné mapovanie medzi názvami identifikátorov dvoch fragmentov kódu. V takom prípade dokážeme detekovať aj premenovanie identifikátorov. Niektoré normalizácie sú uvedené v tabuľke 2.2.

Pri textovom porovnávaní sa môže vyskytnúť niekoľko problémov. Ak v kóde prevedieme niektoré špecifické zmeny, ktoré nemajú vplyv na funkčnosť a ani štruktúru kódu, nedokážeme detekovať takýto kód ako zhodný.

1. Pridanie alebo odobranie prázdnych riadkov.
2. Zmena názvov identifikátorov.

Operácia	Element jazyka	Príklad	Výsledok
1	Reťazec	”Abort”	”...”
2	Literál	’y’	’.’
3	Celé číslo	42	1
4	Desatinné číslo	0.314159	1.0
5	Identifikátor	counter	p
6	Základné typy	int, long, float, double, ...	num
7	Názov funkcie	main()	foo()

Tabulka 2.2: Normalizácie zdrojového kódu podľa [24].

3. Pridanie alebo odobranie zátvoriek okolo nejakého príkazu.

Porovnávanie zdrojových kódov touto technikou je veľmi rýchle, ale nedokáže detekovať sofistikovanejšie zmeny.

Tokenovo-orientované porovnávanie

Pri tokenovo-orientovanom porovnávaní je zdrojový text rozdelený na sekvenciu tokenov. Tieto sekvencie následne porovnávame a hľadáme rovnaké subsekvencie spoločné v oboch sekvenciách. Táto technika je v porovnaní s textovo-orientovanou obvykle odolnejšia proti zmenám kódu ako je napríklad formátovanie, zmeny komentárov a názvy identifikátorov. Zdrojový kód môžeme rozdeliť na tokeny pomocou lexikálneho analyzátoru [81].

Je možné previezť niektoré ďalšie transformácie nad sekvenciou tokenov. Napríklad zameniť tokeny reprezentujúce názvy identifikátorov za špeciálny token.

Na porovnávanie sekvencií tokenov sa používajú suffix-tree algoritmy založené na porovnávaní podreťazcov. Nájdené zhodné úseky sú lokalizované vzhľadom na sekvenciu tokenov a je potrebné mapovanie tokenov na riadky kódu, aby sme dané úseky identifikovali priamo v zdrojovom kóde.

Stromovo-orientované porovnávanie

Pri tomto type porovnávania je zdrojový kód rozparsovaný na abstraktný syntaktický strom (AST) pomocou syntaktického parseru [82] pre daný programovací jazyk. Získané AST sú ďalej porovnávané pomocou stromových algoritmov 2.2. Parsovaním AST získame kompletné informácie o zdrojovom kóde. Keď v stromovej reprezentácii zahodíme názvy premenných a hodnoty literálov a konštánt, je možné použiť sofistikovanejšie metódy na porovnávanie.

Zatiaľ čo je AST zostavovaný z lexikálnej abstrakcie zdrojového kódu iba parametrizovaním listov AST, štrukturálna abstrakcia parametrizuje celé príslušné podstromy. Uvažujme fragment kódu `a[?] = x`, ktorý sa môže vyskytovať v dvoch dokumentoch ako `a[i] = x;` a `a[i+1] = x;`, kde v prvom výskyte ide o argument reprezentovaný listom v strome, ale v druhom výskyte je argument štruktúrovaný a je reprezentovaný ako binárny uzol AST. Je vidieť, že štrukturálna abstrakcia je všeobecnejšia ako AST.

Porovnávanie grafov závislostí programu

Graf závislostí programu (PDG) je o krok bližšie k získaniu vysokej abstrakcie zo zdrojového kódu pomocou uvažovania sémantických informácií kódu. PDG obsahuje informácie o toku riadenia a toku dát v programe, a preto nesie sémantické informácie. Následne tieto grafy získané zo zdrojových kódov porovnávame a takto nájdeme sémanticky zhodné fragmenty kódu. Tento postup poskytuje oveľa väčšiu úroveň abstrakcie, ale má najväčšie nároky na zložitosť a čas výpočtu.

Porovnávanie metrík programu

Tento prístup je odlišný od predchádzajúcich. Najskôr získame rôzne metriky zo zdrojových kódov a následne porovnáme tieto vektory metrík namiesto zdrojových kódov. Za metrické údaje berieme napríklad počet riadkov kódu, volaní funkcií, príkazov vetvenia, cyklov, priradení, atď. Tieto metrické údaje môžeme podľa miery abstrakcie získať z textovej reprezentácie zdrojového kódu, zo sekvencie tokenov, z AST alebo PDG.

Hybridné porovnávanie

Existuje niekoľko ďalších techník porovnávania zdrojových kódov, ktoré používajú hybridné metódy (hybridná reprezentácia kódu a/alebo algoritmy). Tieto techniky môžeme klasifikovať podľa predchádzajúcich kategórií.

Napríklad použitie AST a suffix-tree kombinuje schopnosti detekcie AST s rýchlosťou suffix-tree.

Kapitola 3

Lexikálna analýza

Lexikálna analýza je proces, pri ktorom sa vstupná postupnosť znakov rozdelí na lexémy - lexikálne jednotky (napr. identifikátory, čísla, kľúčové slová, operátory, ...). Tieto lexémy sú reprezentované vo forme tokenov, ktoré sa ďalej spracovávajú. V následujúcej kapitole si popíšeme tento proces podľa [4] a uvedieme si nástroje, ktoré nám generujú lexikálne analyzátory definované určitou gramatikou.

3.1 Lexikálna analýza

Lexikálny analyzátor býva najčastejšie prvá fáza prekladača alebo úvodná fáza pri spracovaní zdrojového kódu, či inak štruktúrovaného textu. Jeho hlavnou úlohou je čítať znaky zo vstupu, zoskupovať ich do lexémov a na svoj výstup produkovať sekvenciu tokenov pre každý lexém v zdrojovom kóde. Túto sekvenciu tokenov sa ďalej používa syntaktický analyzátor.

Vzhľadom k tomu, že lexikálny analyzátor je prvá fáza spracovania zdrojového kódu, môže vykonávať aj ďalšie úlohy. Napríklad odstraňovanie bielych znakov, informovanie o chybách pri spracovávaní vstupu (napríklad pri nerozpoznaní lexému) alebo udržovať pozíciu aktuálne načítaného lexému, čo uľahčí jeho identifikáciu v zdrojovom kóde. Pokiaľ zdrojový jazyk obsahuje funkcie makroprocesoru, potom môžu byť tieto funkcie vykonané v priebehu lexikálnej analýzy.

Niekedy je lexikálna analýza rozdelená do dvoch krokov [4]:

1. Skenovanie, jednoduchý proces, ktorý nevyžaduje tokenizáciu. Slúži na odstránenie komentárov a zlúčenie po sebe idúcich bielych znakov do jedného.

2. Lexikálna analýza, zložitejší proces, ktorý produkuje sekvenciu tokenov.

3.1.1 Základné pojmy

Pri lexikálnej analýze používame výrazy token, vzor a lexém so špecifickým významom:

Token je pár, pozostávajúci z názvu typu tokenu a voliteľného atribútu – hodnoty. Typ tokenu je abstraktná reprezentácia typu lexikálnej jednotky. Napríklad konkrétne kľúčové slovo alebo reťazec vstupných znakov značia identifikátor.

Vzor je popis formy, ktorú má lexém príslušného tokenu. V tomto prípade, kľúčové slovo ako token, má vzor sekvenciu znakov daného kľúčového slova. Pre identifikátory a ostatné tokeny je vzor oveľa zložitejší a porovnáva sa s mnohými reťazcami.

Lexém je reťazec znakov v zdrojovom kóde, ktorá zodpovedá vzoru daného tokenu a je identifikovaná lexikálnym analyzátorom ako inštancia daného tokenu.

V tabuľke 3.1 sú typické tokeny, niekoľko príkladov na lexémy a neformálne popísané vzory.

Typ tokenu	**Príklad lexémov**	**Popis vzoru**
CONST	const	const
IF	if	if
OPERATOR	+,-,*,/,%	+ alebo - alebo * alebo / alebo %
IDENTIFIER	i, count, X,Y	písmeno následované písmenami a číslicami
FLOATING_LITERAL	3.14, 2e-4, .75	desatinná konštanta
STRING_LITERAL	"Hello world!"	postupnosť znakov v uvodzovkách

Tabulka 3.1: Príklady symbolov

V mnohých programovacích jazykoch pokrýva následujúca klasifikácia väčšinu tokenov:

- Jeden token pre každé kľúčové slovo. Vzor kľúčového slova je rovnaký ako kľúčové slovo samo.
- Tokeny pre všetky operátory buď individuálne, alebo v triedach ako porovnávacie operátory, aritmetické a iné.
- Jeden token reprezentujúci všetky identifikátory.
- Jeden alebo viac tokenov reprezentujúce konštanty, ako napríklad čísla a reťazcové literály.
- Token pre každý interpunkčný symbol, ako napríklad zátvorky, čiarka, bodkočiarka, a iné.

3.1.2 Atribúty tokenov

V prípade, že jednému vzoru zodpovedá viac lexémov, lexikálny analyzátor musí mať prídavnú informáciu o lexéme, ktorý vyhľadal. Napríklad, vzoru pre token číslo zodpovedajú 0 aj 1, ale je veľmi dôležité vedieť, ktorý lexém bol nájdený v zdrojovom kóde. Preto v mnohých prípadoch lexikálny analyzátor vracia nielen typ tokenu, ale aj atribút, ktorý popisuje hodnotu reprezentujúcu tokenom.

Budeme predpokladať, že token má len jeden atribút, aj keď tento atribút môže byť štruktúra obsahujúca niekoľko informácií. Najdôležitejšia je typ tokenu. Najčastejšia štruktúra obsahuje naviac hodnotu lexému a jeho pozíciu.

3.1.3 Chyby pri lexikálnej analýze

Bez podpory iných komponentov, je pre lexikálnu analýzu náročné detekovať chybu v zdrojovom kóde. Napríklad, ak prvýkrát narazíme na reťazec `fi` v zdrojovom kóde jazyku C v kontexte `fi (a == f(x))`, lexikálny analyzér nemôže povedať, či fi je chybne napísané kľúčové slovo if, alebo nedeklarovaný identifikátor funkcie. Pretože fi je platný lexém pre token identifikátor, lexikálny analyzátor musí vrátiť token identifikátor a nechať ďalšiu fázu ošetriť túto chybu spôsobenú zámenou znakov.

Predpokladajme však, že vzniknutá situácia, v ktorej lexikálny analyzátor je neschopný pokračovať, pretože žiadny zo vzorov pre tokeny sa nezhoduje s ľubovoľným prefixom zostávajúceho vstupu. Najjednoduchšia stratégia zotavenia je „panický mód". Zmažeme nasledujúce znaky zo zostávajúceho vstupu pokiaľ lexikálny analyzátor nenájde správne formátovaný token na začiatku vstupu, ktorý zostal. Táto technika môže zmiasť následujúcu fázu po lexikálnej analýze, ale v interaktívnom prostredí môže byť adekvátna.

Ďalšie možné spôsoby zotavenia z chyby sú:

- Zmazanie jedného znaku zo zostávajúceho vstupu.
- Vloženie chýbajúceho znaku do zostávajúceho vstupu.
- Zámena znaku za iný znak.
- Vymeniť dva susedné znaky.

Transformácie ako tieto sa môžu pokúsiť opraviť vstup. Najjednoduchšie z týchto stratégií je zistiť, či prefix zostávajúceho vstupu môže byť transformovaný na platný lexém jedinou transformáciou. Táto stratégia dáva zmysel, pretože v praxi väčšina lexikálnych chýb zahŕňa jeden znak. Všeobecnejšou korekčnou stratégiou je nájdenie najmenšieho počtu zmien, potrebných k transformácii zdrojového programu tak, aby pozostával len z validných lexémov. Ale tento prístup je v praxi považovaný za moc náročný, aby stál za to úsilie.

3.1.4 Načítavanie vstupu

Ešte pred popisom rozpoznávania lexémov zo vstupu, ukážeme niektoré jednoduché, ale veľmi dôležité úlohy – spôsoby načítania vstupného zdrojového kódu. Táto úloha sa môže zdať ťažká, pretože často musíme skúmať jeden alebo viac znakov za následujúcim lexémom ešte pred tým, ako sme si istý, že máme správny lexém. Napríklad, nemôžeme rozpoznať koniec identifikátoru, pokiaľ nenarazíme na znak, ktorý nie je písmeno alebo číslica a tým pádom nie je platný znak pre identifikátor. V jazyku C, operátory pozostávajúce z jedného znaku, napríklad $-$, $=$, alebo $<$ môžu byť začiatkom dvojznakových operátorov ako napríklad $->$, $==$, alebo $<=$.

Dvojica bufferov

Vzhľadom na veľké množstvo času, potrebného na spracovanie znakov, a veľké množstvo znakov, ktoré musíme spracovať, boli vyvinuté špeciálne techniky s vyrovnávacou pamäťou (bufferom), ktoré znížia réžiu pri spracovávaní jedného znaku. Dôležitá schéma používa dve vyrovnávacie pamäte, ktoré sú striedavo naplňované. Každý buffer má rovnakú veľkosť N, ktorá je spravidla rovnaká, ako veľkosť bloku na disku, napríklad 4096 bajtov. Použitím jediného systémového volania načítame N znakov do vyrovnávacej pamäte, čo je omnoho výhodnejšie, ako volať jadro operačného systému pre načítanie každého znaku. Ak na vstupe zostáva menej ako N znakov, špeciálny znak, EOF, označuje koniec zdrojového kódu. Tento znak je odlišný od akéhokoľvek znaku, ktorý sa môže vyskytnúť v zdrojovom kóde.

Vo vstupnom bufferi udržujeme dva ukazateľe:

- Ukazateľ **začiatok** ukazuje na začiatok súčastného lexému, ktorého obsah sa snažíme zistiť.
- Ukazateľ **vpred** prechádza znaky pokiaľ nenájde zhodu so vzorom.

Akonáhle je lexém rozpoznaný, ukazateľ vpred značí koniec lexému. Následne, po zaznamenaní lexému ako hodnotu atribútu tokenu je ukazateľ začiatok nastavený na znak následujúci za ukazateľom vpred. Postupujúci ukazateľ vpred musí najskôr zistiť, či nedosiahol koniec bufferu, a ak áno, musí načítat druhý buffer zo vstupu.

3.1.5 Špecifikácia tokenov

Regulárne výrazy sú dôležitým prostriedkom pre špecifikáciu vzorov lexémov. Nedokážu vyjadriť vsšetky možné vzory, ale sú veľmi efektívne pri špecifikácii takých typov vzorov, ktoré pre tokeny práve potrebujeme. V tejto časti si popíšeme formálny zápis regulárnych výrazov a následne si ukážeme ako sú tieto regulárne výrazy použité pri lexikálnej analýze.

Reťazce a jazyky

Abeceda je konečná, neprázdna množina symbolov. Typickými príkladmi symbolov sú písmená, číslice a interpunkčné znamienka. Množina $\{0, 1\}$ je binárna abeceda. ASCII je dôležitý príklad abecedy, ktorý je použitý v mnohých softvérových systémoch. Unicode, ktorý obsahuje približne 100 000 znakov z abecied celého sveta je ďalšou rovnako dôležitou abecedou.

Reťazec nad abecedou je konečná postupnosť symbolov danej abecedy. V teórii jazykov, sú pojmy „veta"a „slovo"použité ako synonymá pre „reťazec". Dĺžka reťazca sa zapisuje ako $|s|$, je počet symbolov vyskytujúcich sa v s. Prázdny reťazec označovaný ako ε je reťazec s dĺžkou nula.

Jazyk je konečná množina reťazcov nad určitou abecedou. Táto definícia je veľmi všeobecná. Abstraktné jazyky ako 0, prázdna množina, alebo $\{\varepsilon\}$ sú jazyky splňujúce definíciu. Ale tiež množina všetkých syntakticky správnych programov v jazyku C a množina všetkých gramaticky korektných viet v angličtine.

Ak x a y sú reťazce, potom **konkatenácia** x a y značená ako xy je reťazec, vzniknutý pripojením y za x. Prázdny reťazec je identita nad konkatenáciou. To znamená, že pre každý reťazec s, $\varepsilon s = s\varepsilon = s$.

Ak konkatenáciu považujeme za násobenie, môžeme definovať mocninu reťazca ako:

$$s^0 = \varepsilon \tag{3.1}$$

$$\forall i > 0 : s^i = s^{i-1}s \tag{3.2}$$

Operácie nad jazykmi

V lexikálnej analýze sú najdôležitejšie operácie nad jazykmi zjednotenie, konkatenácia a iterácia, ktoré sú formálne definované v tabuľke 3.2.

Regulárne výrazy

Regulárne výrazy sú budované rekurzívne z menších regulárnych výrazov pomocou pravidiel popísaných nižšie. Každý regulárny výraz r popisuje jazyk $L(r)$, ktorý je tiež definovaný rekurzívne z jazykov popisujúcich podvýrazy výrazu r. Ďalej máme pravidlá, ktoré definujú regulárny výraz r nad abecedou Σ a jazyky, ktoré tieto regulárne výrazy popisujú.

Základné dve pravidla sú:

1. ε je regulárny výraz, a jazyk $L(\varepsilon)$ je $\{\varepsilon\}$.

Operácia	**Definícia a zápis**
Zjednotenie jazykov L a M	$L \cup M = \{s \mid s \in L \vee s \in M\}$
Konkatenácia jazykov L a M	$LM = \{sl \mid s \in L \wedge l \in M\}$
n-tá mocnina jazyka L	$L^0 = \{\varepsilon\}$ $L^n = L \cdot L^{n-1},\ pre\ n \geq 1$
Iterácia jazyka L	$L^* = \bigcup_{n>0} L^n$
Pozitívna iterácia jazyka L	$L^+ = \bigcup_{n\geq 1} L^n$

Tabulka 3.2: Operácie nad jazykmi

2. Ak a je symbol z Σ, potom a je regulárny výraz, a $L(a) = \{a\}$

Ďalej máme spôsoby budovania regulárnych výrazov. Predpokladajme, že r a s sú regulárne výrazy popisujúce jazyky $L(r)$ a $L(s)$:

1. $(r)|(s)$ je regulárny výraz popisujúci jazyk $L(r) \cup L(s)$
2. $(r)(s)$ je regulárny výraz popisujúci jazyk $L(r)L(s)$
3. $(r)*$ je regulárny výraz popisujúci jazyk $(L(r))*$

Aby sme odstránili zbytočné zátvorky okolo regulárnych výrazov, určíme si konvenciu priority operátorov:

1. unárny operátor * má najvyššiu prioritu,
2. konkatencia ma druhú najvyššiu prioritu,
3. binárny operátor | má najnižšiu prioritu, a je zľava asociatívny.

Jazyk, ktorý je definovaný regulárnym výrazom sa nazýva regulárna množina. Ak dva regulárne výrazy r a s popisujú rovnakú regulárnu množinu, hovoríme, že sú ekvivalentné. Napríklad $(a|b) = (b|a)$. Existuje niekoľko algebrických pravidiel pre regulárne výrazy. Niektoré z nich sú uvedené v tabuľke 3.3

Pravidlo	**Popis**
$r\|s = s\|r$	operácia | je komutatívna
$r\|(s\|t) = (s\|r)\|t$	operácia | je asociatívna
$r(st) = (sr)t$	operácia konkatenácia je asociatívna
$r(s\|t) = rs\|rt$	konkatenácia distributívna nad operáciou |
$\varepsilon r = r\varepsilon = r$	ε je identitou pre konkatenáciu
$r^* = (r\|\varepsilon)^*$	ε je jednotkou pre iteráciu

Tabulka 3.3: Algebraické pravidlá pre regulárne výrazy

Regulárne výrazy POSIX

Štandard POSIX definuje dve časti regulárnych výrazov [1]:

1. základné regulárne výrazy (BRE),

2. rozšírené regulárne výrazy (ERE).

Základné regulárne výrazy boli navrhnuté pre spätnú kompatibilitu s tradičnou syntaxov, ale za predpokladu spoločného štandardu, ktorý bol prijatý v mnohých nástrojoch pre regulárne výrazy. V BRE syntaxi je väčšina znakov považovaná za literály, ktoré sa zhodujú sami so sebou, napríklad „a". Výnimky sú uvedené v tabuľke 3.4.

Metaznak	**Popis**
.	Zhoduje sa s jedným (akýmkoľvek) znakom. Napríklad výraz „a.c"sa zhoduje s „abc"
[ ]	Zhoduje sa s jedným znakom, uvedeným v hranatých zátvorkách. Môžeme uviesť aj rozsah, napríklad a-z. Napríklad výraz „[abc]"sa zhoduje s „a", „b"alebo s „c". výraz „[a-z]"sa zhoduje s akýmkoľvek malým pismenom
[^]	Zhoduje sa s jedným znakom,ktorý nie je uvedeným v hranatých zátvorkách. Môžeme uviesť aj rozsah, napríklad a-z. Napríklad výraz „[^ abc]"sa zhoduje s akýmkoľvek znakom okrem „a", „b"alebo s „c". výraz „[^ a-z]"sa zhoduje s akýmkoľvek znakom, okrem malého písmena malým pismenom
^	Zhoduje sa s počiatočnou pozíciou v reťazci. V riadkovo orientovaných nástrojoch sa zhoduje so začiatkom riadku.
$	Zhoduje sa s koncovou pozíciou v reťazci. V riadkovo orientovaných nástrojoch sa zhoduje so koncom riadku.
()	Definuje regulárny podvýraz. Reťazec zhodný s regulárnym výrazom definovaným v zátvorkách môže byť použitý neskôr. Podvýraz nazývame blok alebo skupina.
\n	Zhoduje sa s n-tým označeným podvýrazom, kde n je číslo 1 až 9
*	Zhoduje sa s nula alebo viacnásobným výskytom predchádzajúceho prvku. Napríklad výraz „ab*c"sa zhoduje s „ac", „abc", „abbc", „abbbc"atď.
{m,n}	Zhoduje sa s výskytom predchádzajúceho elementu minimálne m-krát a maximálne n-krát. Napríklad výraz a3,5 sa zhoduje s „aaa", „aaaa", „aaaaa", ale nie s „aa"ani s „aaaaaa".

Tabulka 3.4: Základné regulárne výrazy

V základných regulárnych výrazoch musíme pred zátvorky () a {} umiestniť znak „\"inak budú považované za literál.

Rozšírené regulárne výrazy pridávajú nasledujúce následujúce metaznaky:

3.1.6 Rozpoznávanie tokenov

V predchádzajúcom texte sme vysvetlili, ako vyjadriť vzor pomocou regulárnych výrazov. Teraz musíme z vzorov pre všetky potrebné tokeny zostaviť časť kódu, ktorý vo vstupnom zdrojovom kóde nájde lexém zhodujúci sa s jedným zo vzorov.

Metaznak	Popis
?	Zhoduje sa s nula alebo jedným výskytom predchádzajúceho elementu. Napríklad výraz „ba?"sa zhoduje s „b"alebo s „ba"a s ničím iným.
+	Zhoduje sa s jedna alebo viacnásobným výskytom predchádzajúceho prvku. Napríklad výraz „ab*c"sa zhoduje s „abc", „abbc", „abbbc"atď., ale nie s „ac".
|	Operátor výberu, zhoduje sa s výrazom pred, alebo s výrazom za operátorom. Napríklad „abc|def"sa zhoduje s „abc"alebo „def".

Tabulka 3.5: Rozšírené regulárne výrazy

Graf prechodov

Ako medzikrok pri konštruovaní lexykálneho analyzátoru musíme najskôr previesť vzory na vývojový diagram nazývaný graf prechodov (konečný automat).

Diagram prechodov obsahuje uzly nazývané stavy. Každý stav reprezentuje podmienku, ktorá nastane počas spracovávania vstupu pri hľadaní lexému, ktorý zodpovedá jednému vzoru. Orientované hrany spájajú jeden stav s druhým. Hrany sú označované symbolom, alebo množinou symbolov. Ak sme v nejakom stave, a následujúci vstupný znak je napríklad „a", hľadáme hranu z aktuálneho stavu, ktorá je označená symbolom „a". Ak takú hranu nájdeme, posunieme dopredný ukazateľ vo vstupnom bufferi, a aktuálny stav nastavíme na stav, do ktorého hrana ide. Predpokladáme, že diagram je deterministický, čo znamená, že z jedného stavu nejdú dve cesty, označené rovnakým symbolom.

Dôležité konvencie o grafe prechodov sú:

- Niektoré stavy nazývame akceptujúce alebo konečné. Tieto stavy indikujú, že sme rozpoznali lexém.
- Ďalej je veľmi dôležité vrátiť dopredný ukazateľ o jeden znak späť, v prípade ak lexém neobsahuje symbol, ktorý by dosiahol akceptujúci stav.
- Jeden stav je označený ako štartovací. Graf prechodov potom vždy začína v tomto stave predtým, než začne spracovávať vstupný zdrojový kód.

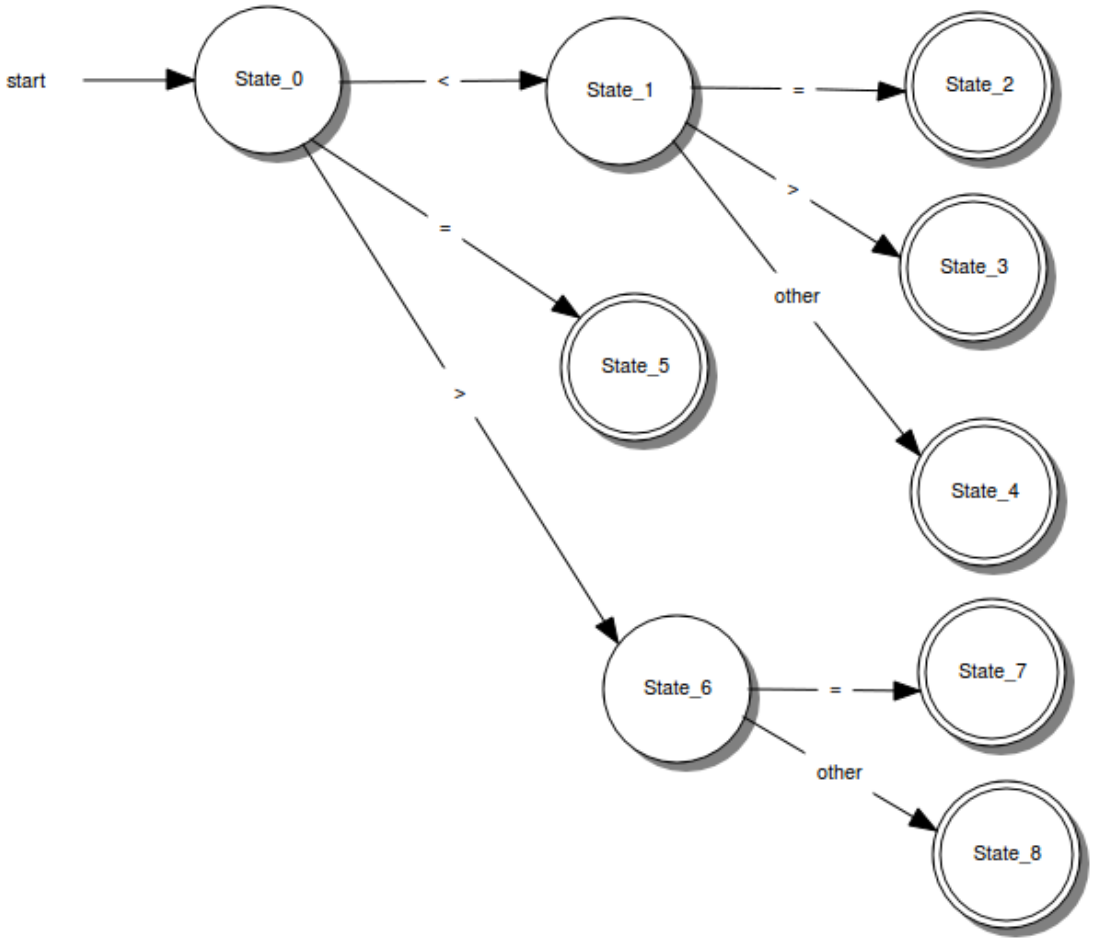
Ako príklad si uvedieme rozpoznanie relačných operátorov. V tabuľke 3.6 sú uvedené vzory lexémov a k nim príslušné tokeny.

Regulárny výraz	Token
<	LT
>	GT
<=	LE
>=	GE
<>	NE
=	EQ

Tabulka 3.6: Operátory

K daným vzorom sme vytvorili graf prechodov, znázornený na obrázku 3.1. Začíname v počiatočnom stave 0. Ak načítame prvý symbol znak <, pokračujeme do stavu 1. Ak

v tomto stave načítame znak =, pokračujeme do stavu 2, ktorý je koncový. V takom prípade sme rozpoznali token LE. Alternatívne, sme v stave 1 a načítame znak >, prejdeme do stavu 3, v ktorom lexikálny analyzátor rozpoznal token NE. Ak sme v stave 1 a príde iný znak než = alebo >, prejdeme do stavu 4, v ktorom sme rozpoznali token LT, avšak posledný znak nepatrí do rozpoznaného lexému, takže posunieme dopredný ukazateľ o jednu pozíciu späť. Obdobne postupujeme v ostatných prípadoch.



Obrázek 3.1: Graf prechodov

Pri rozpoznávaní identifikátorov a kľúčových slov nastáva problém. Existujú dva spôsoby ako rozlíšiť identifikátor od kľúčového slova:

1. Ak definujeme identifikátor pomocou regulárneho výrazu „písmeno(písmeno|číslica)*", tomuto vzoru zodpovedajú aj kľúčové slová. Preto pri rozpoznaní identifikátoru sa musíme najskôr presvedčiť, či nejde o kľúčové slovo. Spravidla máme kľúčové slová definované v tabuľke, potom stačí porovnať rozpoznaný lexém s položkami tejto tabuľky.

2. Vytvoriť samostatný regulárny výraz pre každé kľúčové slovo.

3.2 Nástroje na lexikálnu analýzu

V predchádzajúcej časti sme si vysvetlili lexikálnu analýzu, vysvetlili sme si základné pojmy a princípy lexikálnej analýzy, predovšetkým rozpoznávanie lexémov podľa vzorov definovných pomocou regulárnych výrazov. Ručné programovanie takéhoto lexikálneho analyzátoru je však veľmi náročné a náchylné na chybu. Preto existujú nástroje, ktoré podľa zadaných pravidiel vygenerujú lexikálny analyzátor, alebo dokážu samy analyzovať vstupný zdrojový kód a na výstup produkujú tokeny.

Najznámejší takýto program je Lex. Ďalej si popíšeme jeho variantu v prostredí programovacieho jazyku Python. Mierne odlišným spôsobom sa používa knižnica PyParsing popísana ďalej. Taktiež uvedieme lexikálny analyzátor Boost Spirit a nakoniec spomenieme nástroj Antlr.

3.2.1 Lex

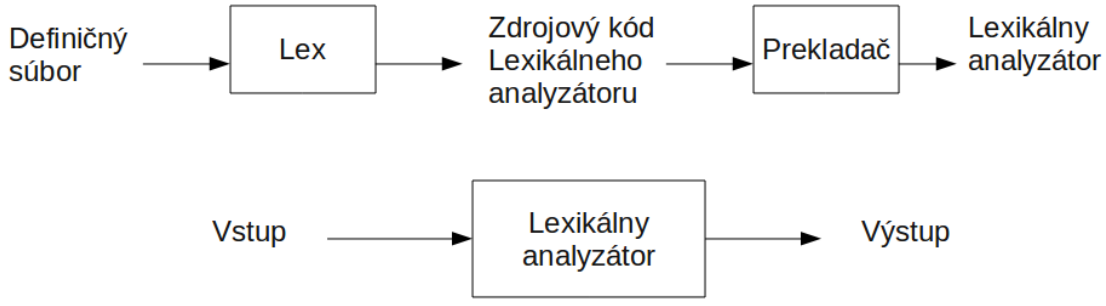
Lex je počítačový program, ktorý podľa zadaných pravidiel vygeneruje zdrojový kód lexikálneho analyzátoru v jazyky C. Často sa Lex používa s generátorom prekladačov yacc. Alternatívou k programom lex a yacc su open-source programy flex a bison[85].

Zdrojový kód programu lex je tabuľka regulárnych výrazov a k nim korešpondujúce fragmenty programu. Tabuľka je preložená na program, ktorý číta vstupný text, kopíruje ho na výstup a delí ho do reťazcov, ktoré sa zhodujú s danými regulárnymi výrazmi. Pri rozpoznaní každého takéhoto reťazcu, je vykonaný zodpovedajúci fragment programu. Rozpoznávanie reťazcov je vykonávané deterministickým konečným automatom generovaným programom lex. Fragmenty programu sú vykonávané v poradí v akom sú rozpoznávané reťazce zo vstupu.

Programy lexikálnej analýzy napísane v programe lex prijímajú aj nejednoznačnú špecifikáciu a vyberajú najdlhšiu možnú zhodu z každého bodu vstupu[50].

Princíp

Zdrojový súbor s definíciami a akciami pre jazyk lex je transformovaný do hosťovského zdrojového jazyka. Zdrojový kód takto vytvoreného lexikálneho analyzátora je následne skompilovaný prekladačom hosťovského jazyka do výsledného lexikálneho analyzátora. Tento program, lexikálny analyzátor, číta vstupný text, ktorý rozdeľuje na reťazce a vykonáva definované akcie. Na obrázku 3.2 je tento proces znázornený.



Obrázek 3.2: Lex

Štruktúra definičného súboru programu lex

Definičný súbor programu lex pozostáva z troch častí, oddelených riadkom, obsahujúcich dvojicu znakov „%%".

Prvá časť „definition section"je definičná, obsahuje definície premenných, názvov a počiatočných podmienok. Definícia má tvar `meno definícia`.

V druhej časti, „rules section", sú definované pravidlá. Definícia má tvar `vzor akcia`, kde vzor je regulárny výraz, ktorý môže používať definície z prvej časti. Akcia je fragment kódu, typicky v jazyku C.

Posledná, tretia, časť nazývaná „user code section"obsahuje pomocný zdrojový kód, funkcie v C jazyku, použité v pravidlách a vo funkci `main()`. Volanie funkcie `yylex()` spustí lexikálnu analýzu.

Príklad definičného súboru je uvedený v prílohe D.1.

3.2.2 PLY

PLY (Python Lex Yacc) je implementácia nástrojov lex a yacc v jazyky python. Pozostáva z dvoch samostatných modulov `lex.py` a `yacc.py`, ktoré sa nachádzajú v balíku ply. Modul `lex.py` sa používa na rozdelenie vstupného textu do sekvencie tokenov podľa definovaných regulárnych výrazov. Druhý modul, `yacc.py`, rozpoznáva syntax jazyka podľa definovanej bezkontextovej gramatiky podľa [13].

Hlavný rozdiel medzi ply a nástrojmi lex a yacc je v tom, že ply nepotrebuje samostatný definičný súbor, z ktorého generuje zdrojový kód analyzátora. Namiesto toho sa ply opiera na introspekciu [1] pri zostavovaní lexikálneho analyzátoru a syntaktického parseru.

Lexikálny analyzátor

Nástroj ply do značnej miery využíva konvencie pre pomenovávanie a introspekciu.

Špecifikácia tokenov Lexikálnemu analyzátoru musíme definovať zoznam názvov (typov) tokenov. Na to slúži premenná `tokens`. Napríklad:

```
tokens = ('NUMBER','PLUS','MINUS','TIMES','DIVIDE','LPAREN','RPAREN')
```

Každý token je špecifikovaný pomocou regulárneho výrazu. Každé takéto pravidlo je definované pomocou deklarácie premennej alebo procedúry so špecifickým prefixom `t_`, za ktorým nasleduje názov tokenu, definovaný v premennej `tokens`. Pre jednoduché tokeny môžeme regulárny výraz špecifikovať následovne:

```
t_PLUS = r'\+'
```

Ak je potrebné vykonať špecifickú akciu pri rozpoznaní tokenu, pravidlo môžeme definovať ako funkciu. Napríklad následovná funkcia rozpoznané číslo konvertuje na celočíselnú premennú:

```
def t_NUMBER(t):
    r'\d+'
    t.value = int(t.value)
    return t
```

[1] Skúmanie, pohľad do vlastného zdrojového kódu.

Pri použití funkcie je regulárny výraz definujúci vzor lexému špecifikovaný v dokumentačnom reťazci. Funkcia vždy prijíma jeden argument, inštanciu `LexToken`. Tento objekt má atribúty `t.type`, typ tokenu, `t.value` lexém zodpovedajúci tokenu, `t.lineno` číslo riadku, na ktorom bol token rozpoznaný a `t.lexpos` pozícia rozpoznaného lexému vo vstupnom zdrojovom kóde.

Vnútorne `lex.py` používa modul `re` na rozpoznávanie vzorov. Pri zostavovaní hlavného regulárneho výrazu sú pravidlá pridávané v následujúcom poradí:

1. Všetky tokeny, definované pomocou funkcií sú pridávané v rovnakom poradí v akom sa vyskytuju v zdrojovom kóde lexikálneho analyzátoru.
2. Tokeny definované pomocou reťazcov sú následne pridávané od najdlhších po najkratšie.

Ignorovanie tokenov V prípade, že chceme daný token ignorovať, funkcia, ktorá ho definuje nebude vracať žiadny výsledok, v opačnom prípade musí vrátiť inštanciu `LexToken`. Alternatívou je v delarácii uviesť prefiz `t_ignore_`, napríklad:

```
t_ignore_COMMENT = r'\#.*'
```

Pozícia lexémov Implicitne `lex.py` nepozná čísla riadkov, kvôli absencii informácie, čo presne znamená riadok (napr. tvar konca riadku). Pre udržovanie informácie o aktuálnom riadku je potrebné inkrementovať premennú `t.lineno` pri prechode na nový riadok.

Literály Literály môžeme špecifikovať zvlášť, pomocou premennej `literals`, ktorá obsahuje znakové literály. V prípade, že lexikálny analyzátor narazí na literál, vráti ho tak ako ho detekoval.

Spracovanie chýb Nakoniec, funkcia `t_error()` je použitá pri spracovávaní chýb pri detekcii nepovolených znakov. Ak chceme takéto chyby ignorovať, môžeme použiť metódu `t.lexer.skip(1)`, ktorá preskočí nepovolený znak a pokračuje v lexikálnej analýze.

Vytvorenie a použitie lexikáleneho analyzátora Po definovaní pravidiel a akcií môžeme jednoducho vytvoriť samotný lexikálny analyzátor:

1. funkcia `lex.lex()` vytvorí lexikálny analyzátor,
2. funkcia `lex.input(data)` nastaví vstupný text, ktorý sa bude analyzovať,
3. funkcia `lex.token()` vráti následujúci token.

Pri vytváraní lexikálneho analyzátora je možné zapnúť optimalizáciu alebo aktivovať ladiace výpisky.

Príklad lexikálneho analyzátora implementovaného pomocou ply je uvedený v prílohe D.2.

3.2.3 PyParsing

PyParsing je alternatívny prístup k tvorbe a vykonávaniu jednoduchých gramatík oproti tradičnému prístupu na báze lex a yacc, alebo oproti použitiu regulárnych výrazov.

PyParsing nevytvára žiadny rozdiel medzi lexikálnou a syntaktickou analýzou. Namiesto toho poskytuje funkcie a triedy pre tvorbu prvkov analyzátora – jeden prvok pre každú vec, s ktorou sa má hľadať zhoda.

V knižnici PyParsing nájdeme triedy pre vytváranie parseru z jednotlivých jednoduchých výrazov do komplexnejších. Výrazy môžeme kombinovať pomocou intuitívnych operátorov + pre zreťazenie (konkatenáciu) jedného výrazu za druhý, a | resp. ^ pre definovanie alternatívy (zhoda na prvú alternatívu, resp. zhoda na najdlhšiu alternatívu). Opakovanie výrazov je definované pomocou tried ako `OnoOrMore`, `ZeroOrMore` a `Optional` [59].

Ako príklad si uvedieme regulárny výraz, ktorý parsuje IP adresu nasledovanú telefónnym číslom v štýle US:

```
(\d{1,3}(?:\.\d{1,3}){3})\s+(\(\d{3}\)\d{3}-\d{4})
```

Rovnaký výraz, zapísaný pomocou PyParsing vyzerá následovne:

```
ipField = Word(nums, max=3)
ipAddr = Combine(ipField + "." + ipField + "." + ipField + "." + ipField)
phoneNum = Combine("(" + Word(nums, exact=3) + ")") + Word(nums, exact=3) +
    "-" + Word(nums, exact=4)
userData = ipAddr + phoneNum
```

Ako je vidieť, zápis pomocou PyParsing je dlhší, ale lepšie čitateľný a pochopiteľnejší.

Prvky knižnice PyParsing

PyParsing (Python Parsing) poskytuje mnoho prvkov, z ktorých si uvedieme najpoužívanejšie [75].

`Literal()` sa zhoduje so zadným textovým literálom. `CaselessLiteral()` robí to isté, ale ignoruje veľkosť písmen. `Supress()` sa zhoduje so zadaným literálom, ale nepridáva ho do výsledku. Pomocou `Keyword()` definujeme kľúčové slovo. Ďalším prvkom je `Word()`, ktorý sa zhoduje s reťazcom zostaveným zo znakov, ktoré prvok prijal ako argument. Môžeme definovať aj regulárny výraz, pomocou prvku `Regexp()`. Existuje mnoho ďalších prvkov, ako `quotedString()`, `delimitedList()`, `makeHTMLTags()` a iné.

Prvky analyzátora môžeme kvantifikovať podobne ako regulárne výrazy, k čomu slúžia funkcie `Optional()`, `ZeroOrMore()`, `OneOrMore()` a niekoľko dalších. Prvky sa dajú pomocou `Group()` zoskupovať a pomocou `Combine()` kombinovať.

3.2.4 Boose Spirit

Boost Spirit je objektovo orientovaný parser a knižnica pre C++. Umožňuje písať gramatiky a formálne popisy podobne s rozšírenou Backus-Naur formou priamo v C++. Tieto špecifikácie gramatiky môžeme ľubovoľne kombinovať s ostatným kódom v C++, a vďaka sile šablón sú ihneď vykonateľné[30].

Vzhľadom na to, že vstupná gramatika a formát výstupu je celý naprogramovaný v C++ nepotrebujeme žiadne samostatné nástroje na kompiláciu, preprocessing alebo integráciu týchto vecí do procesu zostavovania. Toto často umožňuje jednoduchší a efektívnejší kód.

Knižnica pozostáva zo štyroch hlavných častí:

Spirit.Classic je základný kód prevzatý z bývalého Boost Spiritv1.8. Bola pridaná vrstva, ktorá zabezpečuje spätnú kompatibilitu a kód bol presunutý pod menný priestor `boost::spirit::classic`.

Spirit.Qi je knižnica umožňujúca vytvoriť rekurzívne zostupný parser. Je možné použiť vstavaný doménovo-špecifický jazyk pre popis gramatiky, ktorú implementujeme a pre popis pravidiel na ukladanie parsovaných informácií.

Spirit.Lex je knižnica používaná pri vytváraní lexikálnych analyzátorov. Doménovo-špecifický jazyk vstavaný v Spirit.Lex umožňuje definovať regulárne výrazy používané pri rozpoznávaní lexémov, asociovať tieto regulárne výrazy s kódom, ktorý má byť vykonaný kedykoľvek je lexém rozpoznaný.

Spirit.Karma je knižnica umožňujúca vytvoriť kód pre rekurzívne zostupný, typmi dát riadený formátovač výstupu. Vstavaný doménovo-špecifický jazyk je takmer ekvivalentný s jazykom popisu parseru použitým v Spirit.Qi, okrem toho, že je použitý pre popis požadovaného výstupného formátu, ktorý sa má generovať z danej dátovej štruktúry.

3.2.5 Antlr

Antlr (ANother Tool for Language Recognition) je nástroj používaný pri konštrukcii nástrojov pre formálne jazyky ako napríklad prekladače, analyzátory alebo kompilátory. Antlr, podobne ako lex a yacc, je generátor prekladačov a používa sa na generovanie zdrojového kódu parserov, alebo analyzátorov jazyka zo špecifikácie jazyka. Antlr prijíma vstupnú gramatiku ako presný popis jazyka rozšírený o sémantické akcie. Cieľovým jazykom generovaného kódu môže byť Java, C/C++, C#, Python, Ruby, Perl alebo niektorý iný programovací jazyk [66].

Kapitola 4

Programy na porovnávanie dokumentov

Užívatelia pri práci na počítači často potrebujú zistiť rozdiel medzi dvoma súbormi. Niekedy môžu mať dva súbory, kde jeden vznikol niekoľkými úpravami druhého súboru. Alebo dva rôzne súbory vytvorené z jedného originálu dvoma rozdielnymi ľuďmi. V takýchto prípadoch hľadajú program na zistenie rozdielov medzi porovnávanými súbormi. Existuje mnoho rôznych programov na porovnávanie súborov. Väčšinou ide o porovnávanie dvoch súborov riadok po riadku a zobrazenie zmien. Niektoré pracujú v príkazovom riadku alebo majú grafické užívateľské rozhranie (GUI). Obe má svoje výhody. Výstup programu príkazového riadku môžeme ďalej strojovo spracovávať v skripte – napríklad pravidelne ukladať zmeny súboru, čo nám výrazne ušetrí pamäťový priestor. Grafické užívateľské rozhranie je vhodné pri prezeraní rozdielov súborov užívateľom. Program názorne zobrazí oba texty vedľa seba a vyznačí rozdiely. Pri niektorých textoch ako sú zdrojové kódy, dokáže zvýrazňovať syntax. ďalšou možnosťou je trojcestné porovnávanie súborov, kde je jeden originál a dve rôzne kópie. Toto porovnávanie je veľmi vhodné pri práci viac ľudí na jednom súbore. Dokážeme zistiť, kto previedol aké zmeny voči originálu a môžeme obe nové verzie zlúčiť do jednej. Taktiež môžeme porovnávať celé zložky, súbor po súbore. Pre porovnávanie netextových súborov môžeme použiť binárne porovnávanie bajt po bajte. V prípade štruktúrovaných dát ako je XML dokument je vhodné porovnávať dokumenty s ohľadom na štruktúru, čo môže viesť k lepšej čitateľnosti a prehľadnosti rozdielov dokumentov.

V nasledujúcom texte si predstavíme niektoré z programov na porovnávanie rôznych typov súborov.

4.1 Porovnávanie textov

4.1.1 GNU Diffutils

Medzi najznámejšie a najstaršie programy na porovnávanie súborov patrí skupina programov GNU Diffutils. V čase písania tejto práce je k dispozícii verzia 2.8.1 z 5. apríla 2002. GNU Diffutils pozostáva zo štyroch samostatných programov:

- diff – nájde rozdiely medzi dvoma súbormi
- diff3 – trojcestné porovnanie súborov riadok po riadku
- sdiff – porovná dva súbory a zobrazí ich vedľa seba

- cmp – porovná dva súbory bajt po bajte

Popis

Je viac spôsobov ako chápať rozdiely medzi súbormi. Jeden zo spôsobov ako pochopiť rozdiely medzi súbormi je zoznam riadkov, ktoré boli zmazané, vložené alebo zmenené. Po aplikácii týchto zmien dostaneme druhý súbor. `GNU diff` porovnáva dva súbory riadok po riadku, hľadá skupiny riadkov, ktoré sú odlišné a vypíše každú takúto skupinu. Dokáže vypisovať rozdielne riadky v rôznych formátoch, ktoré majú rôzne účely. `GNU diff` vie zobraziť, či sú súbory rozdielne bez prihliadnutia na niektoré odlišnosti ako napríklad veľkosť písmen alebo počet bielych znakov (medzery, tabulátory, prázdne riadky).

Iný spôsob ako chápať rozdiely medzi súbormi je zoznam párov bajtov, ktoré môžu byť rovnaké alebo odlišné. Program `cmp` vypíše rozdiely medzi dvoma súbormi bajt po bajte namiesto riadok po riadku. Z toho vyplýva, že `cmp` je vhodnejší než `GNU diff` pri porovnávaní binárnych súborov. Pre textové súbory je `cmp` užitočný hlavne vtedy, ak chceme vedieť, či sú dva súbory identické alebo či je jeden prefixom druhého.

Program `diff3` bežne porovnáva tri vstupné súbory riadok po riadku, hľadá skupiny riadkov, ktoré sú rôzne a vypíše ich. Jeho výstup je navrhnutý tak, aby bolo ľahké preskúmať dve rôzne verzie toho istého súboru.

Pre názornejšie zobrazenie rozdielu dvoch dokumentov môžeme použiť program `sdiff`, ktorý nám zvisle rozdelí obrazovku na dve časti. V ľavej časti zobrazí prvý dokument, v pravej časti zobrazí druhý dokument a v deliacom priestore medzi nimi zobrazí znaky, ktoré indikujú zmeny spôsobené zmazaním, pridaním alebo zmenením riadku. Takto máme prehľadne zobrazené oba texty jeden vedľa druhého.

História

Program `diff` bol vytvorený na počiatku sedemdesiatych rokov minulého storočia na operačnom systéme UNIX, ktorý sa objavoval v AT&T Bell Labs v Murray Hill, New Jersey. Finálnu verziu, zahrnutú do 5. vydania Unixu v roku 1974, napísal Douglas McIlroy. Tento výskum publikoval v roku 1976 v spolupráci s Jamesom W. Huntom, ktorý vyvíjal počiatočný prototyp programu `diff` [32]. McIlroyova práca predchádzala a bola ovplyvňovaná porovnávacím programom Steva Johnsona na operačnom systéme GECOS a programom `proof` Mika Leska. `proof` tiež vznikol na UNIXe, podobne ako program `diff`, produkoval riadkové zmeny a dokonca používal ostré zátvorky $>$ a $<$ pre vyjadrenie vloženia a zmazania riadku na výstupe programu. Heuristika použitá v týchto prvých aplikáciach bola považovaná za nespoľahlivú. Potenciálny úspech programu `diff` povzbudil McIlroya do výskumu a návrhu robustnejšieho nástroja, ktorý môže byť použitý v rôznych úlohách, ale splní výkonové limity minipočítača PDP-11. Jeho prístup k problému vyústil k spolupráci jednotlivcov z Bell Labs ako Alfred Aho, Elliot Pinson, Jeffrey Ullman a Harold S. Stone.

Dnešnú verziu programu `GNU diff` napísali Paul Eggert, Mike Haertel, David Hayes, Richard Stallman a Len Tower. Wayne Davison navrhol a implementoval unifikovaný výstupný formát. Základný algoritmus, ktorý nástroje GNU Diffutils používajú, je popísaný v článku Eugena Myersa [65] a v článku Millera Webba a Eugene Myersa [62]. Algoritmus bol tiež nezávisle objavený a popísaný v článku E. Ukkonena [77].

V tejto podkapitole boli použité texty z [53] a [39].

4.1.2 Araxis Merge

Axis Merge je vizuálny porovnávací, zlučovací (merging) a synchronizačný program [7]. Porovnáva a zlučuje zdrojové kódy, webové stránky, XML a iné textové súbory. Priamo otvára a porovnáva text zo súborov programov Microsoft Word a Excel, ďalej súbory Open Document, Pdf a Rtf. Porovnáva obrázky a binárne súbory. Synchronizuje zložky aj cez FTP. Podporuje trojcestné porovnávanie. Výstup je možné uložiť vo formáte XML, html, alebo diff.

Program vyvíja firma Araxis pre operačné systémy Windows a Mac OS X pod proprietárnou licenciou.

4.1.3 Beyond Compare

Beyond Compare porovnáva súbory a zložky, dokáže ich zlučovať a synchronizovať [72]. Program vyvíja firma Scooter software pre operačné systémy Windows a Linux pod proprietárnou licenciou.

4.1.4 Ďalšie programy

Kompare [84], program s grafickým rozhraním pre porovnávanie textových dokumentov. Dokáže vytvoriť a aplikovať patch. Slobodný softwér pod GPL licenciou, v prostredí KDE.

Meld [86], jednoduchý program na porovnávanie textových dokumentov a zložiek. Slobodný softwér pod GPL licenciou.

4.2 Porovnávanie XML dokumentov

Pri rozšírení používania XML dokumentu vznikla potreba porovnávať tieto dokumenty. Na túto potrebu zareagovalo viacero spoločností, ktoré vyvinuli nástroje na porovnávanie týchto dokumentov. Rôzne nástroje používajú rôzne algoritmy, od tých jednoduchších, ktoré berú XML dokument ako textový súbor, až po tie komplexné, ktoré dokážu porovnávať dva neusporiadané XML dokumenty. Algoritmy majú tiež rôznu časovú zložitosť a presnosť.

V ďalšom texte si prezentujeme niektoré hlavné programy. Kapitola čerpá niektoré materiály z [3].

4.2.1 DeltaXML

DeltaXML má vedúce postavenie na trhu. Používa techniky založené na LCS. Pri použití Wu [87] a D-Band [65] algoritmov je časová zložitosť algoritmu kvázi-lineárna. Veríme [1], že časová zložitosť je $O(|x|.D)$, kde $|x|$ je celková veľkosť oboch dokumentov a D je editačná vzdialenosť (edit distance) medzi nimi.

Posledná verzia programu podporovala pridávanie kľúčov (pomocou atribútov alebo v DTD), ktoré boli použité pri vylepšení správnosti zhody (napr. porovnávaj osobu podľa jej mena). Podporované je tiež porovnávanie neusporiadaných XML dokumentov.

Pretože algoritmus Wua je aplikovaný na každú úroveň samostatne, výsledok nie je striktne minimálny. Avšak skúsenosti ukazujú, že vo výsledku (90%) je striktne minimálny.

[1] Algoritmus zatiaľ nebol publikovaný.

4.2.2 Logilab XmlDiff

XmlDiff [52] Open Source program na porovnávanie XML dokumentov. Zmeny sú reprezentované pomocou XUpdate alebo pomocou vlastného interného formátu. Ponúka dva rozdielne algoritmy na detekciu zmien.

Pre veľké súbory používa [18]. Tak ako predchádzajúci algoritmus, aj tento aplikuje LCS [65] na každú úroveň a každý typ uzlu. Z toho dôvodu ako DeltaXML, nenájde minimálny rozdiel. Časová zložitosť je $O(l.|D1|.e)$, kde l je počet typov uzlov, $|D1|$ je hĺbka stromu a e je editačná vzdialenosť (edit distance) medzi dokumentami.

Druhý algoritmus je [90, 71]. Ten nájde minimálny počet úprav vkladania a zmazania podľa Taiovho modelu. Presnejšie, vylepšili Zhang Shasha algoritmus pridaním úpravy *swap* medzi uzlom a jeho susedom [11]. Zložitosť algoritmu je kvázi-kvadratická.

4.2.3 Microsoft XmlDiff

Microsoft implementoval sadu nástrojov XmlDiff a Patch [37]. Zdrojové kódy sú voľne dostupné. Výstupný formát je použitý Microsoft XDL. Tento nástroj je podobný s XML TreeDiff [20] vyvíjaný firmou IBM a ktorý je založený na [21] a [90, 71]. Sú použité dva algoritmy. Prvý, *Treewalk-XmlDiff* je rýchly algoritmus, ktorý používa podobné vzorce ako XyDiff na obmedzenie počtu navštívených uzlov počas priechodu stromom. Druhý algoritmus, *Precise-XmlDiff*, je implementovaný pomocou Zhang Shasha algoritmu [90, 71]. Ako model je použitý Taiov model.

4.2.4 XyDiff

XyDiff bol navrhnutý v [56, 55]. Program je Open Source a používa sa v spoločnosti Xyleme [29]. XyDiff je rýchly algoritmus, ktorý podporuje operácie presunu podstromu a DTD ID atribúty. Algoritmus hľadá zhodu veľkých podstromov v oboch dokumentoch a následne propaguje tieto zhody.

XyDiff má časovú a priestorovú zložitosť $O(n.\log(n))$, kde n je veľkosť dokumentov. Avšak vo všeobecnosti tento algoritmus nenájde minimálny výsledok.

4.2.5 MMDiff and XMDiff

S. Chawathe na základe [18, 17] implementoval programy Main Memory Diff a Externall Memory Diff. Programy pracujú s usporiadanými stromami. MMDiff má kvadratickú časovú zložitosť. Vychádza z MMDiffu, ale redukuje počet I/O operácií, pričom je vhodný pre porovnávanie veľkých súborov.

4.2.6 XDiff

Program XDiff [80] porovnáva neusporiadané stromové štruktúry. Časová zložitosť algoritmu je kvadratická.

4.2.7 Ostatné programy

XML treediff [20] vyvíjaný firmou IBM a postavený na algoritmoch [21] a [90, 71]. Bol vyvinutý skôr a má blízko k Microsoft XmlDiff. Používa rovnaký algoritmus (Zhang-Shasha) s podobnou prerezávacou fázou.

Sun tiež vydal nástroj na porovnávanie XML dokumentov pod názvom DiffMK, ktorý počíta rozdiely medzi dvomi XML dokumentami. Tento nástroj je založený na algoritme GNU diff a používa popis XML uzlov namiesto stromu. V podobnom duchu je Open Source program xml diff od DecisionSoft. Tento program používa lineárnu reprezentáciu XML dokumentu, t.j. XML dokument je transformovaný na text a každý riadok je považovaný za uzol. Nad týmto textom sa spustí unix diff. Výkon a kvalita posledných dvoch nástrojov korešponduje výlučne s programom GNU diff.

4.3 Porovnávanie zdrojových kódov

4.3.1 Aqua Data Studio

Aqua Data Studio je komplexné integrované vývojové prostredie (IDE) pre dotazovanie, správu a vývoj databáz [6]. Toto IDE obsahuje nástroj na porovnávanie schém databáz, objekty schém databáz, dotazov, výsledkov dotazov, súborov, adresárov, obsahu schránky a históriu verzií. Výsledky všetkých porovnaní sa dajú uložiť vo formáte HTML.

Produkt vyvíja spoločnosť AquaFold pod proprietárnou licenciou pre systémy Windows, Macintosh, Linux a iné platformy podporujúce Javu.

4.3.2 CodeCompare

CodeCompare je samostatná aplikácia alebo doplnok do IDE VisualStudio [23]. Porovnáva zdrojové kódy pomocou štrukturálneho (pre jazyky C#, C++, VisualBasic a JavaScript) a lexikálneho porovnania vzhľadom na daný programovací jazyk. Zobrazuje zmeny štruktúry tried v dvoch revíziách kódu. Integrácia s mnohými verzovacími systémami. Ďalej podporuje trojcestné porovnávanie a zlučovanie, porovnávanie a synchronizáciu zložiek.

Program je vyvíjaný firmou Devart pre systém Windows. Program ponúka bezplatnú aj platenú rozšírenú verziu.

4.3.3 Compare++

Compare++ je program na porovnávanie zdrojových kódov, textov a zložiek. Používa štruktúrované porovnávanie zdrojových súborov. V súčastnosti sú podporované jazyky C/C++, Java a C#.

Program vyvíja firma Coodesoft pre systém Windows pod proprietárnou licenciou.

4.3.4 OOP-DIFF

OOP-DIFF pracuje so štruktúrou zdrojových kódov v jazyku C#. Program porovnáva logické časti kódu ako metódy, triedy bez ohľadu na poradie [73].

4.4 Ostatné programy

Program fc je alternatívou k programu diff, používanou v systémoch Microsoft Windows. Počiatok tohto programu siaha až k OS MS-DOS 3.3.

ExamDiff Pro je jednoduchý program porovnávajúci dva textové, binárne súbory alebo zložky.

Guiffy SureMerge je nástroj na porovnávanie súborov, zložiek archívov. Podporuje trojcestné porovnávanie, výstup v html.

Ediff je porovnávací nástroj, ktorý je súčasťou editoru emacs, rovnako aj vimdiff v editoru vim.

4.5 Formát výstupu

4.5.1 Textové dokumenty

Prvým programom na porovnávanie textov bol `GNU diff`. Ten zaviedol niekoľko formátov výstupu, ktoré sa dodnes používajú. Implicitný je „Normálny formát výstupu", ktorý zobrazuje len rozdielny text. Ďalej sú dva kontextové formáty výstupu, „Kontextový" a „Unifikovaný" formát výstupu, ktoré v okolí rozdielneho textu zobrazujú niekoľko riadkov pôvodného spoločného textu – kontextu. Samozrejmosťou pri výstupe sú informácie o číslach riadkov, rozsah, resp. dátum zmien. Okrem týchto štandardných formátov výstupu `GNU diff` ponúka aj niektoré ďalšie [53]. V nasledujúcom texte popíšeme každý z formátov. V prílohe C.1 si názorne tieto formáty predvedieme.

Normálny formát výstupu

Normálny formát výstupu programu `GNU diff` vypisuje každú skupinu rozdielnych riadkov bez obklopujúceho kontextu. Niekedy je takýto výstup najlepší spôsob ako ukázať, ktoré riadky boli zmenené bez rozptýlenia okolitými nezmenenými riadkami (avšak takýto výsledok môžeme dosiahnúť aj pri kontextovom alebo unifikovanom formáte, pri použití 0 riadkov ako kontextu). Normálny formát výstupu je implicitný kvôli kompatibilite so staršími verziami `GNU diff`u a štandardu POSIX.

Normálny formát výstupu pozostáva z jedného alebo viac skupín odlišných riadkov. Každá skupina predstavuje jednu oblasť, kde sú súbory rozdielne. Normálny formát výstupu vyzerá následovne:

```
značka zmeny
< súbor1-riadok
< súbor1-riadok...
---
> súbor2-riadok
> súbor2-riadok...
```

Existujú tri typy značiek zmeny. Každá obsahuje číslo riadku alebo čiarkou oddelený rozsah riadkov v prvom súbore, jeden znak indikujúci typ zmeny, ktorá nastala a číslo riadku alebo čiarkou oddelený rozsah riadkov v druhom súbore. Typy značiek zmien:

- `lar` – pridané riadky v rozsahu `r` z druhého súboru za riadok `l` z prvého súboru.
- `fct` – vymenené riadky prvého súboru rozsahu `f` za riadky druhého súboru rozsahu `t`.
- `ldr` – zmazané riadky v rozsahu `r` z prvého súboru za riadkom `l` v druhom súbore.

Každý riadok začína znakom `>` alebo `<` nasleduje medzera a text daného riadku v originálnom súbore. Znaky `>` a `<` sú volené tak, aby bolo na prvý pohľad vidieť o akú zmenu ide. Znak `>` znamená pridaný riadok, znak `<` znamená zmazaný riadok. ©pecíalnym riadkom je oddelovač `---`, ktorý nájdeme v skupine zmenených riadkov. Oddelovač rozdeľuje skupinu na dve časti a tak určí, ktoré riadky majú byť nahradené druhými.

Kontextové formáty výstupu

Zvyčajne, keď pozeráme na rozdiely medzi súbormi, chceme vidieť nezmenené časti súborov blízko riadkov, ktoré sú rozdielne. Tieto blízke časti súborov voláme kontext.

`GNU diff` poskytuje dva formáty výstupu, ktoré zobrazia kontext okolo rozdielnych riadkov: kontextový formát výstupu a unifikovaný formát výstupu. Taktiež môže zobraziť, v ktorej funkcii alebo sekcii súboru sa nachádzajú rozdielne riadky.

Kontextový formát výstupu

Kontextový formát výstupu zobrazuje niekoľko riadkov kontextu okolo rozdielnych riadkov. Tento formát výstupu je štandardný pre distribúciu aktualizácií zdrojových kódov.

Kontextový formát výstupu začína s dvojriadkovou hlavičkou:

```
*** súbor1 čas-poslednej-zmeny-súboru1
--- súbor2 čas-poslednej-zmeny-súboru2
```

ďalej nasleduje niekoľko skupín odlišných riadkov. Každá skupina zobrazuje jednu časť, kde sa súbory líšia. Napríklad:

```
***************
*** súbor1-rozsah-riadkov ****
 súbor1-riadok
 súbor1-riadok...
--- súbor2-rozsah-riadkov ----
 súbor2-riadok
 súbor2-riadok...
```

Riadky kontextu okolo riadkov, ktoré sa líšia, začínajú s dvoma medzerami. Riadky, ktoré sú rozdielne medzi súbormi, začínajú jedným z nasledujúcich znakov a medzerou, ktoré indikujú, o akú zmenu ide:

- `!` Riadok, ktorý je časťou skupiny jedného alebo viac riadkov, ktoré sú zmenené medzi dvoma súbormi. V druhom súbore je zodpovedajúca skupina riadkov označená `!`.
- `+` Riadok vložený do druhého súboru.
- `-` Riadok zmazaný z prvého súboru.

Unifikovaný formát výstupu

Unifikovaný formát výstupu je variácia kontextového formátu výstupu, ktorá je kompaktnejšia, pretože zanedbáva nadbytočné riadky kontextu.

Unifikovaný formát výstupu začína s dvojriadkovou hlavičkou:

```
--- súbor1 čas-poslednej-zmeny-súboru1
+++ súbor2 čas-poslednej-zmeny-súboru2
```

Nasleduje jeden alebo viac skupín odlišných riadkov. Každá skupina zobrazuje jednu časť, kde sa súbory líšia. Napríklad:

```
@@ súbor1-rozsah-riadkov súbor2-rozsah-riadkov @@
riadok-každého-súboru
riadok-každého-súboru...
```

Riadky bežne začínajú medzerou. Tie, ktoré sú v skutočnosti rozdielne, začínajú jedným z nasledujúcich znakov:

- `+` Riadok vložený do prvého súboru.
- `-` Riadok zmazaný z prvého súboru.

Formát výstupu vedľa seba

Formát výstupu vedľa seba je podobný normálnemu formátu výstupu, avšak zobrazené sú naraz oba súbory v dvoch stĺpcoch. V strede medzi stĺpcami sú značky:

- medzera Odpovedajúce si riadky sú rovnaké.
- | Odpovedajúce si riadky sú rozdielne.
- < Súbory sú rozdielne a iba prvý súbor obsahuje riadok.
- > Súbory sú rozdielne a iba druhý súbor obsahuje riadok.
- (Iba prvý súbor obsahuje riadok, ale rozdiel je ignorovaný.
-) Iba druhý súbor obsahuje riadok, ale rozdiel je ignorovaný.
- \ Odpovedajúce si riadky sú rozdielne a iba prvý riadok je neúplný.
- / Odpovedajúce si riadky sú rozdielne a iba druhý riadok je neúplný.

Ostatné formáty výstupu

Niektoré formáty výstupu produkujú skripty, ktoré dokážu vytvoriť z prvého súboru druhý. Podporované sú `ed` (editačné) skripty, spätné `ed` skripty a `RCS` skripty.

`GNU diff` podporuje aj porovnávanie zdrojových kódov jazyka C. Výstup takéhoto porovnania obsahuje všetky riadky z oboch súborov. Riadky spoločné pre oba súbory sa vo výstupe nachádzajú len raz, odlišné riadky sú oddelené pomocou preprocesoru jazyka C makrami `#ifdef name` alebo `#ifndef name`, `#else` a `#endif`. Pri preklade zdrojového kódu môžeme vybrať, ktorá verzia bude skompilovaná.

ďalej môžeme použiť formát výstupu, ktorý špecifikuje označenie jednotlivých skupín textu. Môžeme nadefinovať špecifické značky, ktorými má zmenený text začínať a iné špecifické značky, ktorými má zmenený text končiť. Napríklad pri porovnaní TEXových súborov, môžeme určiť, že zmenené skupiny riadkov budú uzavreté do tagov `\begin{em}` a `\end{em}`. Ostatné riadky ponecháme bez zmien. Podobné úpravy značiek indikujúcich zmenený text môžeme nadefinovať aj pre jednotlivé riadky.

4.5.2 XML dokumenty

Pri porovnávaní textových dokumentov bol priekopníkom program diff, ktorý zaviedol niekoľko formátov výstupu, ktoré sa dodnes používajú ako nepísaný štandard. Avšak pri programoch porovnávajúcich XML dokumenty je situácia iná. Exituje mnoho programov a veľa z nich implementuje vlastný formát výstupu, v ktorom prezentuje rozdiel medzi dokumentami. Niektoré sa zameriavajú na čo najmenšie pamäťové nároky, niektoré majú za cieľ vytvoriť prehľadný výstup, ktorý bude dobre čitateľný a prehľadný.

Dôležitú úlohu hrajú aj formálne vlastnosti týchto formátov – či je výstup reverzibilný, alebo či môžeme výstupy agregovať.

Niektoré formáty sú v tvare editačného skriptu, t.z. zoznam zmien potrebných k transformácii jedného dokumentu na druhý, iné ponúkajú iba rozdiel medzi dokumentami.

V tejto časti si popíšeme základné formáty výstupu rozdielov medzi dvoma XML dokumentami. Ukážky formátov nájdeme v prílohe C.2.

XUpdate

XUpdate je špecifikácia definujúca syntax a sémantiku jazyku pre aktualizáciu XML dokumentov, ktorá je navrhnutá pre použitie nezávisle na implementácii [5]. Popisuje úpravy ako vloženie, zmazanie a editáciu, založené na Selkowom modeli 2.2.1. Uzly ovplyvnené zmenami sú identifikované v originálnom dokumente pomocou XPath výrazov.

Hlavné nedostatky sú:

- Nepodporuje komplexnejšie operácie ako presun podstromu.
- Výstup nemôžeme reverzovať. T.j. z delty[2] $V_1 \to V_2$ nie je možné vytvoriť $V_2 \to V_1$

Reverzácia delty sa intuitívne skladá z nahradenia vloženia za zmazanie a naopak. Tu, ale nie je obsah uzlov, ktoré sú odstránené, uložený v delte, takže reverzovaná delta nevie, ktoré uzly majú byť vložené.

Výhodou používania XPath výrazov je, že intuitívne dávajú kontext, v ktorom sú prevedené zmeny – tagy predchádzajúcich uzlov.

XyDelta

XyDelta [56, 55] je formát navrhnutý v rámci projektu Xyleme pre balík programov `XyDiff Tools` [29]. Popisuje úpravy vloženie, zmazanie, zmena uzlu a presun podstromu. Uzly ovplyvnené zmenami sú identifikované pomocou perzistentných identifikátorov XID. Každá operácia presunu je reprezentovaná párom operácii vloženie a zmazanie na jednom uzle (rovnaké XID).

XyDelta má matematické vlastnosti. Môžu byť agregované, to znamená, že z délt $V_1 \to V_2$ a $V_2 \to V_3$ je možné zostrojiť deltu $V_1 \to V_3$. Ďalej môžu byť reverzované, t.j. zostrojiť deltu $V_2 \to V_1$. Formálny model XyDelta považuje každú XyDelta ako množinu úprav namiesto sekvencie úprav [28]. To umožňuje porovnávanie délt, synchronizáciu délt a aplikovanie iba niektorých časti delty na dokument.

Hlavným nedostatkom XyDelta je to, že vyžaduje perzistentný identifikátor. Okrem toho XID nedáva taký intuitívny pohľad na kontext, v ktorom sa zmeny nachádzajú. Na získanie kontextu zmien z hlavného dokumentu sa používajú práve identifikátory XID.

Microsoft XML Diff Language

Microsoft XML Diff Language (XDL) [37] je proprietárny XML-orientovaný jazyk pre popis rozdielov medzi dvomi XML dokumentami. Inštancia XDL je nazývaná *XDL diffgram*. Narozdiel od predchádzajúcich formátov, XDL je založený na Taiovom modeli 2.2.1. Podobne ako XyDelta, úprava presunu podstromu je reprezentovaná ako dve operácie – vloženie a zmazanie. Uzly ovplyvnené zmazaním alebo aktualizáciou sú identifikované podľa zhody atribútov. XDL obsahuje relatívnu cestu (XDL-path) voči súčasnému „kontextovému uzlu".

[2] Rozdiel medzi dokumentami.

Nevýhoda XDL voči XyDelta je, že diffgramy nemôžu byť reverzované alebo agregované. V porovnaní s XUpdate obsahuje XDL cestu, ktorá nemá takú výpovednú hodnotu, avšak zaberá menej priestoru.

Výhody XDL sú:

- Verifikácia zdrojového a cieľového dokumentu pomocou hashu uloženého v diffgrame.
- Operácia *copy*, ktorá môže v niektorých prípadoch ušetriť priestor.
- XDL je validovateľné pomocou DTD.

Fakt, že XDL používa Taiov model namiesto Selkowho môže byť vnímané ako výhoda aj nevýhoda. Závisí to na požiadavkách aplikácie.

DeltaXML

DeltaXML je formát prezentovaný v [26]. Rozdiel pozostáva z originálneho dokumentu, do ktorého sú pridané atribúty pre popis zmeny ako zmazanie, vloženie alebo aktualizácia. Identifikácia uzlov s rovnakým rodičom a rovnakými názvami elementov je dosiahnutá vložením susedných uzlov, ktoré sú nezmenené (označené atribútom `deltaxml:unchanged`). Inými slovami, existuje triviálne mapovanie medzi deltou a originálnym dokumentom, a delta má rovnaký „vzhľad"ako originálny dokument. Úprava presun podstromu nie je povolená.

Tento formát je priestorovo oveľa náročnejší, ale jednoduchšie čitteľny a pochopiteľný človekom.

XML Update Language

Program `XML TreeDiff` firmy IBM používa formát XML Update Language (XUL) [20] na zobrazenie zmien medzi dvomi XML dokumentami. Každá XUL delta je množina vnorených uzlov, avšak štruktúra originálneho stromu bola redukovaná na potrebné minimum. Stále je však možné, zobraziť štruktúru nezávisle na úpravách.

Kapitola 5

Analýza a návrh aplikácie

Pri vytváraní aplikácie bola venovaná pozornosť podrobnej analýze problému, kvalitnému návrhu riešenia a presnej implementácii.

5.1 Špecifikácia

Táto práca má za úlohu navrhnúť a následne implementovať aplikáciu, ktorá porovnáva rôzne dokumenty a zobrazuje ich rozdiel. Má názvom `mediadiff`.

Aplikácia má byť modulárna, ktorá by mala byť vhodnou náhradou za program `diff`. Dôležitou funkcionalitou je možnosť porovnávania aj iných, ako textových súborov, na rozdiel od programu `diff`.

Keďže rôzne typy súborov sú rôzne interne reprezentované, majú odlišnú sémantiku a štruktúru, musíme pre každý takýto typ vyvinúť špecifický modul, ktorý bude implementovať potrebnú funkcionalitu. Existujú rôzne typy súborov:

- textové súbory,
- konfiguračné súbory,
- zdrojové kódy,
- súbory kancelárskych balíkov,
- pdf súbory,
- XML dokumenty,
- audio súbory,
- video súbory,
- obrázky,
- archívy,
- adresáre a súborové systémy,

a mnohé ďalšie typy súborov.

Cieľom tejto práce je niektoré z týchto typov súborov, navrhnúť a implementovať porovnávanie pre tieto typy súborov. V predchádzajúcej práci [60] sú implementované moduly na

porovnávanie textových a konfiguračných súborov, súborov balíku OpenOffice a porovnávanie obrázkov. Súbežne s touto prácou sa vyvíjajú moduly pre porovnávanie audio súborov, a prepracované porovnávanie obrázkov. Tieto moduly budú následne súčasťou výslednej aplikácie, ktorá porovnáva rôzne typy dokumentov.

Ako základný rozširujúci modul budeme implementovať porovnávanie zdrojových kódov. Tento modul bude môcť porovnávať zdrojové kódy rôznych programovacích jazykov a zobrazovať ich rozdiely. Pretože jednotlivé programovacie jazyky sa od seba líšia jak syntaxou, tak sémantikou, bude tento modul rozdelený na submoduly, každý pre jeden konkrétny programovací jazyk. Primárne sa zameriame na implementačný jazyk Python a ďalej na veľmi používané jazyky C/C++ a Javu.

5.2 Vývojové prostriedky

Pre úspech aplikácie na trhu hrajú veľmi dôležitú úlohu faktory ako platforma, na ktorej aplikácie funguje a s ňou spojená prenositeľnosť programu, rýchlosť aplikácie, závislosť na knižniciach alebo inom softvéri, a v neposlednom rade čas potrebný pre uvedenie na trh. Tieto faktory do značnej miery ovplyvňuje výber programovacieho jazyka a knižníc, vývojový model a vývojové prostriedky ako správa zdrojového kódu a vývojové prostredie. Tieto prvky si v následujúcej časti rozpíšeme, vyberieme tie z nich, ktoré nám pomôžu k dosiahnutiu cieľov.

5.2.1 Vývojový model

Vývojový model priamo ovplyvňuje čas potrebný k vývoju aplikácie a pružnosť v reakcii na zmeny alebo chyby. Existuje rada vývojových modelov ako vodopádový model, iteratívny inkrementálny model, RUP, Agilný vývoj, vývoj riadený testom a mnohé iné. Každý z modelov má svoje výhody a nevýhody. V našom prípade sme si zvolili ako vývojový model iteratívny inkrementálny model. Jeden cyklus (iterácia) pozostáva z následujúcich etáp:

1. špecifikácia,
2. návrh,
3. implementácia,
4. testovanie.

Počet cyklov určíme podľa potreby. V každom cykle získava aplikácia viac funkcionality (inkrement), až po poslednom cykle je aplikácia hotová a funkčná. Jednotlivé iterácie prechádzajú určitými fázami:

1. Prvou fázou je **počiatok**, táto fáza je najkratšia a špecifikuje sa v nej rozsah projektu, požiadavky a riziká.
2. Nasleduje fáza **rozpracovanie**, ktorá je predovšetkým zameraná na analýzu a návrh aplikácie. Postupne, keď sú navrhnuté určité dielčie časti, začína sa s ich implementáciou.
3. Vo fáze **implementácie** je už návrh skoro hotový a naplno sa pracuje na implementácii.

4. V poslednej fáze, **prechod**, prebieha hlavné testovanie, integrácia a nasadenie aplikácie.

Iteratívny inkrementálny model má výhodu oproti vodopádovému modelu v tom, že v prípade chyby v niektorej z etáp, túto chybu v následujúcej iterácii môžeme opraviť, napríklad ak pri implementácii zistíme chybu v analýze, čo pri modeli vodopád nie je možné, keďže ten prechádza etapami len raz.

5.2.2 Programovací jazyk

Výber programovacieho jazyka je dôležitá úloha. Zvolený programovací jazyk priamo ovplyvňuje rýchlosť výslednej aplikácie, jej prenositeľnosť a taktiež efektívnosť a rýchlosť vývoja. Existuje mnoho programovacích jazykov, preto si uvedieme niektoré z nich, s ich výhodami a nevýhodami a nakoniec jeden z nich zvolíme za implementačný jazyk našej aplikácie.

C

Jazyk C je procedurálny kompilovaný programovací jazyk, ktorý vyniká svojou rýchlosťou. Pri zachovaní noriem má veľmi dobrú prenositeľnosť medzi rôznymi operačnými systémami. V jazyku C sa dajú tvoriť modulárne aplikácie a existuje preň veľká sada podporných nástrojov pre vývoj aplikácií.

Nevýhoda je malá efektivita tvorby zdrojového kódu aplikácie, pretože jazyk C je procedurálny, a nemôžeme využiť prvky vyššej abstrakcie, ktorú nám ponúkajú objektovo orientované programovacie jazyky. Absencia automatickej správy pamäte dáva síce programátorovi väčšie možnosti, ale manuálna správa pamäte je viac náchylná k chybám pri programovaní. Ďalšou nevýhodou je absencia rozsiahlejších štandardných knižníc, ktoré by zabezpečovali rýchly vývoj a zároveň dobrú prenositeľnosť aplikácie. Je možnosť použiť knižnice tretích strán, ale tieto knižnice často nie sú voľne šíriteľné alebo neposkytujú potrebnú funkcionalitu, alebo prenositeľnosť.

C++

Programovací jazyk C++ je na rozdiel od jazyku C objektovo orientovaný, kompilovaný s väčším počtom štandardných knižníc. Prenositeľnost, podobne ako pri jazyku C, je pri dodržaní noriem na vysokej úrovni. Kvalitných nástrojov podporujúcich vývoj aplikácii v jazyku C++ je tiež mnoho.

Podobne ako pri jazyku C, C++ nemá automatickú správu pamäte. Rozsah štandardných knižníc je oproti jazyku C väčší, avšak voči iným programovacím jazykom stále minimálny. Problémy s knižnicami tretích strán sú obdobné, ako pri jazyku C.

Java

Java je čisto objektovo orientovaný, interpretovaný programovací jazyk. Má obrovskú podporu štandardných knižníc, výbornú prenositeľnosť, a automatickú správu pamäte. Vďaka tomu, že aplikácia sa spúšťa v prostredí virtuálneho stroja (JVM), je program prenositeľný na také platformy, pre ktoré je implementovaný JVM, čo v súčasnosti zahŕňa väčšinu používaných platforiem. Profesionálnych podporných vývojových prostriedkov je taktiež mnoho.

JVM je veľkou výhodou, ale aj nevýhodou. Ďalšou nevýhodou je závislosť na virtuálnom stroji, ktorý musí byť na cieľovej platforme dostupný.

Python

Python je moderný, dynamicky typovaný, objektovo orientovaný, interpretovaný programovací jazyk. Disponuje veľkým počtom štandardných knižníc určených na rôzne účely, pričom s každou verziou pribúdajú ďalšie nové. Podporuje automatickú správu pamäti. Interpret jazyku Python a kritické časti knižníc sú implementované v jazyku C, takže interpretácia je dostatočne rýchla, prípadne náročné výpočty je možné implementovať v C/C++, skompilovať a následne integrovať so zdrojovým kódom jazyku Python. Interpret jazyku je implicitne predinštalovaný na operačných systémoch GNU/Linux, pre ostatné platformy je zdarma k dispozícii inštalačný balíček. Významnou vlastnosťou jazyku Python je vysoká produktivita z hľadiska tvorby programov, čo je spôsobené jednoduchosťou jazyka a množstvom užitočných štandardných knižníc.

Nevýhodou jazyku Python je jeho rýchlosť oproti kompilovaným jazykom ako C/C++. Táto nevýhoda nie je až tak markantná, pretože náročné úlohy sa dajú implementovať v jazyku C/C++ prepojiť s kódom v jazyku Python [75].

Zvolený jazyk

Keďže v dnešnej dobe je výkon počítačov dostatočný a stále rýchlo napreduje, bude hlavné kritérium pri výbere jazyka efektívnosť implementácie, čo znamená množstvo a potrebné štandardné knižnice, automatická správa pamäti, podporné vývojové prostriedky a v neposlednej rade prenositeľnosť.

Z vyššie spomenutých kandidátov sme implementačným jazykom aplikácie zvolili Python. Vďaka mohutnej podpore štandardných knižníc s rôznym zameraním, vysokej produktivite tvorby programov, rýchlosti výslednej aplikácie s možnosťou optimalizácie zvolených modulov pomocou implementácie v kompilovanom jazyku C/C++, rozšírenosti a rastúcej popularite sa vychádza spomedzi ostatných kandidátov ako najlepší.

5.2.3 Knižnice

Dôležitou časťou návrhu je voľba vhodných knižníc, ktoré budeme používať pri implementácii. Keďže sme ako implementačný jazyk zvolili Python, budeme primárne používať štandardné knižnice ktoré jazyk ponúka. Pri použití štandardných knižníc máme istotu dobrej prenositeľnosti aplikácie. Z knižníc budeme používať tieto:

- `difflib` na porovnávanie jednorozmerných dát,
- `PLY, tokenize, keyword` na lexikálnu analýzu zdrojových kódov,
- `string, re` na prácu s reťazcami,
- `argparse` pre spracovanie argumentov príkazového riadku,
- `os, time, path` pre prácu so súbormi,
- `configparser` pre spracovanie konfiguračných súborov.

  `PLY` je knižnica tretej strany, na linuxe dostupná pomocou balíčkovacieho systému, inak voľne stiahniteľna na [13]. Použitie knižníc bude uvedené ďalej v texte v kapitolách 5.4 a 6.1.

5.2.4 Vývojové prostredie

Pri vývoji aplikácií zastáva dôležitú úlohu voľba vývojového prostredia. Existuje široká paleta, od jednoduchých textových editorov so zvýrazňovaním syntaxe, až po komplexné integrované vývojové prostredia (IDE).

Pre potreby programovania postačuje jednoduchý textový editor a interpret jazyka Python. Avšak táto voľba nie je príliš produktívna v porovnaní s IDE, ktoré ponúka širokú škálu podporných nástrojov ako su napríklad:

- debugger,
- proffiler,
- automatické dopĺňanie kódu,
- refaktorizácia,
- zvýrazňovanie syntaxe a výskytov,
- kontrola syntaxe,
- správa zdrojového kódu,
- profily pri ladení a spúštaní aplikácie,
- automatizované testovanie,
- podpora UML,

a mnoho ďalších užitočných nástrojov, ktoré výrazne uľahčujú a urýchľujú vývoj aplikácii.

Vývojových prostredí je mnohom niektoré sú komerčné s platenou licenciou, ale existuje niekoľko kvalitných a voľne dostupných. Pre potreby vývoja našej aplikácie sme zvolili modulárne vývojové prostredie Eclipse, s doplnkom PyDev. ktoré obsahuje všetky vyššie uvedené nástroje.

5.2.5 Správa zdrojového kódu

Správa zdrojového kódu (SCM) zabezpečuje predovšetkým uchovávanie histórie vykonaných zmien v zdrojovom kóde a v iných pomocných súboroch. SCM eviduje kto, kedy a akým spôsobom zmenil časť súboru. To pomáha udržiavať prehľad zmien, možnosť zistiť stav súboru v minulosti a tiež vrátiť sa k niektorej z predchádzajúcich verzií, v prípade že pri vývoji došlo k chybám. Druhým, rovnako dôležitým faktorom SCM je možnosť spolupráce viacerých programátorov na projekte, pretože SCM strážia a riešia prípadné konflikty.

Udržiavať stav projektu pomocou SCM má viacere výhody, ako bolo uvedené vyššie. Naša aplikácia má byť modulárna, na každom z modulov môže pracovať iný vývojár, preto je použitie SCM nutnosť. Existuje niekoľko systémov SCM, v základe centralizované (CVS, SVN), alebo decentralizované (Git,Bazaar,Mercurial).

Pre potreby vývoja našej aplikácie sme zvolili SCM distribuovaný systém Git. Má viaceré výhody oproti centralizovaným, ale je aj zložitejší na počiatočné pochopenie a naučenie, hlavne pre vývojárov pracujúcich s centralizovaným SCM.

5.3 Analýza problému

Podľa špecifikácie 5.1, budeme navrhovať a implementovať modul, porovnávajúci zdrojové kódy.

Pri porovnávaní textových súborov sme chápali text, ako zoznam riadkov. Ak sme porovnávali dva súbory, porovnávali sme ich riadok po riadku, a v prípade zmeny, sme zobrazili celý riadok. Pri zdrojovom kóde je situácia zložitejšia. Zdrojový kód je postupnosť príkazov zapísaných v určitom programovacom jazyku. Daný jazyk definuje pre každý príkaz určitú syntax a sémantiku.

5.3.1 Typy podobností zdrojového kódu

V kapitole 2.3.2 sme si uviedli dva základné typy podobností zdrojového kódu:

Textová podobnosť , kde dva fragmenty kódu sú textovo odlišné, napríklad zmenou bielych znakov, názvov identifikátorov alebo zmenou príkazov.

Funkčná podobnosť , kde dva fragmenty kódu sú textovo odlišné ale sémanticky rovnaké, napríklad výpočet faktoriálu iteratívnou a rekurzívnou metódou.

Rozpoznanie funkčnej podobnosti

Ak by sme chceli porovnávať zdrojové kódy na základe funkčnej podobnosti, museli by sme pre každý nájdený textový rozdiel pomocou určitej verifikačnej metódy vyhodnotiť tieto dva fragmenty kódu za rovnaké, resp. rozdielne.

Verifikačné metódy by sme mohli zaradiť do nasledovných kategórií [78]:

1. **Model checking**, algoritmický spôsob zistenia, či daný systém spĺňuje dané vlastnosti pomocou systematického prehľadávania stavového priestoru daného systému. Systém môže byť reálny, alebo model, vlastnosti sú špecifikované pomocou temporálnej logiky (LTL,CLT,CTL*,...) alebo iným spôsobom. Veľkou nevýhodou je problém s explóziou stavov.

   Verifikačné nástroje, založené na tomto princípe sú napríklad Spin, SMV, RuleBase (IBM), Incisive Formal Analysis (Cadence), Blast, Magic, Copper, JPF (NASA), SLAM/SDV (Microsoft), 0-In (Mentor Graphics), Magellan (Synopsys), Forte (Intel), NEC, ...

2. **Statická analýza** sa pokúša predísť vykonávaniu skúmaného systému a namiesto toho analyzuje jeho zdrojový kód a zhromažďuje, typicky aproximované, informácie o systéme. Formy statickej analýzy sú napr. typová analýza, analýza toku dát, abstraktná interpretácia. Nevýhodou je produkcia falošných výsledkov, ktoré treba inak overiť.

   Verifikačné nástroje, založené na tomto princípe sú napríklad Coverity, KlocWork, PolySpace, CodeSonar, Purify, Lint, Astrée, FindBugs, Sparse, PREfix/PREfast, AbsInt, ESC, Invader, ...

3. **Dokazovanie teorémou**, deduktívna verifikácia, často podobná klasickému matematickému spôsobu dokazovania teorémov z počiatočných axiómov.

   Verifikačné nástroje, založené na tomto princípe sú napríklad PVS, Coq, Hol, Isabelle, ACL2, Forte, ...

Problém vo vyššie uvedenom postupe spočíva v tom, že nástroje formálnej verifikácie nie sú úplne automatizované. Potrebujú vytvoriť model, ktorý budú skúmať, stanoviť podmienky pred a po vykonaní daného fragmentu alebo stanoviť teorém, ktorý budú dokazovať. To by vyžadovalo zložitú interakciu, s aplikáciou na porovnávanie, ktorá by zároveň bola rádovo dlhšia, ako vyšetrenie podobnosti človekom.

Kvôli uvedeným vlastnostiam nebudeme pri porovnávaní zdrojových kódov detekovať funkčnú podobnosť, ale sústredíme sa na textovú podobnosť.

Rozpoznanie textovej podobnosti

Pri textovej podobnosti rozlišujeme tri typy:

Typ I: Identické fragmenty kódu, až na zmeny bielych znakov a komentárov.

Typ II: Štrukturálne/syntakticky identické fragmenty, až na zmeny identifikátorov, literálov, typov, usporiadania a komentárov.

Typ III: Skopírované fragmenty so zmenami. Zámena príkazov, pridanie alebo zmazanie a naviac zmeny identifikátorov, literálov, typov, usporiadania a komentárov

Aby sme mohli detekovať takéto zmeny, musíme porovnávaný zdrojový kód rozdeliť na časti, medzi ktorými chceme detekovať zmenu. Z výššie uvedených typov to sú:

- biele znaky,
- komentáre,
- identifikátory,
- literály (celočíselné, desatinné, znakové a reťazcové),
- typy.

Takáto transformácia sa nazýva lexikálna analýza, popísaná v kapitole 3.1. Lexikálna analýza, na základe definovanej gramatiky jazyka rozdelí vstupný zdrojový kód na zoznam tokenov. Takto získané zoznamy tokenov porovnáme, pričom môžeme ignorovať zmeny pri určitých typoch tokenov (napr. komentáre, biele znaky, ale aj literály a identifikátory). Po porovnaní musíme výstup, zoznam zmien, transformovať do výstupného formátu, ktorý nakoniec zobrazíme.

5.3.2 Algoritmus

Konceptuálny algoritmus je uvedený v kapitole 2.3.3. Konkrétna verziu algoritmu, s konkrétnymi akciami v jednotlivých fázach:

1. **Preprocessing:** Načítanie zdrojových kódov zo súborov do textových reťazcov, na základe kódovania. V prípade niektorých programovacích jazykov je možné odstrániť nepodstatné časti.
2. **Transformácia:** Na transformáciu použijeme lexikálnu analýzu, ktorá nám zdrojový kód rozdelí na reťazec tokenov. Pri transformácii si musíme uchovať informáciu o pozícii každého tokenu v zdrojovom kóde, pre spätnú lokalizáciu zmien v konkrétnom zdrojovom kóde.

3. **Porovnávanie:** Dva reťazce tokenov porovnáme pomocou algoritmu na nájdenie najdlhšieho spoločného podreťazca, popísaného v kapitole 2.1.2. Výstupom z tejto fáze je zoznam indexov spoločných podzoznamov oboch zoznamov tokenov.

4. **Formátovanie:** Táto fáza pozostáva z dvoch častí. Najskôr zoznam indexov z predchádzajúcej fázy porovnáme s oboma zoznamami tokenov 5.3.2. Dostaneme tak zoznam indexov zmien a rovnakých častí. Následne indexy, ktoré ukazujú do zoznamov tokenov transformujeme na indexy do pôvodných zdrojových súborov 5.3.2.

5. **Postprocessing:** Pomocou získaného zoznamu indexov zmien vizualizujeme konkrétne zmeny v špecifickom formáte. Kvôli zachovaniu kompatibility s pôvodným programom `diff`, budú zachované štandardné formáty – normálny formát výstupu, unifikovaný formát výstupu a kontextový formát výstupu. Tieto formáty sú popísané v kapitole 4.5.1.

Získanie zoznamu zmien

Zmena je definovaná typom zmeny, rozsahom v prvom zozname a rozsahom v druhom zozname. Typ zmeny môže byť následovný:

- Zámena – jeden podzoznam tokenov v prvom zozname je zamenený za iný podzoznam tokenov v druhom zozname.
- Vloženie – podzoznam tokenov v druhom zozname je vložený za konkrétny token v prvom zozname.
- Zmazanie – za konkrétnym tokenom v druhom zozname chýba podzoznam tokenov z prvého zoznamu.
- Zhoda – dva podreťazce tokenov sú zhodné.

V prípade vloženia, resp. zmazania je rozsah v prvom, resp. druhom zozname nulový.

Pri porovnávaní zoznamov, ak zmena pozostáva iba z podzoznamu tokenov, ktoré majú byť ignorované, prehlásime dva podzoznamy za zhodné. Takéto správanie vykazuje aj program `diff`. Nevýhodou je, že ak podzoznam pozostáva z ignorovaných tokenov a jedného tokenu, ktorý nie je ignorovaný, celý podzoznam prehlásime za zmenu.

Pretože v pôvodnom zozname zmien boli dve zmeny vždy oddelené jednou zhodou, v prípade prehlásenia zhody podzoznamov by sa v zozname vyskytli dve až tri zhody po sebe. Tieto zhody budú zlúčené do jednej pre zachovanie konzistencie zoznamu zmien.

Transformácia indexov

Program `diff` je riadkovo orientovaný – text porovnáva po riadkoch a zmeny zobrazuje ako rozsahy riadkov, ktoré sú zmenené. V našom prípade sa indexy v zozname zmien vzťahujú na jednotlivé lexémy v zdrojovom kóde. Z toho vyplýva, že na jednom riadku môže byť viac druhov zmien. Napríklad, ak porovnávame fragment kódu

```
int i = 0, j;
```

s fragmentom

```
int i = 1;
```

dostávame po porovnaní výsledok, že na danom riadku je celočíselný literál `0` zamenený za `1` a zmazané lexémy `,` a `j`.

Aby sme zabezpečili kompatibilitu s programom `diff` a dodržali jeho formáty výstupu, ktoré sú riadkovo orientované, musíme transformovať zoznam zmien s indexmi ukazujúcimi na lexémy, na zoznam zmien s indexmi ukazujúcimi na riadky. Riadok v predchádzajúcom príklade by potom bol označený za zmenený

Postupne prechádzame zoznam zmien lexémov, pre každú zmenu môže nastať jeden alebo viac z troch prípadov:

1. Začiatočný riadok aktuálnej zmeny sa kryje s koncovým riadkom predchádzajúcej zmeny. Prekrytie je v prvom súbore a nejedná sa o typ zmeny vloženie. Ak je typ predchádzajúcej zmeny zhoda, zmenšíme jej rozsah v prvom súbore o jeden riadok. V opačnom prípade je aktuálny typ zmeny zhoda. Potom posunieme začiatok rozsahu v prvom súbore o jeden riadok.

2. Začiatočný riadok aktuálnej zmeny sa kryje s koncovým riadkom predchádzajúcej zmeny. Prekrytie je v druhom súbore a nejedná sa o typ zmeny zmazanie. Ak je typ predchádzajúcej zmeny zhoda, zmenšíme jej rozsah v druhom súbore o jeden riadok. V opačnom prípade je aktuálny typ zmeny zhoda. Potom posunieme začiatok rozsahu v druhom súbore o jeden riadok.

3. Ak je typ predchádzajúcej zmeny zhoda a jeden z rozsahov tejto zmeny je nulový, potom túto zhodu odstránime a zlúčime aktuálnu zmenu s predposlednou.

   (a) Ak typ aktuálnej aj predposlednej zmeny je zmazanie, zlúčime tieto rozsahy.

   (b) Alebo, ak typ aktuálnej aj predposlednej zmeny je vloženie, zlúčime tieto rozsahy.

   (c) Alebo zlúčime rozsahy a typ zmeny nastavíme na zámenu.

Tento postup vyžaduje valídny zoznam zmien, t.j. zmeny typu vloženie, zmazanie alebo zámena sú vždy oddelené zmenou typu zhoda.

5.3.3 Ignorovanie zmien

Hlavnou výhodou, oproti klasickému textovému porovnávaniu, je to, že dokážeme ignorovať zmeny pozostávajúce iba z určitých typov tokenov.

Pretože dokážeme rozpoznať jednotlivé typy tokenov, dokážeme selektívne porovnávať zdrojový kód. Možnosti sú obmedzené iba počtom typov tokenov.

Typickým prípadom môže byť, ak chceme ignorovať zmeny v názvoch identifikátorov. Potom dva fragmenty kódu, ktoré sú zhodné až na názvy identifikátorov by klasické porovnávanie prehlásilo za rozdielne, avšak pri lexikálnom porovnávaní, môžeme práve zmeny v tokenoch typu identifikátor ignorovať. Potom lexikálne porovnanie prehlási dva fragmenty za zhodné.

5.4 Návrh riešenia

Podľa špecifikácie, aplikácia má porovnávať viac typov súborov. Preto aplikáciu rozdelíme na dve časti:

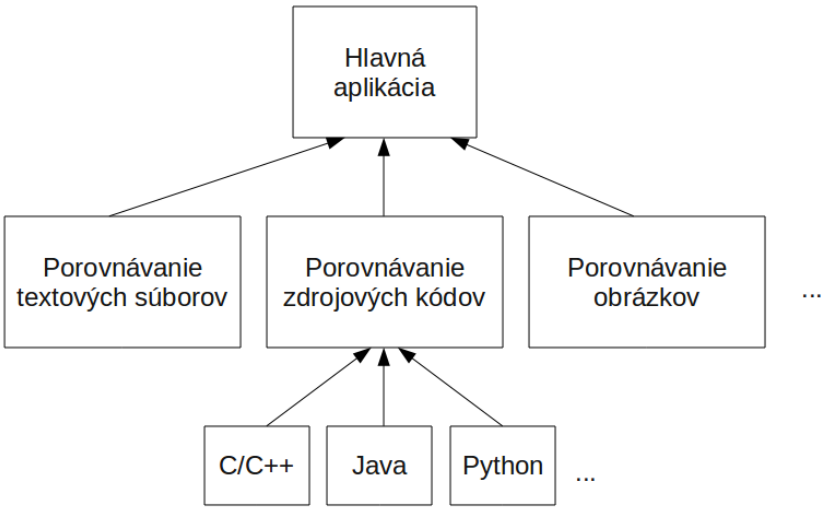
- hlavná aplikácia,

- moduly implementujúce porovnávanie.

Funkcie a popis týchto častí si uvedieme nižšie. Následne definujeme rozhranie a navrhneme diagram tried.

5.4.1 Architektúra aplikácie

Výsledná aplikácia porovnáva rôzne typy dokumentov. Pozostáva z modulov, kde porovnávanie jedného typu dokumentov je implementované v jednom module. Modul môže ďalej obsahovať submoduly. Architektúra je ilustrovaná na obrázku 5.1.



Obrázek 5.1: Architektúra aplikácie

Hlavná aplikácia

Hlavná aplikácia spracuje parametre príkazového riadku a na základe typu súboru (alebo prepínaču) vyberie daný modul, ktorý prevedie porovnávanie. Typy súborov sú rozpoznávané na základe koncoviek, Každý modul definuje, ktoré typy súborov porovnáva a s akou prioritou. Napríklad zdrojový kód sa dá porovnať ako text, ale aj ako zdrojový kód. Priorita definuje, ktorý modul sa implicitne použije.

Voľby prepínačov hlavnej aplikácie sú uvedené v prílohe **??**.

Modul

Každý modul implementuje rovnaké rozhranie. Modul má konfiguračný súbor, pomenovaný rovnako ako modul, vo formáte INI [83]. Tento súbor povinne obsahuje sekcie:

```
# general parameters of this module to be processed in mediadiff
[mediadiff]
extensions=py,h,hh,hpp,hxx,h++,c,cc,cpp,cxx,c++,java
```

```
[priorities]
# priority of each extension, default is the lowest possible
py=100000
c=100000
h=100000
```

Sekcia `[mediadiff]` obsahuje hodnotu `extensions`, ktorá definuje koncovky súborov, ktoré porovnáva daný modul porovnáva. V tejto sekcii môžu v budúcnosti pribudnúť ďalšie potrebné hodnoty. Sekcia `[priorities]` následne obsahuje všetky koncovky uvedené v predchádzajúcej časti, a ku každej koncovke definuje prioritu. V prípade, že viac modulov definuje rovnakú koncovku, porovnávanie bude vykonávať ten modul, ktorý má najnižšiu prioritu.

V prípade potreby samostatného porovnávania daného typu dokumentov, bude modul samostatne spustiteľný, sám porovná zdrojové kódy a zobrazí rozdiel.

Modul môže byť ďalej členený na submoduly, ktoré implementujú funkcionalitu pre daný, konkrétny typ súboru. Napríklad pri porovnávaní video súborov, bude modul implementovať porovnávací algoritmus a submoduly budú načítať a spracovávať vstup do potrebnej, štandardnej podoby.

Modul si sám definuje argumenty a prepínače príkazového riadku. Na toto je použitá knižnica, `argparse`, ktorá umožňuje vybudovať hlavný parser príkazového riadku z subparsrov. Modul si takýto subparser vytvorí a predá ho hlavnej aplikácii. Každý modul bude akceptovať minimálne tieto argumenty:

- Jeden argument, špecifikujúci prvý porovnávaný súbor. Ako druhý súbor sa použije štandardný vstup.
- Dva argumenty, špecifikujúce súbory, ktoré sa budú porovnávať.
- Nápoveda pre konkrétny modul, kde budú uvedené všetky prepínače s ich vysvetlením.

Hlavná aplikácia spracuje parametre príkazového riadku, zvolí požadovaný modul, a predá mu zvyšné spracované parametre. Voľby pre jednotlivé moduly sú v prílohe F

5.4.2 Porovnávanie zdrojových kódov

Podľa špecifikácie 5.1, budeme navrhovať a implementovať modul porovnávajúci zdrojové kódy.

V analýze sme si uviedli algoritmus, podľa ktorého budeme postupovať. Jeho hlavné kroky sú transformácia zdrojového kódu na zoznam tokenov pomocou lexikálnej analýzy, následne porovnanie dvoch takýchto zoznamov a v poslednej fáze, formátovanie a zobrazenie zmien. Modul bude preto pozostávať z týchto troch hlavných častí.

Porovnávanie nie je obmedzené len na zdrojové kódy programovacích jazykov. Môžeme porovnávať akékoľvek jazyky, pre ktoré definujeme gramatiku a implementujeme špecifický submodul lexikálnej analýzy.

Lexikálna analýza

Aby sme dokázali porovnávať rôzne programovacie jazyky, budeme implementovať lexikálny analyzátor pre každý programovací jazyk ako samostatný submodul. Takýto systém nám umožní ľahké rozširovanie celého modulu o porovnávanie ďalšieho programovacieho jazyka jednoduchým pridaním nového submodulu.

Submodul implementuje rozhranie `LexicalParser`, ktoré poskytuje jedinú metódu `GetTokenList()`, ktorá vráti zoznam tokenov. Druhé potrebné rozhranie je `IgnoreTokens`, ktoré poskytuje metódu `Ignore()`. Táto metóda príma token, a podľa jeho typu a nastavenia, ktoré tokeny sa majú ignorovať vracia true, alebo false.

Lexikálny analyzátor budeme implementovať pomocou knižnice PLY. Popis použitia je v kapitole 3.2.2. Implementovať budeme lexikálne analyzátory pre jazyky C/C++, Java a Python (6.1.3).

Porovnávanie

Hlavný modul implementuje porovnávanie zoznamov tokenov pomocou triedy `LexicalMatcher` (6.1.2). Na porovnávanie dvoch zoznamov tokenov použijeme knižnicu `difflib`. Z nej ale využijeme len metódu `get_matching_blocks()` triedy `SequenceMatcher`, ktorá nám vráti zoznam ekvivalentných podzoznamov tokenov. Metóda porovnáva dva zoznamy tokenov pomocou algoritmu Ratcliff/Obershelp popísaného v kapitole 2.1.2. Keďže indexy v zozname ukazujú do zoznamu tokenov, musíme ďalšie transformačné metódy implementovať sami. Algoritmus transformácie indexov sme uviedli v analýze.

Formátovanie

Výstupné formáty budú rovnaké ako v programe `diff` – normálny, kontextový a unifikovaný (4.5). Pre formátovanie využijeme zdrojový kód z knižnice `difflib`, ktorý upravíme pre naše potreby. Rozhranie bude implementovať trieda `SemanticDiff` (6.1.2).

5.4.3 Definícia gramatík

Aby sme mohli implementovať lexikálny analyzátor, musíme mať definovanú gramatiku. V následujúcom texte popíšeme lexikálnu štruktúru jazykov C/C++ a Java. Jazyk Python ponúka štandardnú knižnicu `tokenize`, ktorá implementuje lexikálny analyzátor. Pre naše potreby použijeme túto knižnicu.

C/C++

Lexikálna štruktúra jazykov C a C++ je veľmi podobná, a väčšinou program napísaný v jazyku C je zároveň programom v C++. Preto budeme definovať gramatiku pre jazyk C++. To nám umožní pomocou jedného lexikálneho analyzátoru spracovávať oba jazyky. Definíciu gramatiky [74] nájdeme v prílohe A na CD. Typy tokenov sú:

- **Kľúčové slová**, zoznam slov definovaný v norme.
- **Identifikátory**, definované regulárnym výrazom (r.v.) `[A-Za-z_][\w_]*` s kontrolou, či sa nejedná o kľúčové slovo.
- **Celočíselné konštatnty** môžu byť v decimálne, hexadecimálne alebo oktalové oktalové. Definované r.v. `((0[xX][0-9ABCDEFabcdef]+)|(0[01234567]+)|(\d+))[uUlL]*`
- **Desatinné konštanty**, skladjú sa z celočíselnej časti, desatinnej bodky, desatinnej časti a exponentu. Celočíselná alebo desatinná časť môže chýbať, ale nikdy nie obe. Takisto exponent môže chýbať. Regulárny výraz definujúci desatinné konštanty

`(\d+\.\d*([eE][\+-]?\d+)?[fFlL]?) | (\d*\.\d+([eE][\+-]?\d+)?[fFlL]?) | (\d+[eE][\+-]?\d+[fFlL]?) | (\d+([eE][\+-]?\d+)?[fFlL])`

- **Znakové literály**, regulárny výraz `\'(\\.|[^\\\'])\'`
- **Reťazcove literály**, regulárny výraz`\"((\\.)|[^\\"])*\"`
- **Komentáre** Môžu byť jednoriadkové `//.*\n` alebo viacriadkové `/\*(.|\n)*?\*/`
- **Direktívy preprocesoru**, r.v. `\#.*\n`
- **Operátory**, aritmetické, porovnávacie. Definované v norme.
- **Oddelovače**, zátvorky a interpunkcia. Definované v norme.

Java

Definíciu lexikálnej štruktúry jazyka Java som čerpal z [27]. Gramatika je uvedená v prílohe na A CD. Typy tokenov sú:

- **Kľúčové slová**, zoznam slov definovaný v norme.
- **Identifikátory**, r.v. rovnako ako v C++.
- **Celočíselné konštatnty** , r.v. rovnako ako v C++, rozdiel v prípone.
- **Desatinné konštanty**, r.v. rovnako ako v C++, rozdiel v prípone.
- **Znakové literály**, r.v. rovnako ako v C++.
- **Reťazcove literály**, r.v. rovnako ako v C++.
- **Komentáre**, r.v. rovnako ako v C++.
- **Anotácie**, r.v. `@\w+(\([^\)]*\))?`
- **Operátory**, aritmetické, porovnávacie. Definované v norme.
- **Oddelovače**, zátvorky a interpunkcia. Definované v norme.

5.4.4 Voľby porovnávania

Výhodou pri porovnávaní zoznamov tokenov je to, že zmeny pozostávajúce z určitých špecifikovaných typov tokenov môžeme ignorovať. Aby nebol modul príliš náročný na nastavenie volieb porovnávania, definujeme voľby, ktoré sú spoločné pre väčšinu programovacích jazykov. Ignorovať môžeme zmeny

- čísel – celočíselných alebo desatinných,
- reťazcov alebo znakových literálov,
- operátorov,
- identifikátorov,
- kľúčových slov,
- komentárov.

Kapitola 6

Implementácia a testovanie

Na základe analýzy problému a návrhu aplikácie môžeme pristúpiť k samotnej implementácii. V kapitole 5.4.1 sme si navrhli základnú architektúru aplikácie, ktorá pozostáva z hlavnej aplikácie a modulov, ktoré implementujú porovnávanie konkrétneho typu súboru.

6.1 Implementácia

V tejto časti popíšeme základné prvky implementovanej aplikácie, ich rozhranie a funkcionalitu. Každá trieda má popísané základné metódy a je zobrazená pomocou UML class diagramu. Programová dokumentácia – konkrétnejší popis tried, metód a parametrov je uvedený v prílohe A na CD.

6.1.1 Hlavná aplikácia mediadiff

Hlavná aplikácia je jednoduchá, obsahuje iba funkciu `main()`, ktorá spracuje parametre príkazového riadku a následne zavolá metódu `run()` príslušného modulu. Aplikácia si dynamicky načíta moduly, ktoré sú umiestnené v zložke `./modules`. Moduly sú v separátnych zložkách. Každý modul obsahuje obsahuje minimálne dva súbory:

- Zdrojový kód modulu. Súbor s rovnakým názvom ako modul (zložka) s koncovkou „py". Implementuje funkciu run(parameters), ktorá príjma spracované parametre príkazového riadku. Druhou povinou funkciou je GetArgParser(), ktorý vráti subparser príkazového riadku pre daný modul.
- Konfiguračný súbor. Súbor má rovnaký názov ako modul, s koncovkou „config". Formát a povinné položky sú popísané v 5.4.1.

Po načítaní modulov prejde jednotlivé konfiguračné súbory modulov a zostaví si tabuľku koncoviek s modulmi, ktoré ich spracovávajú. V prípade ak daný typ súboru spracovávajú viaceré moduly, vyberie sa podľa priority prvý z nich.

V ďalšom texte si popíšeme modul, ktorý porovnáva zdrojové kódy a jeho submoduly pre jednotlivé programovacie jazyky.

6.1.2 Modul SemanticDiff

Tento modul je hlavným modulom pre porovnávanie zdrojových kódov. Obsahuje dve triedy:

- `LexicalMatcher`, trieda ktorá porovnáva dva zdrojové kódy. Poskytuje metódy pre získanie zoznamu zmien.
- `SemanticDiff`, trieda, ktorá zobrazuje zoznam zmien. Poskytuje metódy pre rôzne formáty výstupu, definované v 4.5.1.

Okrem týchto hlavných tried, obsahuje funkciu `main()`, ktorá je zavolaná pri spustení modulu ako samostatného programu. Po spustení spracuje argumenty príkazového riadku a porovná zadané súbory so zdrojovými kódmi. Výstup vypíše v špecifikovanom formáte.

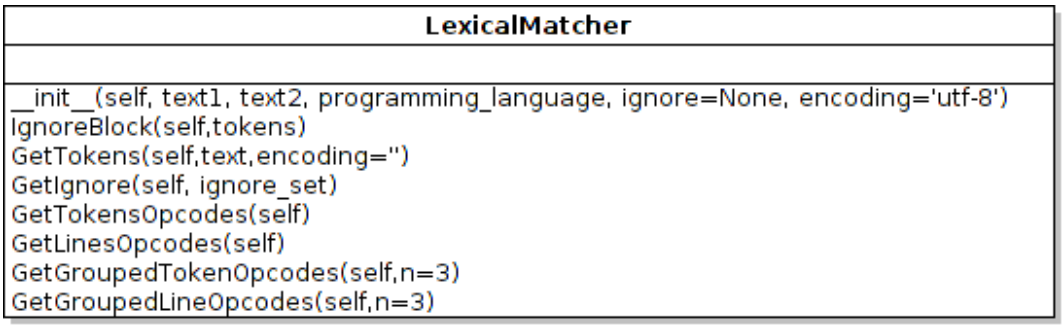
Ďalšia pomocná funkcia je `DetectLanguage()`, ktorá na základe konfiguračného súboru modulu a názvu súboru so zdrojovým kódom určí programovací jazyk.

Poslednou pomocnou funkciou je `GetArgParser()`, ktorá vytvorí subparser príkazového riadku so špecifickými prepínačmi pre modul. Z tohto a ďalších subparsrov bude zostavený hlavný parser aplikácie.

LexicalMatcher

Trieda `LexicalMatcher` pomocou konkrétneho lexikálneho analyzátora získa zoznam tokenov. Takto získané zoznamy tokenov sú porovnané pomocou knižnice `difflib` a pomocou metód `GetLinesOpcodes()` resp. `GetLinesOpcodes()` je možné získať zoznam zmien, vzťahujúci sa priamo na riadky v zdrojovom texte. Pomocou metód `GetTokensOpcodes()` alebo `GetGroupedTokenOpcodes()` je možné získať zoznam zmien s indexami priamo do zoznamu tokenov, ktorý sa dá získať metódou `GetTokens()`.

Ak je v konštruktori triedy definovaný objekt `Ignore`, je možné, podľa nastavenia daného objektu ignorovať zmeny pozostávajúce z tokenov určitých typov. Napríklad, môžeme ignorovať zmeny literálov, názvov identifikátorov alebo komentárov.



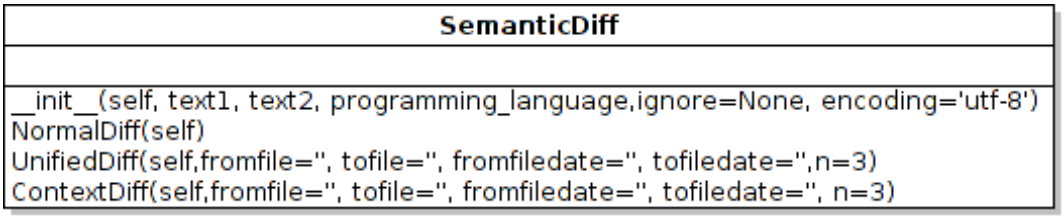
Obrázek 6.1: Trieda LexicalMatcher

SemanticDiff

Trieda `SemanticDiff` formátuje zoznam zmien do textovej podoby. Formáty výstupu sú rovnaké ako v programe `diff`. Špecifikácia týchto formátov je v kapitole 4.5.1. Zdrojový kód týchto metód bol prevzatý z knižnice `difflib` a mierne upravený pre naše potreby.

6.1.3 Submoduly Cpp, Java a Python

Submoduly Cpp, Java a Python implementujú samotnú lexikálnu analýzu. Každý submodul poskytuje rovnaké rozhranie definované triedami `LexicalParser` a `IgnoreTokens`.



Obrázek 6.2: Trieda SemanticDiff

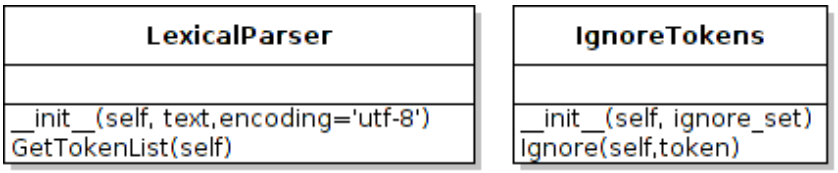
LexicalParser

Lexikálna analýza je implementovaná pomocou triedy `LexicalParser`. V konštruktore je nastavený text zdrojového kódu a pomocou metódu `GetTokenList()` sa prevedie lexikálna analýza a vráti sa zoznam tokenov – objekt triedy `TokenList`.

Lexikálna anlýza je implementovaná pomocou knižnice PLY. V súbore konkrétneho lexikálneho analyzátora pre daný jazyk musí byť definovaná gramatika jazyka. Spôsob definície jazyka je uvedený v kapitole 3.2.2. Samotné gramatiky sú popísané v kapitole 5.4.3.

IgnoreTokens

Pretože každý programovací jazyk má rôzne typy tokenov, je potrebné, aby v module bola implementovaná aj trieda `IgnoreTokens`. Objekt tejto triedy poskytuje metódu `Ignore()`, ktorá prijíma objekt triedy `Token` a na základe nastavení, ktoré tokeny má ignorovať, vracia táto metóda true alebo false.



Obrázek 6.3: Triedy LexicalParser a IgnoreTokens

6.1.4 Pomocé triedy

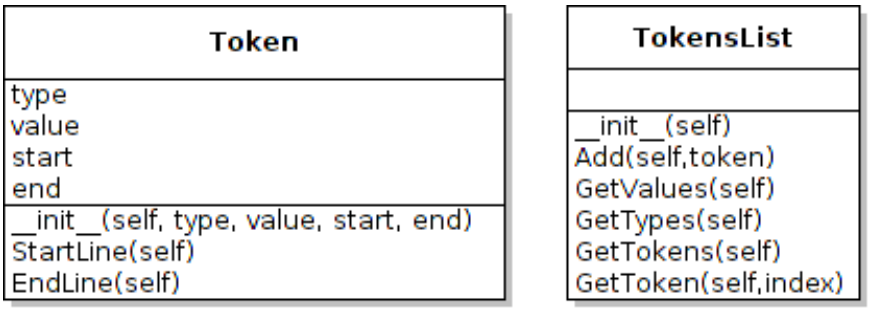
Modul Token implementuje dátov štruktúru reprezentujúcu token a pomocnú štruktúru pre udržovanie zoznamu tokenov. Obsahuje dve triedy, `Token` a `TokensList`. Modul Utils obsahuje pomocné triedy a funkcie.

Token

Trieda `Token` je jednoduchá dátová štruktúra, obsahujúca typ tokenu, zodpovedajúci lexém a pozíciu tohto lexému v zdrojovom kóde. Pozícia je definovaná dvoma položkami – začiatok a koniec, pričom každá položka pozostáva z riadku a stĺpcu (znaku v riadku).

TokensList

Trieda `TokensList` zapúzdruje zoznam tokenov a ponúka špecifické prístupové metódy k tomuto zoznamu. Môžeme získať zoznam typov tokenov, zoznam hodnôt tokenov, alebo jeden token, špecifikovaný indexom.



Obrázek 6.4: Triedy Token a TokensList

Properties

Trieda `Properties` pomocou knižnice `configparser` načíta informácie z konfiguračného súboru. Pomocou metódy `Parameters()` získame implicitné parametre, v našom prípade zoznam podporovaných jazykov a implicitné kódovanie textu zdrojových súborov. Metóda `Languages` vráti zoznam identifikátorov jazykov a metóda `LanguagesExtensions` vráti zoznam koncoviek asociovaných s konkrétnym programovacím jazykom.

Konfiguračný súbor

Konfiguračný súbor pre `SemanticDiff` obsahuje štyri položky:

1. Položka `mediadiff` je povinná pre každý modul, a obsahuje zoznam koncoviek súborov, ktoré modul porovnáva.
2. Položka `priorities` je povinná pre každý modul, a obsahuje koncovky z predchádzajúceho zoznamu s priradenými prioritami, kde menšie číslo má väčšiu prioritu.
3. Položka `languages`, špecifická pre modul `SemanticDiff` obsahuje koncovky zdrojových kódov programovacích jazykov, s priradeným programovacím jazykom – submodulom, ktorý implementuje lexikálnu analýzu.
4. Posledná položka `parameters` obsahuje parametre pre modul. Konkrétne je implicitné kódovanie textu v zdrojových súboroch, a podporované programovacie jazyky.

Príklad konfiguračného súboru je v prílohe B.

6.1.5 Distribúcia a inštalácia

Distribúcia a inštalácia aplikácie prebieha v réžii knižnice `distutils`. Táto štandardná knižnica jazyku Python ponúka množstvo volieb a pritom je veľmi jednoduchá na použitie.

V inštalačnom skripte `setup.py` definujeme funkciu `setup`, ktorá príjma rôzne argumenty ako napr. meno aplikácie, verziu, autora, moduly a ostatné súbory potrebné k inštalácii. Samotná inštalácia prebieha veľmi jednoducho, stačí zadať príkaz `python setup.py install`. V prípade potreby, môžeme jednoducho vytvoriť archív k distribúcii, alebo priamo inštalačný balíček pre rôzne platformy ako napr. exe pre windows, rpm pre fedoru, deb pre ubuntu a iné.

6.2 Testy

V tejto kapitole uvedieme príklady zdrojových kódov s určitými odlišnosťami, ktoré porovnáme. Najskôr oba zdrojové kódy porovnáme originálnym programom `diff` a následne našou aplikáciou `mediadiff`. Ukážeme si aj aplikáciu niektorých volieb pri porovnávaní.

6.2.1 Testovacie dáta

Testovacie dáta pozostávajú z dvoch zdrojových kódov jazyku C – E.3 a E.4.

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5    int n, sum = 1;
6    printf("Please enter a number\n");
7
8    /*
9     * Iterative count factorial
10    */
11    for(int i = 1; i <= n; i++)
12      sum *= i;
13
14   printf("Factorial of %d is %d\n",n,sum);
15
16   return 0;
17 }
```

Algoritmus 6.1: Zdrojový kód `a.c`

Ako je vidieť, v oboch príkladoch sa počíta faktoriál. Avšak medzi ukážkami sú zmeny – názvy identifikátorov, číselne a reťazcové literály, komentáre a operátory.

V tomto príklade bol ako názorná ukážka zvolený jazyk C, avšak porovnávať môžeme aj zdrojové kódy v C++, Jave alebo Pythone.

6.2.2 Výsledky testov

Porovnanie programom `diff`

Výsledok porovnania ukážkových zdrojových kódov E.3 a E.4 originálnym programom `diff`:

```
$ diff ./data/a.c ./data/b.c
5,6c5,6
```

```c
#include <stdio.h>

int main(void)
{
  int number, sum = 1;
  printf("Type a number\n");

  //iterative count factorial
  for(int i = 1; i <= number; ++i)
    sum *= i;

  //print result
  printf("%d! = %d\n",number,sum);

  return 1;
}
```

Algoritmus 6.2: Zdrojový kód `b.c`

```
<   int n, sum = 1;
<   printf("Please enter a number\n");
---
>   int number, sum = 1;
>   printf("Type a number\n");
8,12c8,10
<   /*
<    * Iterative count factorial
<    */
<    for(int i = 1; i <= n; i++)
<      sum *= i;
---
>   //iterative count factorial
>   for(int i = 1; i <= number; ++i)
>     sum *= i;
14c12,13
<   printf("Factorial of %d is %d\n",n,sum);
---
>   //print result
>   printf("%d! = %d\n",number,sum);
```

Ako je vidieť, program porovnal riadky oboch súborov a našiel zmeny.

Porovnanie programom `mediadiff`

Pri porovnaní ukážkových zdrojových kódov E.3 a E.4 implementovaným programom `mediadiff` bez použitia špeciálnych volieb:

```
$ ./SemanticDiff.py ./data/a.c ./data/b.c
5,11c5,9
<   int n, sum = 1;
```

```
<   printf("Please enter a number\n");
<
<   /*
<    * Iterative count factorial
<    */
<    for(int i = 1; i <= n; i++)
---
>   int number, sum = 1;
>   printf("Type a number\n");
>
>   //iterative count factorial
>   for(int i = 1; i <= number; ++i)
14c12,12
<   printf("Factorial of %d is %d\n",n,sum);
---
>   //print result
>   printf("%d! = %d\n",number,sum);
16c15
<   return 0;
---
>   return 1;
```

Výsledok je správny, ale odlišný od predchádzajúceho výsledku. Je to spôsobené tým, že pri lexikálnej analýze ignorujeme biele miesto v kóde.

Keď však porovnáme súbory s ignorovaním názvov identifikátorov, číselných a reťazcových literálov, oba súbory prehlási za zhodné – nenájde žiadny rozdiel.

```
$ ./SemanticDiff.py --ignore-identifiers --ignore-strings --ignore-numbers
--ignore-comments --ignore-operators ./data/a.c ./data/b.c
```

Kôli prehľadnosti budeme ignorovať vždy len jeden typ tokenov.

Porovnanie na základe názvov identifikátorov

```
$ ./SemanticDiff.py --ignore-strings --ignore-numbers --ignore-comments
--ignore-operators ./data/a.c ./data/b.c
5c5
<   int n, sum = 1;
---
>   int number, sum = 1;
11c9
<    for(int i = 1; i <= n; i++)
---
>   for(int i = 1; i <= number; ++i)
14c13
<   printf("Factorial of %d is %d\n",n,sum);
---
>   printf("%d! = %d\n",number,sum);
```

Porovnanie na základe reťazcových literálov

```
$ ./SemanticDiff.py --ignore-identifiers --ignore-numbers --ignore-comments
--ignore-operators ./data/a.c ./data/b.c
6c6
<   printf("Please enter a number\n");
---
>   printf("Type a number\n");
14c13
<   printf("Factorial of %d is %d\n",n,sum);
---
>   printf("%d! = %d\n",number,sum);
```

Porovnanie na základe číselných literálov

```
$ ./SemanticDiff.py --ignore-identifiers --ignore-strings --ignore-comments
--ignore-operators ./data/a.c ./data/b.c
16c15
<   return 0;
---
>   return 1;
```

Porovnanie na základe komentárov

```
$ ./SemanticDiff.py --ignore-identifiers --ignore-strings --ignore-numbers
--ignore-operators ./data/a.c ./data/b.c
8,10c8
<   /*
<    * Iterative count factorial
<    */
---
>   //iterative count factorial
13a12
>   //print result
```

Porovnanie na základe operátorov

```
$ ./SemanticDiff.py --ignore-identifiers --ignore-strings --ignore-numbers
--ignore-comments ./data/a.c ./data/b.c
11c9
<    for(int i = 1; i <= n; i++)
---
>   for(int i = 1; i <= number; ++i)
```

Ďalšie programovacie jazyky

Ukážky porovnávania ďalších programovacích jazykov Java a Python nájdeme v prílohe E.

Rýchlosť aplikácie

Porovnávanie uvedených súborov prebehlo okamžite – trvalo 0,13 s. Pri porovnávaní zdrojového súboru s 10 000 riadkami trvalo porovnanie 2,6 s. Pre porovnanie, veľké súbory porovnal program `diff` za 0,01 s, avšak tieto údaje sú neporovnateľné, keďže `diff` porovnáva riadky oboch súborov a je implementovaný v kompilovanom jazyku C., proti tomu, `mediadiff` porovnáva zdrojové kódy na základe zoznamu tokenov a je implementovaný v interpretovanom jazyku Python. Porovnanie s voľbami ignore výsledný čas nezmenili. Takisto, pri ostatných formátoch výstupu bol čas porovnávania rovnaký.

Kapitola 7

Možné pokračovanie

Táto práca nadväzuje na predchádzajúcu bakalársku prácu, v ktorej som implementoval moduly:

- Porovnávanie textových súborov, so štandardnými formátmi výstupu, ale aj s výstupom formou html stránky. Pri porovnávaní máme pokročilé voľby ignorovania veľkosti písmen, bielych znakov alebo riadkov definovaných pomocou regulárneho výrazu a iné.
- Porovnávanie DTP systému LaTeX, kde sme extrahovali text, ktorý sme následne porovnali.
- Porovnávanie konfiguračných súborov. Porovnávanie je nezávislé na poradí jednotlivých riadkov v súbore.
- Porovnávanie obrázkov s pokročilými voľbami ako detekcia otočenia alebo zrkadlenia, zmeny veľkosti, rýchle porovnávanie a iné.

Paralelne s touto prácou sa vyvíjajú moduly na porovnávanie zvuku a prepracované porovnávanie obrázkov.

Cieľom tejto práce bolo vyvinúť modul porovnávajúci zdrojové kódy programovacích jazykov. Porovnávanie na základe riadkov zdrojových kódov nie je dostatočné, pretože zdrojový kód má určitú syntax a sémantiku presne definovanú gramatikou. Pri sémantickom porovnávaní môžeme napríklad zanedbať zmeny v názvoch identifikátorov, literáloch alebo komentároch. Implicitne ignorujeme zmeny v bielych znakoch.

Keďže aplikácia, `mediadiff`, bola od začiatku navrhovaná ako modulárna, je jednoduché pridaním ďalšieho modulu rozšíriť jej funkčnosť o porovnávanie nového typu dokumentov. Druhou možnosťou je rozšíriť existujúci modul, ktorý porovnáva daný typ súborov, o spracovanie nového formátu daného typu – napríklad modul SemanticDiff rozšíriť o porovnávanie ďalšieho programovacieho jazyka. Ďalej uvedieme konkrétne možné rozšírenia aplikácie.

Porovnávanie ďalších programovacích jazykov

V prípade potreby porovnávať ďalší programovací jazyk, je možné implementovať ďalší submodul lexikálneho analyzátora, ktorý zdrojový kód daného jazyka rozparsuje na zoznam tokenov. Submodul bude implementovaný, podobne ako ostatné submoduly, pomocou knižnice PLY a bude poskytovať potrebné rozhranie. Samotné porovnávanie a formátovanie je už implementované. Obmedzenie nie je na zdrojové kódy programovacích jazykov, ale môžeme

porovnávať akýkoľvek jazyk, ktorému definujeme gramatiku. Ako príklady môžeme uviesť programovacie jazyky C# , PHP, JavaScript, ale aj značkovacie jazyky XML, HTML a iné.

Porovnávanie zdrojových kódov založené na AST

Porovnávanie zdrojových kódov môžeme realizovať aj porovnaním ich abstraktných syntaktických stromov (AST). Tak by sme dosiahli väčšiu mieru abstrakcie pri porovnávaní. V porovnávaných AST by sme mohli lokalizovať zmenu priamo v triede, metóde alebo bloku kódu a takto vyčleniť celú jednotku. Bolo by možné, pri použití vhodného porovnávacieho algoritmu, detekovať premiestnenie časti kódu v súbore, resp. v súboroch.

Postup pri takomto porovnávaní by sa skladal z dvoch častí:

1. Vytvorenie AST zo zdrojového kódu. Na vytvorenie by sme mohli použiť druhú časť, syntaktický analyzátor, z knižnice PLY.
2. Porovnanie stromových štruktúr vhodným algoritmom. Algoritmus by mal detekovať zámenu uzlov v strome a ponúkať pokročilé voľby na ignorovanie určitých zmien v špecifikovaných podstromoch.

Tieto operácie sú však omnoho výpočetne náročnejšie a porovnávanie väčších zdrojových kódov, či dokonca celých projektov by trvalo oveľa dlhší čas, než jednoduché porovnanie po riadkoch resp. implementované sémantické porovnávanie.

Ďalšie moduly

Medzi ďalšie moduly, ktoré by mohli rozšíriť `mediadiff` patrí:

- porovnávanie video súborov,
- porovnávanie XML súborov,
- porovnávanie pdf súborov,
- porovnávanie archívov.

Aplikácia je konzolová a má textový výstup. Bolo by vhodné vytvoriť grafické užívateľské rozhranie k programu ako ďalšie rozšírenie.

Kapitola 8

Záver

Cieľom tejto práce bola teoretická analýza, návrh a implementácia modulárnej aplikácie, `mediadiff`, ktorá porovnáva rôzne typy dokumentov. Stanovený cieľ sa mi podarilo splniť a výsledkom mojej práce je funkčná aplikácia, porovnávajúca rôzne typy dokumentov.

Doposiaľ neexistoval jeden nástroj na porovnávanie rôznych typov dokumentov, ale mnoho programov, ktoré väčšinou porovnávali text alebo zložky, prípadne niektoré programy úzko špecializované na konkrétny typ súboru, ako napríklad xml dokumenty. Absencia jedného nástroju na porovnávanie viacerých typov dokumentov nútila užívateľa hľadať rôzne programy na porovnávanie konkrétneho typu súboru.

Práca nadväzuje na moju predchádzajúcu bakalársku prácu, ktorá implementovala porovnávanie textových súborov, konfiguračných súborov, súborov systému LaTeXa obrázkov. Súbežne s touto prácou sú vyvíjané moduly na porovnávanie zvuku a prepracované porovnávanie obrázkov.

Program `mediadiff` je modulárna aplikácia, kde porovnávanie konkrétneho typu dokumentu je vykonávané príslušným modulom. Každý modul je možné spustiť aj ako samostatný program alebo ho použiť v inom programe ako knižnicu. Modul, ktorý je výsledkom tejto práce porovnáva zdrojové kódy programovacích jazykov. Podporované programovacie jazyky sú C/C++, Java a Python. Modul sa dá jednoducho rozšíriť o porovnávanie ďalších programovacích jazykov. Porovnávanie pozostáva z lexikálnej analýzy zdrojových kódov a následné porovnanie získaných zoznamov tokenov. Tento prístup nám dáva väčšie možnosti, než jednoduché porovnanie po riadkoch. Možnosť ignorovať zmeny v názvoch identifikátorov odhalí obyčajné premenovanie premenných. Ďalšou možnosťou je ignorovanie zmien v číselných alebo reťazcových literáloch, alebo v operátoroch. Dôležitou vlastnosťou je ignorovanie zmien v komentároch. Samozrejmosťou je možnosť tieto voľby ľubovoľne kombinovať. Zmeny v bielych znakoch sa ignorujú automaticky.

Pri testovaní som dosiahol dobré výsledky. Malé súbory sú porovnávane v zlomkoch sekúnd. Porovnávanie súboru s 10 000 riadkami zdrojového kódu trvalo 2,6 s, čo je veľmi dobrá hodnota, vzhľadom na zložité procesy, ktoré sa pri porovnávaní vykonávali a vzhľadom na to, že Python je interpretovaný jazyk.

Počas vývoja tejto práce som sa zoznámil s problematikou porovnávania zdrojových kódov a s rôznymi prístupmi jej riešenia, s rôznymi nástrojmi na automatizáciu lexikálnej analýzy, v ktorých som implementoval lexikálne analyzátory pre dané gramatiky jazykov a v neposlednej rade s jazykom Python a množstvom jeho užitočných štandardných knižníc.

Nakoľko rozsah tejto práce neumožňuje popísať a implementovať porovnávanie viacerých typov dokumentov, je možné aplikáciu rozšíriť o ďalšie moduly. Modul porovnávajúci zdrojové kódy je možné rozšíriť o porovnávanie ďalších programovacích jazykov alebo po-

rovnávanie vykonávať na základe porovnania abstraktných syntaktických stromov. Ďalšie moduly vhodné k implementácii sú porovnávanie video súborov, xml dokumentov, pdf súborov, archívov a iných špecifických typov dokumentov.

Literatura

[1] *Regular Expressions*. The Open Group, druhé vydání, 1997, The Single UNIX ® Specification.

[2] Lexical Analysis With Flex. 2011, [Online; accessed 7-May-2011].
URL http://flex.sourceforge.net/manual/

[3] Abdessalem, T.; Hinnach, Y.; Cobéna, C. G.; aj.: A comparative study of XML diff tools. 2008.

[4] Aho, A. V.; Lam, M. S.; Sethi, R.; aj.: *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, 2006.

[5] Andreas Laux,Lars Martin: XUpdate - XML Update Language [online]. `http://xmldb-org.sourceforge.net/xupdate/`, [cit. 2010-12-26].

[6] AquaFold: Database query tool - Aqua Data Studio is the query analyzer that works with Oracle, DB2, Sybase, MySQL and more. `http://www.aquafold.com/`, [cit. 2011-01-03].

[7] Araxis: Merge for Windows — File Comparison, Merging & More. `http://www.araxis.com/merge/index.html`, [cit. 2011-01-03].

[8] Baker, B. S.: A Program for Identifying Duplicated Code. *Computing Science and Statistics*, ročník 24, 1992: str. 49–57.

[9] Baker, B. S.: On finding duplication and near-duplication in large software systems. In *Proceedings of the Second Working Conference on Reverse Engineering*, WCRE '95, Washington, DC, USA: IEEE Computer Society, 1995, ISBN 0-8186-7111-4, s. 86–95.

[10] Balazinska, M.; Merlo, E.; Dagenais, M.; aj.: Measuring Clone Based Reengineering Opportunities. In *Proceedings of the 6th International Symposium on Software Metrics*, Washington, DC, USA: IEEE Computer Society, 1999, ISBN 0-7695-0403-5, s. 292–303.

[11] Barnard, D. T.; Clarke, G.; Duncan, N.: Tree-to-tree Correction for Document Trees. Technická zpráva, Queen's University, Kingston, Ontario K7L 3N6, Canada, 1995.

[12] Baxter, I. D.; Yahin, A.; Moura, L.; aj.: Clone Detection Using Abstract Syntax Trees. *Proceedings of the 14th International Conference on Software Maintenance (ICSM'98)*, ročník 0, 1998: s. 368–377.

[13] Beazley, D.: PLY (Python Lex-Yacc). 2011, [Online; accessed 7-May-2011].
URL http://www.dabeaz.com/ply/ply.html

[14] Bellon, S.: Detection of Software Clones Tool Comparison Experiment. In *Tool Comparison Experiment presented at the 1st IEEE International Workshop on Source Code Analysis and Manipulation*, 2002.

[15] Bellon, S.; Koschke, R.; Antoniol, G.; aj.: Comparison and Evaluation of Clone Detection Tools. *IEEE Trans. Softw. Eng.*, ročník 33, September 2007: s. 577–591, ISSN 0098-5589.

[16] Burd, E.; Bailey, J.: Evaluating Clone Detection Tools for Use during Preventative Maintenance. In *Proceedings of the Second IEEE International Workshop on Source Code Analysis and Manipulation*, SCAM '02, Washington, DC, USA: IEEE Computer Society, 2002, ISBN 0-7695-1793-5, s. 36–.

[17] Chawathe, S. S.: Comparing Hierarchical Data in External Memory. In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999, ISBN 1-55860-615-7, s. 90–101.

[18] Chawathe, S. S.; Rajaraman, A.; Garcia-Molina, H.; aj.: Change detection in hierarchically structured information. *SIGMOD Rec.*, ročník 25, June 1996: s. 493–504, ISSN 0163-5808.

[19] Cormen, T. H.; et al.: *Introduction to algorithms*. Massachusetts: MIT Press, druhé vydání, 2001, 350–355 s., iSBN 0-262-03293-7.

[20] Curbera, F.: XML TreeDiff [online].
http://alphaworks.ibm.com/tech/xmltreediff/, [cit. 2010-12-26].

[21] Curbera, F.; Epstein, D.: Fast difference and update of xml documents. *XTech,*, 1999.

[22] Davey, N.; Barson, P.; Field, S.; aj.: The Development of a Software Clone Detector. In *International Journal of Applied Software Technology*, 1995, s. 219–236.

[23] Devart: Devart CodeCompares. http://www.devart.com/codecompare/, [cit. 2011-01-03].

[24] Ducasse, S.; Nierstrasz, O.; Rieger, M.: On the effectiveness of clone detection by string matching: Research Articles. *J. Softw. Maint. Evol.*, ročník 18, January 2006: s. 37–58, ISSN 1532-060X.

[25] Ducasse, S.; Rieger, M.; Demeyer, S.: A Language Independent Approach for Detecting Duplicated Code. In *Proceedings of the IEEE International Conference on Software Maintenance*, ICSM '99, Washington, DC, USA: IEEE Computer Society, 1999, ISBN 0-7695-0016-1, s. 109–.

[26] Fontaine, R. L.: A Delta Format for XML: Identifying Changes in XML Files and Representing the Changes in XML. *XML Europe*, 2001.

[27] Gosling, J.; Joy, B.; Steele, G.; aj.: *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005, ISBN 0321246780.

[28] Grégory COBÉNA: *Change Management of semi-structured data on the Web*. Dizertační práce, École Polytechnique, Ecole, 2003.

[29] Grégory Cobéna, A. M., Serge Abiteboul: XyDiff Tools Detecting changes in XML Documents [online]. `http://leo.saclay.inria.fr//software/XyDiff/cdrom/www/xydiff/index-eng.htm`, [cit. 2010-12-26].

[30] Joel de Guzman, H. K.: Boost Spirit. 2011, [Online; accessed 7-May-2011]. URL http://www.boost.org/doc/libs/1_46_1/libs/spirit/doc/html/index.html

[31] Hamming, R. W.: Error detecting and error correcting codes. *Bell System Technical Journal*, ročník 29, č. 2, 1950: s. 147–160.

[32] Hunt, J. W.; McIlroy, M. D.: An Algorithm for Differential File Comparison. Technická Zpráva CSTR 41, Bell Laboratories, Murray Hill, NJ, 1976.

[33] Johnson, J. H.: Substring Matching for Clone Detection and Change Tracking. In *Proceedings of the International Conference on Software Maintenance*, ICSM '94, Washington, DC, USA: IEEE Computer Society, 1994, ISBN 0-8186-6330-8, s. 120–126.

[34] Kamiya, T.; Kusumoto, S.; Inoue, K.: CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.*, ročník 28, July 2002: s. 654–670, ISSN 0098-5589.

[35] Kapser, C.; Godfrey, M. W.: Aiding Comprehension of Cloning Through Categorization. In *Proceedings of the Principles of Software Evolution, 7th International Workshop*, Washington, DC, USA: IEEE Computer Society, 2004, ISBN 0-7695-2211-4, s. 85–94.

[36] Kapser, C. J.; Godfrey, M. W.: Supporting the analysis of clones in software systems: Research Articles. *J. Softw. Maint. Evol.*, ročník 18, March 2006: s. 61–82, ISSN 1532-060X.

[37] Kher, A.: The XML Diff and Patch GUI Tool [online]. `http://msdn.microsoft.com/en-us/library/aa302295.aspx`, [cit. 2010-12-26].

[38] Kim, M.; Sazawal, V.; Notkin, D.; aj.: An empirical study of code clone genealogies. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-13, New York, NY, USA: ACM, 2005, ISBN 1-59593-014-0, s. 187–196.

[39] Kolektív autorov: diff [online]. `http://en.wikipedia.org/wiki/Diffutils`, [cit. 2009-04-21].

[40] Kolektív autorov: Levenshtein distance [online]. `http://en.wikipedia.org/wiki/Levenshtein_distance`, [cit. 2009-05-09].

[41] Kolektív autorov: Longest common subsequence problem [online]. `http://en.wikipedia.org/wiki/Longest_common_subsequence_problem`, [cit. 2009-05-12].

[42] Komondoor, R.; Horwitz, S.: Tool Demonstration: Finding Duplicated Code Using Program Dependences. In *Proceedings of the 10th European Symposium on Programming Languages and Systems*, ESOP '01, London, UK: Springer-Verlag, 2001, ISBN 3-540-41862-8, s. 383–386.

[43] Komondoor, R.; Horwitz, S.: Effective, Automatic Procedure Extraction. In *Proceedings of the 11th IEEE International Workshop on Program Comprehension*, IWPC '03, Washington, DC, USA: IEEE Computer Society, 2003, ISBN 0-7695-1883-4, s. 33–42.

[44] Kontogiannis, K.: Evaluation Experiments on the Detection of Programming Patterns Using Software Metrics. In *Proceedings of the Fourth Working Conference on Reverse Engineering (WCRE '97)*, Washington, DC, USA: IEEE Computer Society, 1997, ISBN 0-8186-8162-4, s. 44–.

[45] Kontogiannis, K. A.; Demori, R.; Merlo, E.; aj.: *Pattern matching for clone and concept detection*. Norwell, MA, USA: Kluwer Academic Publishers, 1996, ISBN 0-7923-9756-8, s. 77–108.

[46] Koschke, R.; Falke, R.; Frenzel, P.: Clone Detection Using Abstract Syntax Suffix Trees. In *Proceedings of the 13th Working Conference on Reverse Engineering*, 2006, ISBN 0-7695-2719-1, s. 253–262.

[47] Krinke, J.: Identifying Similar Code with Program Dependence Graphs. In *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, WCRE '01, 2001, ISBN 0-7695-1303-4, s. 301–309.

[48] Lague, B.; Proulx, D.; Mayrand, J.; aj.: Assessing the Benefits of Incorporating Function Clone Detection in a Development Process. In *Proceedings of the International Conference on Software Maintenance*, 1997, ISBN 0-8186-8013-X, s. 314–321.

[49] Lakhotia, A.; Li, J.; Walenstein, A.; aj.: Towards a Clone Detection Benchmark Suite and Results Archive. In *Proceedings of the 11th IEEE International Workshop on Program Comprehension*, IWPC '03, Washington, DC, USA: IEEE Computer Society, 2003, ISBN 0-7695-1883-4, s. 285–.

[50] Levine, J.; Mason, T.; Brown, D.: *Lex & yacc*. A Nutshell handbook, O'Reilly & Associates, 1992, ISBN 9781565920002.

[51] Li, Z.; Lu, S.; Myagmar, S.; aj.: CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code. *IEEE Trans. Softw. Eng.*, ročník 32, March 2006: s. 176–192, ISSN 0098-5589.

[52] Logilab: XML Diff [online]. `http://www.logilab.org/859`, [cit. 2010-12-26].

[53] MacKenzie, D.; Eggert, P.; Stallman, R.: *Comparing and Merging Files with GNU Diff and Patch*. Bristol: Network Theory, 1997, iSBN 0-9541617-5-0.

[54] Marcus, A.; Maletic, J. I.: Identification of High-Level Concept Clones in Source Code. In *Proceedings of the 16th IEEE international conference on Automated software engineering*, ASE '01, Washington, DC, USA: IEEE Computer Society, 2001, s. 107–.

[55] Marian, A.: Detecting Changes in XML Documents. In *Proceedings of the 18th International Conference on Data Engineering*, ICDE '02, Washington, DC, USA: IEEE Computer Society, 2002, s. 41–.

[56] Marian, A.; Abiteboul, S.; Cobena, G.; aj.: Change-Centric Management of Versions in an XML Warehouse. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001, ISBN 1-55860-804-4, s. 581–590.

[57] Mayrand, J.; Leblanc, C.; Merlo, E.: Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics. In *Proceedings of the 1996 International Conference on Software Maintenance*, ICSM '96, 1996, ISBN 0-8186-7677-9, s. 244–253.

[58] McCreight, E. M.: A Space-Economical Suffix Tree Construction Algorithm. *J. ACM*, ročník 23, April 1976: s. 262–272, ISSN 0004-5411.

[59] McGuire, P.: *Getting Started with Pyparsing*. Safari Books Online, O'Reilly, 2007, ISBN 9780596514235.

[60] Michal Zemko: *Diff pro různé typy dokumentů*. Bakalářská práce, Vysoké učení technické, Fakulta informačních technologií, Brno, 2009.

[61] Michal Zemko: *Diff pro různé typy dokumentů*. Semestrální projekt, Vysoké učení technické, Fakulta informačních technologií, Brno, 2010.

[62] Miller, W.; Myers, E. W.: A File Comparison Program. *Software–Practice and Experience*, ročník 15, č. 11, 1985: s. 1025–1040.

[63] Mishne, G.; Rijke, M. d.: Source code retrieval using conceptual similarity. In *Proceeding of the 2004 Conference on Computer Assisted Information Retrieval (RIAO'04)*, 2004, s. 539–554.

[64] Mitchell, B. S.; Mancoridis, S.: CRAFT: A Framework for Evaluating Software Clustering Results in the Absence of Benchmark Decompositions. In *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, WCRE '01, Washington, DC, USA: IEEE Computer Society, 2001, ISBN 0-7695-1303-4, s. 93–.

[65] Myers, E. W.: An O(ND) Difference Algorithm and its Variations. *Algorithmica*, ročník 1, č. 2, 1986: s. 251–266.

[66] Parr, T.: ANTLR. 2011, [Online; accessed 7-May-2011].
URL http://www.antlr.org/

[67] Rao Kosaraju, S.: Faster algorithms for the construction of parameterized suffix trees. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, FOCS '95, Washington, DC, USA: IEEE Computer Society, 1995, ISBN 0-8186-7183-1, s. 631–.

[68] Ratcliff, J.; Metzener, D.: Pattern Matching: The Gestalt Approach. *Dr. Dobb's Journal*, 1988: str. 46.

[69] Roy, C. K.; Cordy, J. R.: A Survey on Software Clone Detection Research. *SCHOOL OF COMPUTING TR 2007-541, QUEEN'S UNIVERSITY*, ročník 115, 2007.

[70] Selkow, S. M.: The Tree-to-Tree Editing Problem. *Information Processing Letters*, ročník 6, č. 6, 1977: s. 184–186.

[71] Shasha, D.; Zhang, K.: Fast algorithms for the unit cost editing distance between trees. *J. Algorithms*, ročník 11, December 1990: s. 581–621, ISSN 0196-6774.

[72] software, S.: Compare Files, Folders. `http://www.scootersoftware.com/moreinfo.php?zz=moreinfo_compare`, [cit. 2011-01-03].

[73] Software, S.: OOP-DIFF. `http://www.schneidersoft.com/Products/OOP-DIFF/OOP-DIFF.aspx`, [cit. 2011-01-03].

[74] Stroustrup, B.: *The C++ Programming Language*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., třetí vydání, 2000, ISBN 0201700735.

[75] Summerfield, M.: *Programming in Python 3*. Addison-Wesley, druhé vydání, 2009.

[76] Tai, K.-C.: The Tree-to-Tree Correction Problem. *J. ACM*, ročník 26, July 1979: s. 422–433, ISSN 0004-5411.

[77] Ukkonen, E.: Algorithms for Approximate String Matching. *Information and Control*, ročník 64, 1985: s. 100–118.

[78] Vojnar, T.: Formal Analysis and Verification. 2011.

[79] Wahler, V.; Seipel, D.; Gudenberg, J. W. v.; aj.: Clone Detection in Source Code by Frequent Itemset Techniques. In *Proceedings of the Source Code Analysis and Manipulation, Fourth IEEE International Workshop*, Washington, DC, USA: IEEE Computer Society, 2004, ISBN 0-7695-2144-4, s. 128–135.

[80] Wang, Y.; DeWitt, D.; Cai, J.-Y.: X-Diff: an effective change detection algorithm for XML documents. In *Data Engineering, 2003. Proceedings. 19th International Conference on*, 2003, s. 519 – 530.

[81] Wikipedia: Lexical analysis — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/w/index.php?title=Lexical_analysis&oldid=405093223, 2010, [Online; accessed 3-January-2011].

[82] Wikipedia: Parsing — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/w/index.php?title=Parsing&oldid=404926563, 2010, [Online; accessed 3-January-2011].

[83] Wikipedia: INI file — Wikipedia, The Free Encyclopedia. 2011, [Online; accessed 12-May-2011].
URL http://en.wikipedia.org/w/index.php?title=INI_file&oldid=424477463

[84] Wikipedia: Kompare — Wikipedia, The Free Encyclopedia. 2011, [Online; accessed 19-Feb-2011].
URL http://en.wikipedia.org/w/index.php?title=Kompare&oldid=422726899

[85] Wikipedia: Lex (software) — Wikipedia, The Free Encyclopedia. 2011, [Online; accessed 7-May-2011].
URL http://en.wikipedia.org/w/index.php?title=Lex_(software)&oldid=426971668

[86] Wikipedia: Meld (software) — Wikipedia, The Free Encyclopedia. 2011, [Online; accessed 19-Feb-2011].
URL http://en.wikipedia.org/w/index.php?title=Meld_(software)&oldid=424519298

[87] Wu, S.; Manber, U.; Myers, G.; aj.: An O(NP) sequence comparison algorithm. *Inf. Process. Lett.*, ročník 35, September 1990: s. 317–323, ISSN 0020-0190.

[88] Yang, W.: Identifying syntactic differences between two programs. *Softw. Pract. Exper.*, ročník 21, June 1991: s. 739–755, ISSN 0038-0644.

[89] Zhang, K.: A constrained edit distance between unordered labeled trees. *Algorithmica*, ročník 15, 1996: s. 205–222, ISSN 0178-4617.

[90] Zhang, K.; Shasha, D.: Simple fast algorithms for the editing distance between trees and related problems. *SIAM J. Comput.*, ročník 18, December 1989: s. 1245–1262, ISSN 0097-5397.

[91] Zhang, K.; Statman, R.; Shasha, D.: On the editing distance between unordered labeled trees. *Inf. Process. Lett.*, ročník 42, May 1992: s. 133–139, ISSN 0020-0190.

Příloha A

Obsah CD

\doc\	-	programová dokumentácia
\examples\	-	príklady zdrojových kódov
\grammars	-	definície gramatík implementovaných jazykov
\mediadif\	-	implementovaná aplikácia
\mediadif\modules \SemanticDiff\	-	implementácia modulu porovnávajúceho zdrojové kódy
\README	-	návod na použitie
\INSTALL	-	návod na inštaláciu

Příloha B

Konfigrační soubor

```
# general parameters of this module to be processed in mediadiff
[mediadiff]
extensions=py,h,hh,hpp,hxx,h++,c,cc,cpp,cxx,c++,java

[priorities]
# priority of each extension, default is the lowest possible
py=100000
c=100000
h=100000

# extensions of languages
[languages]
py=Python
c=C++
cc=C++
cpp=C++
cxx=C++
c++=C++
h=C++
hh=C++
hpp=C++
hxx=C++
h++=C++
java=Java

# default values of parameters
[parameters]
encoding=utf-8
languages=Python,C,C++,Java
```

Příloha C

Príklady formátov výstupu

C.1 Textové dokumenty

Vzorové texty a niektoré uvedené formáty výstupu sú prebrané z [53].

C.1.1 Vzorové texty

Súbor lao:

```
The Way that can be told of is not the eternal Way;
The name that can be named is not the eternal name.
The Nameless is the origin of Heaven and Earth;
The Named is the mother of all things.
Therefore let there always be non-being,
  so we may see their subtlety,
And let there always be being,
  so we may see their outcome.
The two are the same,
But after they are produced,
  they have different names.
```

Súbor tzu:

```
The Nameless is the origin of Heaven and Earth;
The named is the mother of all things.

Therefore let there always be non-being,
  so we may see their subtlety,
And let there always be being,
  so we may see their outcome.
The two are the same,
But after they are produced,
  they have different names.
They both may be called deep and profound.
Deeper and more profound,
The door of all subtleties!
```

V týchto príkladoch sú niektoré riadky z prvého súboru zmazané, pridané do súboru lao alebo zmenené v oboch súboroch navzájom.

C.1.2 Normálny formát výstupu

Príklad normálneho formátu výstupu:

```
1,2d0
< The Way that can be told of is not the eternal Way;
< The name that can be named is not the eternal name.
4c2,3
< The Named is the mother of all things.
---
> The named is the mother of all things.
>
11a11,13
> They both may be called deep and profound.
> Deeper and more profound,
> The door of all subtleties!
```

C.1.3 Kontextový formát výstupu

Príklad kontextového formátu výstupu:

```
*** lao 2002-02-21 23:30:39.942229878 -0800
--- tzu 2002-02-21 23:30:50.442260588 -0800
***************
*** 1,7 ****
- The Way that can be told of is not the eternal Way;
- The name that can be named is not the eternal name.
  The Nameless is the origin of Heaven and Earth;
! The Named is the mother of all things.
  Therefore let there always be non-being,
    so we may see their subtlety,
  And let there always be being,
--- 1,6 ----
  The Nameless is the origin of Heaven and Earth;
! The named is the mother of all things.
!
  Therefore let there always be non-being,
    so we may see their subtlety,
  And let there always be being,
***************
*** 9,11 ****
--- 8,13 ----
  The two are the same,
  But after they are produced,
    they have different names.
+ They both may be called deep and profound.
+ Deeper and more profound,
+ The door of all subtleties!
```

C.1.4 Unifikovaný formát výstupu

Príklad unifikovaného formátu výstupu:

```
--- lao 2002-02-21 23:30:39.942229878 -0800
+++ tzu 2002-02-21 23:30:50.442260588 -0800
@@ -1,7 +1,6 @@
-The Way that can be told of is not the eternal Way;
-The name that can be named is not the eternal name.
 The Nameless is the origin of Heaven and Earth;
-The Named is the mother of all things.
+The named is the mother of all things.
+
 Therefore let there always be non-being,
   so we may see their subtlety,
 And let there always be being,
@@ -9,3 +8,6 @@
 The two are the same,
 But after they are produced,
   they have different names.
+They both may be called deep and profound.
+Deeper and more profound,
+The door of all subtleties!
```

C.1.5 Formátu výstupu vedľa seba

Príklad formátu výstupu vedľa seba:

```
The Way that can be told of is n   <
The name that can be named is no   <
The Nameless is the origin of He       The Nameless is the origin of He
The Named is the mother of all t   |    The named is the mother of all t
                                   >
Therefore let there always be no       Therefore let there always be no
  so we may see their subtlety,          so we may see their subtlety,
And let there always be being,         And let there always be being,
  so we may see their outcome.           so we may see their outcome.
The two are the same,                  The two are the same,
But after they are produced,           But after they are produced,
  they have different names.             they have different names.
                                   >    They both may be called deep and
                                   >    Deeper and more profound,
                                   >    The door of all subtleties!
```

Aby sa nám výsledok zmestil na stránku, museli sme obmedziť šírku textu na 72 znakov.

C.1.6 Formát výstupu ndiff

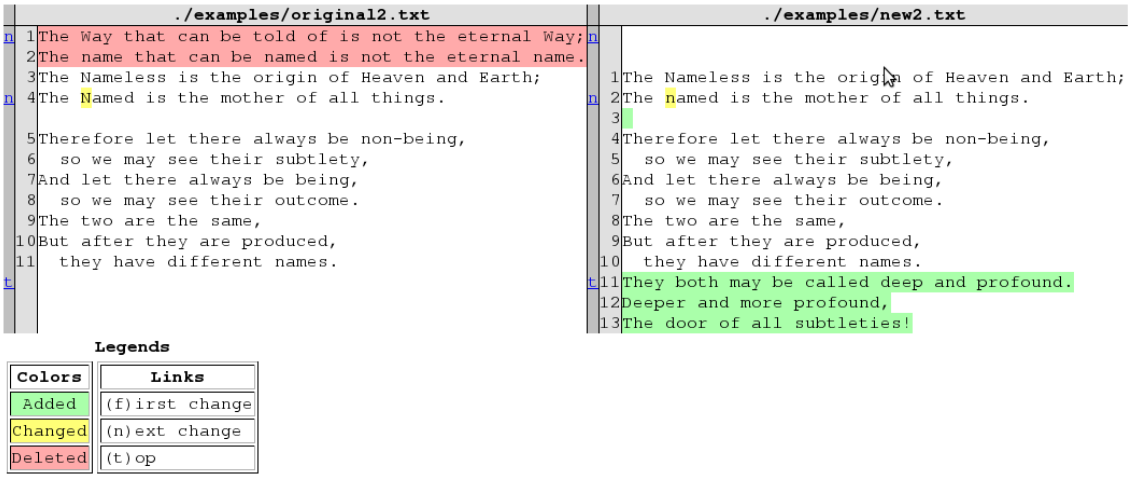
Riadky, ktoré sú rovnaké, začínajú dvoma medzerami, riadky, ktoré sú zmazané z prvého súboru začínajú znakom - a medzerou, vložené riadky do druhého súboru začínajú znakom

+. Špeciálne riadky začínajúce znakom ? majú informatívny charakter a sú vložené pod riadky, ktoré sa líšia iba v určitej miere. V tomto pomocnom riadku ukazujú znaky ^ na znaky predchádzajúceho riadku, ktoré sú odlišné.

Príklad formátu výstupu ndiff:

```
- The Way that can be told of is not the eternal Way;
- The name that can be named is not the eternal name.
  The Nameless is the origin of Heaven and Earth;
- The Named is the mother of all things.
?     ^
+ The named is the mother of all things.
?     ^
+
  Therefore let there always be non-being,
    so we may see their subtlety,
  And let there always be being,
    so we may see their outcome.
  The two are the same,
  But after they are produced,
    they have different names.
+ They both may be called deep and profound.
+ Deeper and more profound,
+ The door of all subtleties!
```

C.1.7 HTML výstup



Obrázek C.1: Výstup vo formáte html

C.2 XML dokumenty

Vzorové dáta a niektoré formáty výstupu sú prebrané z [3].

C.2.1 Vstupné data

Vstupné data:

```
<catalog>
<product>
    <name>Notebook</name>
    <description>2200MHz Pentium4</description>
    <price>$1999</price>
  </product>
  <product>
    <name>Digital Camera</name>
    <description>Fuji FinePix 2600Z</description>
    <status>Not Available</status>
  </product>
</catalog>
```

```
<catalog>
  <product>
    <name>Notebook</name>
    <description>2200MHz Pentium4</description>
    <price>$1999</price>
  </product>
  <product>
    <name>Digital Camera</name>
    <description>Fuji FinePix 2600Z</description>
    <price>$299</price>
  </product>
</catalog>
```

C.2.2 XUpdate

```
<xupdate:modifications version="1.0"
xmlns:xupdate="http://www.xmldb.org/xupdate">
  <xupdate:insert-after select="/catalog[1]/product[2]/description[1]" >
    <xupdate:element name="price">
      $299
    </xupdate:element>
  </xupdate:insert-after>
  <xupdate:remove select="/catalog[1]/product[2]/status[1]" />
</xupdate:modifications>
```

C.2.3 XyDelta

```
<xydelta v1_XidMap="(1-15)" v2_XidMap="(1-11;16-17;14-15)">
  <delete xid="(12-13)" parent="14" position="3">
    <status>Not Available</status>
  </delete>
  <insert xid="(16-17)" parent="14" position="3">
    <price>$299</price>
```

```
  </insert>
</xydelta>
```

C.2.4 XDL

```
<xd:xmldiff srcDocHash="......"
xmlns:xd="http://schemas.microsoft.com/xmltools/2002/xmldiff" >
  <xd:node match="1">
    <xd:node match="2">
      <xd:change match="3" name="price">
        <xd:change match="1">$299</xd:change>
      </xd:change>
    </xd:node>
  </xd:node>
</xd:xmldiff>
```

C.2.5 DeltaXML

```
<catalog
xmlns:deltaxml="http://www.deltaxml.com/ns/well-formed-delta-v1"
deltaxml:delta="WFmodify" >
  <product deltaxml:delta="unchanged"/>
  <product deltaxml:delta="WFmodify">
    <name deltaxml:delta="unchanged"/>
    <description deltaxml:delta="unchanged"/>
    <deltaxml:exchange>
      <deltaxml:old>
        <status deltaxml:delta="delete">Not Available</status>
      </deltaxml:old>
      <deltaxml:new>
        <price deltaxml:delta="add">$299</price>
      </deltaxml:new>
    </deltaxml:exchange>
  </product>
</catalog>
```

Príloha D

Lexikálne analyzátory

D.1 Lex

Príklad definičného súboru pre program lex [2]:

```
/* scanner for a toy Pascal-like language */

        %{
        /* need this for the call to atof() below */
        #include math.h>
        %}

        DIGIT    [0-9]
        ID       [a-z][a-z0-9]*

        %%

        {DIGIT}+    {
                    printf( "An integer: %s (%d)\n", yytext,
                            atoi( yytext ) );
                    }

        {DIGIT}+"."{DIGIT}*        {
                    printf( "A float: %s (%g)\n", yytext,
                            atof( yytext ) );
                    }

        if|then|begin|end|procedure|function        {
                    printf( "A keyword: %s\n", yytext );
                    }

        {ID}        printf( "An identifier: %s\n", yytext );

        "+"|"-"|"*"|"/"   printf( "An operator: %s\n", yytext );

        "{"[\^{}}\n]*"}"     /* eat up one-line comments */
```

```
[ \t\n]+          /* eat up whitespace */

.           printf( "Unrecognized character: %s\n", yytext );

%%

main( argc, argv )
int argc;
char **argv;
    {
    ++argv, --argc;  /* skip over program name */
    if ( argc > 0 )
            yyin = fopen( argv[0], "r" );
    else
            yyin = stdin;

    yylex();
    }
```

D.2 PLY

Príklad lexikálneho analyzátora implementovaného pomocou PLY [13] :

```
# ------------------------------------------------------------
# calclex.py
#
# tokenizer for a simple expression evaluator for
# numbers and +,-,*,/
# ------------------------------------------------------------
import ply.lex as lex

# List of token names.   This is always required
tokens = (
   'NUMBER',
   'PLUS',
   'MINUS',
   'TIMES',
   'DIVIDE',
   'LPAREN',
   'RPAREN',
)

# Regular expression rules for simple tokens
t_PLUS    = r'\+'
t_MINUS   = r'-'
t_TIMES   = r'\*'
t_DIVIDE  = r'/'
```

```
t_LPAREN  = r'\('
t_RPAREN  = r'\)'

# A regular expression rule with some action code
def t_NUMBER(t):
    r'\d+'
    t.value = int(t.value)
    return t

# Define a rule so we can track line numbers
def t_newline(t):
    r'\n+'
    t.lexer.lineno += len(t.value)

# A string containing ignored characters (spaces and tabs)
t_ignore  = ' \t'

# Error handling rule
def t_error(t):
    print "Illegal character '%s'" % t.value[0]
    t.lexer.skip(1)

# Build the lexer
lexer = lex.lex()

# Test it out
data = '''
3 + 4 * 10
  + -20 *2
'''

# Give the lexer some input
lexer.input(data)

# Tokenize
while True:
    tok = lexer.token()
    if not tok: break      # No more input
    print tok
```

Výstup:

```
$ python example.py
LexToken(NUMBER,3,2,1)
LexToken(PLUS,'+',2,3)
LexToken(NUMBER,4,2,5)
LexToken(TIMES,'*',2,7)
LexToken(NUMBER,10,2,10)
LexToken(PLUS,'+',3,14)
```

```
LexToken(MINUS,'-',3,16)
LexToken(NUMBER,20,3,18)
LexToken(TIMES,'*',3,20)
LexToken(NUMBER,2,3,21)
```

Příloha E

Testy

E.1 Java

Testovacie dáta:

```
1  import java.utils;
2
3  /**
4   * Simple class
5   */
6  class HelloWorldApp
7  {
8      public static void main(String[] args)
9      {
10         //print hello world
11         System.out.println("Hello World!");
12     }
13 }
```

Algoritmus E.1: Zdrojový kód `a.java`

```
1  import java.lang;
2
3  /**
4   * Main class
5   */
6  class Main
7  {
8      public static void main(String[] args)
9      {
10         System.out.println("Ahoj svete!");
11     }
12 }
```

Algoritmus E.2: Zdrojový kód `b.java`

Porovnanie bez pokročilých volieb:

```
$ ./SemanticDiff.py ./data/a.java ./data/b.java
1,6c1,6
< import java.utils;
<
< /**
<  * Simple class
<  */
< class HelloWorldApp
---
> import java.lang;
>
> /**
>  * Main class
>  */
> class Main
10,11c10
<         //print hello world
<      System.out.println("Hello World!");
---
>      System.out.println("Ahoj svete!");
```

Pri ignorovaní zmien v identifikátoroch, komentároch a reťazcových literáloch budú zdrojové kódy rovnaké:

```
$ ./SemanticDiff.py --ignore-identifiers --ignore-comments --ignore-strings ./data/a.ja
```

Porovnanie na základe názvov identifikátorov:

```
$ ./SemanticDiff.py  --ignore-comments --ignore-strings
./data/a.java ./data/b.java
1c1
< import java.utils;
---
> import java.lang;
6c6
< class HelloWorldApp
---
> class Main
```

Porovnanie na základe komentárov:

```
$ ./SemanticDiff.py --ignore-identifiers  --ignore-strings
./data/a.java ./data/b.java
3,5c3,5
< /**
<  * Simple class
<  */
---
> /**
>  * Main class
```

```
>  */
10d9
<          //print hello world
```

Porovnanie na základe reťazcových literálov:

```
$ ./SemanticDiff.py --ignore-identifiers --ignore-comments
./data/a.java ./data/b.java
11c10
<      System.out.println("Hello World!");
---
>      System.out.println("Ahoj svete!");
```

E.2 Python

Testovacie dáta:

```
1 def foo(a,b,c):
2         a = 1
3         b = 2
4
5         for i in range(0,c):
6                 print(i)
```

Algoritmus E.3: Zdrojový kód `a.py`

```
1 def foo(x,y,z):
2         y = 0x2
3         x = 0x1
4
5         #sequence of numbers
6         for i in range(0,z):
7                 print(i)
```

Algoritmus E.4: Zdrojový kód `b.py`

Porovnanie bez pokročilých volieb:

```
$ ./SemanticDiff.py ./data/a.py ./data/b.py 1,5c1,6
< def foo(a,b,c):
<  a = 1
<  b = 2
<
<  for i in range(0,c):
---
> def foo(x,y,z):
>  y = 0x2
>  x = 0x1
>
>  #sequence of numbers
>  for i in range(0,z):
```

Pri ignorovaní zmien v identifikátoroch, komentároch a číselných literáloch budú zdrojové kódy rovnaké:

```
$ ./SemanticDiff.py --ignore-identifiers --ignore-numbers --ignore-comments
./data/a.py ./data/b.py
```

Porovnanie na základe názvov identifikátorov:

```
$ ./SemanticDiff.py  --ignore-numbers --ignore-comments
./data/a.py ./data/b.py 1,3c1,3
< def foo(a,b,c):
<  a = 1
<  b = 2
---
> def foo(x,y,z):
>  y = 0x2
>  x = 0x1
5c6
<  for i in range(0,c):
---
>  for i in range(0,z):
```

Porovnanie na základe číselných literálov:

```
$ ./SemanticDiff.py --ignore-identifiers --ignore-comments
./data/a.py ./data/b.py 2,3c2,5
<  a = 1
<  b = 2
---
>  y = 0x2
>  x = 0x1
>
>  #sequence of numbers
```

Porovnanie na základe komentárov:

```
$ ./SemanticDiff.py --ignore-identifiers --ignore-numbers
./data/a.py ./data/b.py 3c3,3
<  b = 2
---
>  x = 0x1
>
>  #sequence of numbers
```

Příloha F

Manuál

F.1 Modul SemanticDiff

```
$ ./SemanticDiff.py -h
usage: SemanticDiff [-h] [--language {Python,Java,C++}] [--encoding ENCODING]
                    [-n] [-u] [-U NUM] [-c] [-C NUM] [--ignore-numbers]
                    [--ignore-strings] [--ignore-operators]
                    [--ignore-identifiers] [--ignore-keywords]
                    [--ignore-comments]
                    file1 file2

Compare two files of source code.

positional arguments:
  file1                 Name of first compared file
  file2                 Name of second compared file.

optional arguments:
  -h, --help            show this help message and exit

Options:
  Options for semantic diff.

  --language {Python,Java,C++}
                        Programming language of source file.
  --encoding ENCODING   Encoding of files, default is UTF-8.

Output:
  Options for output format.

  -n, --normal          Output a normal diff.
  -u                    Output 3 lines of unified context.
  -U NUM, --unified NUM
                        Output NUM lines of unified context.
  -c                    Output 3 lines of copied context.
  -C NUM, --context NUM
```

```
                        Output NUM lines of copied context.

Ignore:
  Options for ignore specific tokens.

  --ignore-numbers      Ignore numbers.
  --ignore-strings      Ignore strings.
  --ignore-operators    Ignore operators.
  --ignore-identifiers  Ignore name of identifiers.
  --ignore-keywords     Ignore keywords.
  --ignore-comments     Ignore comments.
```

F.2 Modul TextDiff

```
$ ./TextDiff.py -h
usage: TextDiff [-h] [--encoding ENCODING] [-n] [-u] [-U NUM] [-c] [-C NUM]
                [-d] [-v] [-i] [-w] [-B] [-I RE]
                file1 file2

Compare two text files line by line.

positional arguments:
  file1                 Name of first compared file
  file2                 Name of second compared file.

optional arguments:
  -h, --help            show this help message and exit

Options:
  Options for text diff.

  --encoding ENCODING   Encoding of files, default is UTF-8.

Output:
  Options for output format.

  -n, --normal          Output a normal diff.
  -u                    Output 3 lines of unified context.
  -U NUM, --unified NUM
                        Output NUM lines of unified context.
  -c                    Output 3 lines of copied context.
  -C NUM, --context NUM
                        Output NUM lines of copied context.
  -d, --differ          Output differ style format. Don't support ignore
                        options.
  -v, --html            Output as html page. Don't support ignore options.

Ignore:
```

```
Ignore options.

-i, --ignore-case     Ignores case differences in file contents.
-w, --ignore-whitespace
                      Ignores white spaces.
-B, --ignore-blanklines
                      Ignores changes whose lines are all blank.
-I RE, --ignore-matchinglines RE
                      Ignores changes whose lines all match RE.
```