Technická univerzita v Košiciach
Fakulta elektrotechniky a informatiky

Návrh a implementácia jazyka stromových transformácií

Diplomová práca

2014 **Bc. Matúš Sulír**

Technická univerzita v Košiciach
Fakulta elektrotechniky a informatiky

Návrh a implementácia jazyka stromových transformácií

Diplomová práca

Študijný program:	Informatika
Študijný odbor:	Informatika
Školiace pracovisko:	Katedra počítačov a informatiky (KPI)
Školiteľ:	Ing. Slavomír Šimoňák
Konzultant:	

Košice 2014 **Bc. Matúš Sulír**

Abstrakt v SJ

Diplomová práca sa zaoberá transformáciou stromu, teda vyhľadávaním v strome podľa vzoru a jeho následnou zmenou. Nová knižnica Treepace, ktorá bola navrhnutá a implementovaná v jazyku Python, ponúka aplikačné programové rozhranie (API) a doménovo-špecifický jazyk na vyhľadávanie a náhradu v stromoch. API sa konceptuálne podobá rozhraniu k textovým regulárnym výrazom jazyka Python. Jazyk je stručný, keďže celú transformáciu je možné zapísať na jeden riadok. Ak je to možné, stratégia nahrádzania sa zvolí automaticky tak, aby bola transformácia jednoznačná. Podporovaná je obojsmerná integrácia s hostiteľským jazykom – je možné používať nielen prvky vnoreného jazyka z hostiteľského, ale i naopak. Hodnotami uzlov môžu byť ľubovoľné objekty, nie iba reťazce. Vďaka možnosti dediť z triedy uzla je možné reagovať na každú zmenu stromu počas transformácie a dosiahnuť tak mapovanie uzlov na externé objekty.

Kľúčové slová

stromy, transformácia, vzor, náhrada

Abstrakt v AJ

This master's thesis deals with tree transformation, which means searching in a tree according to a pattern, followed by its change. The new library Treepace, which was designed and implemented in Python, offers an application programming interface (API) and a domain-specific language for searching and replacing functionality for trees. The API is conceptually similar to the textual regular expression interface of the Python language. The language is terse because it is possible to write the whole transformation in one row. If it is possible, the replacing strategy is automatically selected in a way that the transformation is unambiguous. Bidirectional integration

with the host language is also supported – it is possible to use not only the embedded language elements from the host language, but also vice versa. Any objects, not only strings, can be used as values of nodes. Because of the possibility to inherit from the node class it is possible to react to each tree change during the transformation and achieve mapping of the nodes to the external objects.

Kľúčové slová v AJ

tree, transformation, pattern, replacement

TECHNICKÁ UNIVERZITA V KOŠICIACH

FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Katedra počítačov a informatiky

ZADANIE DIPLOMOVEJ PRÁCE

Študijný odbor: **9.2.1 Informatika**

Študijný program: **Informatika**

Názov práce:

Návrh a implementácia jazyka stromových transformácií

Language of tree transformation design and implementation

Študent: **Bc. Matúš Sulír**

Školiteľ: **Ing. Slavomír Šimoňák, PhD.**

Školiace pracovisko: **Katedra počítačov a informatiky**

Konzultant práce:

Pracovisko konzultanta:

Pokyny na vypracovanie diplomovej práce:

1. Štúdium a analýza súčasného stavu v oblasti stromových transformácií.
2. Návrh a implementácia jazyka stromových transformácií.
3. Zhodnotenie dosiahnutých výsledkov.
4. Vypracovanie dokumentácie podľa pokynov vedúceho práce.

Jazyk, v ktorom sa práca vypracuje: slovenský

Termín pre odovzdanie práce: 02.05.2014

Dátum zadania diplomovej práce: 31.10.2013

doc. Ing. Jaroslav Porubän, PhD.
vedúci garantujúceho pracoviska

TECHNICKÁ UNIVERZITA V KOŠICIACH FAKULTA ELEKTROTECHNIKY A INFORMATIKY -11-

prof. Ing. Liberios Vokorokos, PhD.
dekan fakulty

Čestné vyhlásenie

Vyhlasujem, že som diplomovú prácu vypracoval(a) samostatne s použitím uvedenej odbornej literatúry.

Košice 2. 5. 2014 .....

Vlastnoručný podpis

Poďakovanie

Chcel by som sa poďakovať svojej rodine za vytvorenie optimálnych podmienok počas štúdia a tvorby diplomovej práce.

Moja vďaka taktiež patrí vedúcemu práce Ing. Slavomírovi Šimoňákovi, PhD., za cenné rady a pripomienky ohľadom návrhu knižnice Treepace a samotnej práce. Za radu počas tvorby semestrálneho projektu by som sa chcel poďakovať i doc. Ing. Jaroslavovi Porubänovi, PhD.

Predhovor

O transformáciu stromov som za začal zaujímať už vrámci mojej bakalárskej práce, zaoberajúcej sa generovaním dekodérov inštrukcií a disassemblerov. Vďaka početným transformáciám stromu sa v nej zo špecifikačného tvaru súboru postupne stával vykonateľný. Transformácie však boli vykonávané manuálne, v bežnom jazyku všeobecného použitia (Java), a to návštevou jednotlivých uzlov stromu.

Napadlo mi, či by sa proces zmeny stromu nedal zapísať samostatným jazykom. Po istom čase som sa začal zahrávať s myšlienkou jazyka všeobecného použitia, kde by transformácia stromov predstavovala hlavný, či dokonca jediný spôsob vykonávania programu. To sa ukázalo byť síce zaujímavé, no veľmi nepraktické. Zvolil som teda cestu tvorby doménovo-špecifického jazyka, vnoreného do existujúceho jazyka všeobecného použitia (Python) pomocou reťazca.

Ako to už býva, tvorba tejto diplomovej práce nebola priamočiara. Vyskytlo sa viacero alternatív, z ktorých nakoniec len jedna bola realizovaná. Napríklad, syntax jazyka prešla zložitým vývojom a s prvotným návrhom už nemá veľa spoločné. Verím však, že knižnica a jazyk, ktoré budú vrámci práce predstavená, sa v praxi stanú užitočným pomocníkom pri celého radu úloh týkajúcich sa stromov a ich transformácie.

Obsah

Zoznam obrázkov

Zoznam tabuliek

Zoznam symbolov a skratiek

API Application Programming Interface

AST abstract syntax tree – abstraktný syntaktický strom

DSL Domain Specific Language (doménovo-špecifický jazyk)

HTML HyperText Markup Language

PEG Parsing Expression Grammar

SQL Structured Query Language

XML eXtensible Markup Language

Slovník termínov

Acyklický znamená neobsahujúci uzavreté cykly.

Aplikačné programové rozhranie predstavuje súhrn tried, metód, funkcií a iných prvkov jazyka, navonok dostupných pre používateľa knižnice.

Arita alebo árnosť je vo všeobecnosti počet prvkov funkcie, v tejto práci i počet dcérskych uzlov určitého uzla.

Doménovo-špecifický je určený pre určitú aplikačnú doménu, teda pre oblasť použitia.

Notifikácia značí oznámenie o zmene stavu.

Podstrom je presne vymedzená časť hlavného stromu.

Serializačný formát slúži na ukladanie údajových štruktúr z pamäti počítača napr. na disk.

Transformácia je zmena údajovej štruktúry zo vstupnej na výstupnú podľa určitých pravidiel.

Úvod

Predkladaná diplomová práca sa zaoberá transformáciami stromov, čiže premenou stromovej štruktúry zo zdrojového tvaru na cieľový na základe určitých pravidiel.

Zhrnutie niektorých princípov popísaných v tejto práci je tiež možné nájsť v článku [1].

Motivácia

So stromami sa v informatike stretávame veľmi často, niekedy si to ani neuvedomujeme. Mnoho prekladačov, ale i programov na spracúvanie počítačových jazykov všeobecne, ukladá spracovávanú vetu jazyka vo forme abstraktného syntaktického stromu – AST. Adresárová štruktúra súborových systémov, ktoré používame, tiež tvorí strom. Každá webová stránka, ktorú otvoríme, je v prehliadači reprezentovaná stromom.

Napriek takmer všadeprítomnému výskytu stromov sa tieto štruktúry často spracúvajú ručne. Hoci existujú jazyky, ktoré majú za cieľ práve transformáciu stromov, nie vždy je ich použitie vhodné.

Ako príklad si vezmime XML jazyky, napr. XSLT alebo XQuery. Drobnou nevýhodou je nutnosť zahrnúť do vytváraného programu knižnice, ktorých veľkosť častokrát presahuje program samotný. To však už v dnešnej dobe nie je taký veľký problém ako v minulosti.

Spomínané jazyky sú samostatné, čiže programy v nich zapísané je možné spúšťať bez programu v hostiteľskom jazyku. To môže byť na prvý pohľad výhoda – pokým nepožadujeme, aby boli volané z iného jazyka všeobecného použitia. Vtedy je prvoradá stručnosť, jednoduchosť a obojsmerná integrácia s hostiteľským jazykom, čo

súčasné jazyky nie vždy spĺňajú.

Hlavným nedostatkom XML transformačných jazykov je, že nemáme možnosť sledovať zmeny spôsobené vykonávaním transformačného programu. Jediným výstupom je cieľový strom. Nemôžeme teda program jednoducho ladiť postupným výpisom pridávaných a menených uzlov, graficky znázorňovať transformáciu pomocou animácie, či jednoducho premenúvať súbory v adresárovej štruktúre podľa toho, ako sa menia uzly v pamäti.

Napokon, hoci je bežné meniť objekty v jazyku všeobecného použitia na XML tak, že sa ich členské premenné vhodne konvertujú na reťazce, nemožno tvrdiť, že uzlami XML stromu sú samotné objekty.

Ciele

Ako už bolo spomenuté, pre náš vnorený jazyk sú podstatné najmä tieto štyri vlastnosti:

- stručnosť,
- všeobecnosť,
- jednoduchosť použitia API
- a obojstranná integrácia s hostiteľským jazykom.

Ideálnym príkladom stručnosti a jednoduchosti API sú regulárne výrazy. Hostiteľské jazyky vo spravidla ponúkajú funkciu, ktorá má tri parametre - pôvodný text, hľadaný vzor a náhrada. Navrhovaný jazyk má ambíciu stať sa akýmsi „regulárnym výrazom pre stromy“, i keď toto vyjadrenie nie je presné.

Pod všeobecnosťou rozumieme, že hodnotami uzlov stromu sú doslova objekty hostiteľského jazyka – nie iba ich textové reprezentácie.

Obojstranná integrácia s hostiteľským jazykom je podrobnejšie popísaná v sekcii 6.1.3.

Ďalším cieľom bude možnosť sledovania zmien uzlov počas transformácie. Jednotlivé uzly budú „namapované“ na externé objekty – textové, grafické, diskové a pod., pričom zmenou uzlu v pamäti dosiahneme automaticky zmenu externého objektu.

Knižnica, resp. jazyk sa volá Treepace – skratka slovného spojenia „Tree pattern replace“, voľne preloženého ako „nahrádzanie vzorov v strome“.

1 Formulácia úlohy

Znenie zadania je možné upresniť, podrobnejšie opísať a doplniť takto:

1. Analyzovať možnosti existujúcich systémov na transformáciu stromov, opísať ich výhody a nevýhody.

2. Vyskúšať vybrané transformačné jazyky napísaním toho istého jednoduchého programu v každom z nich.

3. Navrhnúť vnorený jazyk na transformáciu stromov v programovacom jazyku Python. Vnorenosťou sa myslí, že vetu jazyka bude možné zapísať napr. ako reťazec a zavolaním príslušnej metódy ju vykonať.

4. Zabezpečiť, aby nielen program v Pythone mohol volať programy v transformačnom jazyku, ale aby i transformačný jazyk mal možnosť volať funkcie Pythonu, čím sa dosiahne obojsmerná integrácia.

5. Zabezpečiť, aby program v Pythone vedel reagovať na každú zmenu stromu vykonanú transformačným jazykom – napr. vypísaním, grafickým znázornením a pod.

6. Overiť praktickosť použitia vytvorením programu napísaného v Pythone a novovytvorenom jazyku, ktorý bude používať transformácie stromov a reagovať na ich zmeny.

7. Vypracovať používateľskú a systémovú príručku.

2 Stromy

V informatike existuje viacero definícií pojmu strom. Aby sme mohli zadefinovať strom, je najprv potrebné oboznámiť sa s grafmi.

2.1 Grafy

Graf je dvojica $G = (V, E)$, kde V je množina vrcholov (uzlov) a E je množina hrán [2]. Ak máme dva vrcholy $v_1, v_2 \in V$, hrana v_1v_2 medzi nimi môže byť:

- neorientovaná, pokiaľ ide o neusporiadanú dvojicu, teda množinu obsahujúcu dva prvky: $\{v_1, v_2\}$ [3],
- orientovaná, ak sa jedná o usporiadanú dvojicu (v_1, v_2) [4].

V prípade vrcholovo-značeného grafu môžeme pomocou funkcie $l : V \to L$ každému uzlu priradiť dodatočnú informáciu, tzv. značku, z ľubovoľnej množiny značiek L [5] (teda i nekonečnej).

Graf je súvislý, ak medzi všetkými dvojicami uzlov a, b existuje aspoň jedna cesta vytvorená z hrán av_1, v_1v_2, ..., v_nb [6] (jednotlivé uzly sú unikátne, orientáciu hrán zanedbáme). V prípade acyklického grafu medzi ľubovoľnou dvojicou uzlov existuje najviac jedna cesta [7].

2.2 Definícia stromu

Podľa [8] je strom súvislý acyklický graf. Ak prehlásime určitý uzol za koreň, všetky hrany budú otočené smerom od koreňa (od rodičov k potomkom). Vtedy hovoríme o prirodzenej orientácii hrán [9]. Pokiaľ je určené poradie potomkov každého uzla, jedná sa o usporiadaný strom [10].

Odteraz budeme pod pojmom „strom“ implicitne rozumieť usporiadaný, vrcholovo-značený strom s určeným koreňom a prirodzenou orientáciou hrán.

Z praktického hľadiska budeme strom tiež chápať ako údajovú štruktúru, napr. v pamäti počítača, skladajúcu sa z uzlov prepojených s inými uzlami pomocou referencií. Acyklickosť nebude vynucovaná ani kontrolovaná, no budeme ju predpokladať, že programátor aplikácií ju neporuší.

2.3 Uzly stromu

Všetky uzly okrem koreňa sú pripojené práve k jednému rodičovskému uzlu [11]. Podľa [12] delíme uzly na:

- vnútorné uzly – teda uzly, ktoré majú potomkov,
- listy – uzly bez potomkov.

2.4 Arita stromu

Strom je n-árny, pokiaľ jeho uzly majú najviac n potomkov. Ak $n = 2$, ide o tzv. binárny strom [13].

Všeobecný strom nemá určené obmedzenie na počet potomkov uzlov. V ďalšom texte sa budeme zaoberať všeobecnými stromami.

2.5 Využitie stromov

Jedným z najčastejších použití je abstraktný syntaktický strom – AST. Ide o stromovú reprezentáciu vety počítačového jazyka, v ktorej sú zanedbané syntaktické detaily [14].

Stromovú povahu majú častokrát i dokumenty. Pojem dokument má dva, čiastočne prelínajúce sa významy. Prvým je súbor obsahujúci štruktúrované údaje – zoznam žiakov podľa tried, rodokmeň a pod. Druhým významom sú vizuálne zobrazované dokumenty, napríklad vo formáte HTML.

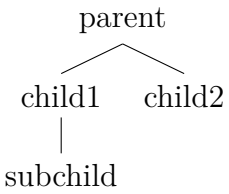
Uzly stromu môžu byť tiež „pripojené“ k externým objektom, ako sú napr. adresáre súborového systému alebo položky databázy. Táto myšlienka je podrobnejšie popísaná v kapitole 3.8.

2.6 Reprezentácia

Aby sme mohli zapísať transformácie, musíme najprv vedieť reprezentovať konkrétne stromy.

2.6.1 Grafická reprezentácia

Pri grafickom znázornení stromov sa koreňový uzol zvyčajne kreslí navrchu, dcérske uzly o úroveň nižšie. Súrodenecké uzly sa nachádzajú v jednej horizontálnej úrovni (Obrázok 2 – 1).

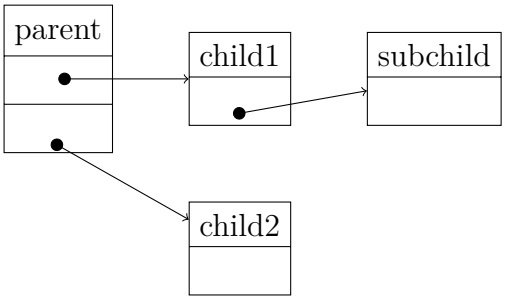


Obr. 2 – 1: Grafická reprezentácia stromu

2.6.2 Interná reprezentácia

V operačnej pamäti je možné strom intuitívne reprezentovať pomocou štruktúr obsahujúcich hodnotu uzla a zoznam pamäťových referencií na dcérske (a prípadne aj

rodičovské) uzly. Znázornenie takejto reprezentácie sa nachádza na Obrázku 2 – 2.



Obr. 2 – 2: Interná reprezentácia stromu v pamäti

2.6.3 Zátvorková syntax

Štruktúru stromu je možné v stručnej textovej podobe zapísať pomocou vyvážených zátvoriek [15]. Pridaním hodnôt uzlov dostaneme zo stromu zobrazeného na Obrázku 2 – 1 takúto textovú reprezentáciu:

```
(parent (child1 (subchild) child2))
```

Vidíme, že stručnosť prevláda na úkor prehľadnosti.

2.6.4 Zápis odsadením textu

Ak miesto zátvoriek použijeme odsadenie textu, dostaneme reprezentáciu dostatočne stručnú a zároveň prehľadnú.

```
parent
        child1
                subchild
        child2
```

Chýba však možnosť zápisu stromu na jeden riadok, ktorá sa v prípade transformácií často môže hodiť.

2.6.5 XML

Rozšírenou reprezentáciou pre ukladanie stromov v externej pamäti je XML (eXtensible Markup Language). XML je textový serializačný formát a sada štandardizovaných API (aplikačných programových rozhraní) pre prístup k dokumentom.

```
<parent>
    <child1>
        Text 1 <subchild /> text 2
    </child1>
    <child2 attribute="value" />
</parent>
```

Výhodou XML je ľahká čitateľnosť pre človeka a veľké množstvo existujúcich nástrojov určených na spracovanie XML dokumentov. Medzi nevýhody patrí veľkosť dokumentov – súbor obsahuje množstvo informácií, ktoré nie sú nevyhnutné.

Ako je možné vidieť v ukážke, XML podporuje tiež atribúty, ale i tzv. zmiešaný obsah [16], čiže uzly obsahujúce text „poprekladaný“ dcérskymi uzlami.

2.7 Informácie obsiahnuté v uzloch

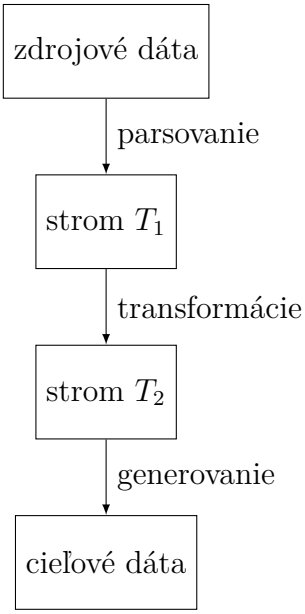
Uzly stromu môžu byť nositeľom užitočnej informácie, napr. celé či desatinné číslo, reťazec alebo štruktúra obsahujúca viacero položiek rôznych typov. Pri každom type stromu, resp. pri každej jeho reprezentácii pritom ide o iný druh a štruktúru informácie.

Napríklad, v prípade adresárov na disku môže ísť o samotný názov súboru, teda reťazec. V prípade XML atribútov, ak uvažujeme, že každý atribút je uložený ako samostatný uzol, ide o položku v tvare:

`attribute=value`

3 Transformácia stromov

Transformácia je zmena stromu zo zdrojového na cieľový podľa určitých pravidiel. Bežný prístup k transformácii stromov je znázornený na Obrázku 3 – 1.



Obr. 3 – 1: Bežný prístup k transformácii

3.1 Spôsoby transformácie

V zásade môžeme rozlíšiť dva rôzne prístupy k transformácii:

- postupnou úpravou existujúceho stromu,
- vytvorením nového (prázdneho) stromu a postupným pridávaním uzlov.

Prvý spôsob sa viac hodí, ak potrebujeme len menšie zmeny a druhý, ak väčšie, no toto pravidlo nie je striktné, keďže obidva spôsoby sú zameniteľné.

3.2 Postup pri transformácii

Transformácia stromu sa vo všeobecnosti skladá z dvoch základných krokov:

1. nájsť relevantnú časť stromu,
2. nahradiť určitú časť stromu inou časťou, prípadne vygenerovať úplne novú časť.

Tieto dva kroky sa môžu opakovať.

3.3 Využitie

Nasleduje krátky prehľad spôsobov využitia transformácie stromov.

3.3.1 Transformácia dokumentov

Často je potrebné transformovať dokumenty do inej podoby – filtrovať záznamy, vytvoriť vizuálnu podobu dokumentu a pod. Na túto úlohu sa dobre hodia XML jazyky.

3.3.2 Spracovanie jazykov a programov

Akonáhle máme strom, môžeme analyzovať vety jazykov, či už prirodzených alebo počítačových. Stromový tvar sa dobre hodí na transformácie AST a generovanie kódu.

3.3.3 Matematické operácie

Uzly predstavujú operátory a operandy matematických operácií, transformácie samotné výpočty. V stromovom tvare sa celkom dobre dajú napr. zjednodušovať výrazy, odstraňovať nadbytočné vetvy, ktoré netreba počítať a pod.

3.4 Manuálna transformácia

Ak potrebujeme ručne nájsť časť stromu podľa daného pravidla, môžeme použiť niektorý zo známych algoritmov pre problém nazvaný „subtree matching“, popísaných v [17].

3.5 Návrhový vzor Visitor

Na riešenie prechodov stromu, či už je možné využiť návrhový vzor Visitor [18]. Každý prechod stromom je reprezentovaný samostatnou triedou, pričom jednotlivé jej metódy reagujú na príslušné typy uzlov.

3.6 XML jazyky

Pokiaľ máme k dispozícii strom vo forme XML dokumentu (alebo je pre nás dostatočne výhodné ho tohto formátu previesť), môžeme použiť niektorý z bežne dostupných XML jazykov – na samotný výber podstromu XPath, na transformáciu XSLT, XQuery alebo XQuery Update.

3.6.1 XPath

XPath [19] je jazyk slúžiaci na adresovanie častí XML dokumentu podľa zadaných kritérií. Nevykonáva teda samotnú transformáciu stromov. Používa sa najmä ako jazyk vložený v inom programovacom jazyku, napr. XSLT (sekcia 3.6.3) alebo XQuery (3.6.4).

Výraz v jazyku XPath sa skladá z lokačných krokov oddelených lomkou [20], pričom každý krok pozostáva z osi a testu uzla [21]:

$$os_1 :: test_1/os_2 :: test_2/ \ldots /os_n :: test_n$$

Os predstavuje mapovanie z uzla na množinu uzlov [22], pričom tieto uzly sú k tomu prvému v určitom vzťahu – „priami potomkovia“, „nasledujúci súrodenci“ a pod. K dispozícii je celkovo 13 osí [19].

Test uzla môže v hranatých zátvorkách obsahovať kvalifikátory, teda dodatočné podmienky, ktoré musí uzol spĺňať [20]. Pre názornosť je tu uvedený príklad na XPath výraz, ktorý vyberie prvý odstavec tretej sekcie článku uloženého v XML dokumente:

```
/article/section[3]/paragraph[1]
```

3.6.2 Regular XPath

Logický základ jazyka XPath sa nazýva Core XPath [23]. Ak k nemu pridáme možnosť používať tranzitívny uzáver (*), dostaneme formalizmus Regular XPath [24]. Ten je implementovaný v jazyku XSeq [25].

Nasleduje príklad na regulárny XPath výraz vracajúci množinu všetkých ľudí, ktorí majú v rodokmeni buď otca s modrými očami, alebo (smerom od rodičov k po-

tomkom) mužského predchodcu s modrými očami a potom striedavo s hnedými a zelenými:

```
//man[@eyes=blue]/(man[@eyes=brown]/man[@eyes=green])*/man
```

Vyjadrovacie možnosti jazyka sa významne rozšíria. Regular XPath však na rozdiel od regulárnych výrazov nad reťazcami stále neobsahuje alternatívu („|“) a „syntaktický cukor“ (znak ? a +).

3.6.3 XSLT

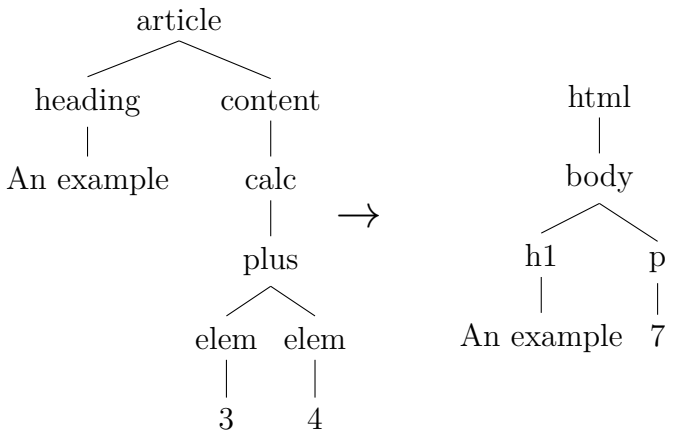
XSLT je prevažne deklaratívny jazyk, ktorý na základe šablóny vytvorí z pôvodného XML dokumentu úplne nový dokument [26], neupravuje teda pôvodný. Je tiež nutné poznamenať, že výstupom môže byť nielen strom, ale ľubovoľný text.

Medzi výhody patrí fakt, že ide o turingovsky kompletný jazyk [27]. Tým pádom je v ňom možné zapísať ľubovoľný algoritmus.

Najväčšou nevýhodou je, že i jednoduché programy sa zapisujú príliš zdĺhavo, keďže samotný program sa zapisuje vo forme XML dokumentu. Od určitej úrovne zložitosti sa XSLT programy stávajú ťažko pochopiteľnými.

Nasleduje ukážka XSLT kódu, ktorý z dokumentu obsahujúceho významové značky a jednoduchý matematický príklad vytvorí HTML dokument. Grafické znázornenie vzorových vstupných a výstupných dát tejto transformácie je na Obrázku 3 – 2

```
<xsl:stylesheet version="1.0"
        xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/article">
        <html>
                <body><xsl:apply-templates/></body>
        </html>
</xsl:template>
```



Obr. 3 – 2: Transformácia dokumentu

```
<xsl:template match="//heading">
        <h1><xsl:apply-templates/></h1>
</xsl:template>

<xsl:template match="//content">
        <p><xsl:apply-templates/></p>
</xsl:template>

<xsl:template match="//calc/plus">
        <xsl:value-of select="sum(elem)"/>
</xsl:template>
</xsl:stylesheet>
```

3.6.4 XQuery

Pomocou jazyka XQuery je možné jednak získať časť XML dokumentu, no tiež vytvoriť nový XML dokument na základe získaných výsledkov. XQuery výrazy sa tiež nazývajú FLWOR, pretože sa skladajú z nasledujúcich častí [28]:

- `for` – deklarácia premenných,
- `let` – definície premenných (zvyčajne odkazujú na časti dokumentu pomocou

XPath výrazu),

- `where` – filtrovanie výsledkov,
- `order by` – zoradenie,
- `return` – vrátenie výsledku.

3.6.5 XQuery Update

Samotné XQuery síce môže vrátiť nový XML dokument, no nedokáže modifikovať existujúci. Toto obmedzenie odstraňuje špecifikácia XQuery Update [29].

Ide jazyk obsahujúci viacero konštrukcií bežne používaných v imperatívnych jazykoch všeobecného použitia. Jeho cieľom je tak nielen doplniť, ale čiastočne i nahradiť GPL. Syntax je prehľadnejšia ako v prípade jazyka XSLT, no na stručnosť stále nie je kladený dostatočný dôraz. Na prvý pohľad nesie niektoré prvky jazyka SQL. Nasleduje ukážka zdrojového kódu pre transformáciu z Obrázka 3 – 2.

```
rename node $doc as "html";
variable $old := $doc/*;
delete node $doc/*;
insert node <body/> as first into $doc;
insert node $old as first into $doc/body;

rename node $doc//heading as "h1";
rename node $doc//content as "p";
for $i in $doc//calc return (
    replace node $i with sum($i/plus/elem)
);

$doc
```

3.7 Špecializované systémy

Teraz spomenieme niektoré transformačné jazyky vytvorené pre konkrétnu aplikačnú doménu.

3.7.1 JastAdd

JastAdd je implementáciou konceptu „Rewritable reference attributed grammars“ [30]. Jedná sa o jazyk určený predovšetkým na transformáciu AST, nie je teda dostatočne všeobecný. Navyše nespĺňa jeden z našich hlavných cieľov – stručnosť.

3.7.2 Tregex

Tregex [31] vznikol vylepšením jazyka TGrep2, zameraného na oblasť lingvistiky. Ten má za úlohu vybrať zo súboru stromov také, v ktorých sa nachádza daný vzor [32]. Jeho vzor má tvar:

$$head\ R_1\ node_1\ R_2\ node_2\ \ldots\ R_n\ node_n$$

$Head$ je vzor pre tzv. hlavný uzol. Všetky ostatné uzly $node_i$ musia byť vo vzťahu R_i k hlavnému uzlu. Pomocou zátvoriek je však možné vyjadriť aj vzťahy medzi samotnými vedľajšími uzlami [32]. Existuje až 45 rôznych vzťahov a každý je syntakticky vyjadrený pomocou interpunkčných znamienok. To spôsobuje začiatočníkom nutnosť neustále nahliadať do manuálu.

3.7.3 Tsurgeon

Tsurgeon [31] je doplnkom jazyka Tregex o možnosť manipulácie so stromami. Je teda tiež zameraný na spracovanie prirodzených jazykov.

Pri hľadaní jednotlivých uzlov pomocou Tregex výrazu existuje možnosť uložiť ich do „premennej“ s určitým menom. Výraz v jazyku Tsurgeon potom pozostáva z príkazu, napr. `relabel` alebo `delete` a argumentov: názvy uzlov, nová hodnota a pod. [31] V istom zmysle, i keď veľmi vzdialene, je podobný jazyku XQuery Update (sekcia 3.6.5).

3.8 Mapovanie uzlov na externé objekty

Určité časti stromu môžu byť „namapované“ na objekty vonkajšieho sveta. Nasleduje prehľad vybraných spôsobov využitia mapovania uzlov stromu na externé objekty.

3.8.1 Súbory

Jednotlivé uzly budú predstavovať adresáre a súbory. Transformáciou bude možné dosiahnuť napr.:

- utriedenie súborov podľa ich názvov a metadát do priečinkov a podpriečinkov,
- hromadné premenovanie na základe pravidiel,
- vyhľadávanie (filtrovanie) na základe kritérií.

V tomto prípade bude určitý podstrom namapovaný na koreňový adresár disku. Samozrejme, takýto strom nebude celý v pamäti. Čítať sa bude z disku podľa potreby, v pamäti sa budú nachádzať len určité úseky vo forme „cache“. Pri zápise sa takmer okamžite premietnu zmeny do ozajstného súborového systému.

Inou možnosťou je namapovať len konkrétny adresár – v tomto prípade je možné ho rekurzívne prečítať celý naraz (v prípade, že neobsahuje príliš veľa podadresárov).

Čiastočne podobnú myšlienku využíva systém DeepFS. Najprv špecializovaný nástroj rekurzívne prejde existujúci súborový systém a na základe neho vytvorí XML databázu v systéme BaseX. Táto databáza je kvôli možnosti rýchlejšieho prístupu indexovaná a obsahuje [33]:

- kompletný strom názvov súborov a adresárov, vrátane atribútov (prístupové práva a pod.),
- metadáta získané z obsahu súborov – napr. autor nahrávky a názov piesne pre zvukový súbor, počet strán pre dokumenty atď.,
- samotný obsah súborov.

Potom je k vytvorenej XML databáze možné pristupovať existujúcimi XML nástrojmi [34], napr. dopytovacím jazykom XQuery je možné filtrovať súbory podľa metadát, či vďaka XQuery Update ich dokonca meniť.

Okrem toho, v operačných systémoch na báze Linuxu je možné túto databázu pripojiť ako bežný súborový systém [34] a umožniť klasickým programom (nepodporujúcim XML jazyky) čítať a zapisovať dáta.

3.8.2 Stromová/grafová databáza

Ak by sme mali mapovanie na disk, mohli by sme ho použiť ako jednoduchú grafovú databázu. Zbavili by sme sa niektorých problémov súčasných programov – napr. už by nebolo potrebné žiadne objektovo-relačné mapovanie. Zmizlo by tiež rozhranie medzi jazykom všeobecného použitia a dotazovacím (SQL) jazykom. Mali by sme jeden jazyk, ktorý rieši vyhľadávanie podľa zložitých kritérií, no zároveň dokáže spracovať aplikačnú logiku.

Medzi existujúce platformy implementujúce myšlienku grafových databáz patrí Neo4j [35]. Na dopytovanie a manipuláciu grafov ponúka deklaratívny jazyk Cypher [36].

3.8.3 Vektorová grafika a GUI

Scény vo vektorovej grafike sa už dnes bežne reprezentujú stromami. Vhodne načasované transformácie sa môžu navonok prejaviť ako animácia.

Je tiež bežné reprezentovať stromom grafické používateľské rozhranie – vnorenie a stav jednotlivých komponentov.

3.8.4 Vizualizácia transformácií

Často môže byť potrebné vizualizovať priebeh transformácie, jednak kvôli výuke a jednak s cieľom ladenia transformačného programu. Jednotlivé uzly stromu teda môžu byť previazané s ich grafickými reprezentáciami. Zmeny stromu v pamäti sa postupne prejavujú napr. presunom uzla na obrazovke.

4 Tvorba doménovo-špecifických jazykov

Jazyk na transformáciu stromov možno zaradiť medzi doménovo špecifické (DSL), pretože je zameraný na jednu konkrétnu činnosť.

4.1 Spôsob kompozície

Kompozícia je spôsob spojenia programov napísaných v dvoch rozličných jazykoch do jedného programu. Podľa [37] delíme spôsoby kompozície jazykov na:

- rozšírenie,
- zjednotenie,
- samo-rozšírenie (self-extension).

V našom prípade pôjde o samo-rozšírenie, keďže rozlišujeme nadradený a podradený jazyk; a vnorený jazyk bude implementovaný v hostiteľskom bez úpravy jeho kompilátora. Vrámci tohto spôsobu rozširovania rozlišujeme:

- knižničné vnorenie, nazývané aj rýdze [38] (pure embedding),
- vnorenie pomocou reťazca [37] – „string embedding“.

4.1.1 Knižničné vnorenie

Doménovo špecifické pojmy sú v tomto prípade vyjadrené konštrukciami jazyka všeobecného použitia, teda funkciami, metódami a pod. Hypotetický príklad stromovej transformácie by mohol vyzerať takto:

```
tree.children('article').replace('html').add_child('body')
tree.descendants('heading').replace('h1')
tree.descendants('content').replace('p')
```

```
subtree = tree.descendants('calc').child('plus').children()
result = str(int(subtree.leaves[1]) + int(subtree.leaves[2]))
subtree.replace(result)
```

Jednou z výhod je jednoduchšia implementácia, keďže nie je potrebná samostatná syntaktická analýza. Na druhej strane, niektoré zápisy sú zbytočne zdĺhavé a informácie vzťahujúce sa ku konkrétnej transformácii sa môžu ľahko stratiť v záplave všeobecných pojmov.

O čosi výraznejšou výhodou je kontrola správnosti kódu počas prekladu hostiteľského programu – budeme upozornení napr. na chýbajúcu zátvorku alebo preklep v názve metódy. Avšak pri interpretovanom a dynamicky typovanom hostiteľskom jazyku sa táto výhoda takmer stráca. Po prvé, fázy prekladu a spúšťania programu už nie sú striktne oddelené. Po druhé, niektoré chyby sa často prejavia až pri vykonávaní.

4.1.2 Vnorenie reťazcom

Pri tomto spôsobe kompozície sa veta vnoreného jazyka zapíše ako reťazec v hostiteľskom jazyku a zavolá sa funkcia, ktorá vykoná príslušné operácie na základe obsahu reťazca. Nasleduje príklad:

```
tree.transform('''
  article -> html < body
  heading -> h1
  content -> p
  calc < plus < elem<{.}>, elem<{.}> -> [text(num($1) + num($2))]
''')
```

Vnorenie pomocou reťazca umožňuje používať voľnejšiu a stručnejšiu syntax. Je tiež jednoduchšie zabezpečiť, aby takýto jazyk mohol byť v budúcnosti implementovaný aj v inom GPL – napr. okrem Pythonu aj v JavaScripte – keďže nie je závislý od konštrukcií hostiteľského jazyka.

Ideálne je kombinovať knižničné a reťazcové vnorenie podľa potreby, ako tomu bude aj v našom prípade. Dôraz sa však bude klásť na programy zapísané ako reťazec.

4.2 Syntaktická analýza DSL vnorených reťazcom

Menšou nevýhodou jazykov vnorených reťazcom je nutnosť samostatnej syntaktickej analýzy.

4.2.1 Generatívne gramatiky

Klasické gramatiky, ako napr. bezkontextové, spočívajú v uvedení pravidiel, pomocou ktorých je možné generovať všetky vety daného jazyka [39].

4.2.2 Gramatiky typu PEG

Na rozdiel od bežných prístupov, gramatiky typu PEG (Parsing expression grammars) [40] sú založené na rozpoznávaní.

Medzi hlavné črty PEG patrí spojenie lexikálnej a syntaktickej gramatiky (resp. analýzy) do jedného celku.

Na rozdiel od gramatiky v tvare EBNF, kde sú možnosti alternatívy (označovanej „|“) neusporiadané, PEG využíva usporiadanú alternatívu (označovanú „/“). Tá umožňuje zápis gramatiky bez akejkoľvek nejednoznačnosti pri rozpoznávaní, keďže vybraná bude vždy prvá platná alternatíva.

5 Aplikačné programové rozhranie

Aby bolo možné používať vnorený jazyk, je potrebné definovať API, ktoré ho prepojí s hostiteľským jazykom [41].

5.1 Údajové štruktúry

Knižnica Treepace navonok poskytuje pre jej používateľov tri základné údajové štruktúry – uzol, strom a podstrom.

5.1.1 Uzol

Uzol je objekt typu `Node`. Každý uzol obsahuje hodnotu ľubovoľného typu (atribút `value`), má zoznam referencií na dcérske uzly (atribút `children`) a taktiež referenciu na rodičovský uzol (`parent`), ktorá má v prípade koreňového uzla hodnotu `None`.

Základné operácie nad uzlom sú:

- zmena hodnoty uzla,
- pridanie dcérskeho uzla na danú pozíciu (index) v zozname potomkov istého uzla,
- odpojenie uzla od jeho rodiča (javí sa ako zmazanie).

Trieda uzla obsahuje aj ďalšie užitočné metódy, no tie sú odvodené z uvedených základných operácií.

Od triedy `Node` je možné dediť a dosiahnuť tak špecifické správanie pri zmenách. Napríklad, trieda `LogNode`, obsiahnutá priamo v knižnici, vypisuje pri každej zmene

informácie v textovej podobe na štandardný výstup. Pri tvorbe odvodenej triedy stačí implementovať uvedené tri operácie, konštruktor a v prípad potreby deštruktor.

5.1.2 Strom

Strom je definovaný referenciou na koreňový uzol. Reprezentuje ho inštancia triedy `Tree`. Stromom je možné prechádzať, napr. spôsobom „pre-order“. Nad objektom stromu sa tiež vykonávajú metódy, ktoré tvoria jadro knižnice Treepace – vyhľadávanie a nahrádzanie podľa vzoru.

5.1.3 Podstrom

Často potrebujeme pracovať len s určitou, presne vymedzenou časťou stromu. Keďže strom sa reprezentuje iba referenciou na koreňový uzol, nemožno povedať, že podstrom je strom.

Preto vznikol pojem podstrom (trieda `Subtree`). Podstrom je reprezentovaný množinou referencií na uzly z hlavného stromu. V implementácii obsahuje aj referenciu na koreňový uzol podstromu.

Bolo by možné rovnakým spôsobom reprezentovať aj hlavný strom, čím by sa vylúčila nutnosť existencie samostatnej triedy pre podstrom. To by však prácu s uzlami robilo nesmierne obtiažnou – napr. pri každom pridaní uzla by sme museli určiť, do ktorých stromov bude patriť. Okrem toho, strom a podstrom majú mierne odlišnú množinu operácií. Tú časť, ktorá je spoločná, je implementovaná v triede, z ktorej strom i podstrom dedia.

5.2 Metódy na hľadanie a nahrádzanie

Rozhranie API na vyhľadávanie a nahrádzanie v strome bolo navrhnuté tak, aby sa čiastočne podobalo známym funkciám na prácu s regulárnymi výrazmi v štandardnej knižnici jazyka Python. Nad stromom je možné vykonať predovšetkým tieto metódy:

- výber podstromov, ktoré zodpovedajú vzoru:
  - kdekoľvek v strome,
  - začínajúc koreňom;
- zistenie, či celý strom zodpovedá danému vzoru,
- nahradenie podstromov zodpovedajúcich vzoru inými podstromami,
- vykonanie transformačného programu (cyklické nahrádzanie).

5.2.1 Hľadanie zhody

Na základe vzoru zapísaného vo forme reťazca hľadáme časť stromu, ktorá mu zodpovedá. Príklad vyzerá takto:

```
for match in tree.search('book < name < {.}'):
    print(match.group(1).value)
```

V strome hľadáme uzol s hodnotou „book“, ktorý má potomka „name“ a uložíme hodnotu pod-potomka do skupiny číslo 1 (skupiny sa označujú zloženými zátvorkami). Z objektu typu „zhoda“ následne túto skupinu získame. Výsledkom je podstrom – v našom prípade však len jeden uzol, ktorého hodnotu, teda názov knihy, zistíme pomocou atribútu „value“.

V prípade, že hľadaný podstrom sa musí začínať koreňovým uzlom stromu, použijeme inú metódu objektu stromu, no inak je príklad podobný:

```
for match in tree.match('books < book < name < {.}'):
    print(match.group(1).value)
```

Takto zistíme, či celý strom od koreňa k listom zodpovedá vzoru, teda či výsledkom vyhľadávania sú všetky jeho uzly:

```
if tree.fullmatch('books < book < . < . '):
        print('The whole tree matches the pattern.')
```

5.2.2 Nahrádzanie

Kľúčovou schopnosťou navrhovaného jazyka, resp. knižnice, je nahrádzanie častí stromu inými podstromami. Takto prebieha náhrada všetkých zhôd podľa jedného pravidla:

```
tree.replace('{book < author} < Shakespearw', '$1 < Shakespeare')
```

Nájdený uzol „book“ spolu s potomkom „author“ sa uloží do skupiny číslo 1. Pri náhrade tieto dva uzly zostanú nezmenené – odkazujeme na ne pomocou spätnej referencie „$1“. Nahradí sa iba dcérsky uzol, v ktorom sa opraví preklep.

Metóda `replace` nemusí mať ako druhý argument reťazec. Je možné použiť objekt reprezentujúci funkciu, ktorá sa zavolá pre každý nájdený podstrom (tzv. callback).

Napokon, najzložitejšou operáciou je vykonanie transformačného programu, ktorý môže pozostávať z viacerých pravidiel.

```
tree.transform('''
 pattern_1 -> replacement_1
 pattern_2 -> replacement_2
''')
```

Na rozdiel od samotného nahrádzania, každé pravidlo sa bude vykonávať dovtedy, kým bude nachádzaná zhoda. Navyše, vykonávanie celého zoznamu pravidiel sa bude

opakovať, kým aspoň v jednom pravidle bude nájdená zhoda.

5.2.3 Dodanie hodnôt z hostiteľského jazyka

Častokrát potrebujeme hľadať alebo nahrádzať hodnoty, ktoré nie sú konštantné, ale pochádzajú zo vstupu od používateľa. Naivným riešením je spájanie reťazcov:

```
value = input()
treee.search('name < "' + value + '"')
```

S ním je však spojené bezpečnostné riziko. Preto knižnica Treepace ponúka možnosť dodať vnorenému jazyku premenné pomocou parametrov metód. Jazyk Python podporuje pomenované argumenty, preto je použitie elegantné:

```
value = input()
treee.search('name < [val]', val=value)
```

Pomenované argumenty sú akceptované všetkými metódami na vyhľadávanie a nahrádzanie, resp. transformáciu v strome.

5.3 Doplnkové funkcie

Knižnica Treepace podporuje tiež načítanie a ukladanie stromov do formátov popísaných v sekcii 2.6. Konkrétne ide o zátvorkovú syntax, zápis odsadením textu a XML.

5.3.1 Prevod XML do stromu

Prevod z XML do stromu prináša niekoľko problémov, keďže XML okrem samotných značiek, ktoré vytvárajú čisto stromovú štruktúru, obsahuje aj:

- atribúty,

- textové hodnoty,
- zmiešaný obsah, čiže značky ľubovoľne premiešané s textom.

Vidíme to na nasledujúcom príklade:

```
<shop>
    <item price="4" material="plastic">
        Some <b>textual</b> description.
    </item>
</shop>
```

Výsledkom prevodu je údajová štruktúra (v notácii jazyka Python):

```
Node("shop", [
    Node("item", [
        Node({'price': "4"}),
        Node({'material': "plastic"}),
        Node({'xmltext': "Some "}),
        Node("b", [
            Node({'xmltext': "textual"})]),
        Node({'xmltext': " description."})])])
```

Samotné značky sú uložené ako uzly s textovou hodnotou. Každý atribút sa uloží ako uzol obsahujúci slovník (asociatívne pole) s jednou položkou. Text je uložený ako slovník s položkou obsahujúcou kľúč „xmltext“.

5.3.2 Prevod ostatných formátov

Pri prevode medzi zátvorkovou syntaxou, resp. odsadeným textom a údajovou štruktúrou v knižnici Treepace obsahujú všetky uzly ako hodnotu textový reťazec. V prípade prevodu odsadeného textu na uzly sa typ odsadenia (tabulátory alebo istý počet medzier) automaticky rozpozná.

6 Transformačný jazyk

Teraz pristúpime k návrhu samotného vyhľadávacieho a transformačného jazyka, čiže reťazca, na základe ktorého sa budú vyhľadávať a nahrádzať časti stromov. Transformačné pravidlo má vo všeobecnosti tvar:

```
vzor -> náhrada
```

Na ľavej strane je reťazec, ktorý tiež bude môcť byť parametrom vyhľadávacích metód (`search` a pod.).

Postupne budú uvádzané čo najjednoduchšie príklady znázorňujúce jednotlivé stavebné prvky a funkcionality jazyka. Nakoniec bude uvedená gramatika vo forme PEG.

6.1 Vzor pre jeden uzol

Najprv musíme definovať, kedy vzor pre jeden uzol zodpovedá konkrétnemu uzlu v strome.

6.1.1 Akákoľvek hodnota

Najzákladnejší vzor, ktorému zodpovedá uzol s akoukoľvek hodnotou, je bodka: „`.`“ – analogicky k ľubovoľnému znaku v regulárnych výrazoch. Typ hodnoty sa nerozlišuje.

6.1.2 Text

Jednoduchým vzorom je textový reťazec. Vyhľadáva sa uzol, ktorého textová reprezentácia (výsledok volania funkcie `str` nad jeho hodnotou) sa presne rovná reťazcu

uvedenému vo vzore. V prípade, že hľadaný text neobsahuje len alfanumerické znaky a podčiarnik, musí sa uviesť v úvodzovkách. Príklady:

```
a_node9
324
"Some text"
"6.7"
```

Vidíme, že čísla sú v tomto prípade pred porovnávaním prevedené na reťazec.

6.1.3 Kód

Vďaka tomu, že jazyk Python obsahuje funkciu `eval` slúžiacu na vykonanie ľubovoľného kódu, test uzla môže byť akýkoľvek výraz v tomto jazyku, ktorý sa dá pretypovať na hodnotu typu `bool`. V jazyku Treepace umiestňujeme kód do hranatých zátvoriek:

```
[log(3 + 0.5) > 1]
```

Aby bola táto funkcionalita užitočná, musíme sa vedieť odvolať na aktuálny, práve testovaný uzol. Funkcia `eval` ponúka možnosť definovať prostredie pozostávajúce z premenných, ktoré budú pri vykonávaní kódu dostupné. Knižnica Treepace do tohto prostredia automaticky umiestni premennú `node`, ktorá obsahuje referenciu na aktuálny uzol. Tým je možné vytvoriť predikát:

```
[log(node.value + 0.5) > 1]
```

Keďže konštrukcia `node.value` sa používa pomerne často, existuje možnosť zapísať ju skrátene pomocou podčiarnika:

```
[log(_ + 0.5) > 1]
```

Samozrejme, je možné tvoriť i zložitejšie predikáty:

```
["substring" in _.lower() and len(_) == 15]
```

Keďže v jazyku Python je funkcia tiež objektom, s výhodou môžeme využiť dodávanie hodnôt z hostiteľského jazyka, popísané v sekcii 5.2.3. Nasleduje ukážka, v ktorej definujeme vlastnú funkcia a vzápätí ju použijeme vo vzore jazyka Treepace.

```
def even(num):
    return num % 2 == 0
tree.search('id < [even(_)]', even=even)
```

Pre skrátenie zápisu môžeme použiť lambda-výrazy:

```
tree.search('id < [even(_)]', even=lambda x: x % 2 == 0)
```

Pri testovaní pomocou predikátu sa najviac prejavuje všeobecnosť jazyka Treepace, keďže jednotlivé uzly môžu byť napr. objekty reprezentujúce súbory a vzor môže vybrať všetky tie, ktoré sú práve otvorené na čítanie.

6.2 Vzor pre viacero uzlov

Keďže ide o stromový jazyk, vzor zvyčajne obsahuje viacero uzlov, ktoré sú navzájom v určitom rozpoložení.

6.2.1 Teoretický základ

Inšpirujeme sa jazykom XPath (sekcia 3.6.1), z ktorého vylúčime kvalifikátory (resp. ich zlúčime s testami uzlov) a ďalšie rozšírené vlastnosti.

Predpokladáme, že máme k dispozícii:

- množinu všetkých uzlov stromu, nazvanú V,
- testy uzlov – unárne predikáty nad uzlom t_i (napr. porovnanie hodnoty uzla s konštantou),

- binárne relácie medzi uzlami: $R_{i,j}$, napríklad „uzly x, y, kde x má nepriameho potomka y“.

Ak zanedbáme niektoré syntaktické detaily, môžeme povedať, že každý výraz (vzor) má tvar:

$$t_1,\ R_{1,2},\ t_2,\ R_{2,3},\ t_3,\ R_{3,4},\ \ldots\ t_{n-1},\ R_{n-1,n},\ t_n. \tag{6.1}$$

Výsledkom jeho vykonania je množina všetkých takých uzlov $u \in V$, že $t_n(u)$ je pravdivé a zároveň platí:

$$\forall i(1 \leq i < n) : t_i(v_i) \wedge (v_i, w_i) \in R_{i,i+1} \wedge t_{i+1}(w_i), \quad v_i, w_i \in V. \tag{6.2}$$

Podstatou je, že XPath výraz:

- vracia množinu uzlov zodpovedajúcu poslednému testu (t_n)
- a os (relácia) sa vzťahuje vždy k množine uzlov zodpovedajúcich testu, ktorý os predchádza.

Prvá spomínaná vlastnosť nám nevyhovuje – v našom jazyku potrebujeme vrátiť všetky uzly vo výraze, ktoré budú dohromady tvoriť súvislý podstrom. Výraz však stále bude mať tvar (6.1) a podmienka (6.2) bude tiež splnená.

6.2.2 Syntax relácií

Navrhovaný jazyk bude podporovať nasledujúce vzťahy (relácie):

- uzol a má priameho potomka b: `a < b`,
- uzol a má rodiča b: `a > b`,
- uzol a má priamo nasledujúceho súrodenca b: `a, b`,
- uzol a má súrodenca b: `a & b`.

Základom je priamy vzťah rodič-potomok, zapisovaný pomocou ľavej ostrej zátvorky.

```
parent < child
root < . < subchild
```

V nasledujúcom príklade je A priamym potomkom uzla B; C je priamo nasledujúcim súrodencom uzla B.

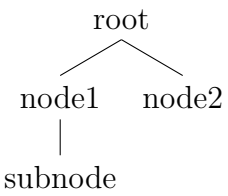
```
A < B, C
```

Vzťah, keď má byť uzol ľubovoľným súrodencom daného uzla (nie však sebou samým), zapisujeme pomocou znaku „&“:

```
A < B & C
```

6.2.3 Relácia „rodič“

Predpokladajme, že chceme nájsť podstrom znázornený na Obrázku 6 – 1.



Obr. 6 – 1: Príklad hľadaného podstromu

Zápis v jazyku Treepace by mohol byť:

```
root < node1 < subnode > ., node2
```

Všimnime si, že za reláciou „rodič“ (`>`) sa prakticky vždy zapisuje bodka (ľubovoľný uzol), ktorá je pomerne mätúca. Lepší zápis je:

```
root < node1 <subnode>, node2
```

Medzi reláciu „rodič“ a bezprostredne za ňou nasledujúcu reláciu teda budeme implicitne vkladať test uzla „`.`“. Pôvodná syntax, teda `child > parent` bude zároveň zakázaná, keďže je nahraditeľná výrazom `parent < child`.

6.2.4 Skupiny a spätné referencie

Analogicky ako v mnohých implementáciách regulárnych výrazov, i v jazyku Treepace je možné používať tzv. skupiny a spätné referencie. Skupina sa označuje zloženými zátvorkami a je automaticky číslovaná, začínajúc jednotkou. Reprezentovaná je podstromom, teda súvislou množinou uzlov s jedným koreňom. Skupiny sa môžu vnárať. V nasledujúcom príklade sa podstrom `a < b` uloží do skupiny č. 1 a podstrom obsahujúci jediný uzol `b` do skupiny 2.

```
{a < {b}}, c
```

Skupina č. 0 je vždy implicitne zaznamenávaná a predstavuje celý nájdený podstrom.

Pomocou spätných referencií, zapisovaných znakom doláru nasledovaným číslom, je možné odkazovať na už uložené skupiny. Spätné referencie vo vyhľadávacom vzore majú najväčší význam pri použití predikátov.

```
{[_ != "a"] < [_ != "b"]}, $1
```

Uvedenému vzoru zodpovedá podstrom z Obr. 6 – 2a, nie však už podstrom na Obrázku 6 – 2b.

(a) Zodpovedajúci podstrom (b) Nezodpovedajúci podstrom

Obr. 6 – 2: Príklad na spätné referencie vo vzore

Zložitejšie vzťahy medzi rôznymi uzlami vrámci vyhľadávaného stromu je možné vyjadriť odvolaním sa na skupinu vrámci predikátu v jazyku Python:

```
{.} <  [cos(_) >= sin(group(1).root.value)]
```

Často používaný zápis `group(n).root.value` má vnútri zdrojového kódu predikátu ekvivalent `$n`:

```
{.} <  [cos(_) >= sin($1)]
```

6.3 Náhrada

Ak zanedbáme detaily, náhrada má podobný tvar ako vzor, ale s drobnými obmenami:

$$v_1,\ R_{1,2},\ v_2,\ R_{2,3},\ v_3,\ R_{3,4},\ \ldots\ v_{n-1},\ R_{n-1,n},\ v_n,$$

pričom každé v_i predstavuje hodnotu novovytvoreného uzla u_i a $R_{i,i+1}$ sú relácie, ktoré musia platiť medzi uzlami u_i a u_{i+1}.

6.3.1 Uzly

Ako hodnotu nového uzla, ktorým sa má nahradiť pôvodný uzol, môžeme uviesť konštantu – či už bez úvodzoviek, alebo s nimi – podobne, ako v prípade vzoru.

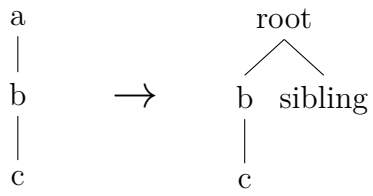
Je tiež možné použiť zdrojový kód v jazyku Python. Hodnota, ktorú výraz vráti, sa uloží ako hodnota práve vytváraného uzla. Nasleduje príklad reťazca náhrady.

```
[abs($1) + 4]
```

V prípade použitia spätných referencií na nájdené podstromy, ktoré pozostávajú z viacerých uzlov, sa predchádzajúca i nasledujúca relácia vzťahuje ku koreňovému uzlu skupiny.

```
tree.replace('a < {b < c}', 'root < $1, sibling')
```

Grafické znázornenie tohto nahradenia je na Obrázku 6 – 3.



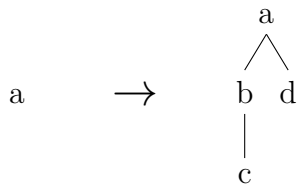
Obr. 6 – 3: Použitie spätnej referencie v náhrade

6.3.2 Relácie

V náhrade je možné používať reláciu „mať potomka“ (`<`) a „mať nasledujúceho súrodenca“ (`,`). Znak `>` nastaví aktuálny uzol na rodiča momentálne aktívneho uzla – zmení teda uzol, na ktorý sa bude vzťahovať nasledujúca relácia. Príklad:

```
tree.replace('a', 'a < b <c>, d')
```

Efekt je zobrazený na Obr. 6 – 4



Obr. 6 – 4: Použitie relácií v náhrade

6.4 Gramatika

Gramatika celej transformácie je:

Rule ⟵ *Pattern* '`->`' *Replacement*

Keďže v prípade PEG gramatík je potrebné brať do úvahy aj tzv. biele znaky ako medzery a tabulátory, zadefinujeme pomocné pravidlo:

$$\_ \longleftarrow (\text{' '} / \text{'\textbackslash t'})*$$

Vzor má gramatiku:

$$\begin{aligned}
Pattern &\longleftarrow Group\ (RelGroup)* \\
Group &\longleftarrow Node\ /\ (GroupStart\ Pattern\ GroupEnd) \\
RelGroup &\longleftarrow (Relation\ Group)\ /\ ParentAny \\
Node &\longleftarrow Any\ /\ Constant\ /\ Code\ /\ Reference \\
Any &\longleftarrow _\ \text{'.'}\ _ \\
Constant &\longleftarrow _\ ((\text{'\textbackslash w'}+)\ /\ (\text{'"'}\ (!\ \text{'"'}\ .)+\ \text{'"'}))\ _ \\
Code &\longleftarrow _\ \text{'['}\ PythonCode\ \text{']'}\ _ \\
PythonCode &\longleftarrow ExprPart+ \\
ExprPart &\longleftarrow (\ !\ (\text{'['}\ /\ \text{']'})\ .)+\ /\ (\text{'['}\ ExprPart\ \text{']'}) \\
Reference &\longleftarrow _\ \text{'\$'}\ ReferenceNum\ _ \\
ReferenceNum &\longleftarrow \text{'\textbackslash d'}+ \\
GroupStart &\longleftarrow _\ \text{'\{'}\ _ \\
GroupEnd &\longleftarrow _\ \text{'\}'}\ _ \\
Relation &\longleftarrow Child\ /\ Sibling\ /\ NextSibling \\
Child &\longleftarrow _\ \text{'<'}\ _ \\
Sibling &\longleftarrow _\ \text{'\&'}\ _ \\
NextSibling &\longleftarrow _\ \text{','}\ _ \\
ParentAny &\longleftarrow _\ \text{'>'}\ _
\end{aligned}$$

Napokon, náhrada:

$$
\begin{aligned}
Replacement &\longleftarrow ReplNode\ (ReplRelNode)* \\
ReplNode &\longleftarrow Constant\ /\ Code\ /\ Reference \\
ReplRelNode &\longleftarrow (ReplRelation\ Node)\ /\ ParentAny \\
ReplRelation &\longleftarrow Child\ /\ NextSibling
\end{aligned}
$$

7 Vykonávanie transformačného programu

V tejto kapitole bude jednak podrobnejšie popísaný efekt nahrádzania a taktiež opísaný postup hľadania vzoru a vykonania transformácie.

7.1 Základný postup

Začiatok postupu je spoločný pre všetky metódy:

1. Syntaktická analýza reťazca reprezentujúceho vzor, náhradu, resp. transformáciu.
2. Prevod AST na zoznam, resp. zoznamy inštrukcií.
3. Vykonanie inštrukcií na vyhľadávacom stroji. Výsledkom je zoznam nájdených podstromov.

Ak navyše chceme nájdené výsledky nahradiť (metóda `replace`), pokračuje sa takto:

4. Zistenie, či sa nájdené podstromy neprekrývajú. Ak áno, vyvolá sa výnimka.
5. Pre každý nájdený podstrom:
   (a) pomocou stroja na tvorbu náhrady vytvorí nový podstrom
   (b) a starý sa ním nahradí.

V prípade transformácie (metóda `transform`) sa kroky 3, 4 a 5 vykonávajú cyklicky.

7.2 Syntaktická analýza

Na syntaktickú analýzu je použitá knižnica Parsimonious, dostupná v balíčkovacom systéme Pythonu. Za pomoci PEG gramatiky zapísanej formou reťazca sa vďaka

nej zo vstupného textu v jazyku Treepace (vzor, náhrada alebo obe súčasti) vytvorí abstraktný syntaktický strom.

7.3 Prevod AST na inštrukcie

Keďže forma abstraktného syntaktického stromu nie je vhodná na priame vykonávanie, prevedie sa do formy zoznamu virtuálnych, pomerne vysokoúrovňových inštrukcií.

7.3.1 Inštrukcie vyhľadávacieho stroja

Nasleduje stručný prehľad inštrukcií vyhľadávacieho stroja.

- FIND p vyhľadáva uzol, ktorý zodpovedá predikátu p.
- REL r nastavuje typ aktuálnej relácie na r. Nasledujúca inštrukcia typu FIND ho potom využije.
- GRPS n predstavuje začiatok skupiny s číslom n,
- GRPE n koniec skupiny n.
- REF n je spätná referencia na skupinu číslo n.

Proces prevodu je nasledovný. Na začiatku sa vytvorí:

- prázdny zoznam inštrukcií,
- počítadlo začiatku skupiny $s = 0$,
- množina čísel skupín, ktoré už skončili: $E = \{\}$.

Uzly syntaktického stromu sa navštívia metódou „post-order“. Schéma $\mathcal{T}_S$ znázorňuje preklad z AST do zoznamu inštrukcií. Pri návšteve uzla typu uvedeného v ľavej časti pravidla sa vykoná pseudokód uvedený v pravej časti. Ak je za typom uzla uvedená hodnota v ostrých zátvorkách, jedná sa o text, ktorý bol získaný pri syntaktickej analýze. Funkcia „add“ pridá danú inštrukciu na koniec zoznamu inštrukcií.

$$\mathcal{T}_S[\![\ \textbf{Any}\ ]\!] = \boxed{\text{add(FIND 'True')}}$$

$$\mathcal{T}_S[\![\ \textbf{Constant}\langle c\rangle\ ]\!] = \boxed{\text{add(FIND 'str(\_) == str(}c\text{)')}}$$

$$\mathcal{T}_S[\![\ \textbf{PythonCode}\langle c\rangle\ ]\!] = \boxed{\text{add(FIND } c\text{)}}$$

$$\mathcal{T}_S[\![\ \textbf{ReferenceNum}\langle n\rangle\ ]\!] = \boxed{\begin{array}{l}\text{if } n \notin \text{E:}\\ \quad\quad \text{raise Exception('group not yet ended')}\\ \text{add(REF } n\text{)}\end{array}}$$

$$\mathcal{T}_S[\![\ \textbf{GroupStart}\ ]\!] = \boxed{\begin{array}{l}\text{s} := \text{s} + 1\\ \text{add(GRPS s)}\end{array}}$$

$$\mathcal{T}_S[\![\ \textbf{GroupEnd}\ ]\!] = \boxed{\begin{array}{l}\text{e} := \max(\{1, 2, \ldots, \text{s}+1\} - \text{E})\\ \text{E} := \text{E} \cup \{\text{e}\}\\ \text{add(GRPE e)}\end{array}}$$

$$\mathcal{T}_S[\![\ \textbf{Child}\ ]\!] = \boxed{\text{add(REL child)}}$$

$$\mathcal{T}_S[\![\ \textbf{Sibling}\ ]\!] = \boxed{\text{add(REL sibling)}}$$

$$\mathcal{T}_S[\![\ \textbf{NextSibling}\ ]\!] = \boxed{\text{add(REL next\_sib)}}$$

$$\mathcal{T}_S[\![\ \textbf{ParentAny}\ ]\!] = \boxed{\begin{array}{l}\text{add(REL parent)}\\ \mathcal{T}_S[\![\ \textbf{Any}\ ]\!]\end{array}}$$

7.3.2 Inštrukcie stroja na tvorbu náhrady

Inštrukciami stroja na tvorbu náhrady sú:

- ADD c pridá do vytváraného stromu nový uzol s hodnotou, ktorá je výsledkom vykonania výrazu c.
- AREL r – nastavuje reláciu, ktorá sa použije v nasledujúcej inštrukcii ADD.

- AREF n pridá do stromu podstrom uložený v skupine číslo n.
- GPAR nastaví aktuálny uzol na rodiča aktuálneho uzla.

Schéma prekladu $\mathcal{T}_R$ je jednoduchšia. Na pravej strane je priamo uvedená generovaná inštrukcia. Význam ostatných konštrukcií je rovnaký, ako v prípade schémy $\mathcal{T}_S$.

$$\begin{aligned}
\mathcal{T}_R[\![\, \textbf{Constant}\langle c\rangle \,]\!] &= \boxed{\text{ADD repr}(c)} \\
\mathcal{T}_R[\![\, \textbf{PythonCode}\langle c\rangle \,]\!] &= \boxed{\text{ADD } c} \\
\mathcal{T}_R[\![\, \textbf{ReferenceNum}\langle n\rangle \,]\!] &= \boxed{\text{AREF } n} \\
\mathcal{T}_R[\![\, \textbf{Child} \,]\!] &= \boxed{\text{AREL child}} \\
\mathcal{T}_R[\![\, \textbf{NextSibling} \,]\!] &= \boxed{\text{AREL next_sib}} \\
\mathcal{T}_R[\![\, \textbf{ParentAny} \,]\!] &= \boxed{\text{GPAR}}
\end{aligned}$$

7.4 Hľadanie podľa vzoru

Vyhľadávanie v strome prebieha vykonávaním inštrukcií virtuálneho stroja navrhnutého na tento účel.

7.4.1 Abstraktný vyhľadávací stroj

Abstraktný stroj hľadajúci v strome pozostáva zo zoznamu prehľadávacích vetiev. Každá vetva je pätica:

$$(\mathit{groups}, \mathit{matches}, \mathit{node}, \mathit{rel}, \mathit{instrs})$$

Význam jednotlivých položiek:

- *groups* – množina čísel skupín, v ktorých sa momentálne vrámci vzoru nachádzame,

- *matches* – zoznam nájdených podstromov – pre každú skupinu jeden,
- *node* – aktuálny (kontextový) uzol,
- *rel* – aktuálna relácia,
- *instrs* – zoznam inštrukcií, ktoré sa ešte nevykonali.

7.4.2 Inicializácia

Pred vyhľadávaním v strome (vrámci metódy `search`, `replace` alebo `transform`) bude stroj obsahovať jednu vetvu, inicializovanú na hodnotu:

$$(\{0\}, (\{\}), \mathit{root}, \mathsf{descendant}, \mathit{instructions})$$

Skupina č. 0 predstavuje celý nájdený podstrom. Zoznam nájdených podstromov obsahuje jednu prázdnu množinu, resp. prázdny podstrom – pre skupinu 0. Uzol *root* je koreňový uzol, descendant je relácia „nepriamy potomok alebo uzol samotný“. Zoznam *instructions* obsahuje všetky inštrukcie získané z daného AST (ktorý bol získaný zo vzoru).

V prípade metód `match` a `fullmatch` bude situácia rovnaká, no aktuálna relácia sa nastaví na id, teda identitu.

7.4.3 Algoritmus vyhľadávania

Pre každú vetvu, ktorá ešte obsahuje inštrukcie, sa odstráni zo zoznamu a následne vykoná prvá inštrukcia. Ak ide o inštrukciu FIND, aktuálna vyhľadávacia vetva sa nahradí vetvami zo zoznamu vetiev, ktoré inštrukcia vrátila. Proces sa opakuje dovtedy, kým existuje aspoň jedna vetva s neprázdnym zoznamom inštrukcií.

Výsledok vyhľadávania bude uložený v zoznamoch *matches* jednotlivých vetiev.

7.4.4 Význam vyhľadávacích inštrukcií

Schéma vykonávania $\mathcal{E}_S$ pomocou pseudokódu vyjadruje význam vykonávania jednotlivých typov inštrukcií. Aktuálna vetva je uložená v premennej „branch“.

$\mathcal{E}_S[\![\mathsf{FIND}\ p]\!] =$

```
branches := [ ]
∀v : ((branch.node, v) ∈ branch.rel) ∧ p(v):
    new := copy(branch)
    ∀g ∈ new.groups:
        new.match[g] := new.match[g] ∪ {v}
    new.node := v
    branches := branches + [new]
return branches
```

$\mathcal{E}_S[\![\mathsf{REL}\ r]\!] =$ `branch.relation := r`

$\mathcal{E}_S[\![\mathsf{GRPS}\ n]\!] =$

```
branch.groups := branch.groups ∪ {n}
branch.match := branch.match + [{}]
```

$\mathcal{E}_S[\![\mathsf{GRPE}\ n]\!] =$ `branch.groups := branch.groups − {n}`

$\mathcal{E}_S[\![\mathsf{REF}\ n]\!] =$

```
instrs := generate(branch.match[n])
branch.instrs := branch.instrs + instrs
```

Funkcia „generate“ vygeneruje z podstromu taký zoznam inštrukcií, ktorý vyhľadá zhodný podstrom.

7.5 Tvorba náhrady

Keď máme nájdené všetky výskyty vzoru, môžeme pokračovať tvorbou stromov, ktoré sa použijú ako náhrady za staré podstromy.

7.5.1 Abstraktný stroj na tvorbu náhrady

Abstraktný stroj, ktorý slúži na tvorbu nového stromu, je pätica:

$$(match, instrs, node, rel, tree)$$

Význam jednotlivých položiek je nasledovný:

- *match* – konkrétna nájdená zhoda, čiže zoznam podstromov,
- *instrs* – zoznam inštrukcií, ktoré sa ešte nevykonali,
- *node* – aktuálny (kontextový) uzol,
- *rel* – aktuálna relácia,
- *tree* – vytváraný strom.

7.5.2 Algoritmus tvorby náhrady

Na začiatku sa inicializuje zhoda (*match*) z údajov získaných pomocou vyhľadávacieho stroja. Zoznam inštrukcií *instrs* sa získal prekladom z AST. Zvyšné položky majú nulovú hodnotu (`None`). Následne sa začnú vykonávať inštrukcie.

7.5.3 Význam inštrukcií

Pre inštrukcie stroja na tvorbu náhrady je definovaná schéma vykonávania $\mathcal{E}_R$. Virtuálny stroj je dostupný prostredníctvom premennej „vm“.

$$\mathcal{E}_R[\![\ \mathsf{ADD}\ c\ ]\!] = \boxed{\begin{array}{l}\text{node := Node}(c\text{(vm.match)})\\ \text{if vm.tree = None:}\\ \quad\text{vm.tree := Tree(node)}\\ \text{else:}\\ \quad\text{build\_relation(vm.rel, vm.node, node)}\\ \text{vm.node := node}\end{array}}$$

$$\mathcal{E}_R[\![\ \mathsf{AREL}\ r\ ]\!] = \boxed{\text{vm.rel := } r}$$

$$\mathcal{E}_R[\![\ \mathsf{AREF}\ n\ ]\!] = \boxed{\begin{array}{l}\text{tree := to\_tree(vm.match[}n\text{])}\\ \text{if vm.tree = None:}\\ \quad\text{vm.tree := tree}\\ \text{else:}\\ \quad\text{build\_relation(vm.rel, vm.node, tree.root)}\\ \text{vm.node := tree.root}\end{array}}$$

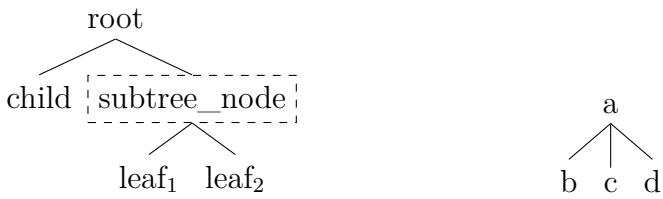
$$\mathcal{E}_R[\![\ \mathsf{GPAR}\ ]\!] = \boxed{\text{vm.node := vm.node.parent}}$$

Funkcia „build_relation“ pridá uzol k danému uzlu tak, aby medzi nimi vznikla určená relácia.

7.6 Nahradenie stromu novým stromom

Teraz je už možné pristúpiť k samotnému nahradeniu nájdených podstromov novými.

Nie každý podstrom je však možné nahradiť ľubovoľným iným stromom. Predpokladajme, že v strome na Obrázku 7 – 1a bol nájdený podstrom obsahujúci jediný uzol, zvýraznený rámčekom. Tento podstrom chceme nahradiť stromom z Obr. 7 – 1b. Nie je vôbec jasné, ako by mal vyzerať výsledný strom. Otázne je predovšetkým výsledné umiestnenie uzlov b, c, d, leaf_1 a leaf_2.



(a) Strom s nájdeným podstromom (b) Nejednoznačná náhrada

Obr. 7 – 1: Príklad nejednoznačnosti náhrady

Podobných prípadov existuje nespočetné množstvo. Knižnica Treepace preto postupuje nasledovne. Je v nej definovaný zoznam stratégií, pričom každá stratégia pozostáva:

- z testu, či je možné vykonať ju nad daným pôvodným stromom+podstromom a náhradou s jednoznačným efektom
- a z algoritmu na samotné vykonanie nahradenia.

Stratégie sú usporiadané podľa priority od najvyššej po najnižšiu. Postupne sa testujú a keď sa nájde použiteľná, nahradenie sa vykoná podľa nej. Ak sa nenájde žiadna vyhovujúca, vyvolá sa výnimka.

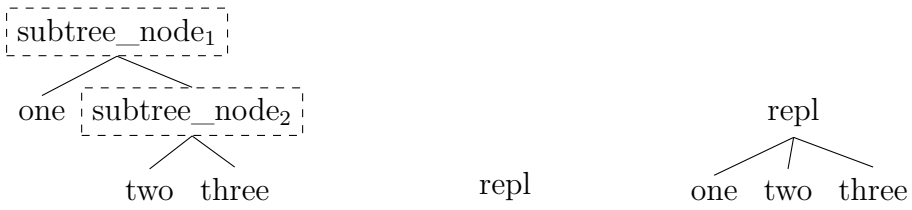
V aktuálnej verzii knižnice Treepace je definovaných päť stratégií, ktoré budú teraz stručne popísané.

7.6.1 Rovnaký tvar

Najjednoduchší prípad je, keď majú starý podstrom a nový strom rovnaký tvar, teda každý uzol v podstrome má rovnaký počet potomkov ako v náhrade. Vtedy sa postupne nastavia hodnoty uzlov v podstrome na zodpovedajúce hodnoty z nového stromu.

7.6.2 Jeden uzol

V tomto prípade je náhradou strom pozostávajúci z jediného uzla. Tento uzol bude mať po vykonaní nahradenia všetky dcérske uzly pôvodného podstromu. Táto stratégia je znázornená na Obrázku 7 – 2. Podstrom obsahujúci uzly subtree_node$_1$ a subtree_node$_2$ je nahradený uzlom „repl“.

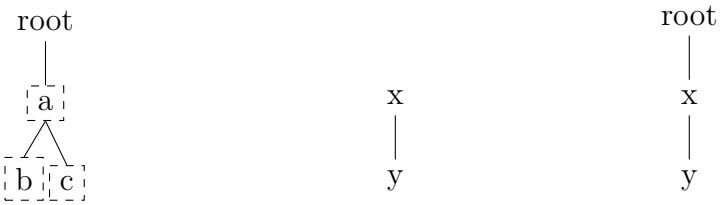


(a) Strom s nájdeným podstromom (b) Náhrada (c) Výsledok

Obr. 7 – 2: Nahradenie jedným uzlom

7.6.3 Žiadni potomkovia mimo podstromu

Túto stratégiu je možné aplikovať, ak žiaden uzol starého podstromu nemá vrámci hlavného stromu potomka, ktorý nepatrí do podstromu. Vtedy sa koreňový uzol podstromu jednoducho odpojí a nahradí sa koreňovým uzlom nového stromu. Príklad je na Obrázku 7 – 3.



(a) Strom s nájdeným podstromom (b) Náhrada (c) Výsledok

Obr. 7 – 3: Žiadni potomkovia mimo podstromu

7.6.4 Rovnaký počet listov

Ak má starý podstrom a nový strom rovnaký počet listov, potomkovia každého listu pôvodného podstromu sa stanú potomkami zodpovedajúceho listu nového stromu.

Navyše musí byť splnená podmienka, že vnútorné uzly podstromu (vrátane koreňa) nemajú potomkov mimo podstromu.

7.6.5 Rovnaký počet čiastočných listov

Listy podstromu, ktoré však majú v hlavnom strome strome potomkov, nazvime čiastočnými listami.

Ak sa počet čiastočných potomkov nahrádzaného podstromu rovná počtu listov nového stromu, môžeme použiť túto stratégiu. Spôsob nahrádzania je podobný ako v predchádzajúcom prípade, no potomkovia sa pripájajú len k čiastočným listom. Spomínaná dodatočná podmienka musí byť tiež splnená.

8 Vyhodnotenie

V tejto kapitole bude okrem iného uvedená ukážka demo-aplikácie vytvorenej s použitím knižnice Treepace.

8.1 Využitie

Knižnica Treepace má širokú škálu využitia:

- vyhľadávanie a nahrádzanie v XML dokumentoch,
- transformácia AST,
- nahrádzanie v stromovitých štruktúrach, napr. v adresároch,
- grafické aplikácie – vizualizácia

a iné.

8.2 Vyjadrovacie schopnosti

Doménovo-špecifické jazyky sú charakteristické tým, že za cenu vyššej abstrakcie nemusia vždy ponúkať všetky možnosti jazyka všeobecného použitia. V knižnici Treepace však existuje viacero spôsobov vykonávania transformácií.

Na najvyššej úrovni pracujú metódy `transform` a `replace`. Keďže ponúkajú najvyššiu abstrakciu, sú aj najstručnejšie. Na druhej strane, v niektorých situáciách nie sú aplikovateľné – vtedy vyvolajú výnimku.

Ďalšiu úroveň predstavuje volanie metódy `replace` s „callback“ funkciou namiesto reťazca náhrady.

O čosi nižšiu abstrakciu ponúka volanie metódy `search` a následná manuálna úprava stromu podľa vráteného výsledku.

V prípade, že ani jedna spomínaná metóda nie je použiteľná, môžeme hľadanú časť stromu nájsť ručne jeho prechádzaním pomocou atribútov `child` a `parent`.

8.3 Porovnanie s existujúcimi riešeniami

Svojou podstatou sa Treepace najviac podobá jazykom XPath, XQuery Update a Tregex / Tsurgeon.

8.3.1 Možnosti

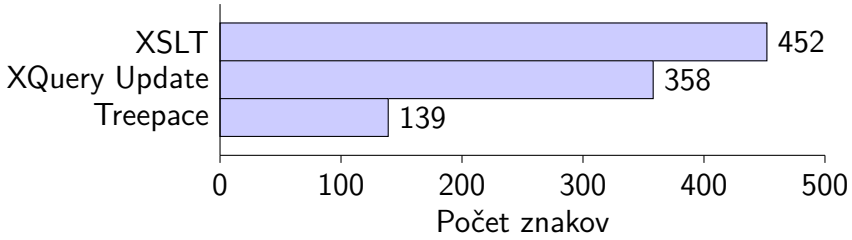
Porovnanie vybraných možností knižnice Treepacee a súvisiacich jazykov sa nachádza v Tabuľke 8 – 1.

Tabuľka 8 – 1: Porovnanie možností jazyka Treepace a iných jazykov

	Treepace	XPath	XQuery Update	Tsurgeon	JastAdd
Vyhľadávanie	✓	✓	✓	✓	✓
Nahrádzanie	✓	✗	✓	✓	✓
Vzor a náhrada jedným riadkom	✓	✗	✗	✗	✗
Hodnota uzla je ľubovoľný objekt	✓	✗	✗	✗	✓

8.3.2 Stručnosť

Na Obrázku 8 – 1 sa nachádza graf závislosti počtu znakov zdrojového kódu transformácie dokumentu, ktorá bola znázornená Obrázkom 3 – 2, od jazyka, v ktorom je transformácia zapísaná.



Obr. 8 – 1: Závislosť počtu znakov kódu transformácie od jazyka

Jednoznačne z nej vyplýva stručnosť jazyka Treepace oproti existujúcim jazykom.

8.3.3 Analógia ku knižniciam regulárnych výrazov

Knižnica Treepace bola navrhovaná tak, aby sa jej API čo najviac podobalo dostupným funkciám na prácu s textovými regulárnymi výrazmi v jazyku Python. To môže znížiť čas učenia sa nového jazyka, resp. API.

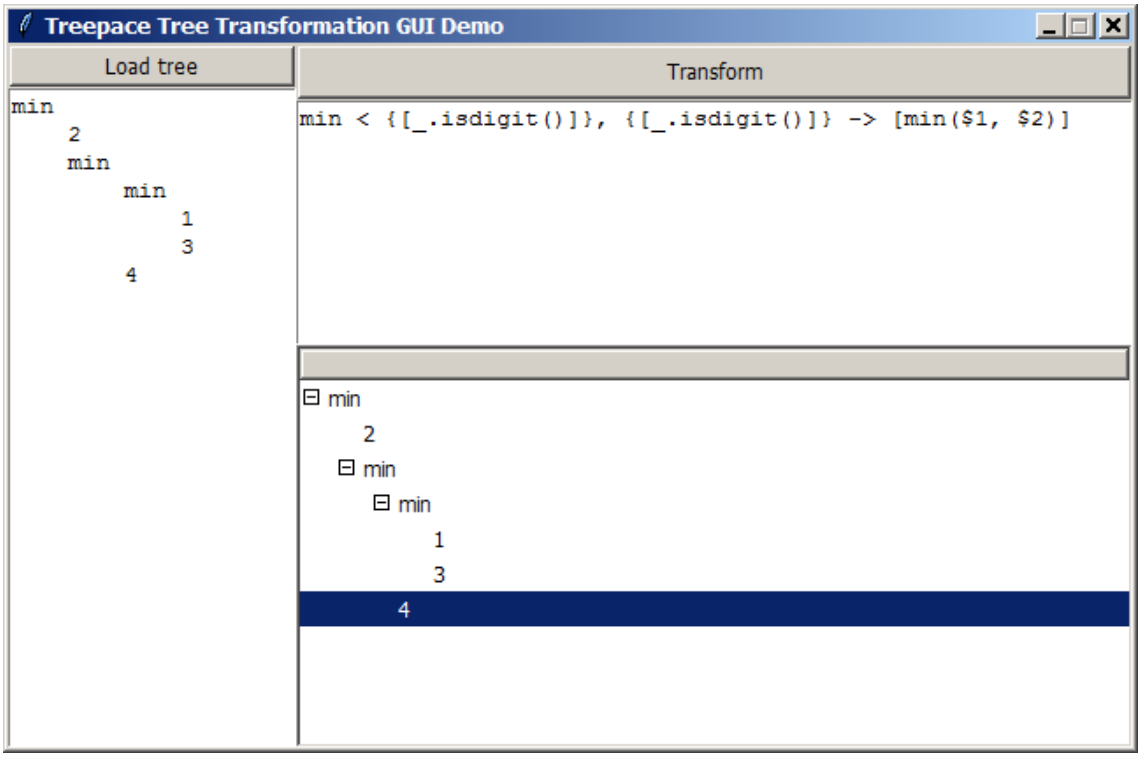
8.4 Ukážková aplikácia

Okrem transformácie dokumentov, ktorá už bola predvedená, je knižnica Treepace užitočná aj pri mapovaní uzlov na prvky grafického používateľského rozhrania. Na Obrázku 8 – 2 je zobrazený stav ukážkovej aplikácie po spustení.

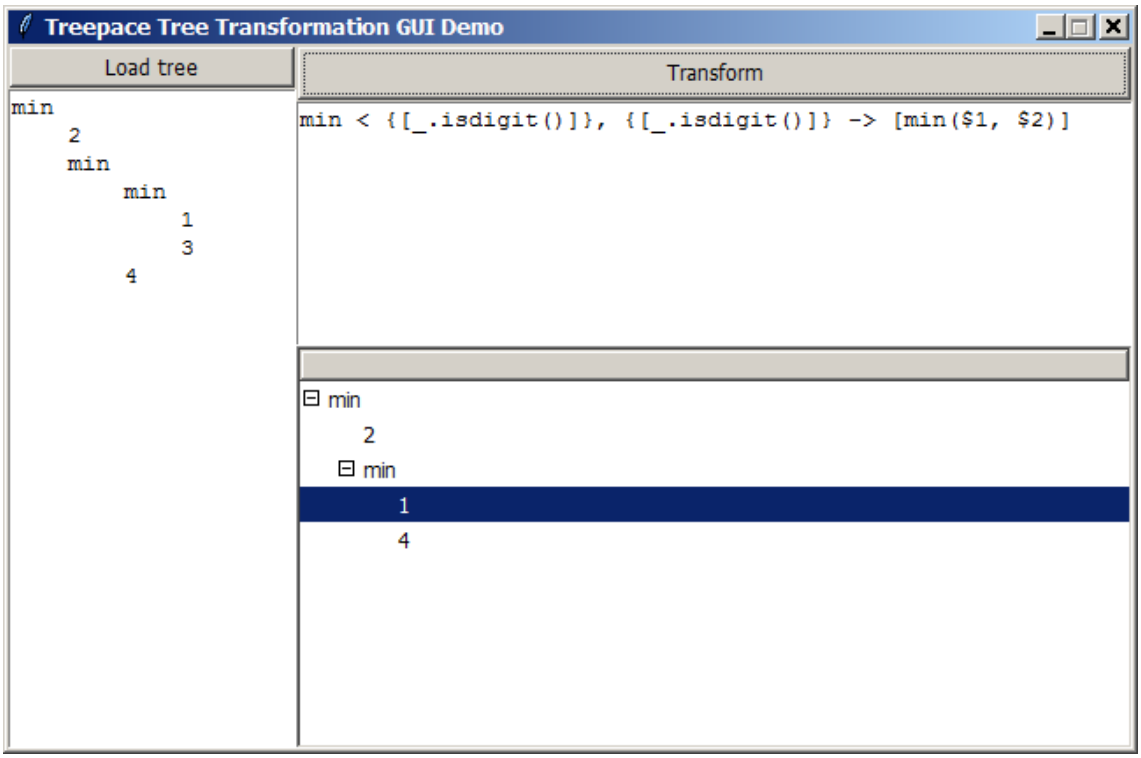
Uzly stromu v aplikácii sú typu `GuiNode`, ktorá je odvodená od `Node`. Okrem štandardného správania táto trieda pri každej zmene uzla aktualizuje GUI. Vďaka tomu môžeme priebeh transformácie priebežne sledovať, čo je znázornené na obrázku 8 – 3.

8.5 Rozšíriteľnosť

Knižnica a jazyk boli navrhované tak, aby boli jednoducho rozšíriteľné – napr. o nové typy relácií, inštrukcie a pod.



Obr. 8 – 2: Demo-aplikácia po načítaní stromu



Obr. 8 – 3: Demo-aplikácia počas transformácie

9 Záver

Cieľom práce bolo navrhnúť a implementovať doménovo-špecifický jazyk na transformáciu stromov. Tento jazyk bol navrhnutý ako vnorený v jazyku všeobecného použitia, konkrétne Python.

Knižnica Treepace ponúka API na vyhľadávanie v strome podľa daného vzoru, náhradu nájdených častí a transformáciu, teda spojenie dvoch predchádzajúcich funkcionalít.

Na rozdiel od existujúcich systémov, uzly stromu v knižnici Treepace môžu byť ľubovoľné objekty – od adresárov na disku až po grafické objekty na obrazovke.

Jazyk Treepace ponúka vysokú stručnosť a zároveň dostatočnú prehľadnosť zápisu. Bežné transformácie je možné zapísať na jeden riadok.

Keďže sa jedná o nesmierne rozsiahlu problematiku, možnosti rozšírenia knižnice sú takmer neobmedzené. Rozšírenia sa môžu uberať dvoma hlavnými smermi – zlepšením možností vzoru alebo náhrady.

Vzor je možné rozšíriť napr. pridaním dodatočných relácií (nepriamy potomok a pod.) alebo zjednodušením niektorých často používaných vzorov pre jeden uzol (regulárne výrazy nad reťazcom a iné). Zaujímavým riešením môže byť pridanie podpory pre vzory na štýl jazyka Regular XPath.

Náhradu sa dá rozšíriť pridaním ďalších stratégií (podmienok a spôsobov) nahrádzania, ak ešte nejaké existujú. Je pri tom nutné dbať na to, aby bola každá transformácia jednoznačná a jej výsledok pre programátora ľahko a intuitívne odhadnuteľný.

Vylepšiť je tiež možné integráciu medzi vzorom a náhradou, napr. explicitným určením, ktorý uzol vo vzore zodpovedá danému uzlu v náhrade. Tým by sa vylepšila podpora pre systém notifikácií o zmene konkrétnych uzlov a bolo by možné jed-

noduchšie tvoriť programy využívajúca mapovanie na externé objekty. Na druhej strane, jazyk by stratil časť svojej deklaratívnosti a stručnosti.

Literatúra

[1] SULÍR, Matúš – ŠIMOŇÁK, Slavomír: *Methods and Application of Tree Transformations*. In: Electrical Engineering and Informatics III, Proceeding of the Faculty of Electrical Engineering and Informatics of the Technical University of Košice. Košice, Slovak Republic: Faculty of Electrical Engineering and Informatics, Technical University of Košice, 2013, s. 373–377.

[2] RUOHONEN, Keijo: *Graph Theory* [online]. 2013. [cit. 2013-11-02]. Dostupné na internete: `http://math.tut.fi/~ruohonen/GT_English.pdf`.

[3] LI, Jerome: *Introduction to Graph Theory* [online]. 2011. [cit. 2014-03-21]. Dostupné na internete: `http://www.ugrad.cs.ubc.ca/~cs221/2010W2/handouts/wolfman/graph-theory-6up.pdf`.

[4] CHIMANI, Markus: *Bend-minimal orthogonal drawing of non-planar graphs*. Diplomová práca. Vienna: Vienna University of Technology, 2004.

[5] WILKE, Thomas: *Alternating Tree Automata, Parity Games, and Modal μ-Calculus*. In: Bulletin of the Belgian Mathematical Society Simon Stevin, 8.2, 2001. s. 359–391.

[6] XIAO, Lin – BOYD, Stephen – LALL, Sanjay: *A scheme for robust distributed sensor fusion based on average consensus*. In: Fourth International Symposium on Information Processing in Sensor Networks. IEEE, 2005. s. 63–70.

[7] KRIEG, Mark L: *A tutorial on Bayesian belief networks*. Edinburgh, Australia: DSTO Electronics and Surveillance Research Laboratory, 2001. 64 s.

[8] THISTLETHWAITE, Morwen B.: *A spanning tree expansion of the Jones polynomial*. In: Topology, 26.3, 1987. s. 297–309.

[9] AIOLLI, Fabio – DA SAN MARTINO, Giovanni – SPERDUTI, Alessandro:

Route kernels for trees. In: Proceedings of the 26th Annual International Conference on Machine Learning. ACM, 2009. s. 17–24.

[10] GESSEL, Ira M. – SEO, Seunghyun: *A refinement of Cayley's formula for trees*. In: Electron, 11(2), R27, 2006.

[11] AHO, Alfred V. – ULLMAN, Jeffrey D.: *Foundations of computer science*. New York: Computer Science Press, 1992. ISBN 07-1678-233-2.

[12] FLECK, Margaret M.: *Building Blocks for Theoretical Computer Science*. (Version 1.3.) Urbana-Champaign: University of Illinois, 2013. 271 s.

[13] GOTSHALKS, Gunnar: *Tree Definitions & Types of Trees* [online]. 2011. [cit. 2013-11-02]. Dostupné na internete: `http://www.eecs.yorku.ca/course_archive/2008-09/F/2011/slides/17-TreeFundamentals.pdf`.

[14] JONES, Joel: *Abstract syntax tree implementation idioms*. In: Proceedings of the 10th Conference on Pattern Languages of Programs (PLoP2003). 2003.

[15] JACOBSON, Guy: *Space-efficient static trees and graphs*. In: 30th Annual Symposium on Foundations of Computer Science. IEEE, 1989. s. 549–554.

[16] BRAY, Tim et al. *Extensible markup language (XML)*. In: World Wide Web Journal, 2.4, 1997. s. 27–66.

[17] CSERKÚTI, Péter: *Survey on Subtree Matching*. In: International Conference on Intelligent Engineering Systems, INES '06. IEEE, 2006. s. 216–221.

[18] SULÍR, Matúš: *Rozšírenie platformy emuStudio*: Bakalárska práca. Košice: Technická Univerzita v Košiciach, 2012.

[19] CLARK, James et al.: *XML Path Language (XPath)* [online]. 1999. [cit. 2013-11-03]. Dostupné na internete: `http://www.w3.org/TR/xpath`.

[20] GENEVÈS, Pierre: *Course: The XPath Language* [online]. Grenoble: University of Grenoble, 2014. [cit. 2014-03-23]. Dostupné na internete: `http://wam.inrialpes.fr/courses/PG-MoSIG13/xpath.pdf`

[21] GOTTLOB, Georg – KOCH, Christoph – PICHLER, Reinhard: *XPath processing in a nutshell*. In: ACM SIGMOD Record, 32.2. ACM, 2003. s. 21–27.

[22] CASSIDY, Steve: *Generalizing XPath for directed graphs*. In: Extreme Markup Languages, Montréal, 2003.

[23] GOTTLOB, Georg – KOCH, Christoph: *Monadic queries over tree-structured data*. In: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science. IEEE, 2002. s. 189–202.

[24] TEN CATE, Balder: *The expressivity of XPath with transitive closure*. In: Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems. ACM, 2006. s. 328–337.

[25] MOZAFARI, Barzan – ZENG, Kai – ZANIOLO, Carlo: *High-Performance Complex Event Processing over XML Streams*. In: Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data. ACM, 2012. s. 253–264.

[26] CLARK, James et al.: *XSL Transformations (XSLT)* [online]. 1999. [cit. 2013-11-03]. Dostupné na internete: `http://www.w3.org/TR/xslt`.

[27] KEPSER, Stephan.: *A proof of the Turing-completeness of XSLT and XQuery*. Technical report SFB 441. Tubingen: Eberhard Karls Universitat Tubingen, 2002.

[28] KAY, Michael: *Learn XQuery in 10 Minutes*. DataDirect Technologies, 2006.

[29] ROBIE, Jonathan et al.: *XQuery update facility 1.0* [online]. 2011. [cit. 2013-11-03]. Dostupné na internete: `http://www.w3.org/TR/xquery-update-10/`.

[30] EKMAN, Torbjörn – HEDIN, Görel. *Rewritable reference attributed grammars*. In: ECOOP 2004 – Object-Oriented Programming. Springer Berlin Heidelberg, 2004. s. 147–171.

[31] LEVY, Roger – ANDREW, Galen. *Tregex and Tsurgeon: tools for querying and manipulating tree data structures*. In: Proceedings of the fifth international conference on Language Resources and Evaluation. 2006. s. 2231–2234.

[32] ROHDE, Douglas L.T.: *TGrep2 User Manual* [online]. 2005. [cit. 2014-03-23]. Dostupné na internete: `http://www.cs.cmu.edu/afs/cs.cmu.edu/project/cmt-55/OldFiles/lti/Courses/722/Spring-08/Penn-tbank/Tgrep2/tgrep2_manual.pdf`.

[33] HOLUPIREK, Alexander: *Declarative Access to Filesystem Data: New application domains for XML database management systems*: Dizertačná práca. Konstanz: Universität Konstanz, 2012.

[34] HOLUPIREK, Alexander – GRÜN, Christian – SCHOLL, Marc H.: *BaseX & DeepFS joint storage for filesystem and database*. In: Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology. ACM, 2009. s. 1108–1111.

[35] RODRIGUEZ, Marko A. – NEUBAUER, Peter: *Constructions from dots and lines*. In: Bulletin of the American Society for Information Science and Technology, 36(6). Wiley Online Library, 2010. s. 35–41.

[36] HOLZSCHUHER, Florian – PEINL, René: *Performance of Graph Query Languages: Comparison of Cypher, Gremlin and Native Access in Neo4j*. In: Proceedings of the Joint EDBT/ICDT 2013 Workshops. ACM, 2013. s. 195–204.

[37] ERDWEG, Sebastian – GIARRUSSO, Paolo G. – RENDEL, Tillmann: *Language composition untangled*. In: Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications. ACM, 2012. s. 7.

[38] HUDAK, Paul: *Modular domain specific languages and tools*. In: Proceedings of the Fifth International Conference on Software Reuse. IEEE, 1998. s. 134–142.

[39] CHOMSKY, Noam: *On certain formal properties of grammars*. In: Information and control, 2(2). 1959. s. 137–167.

[40] FORD, Bryan: *Parsing expression grammars: a recognition-based syntactic foundation*. In: ACM SIGPLAN Notices. ACM, 2004. s. 111–122.

[41] BRAVENBOER, Martin – VISSER, Eelco: *Designing Syntax Embeddings and Assimilations for Language Libraries*. In: Models in Software Engineering: Workshops and Symposia at MoDELS 2007. Lecture Notes in Computer Science. Heidelberg: Springer, 2008. Volume 5002, s. 34–46.

Zoznam príloh

Príloha A CD médium

Príloha B Používateľská príručka

Príloha C Systémová príručka

Príloha D Článok v anglickom jazyku

Technická univerzita v Košiciach
Fakulta elektrotechniky a informatiky

Návrh a implementácia jazyka stromových transformácií

Diplomová práca – Príloha B

Používateľská príručka

2014

Bc. Matúš Sulír

Obsah

1 Úvod

Knižnica Treepace slúži na transformáciu stromov.

Webový návod s príkladmi zdrojových kódov a grafickým zobrazením výsledkov ich vykonávania je možné nájsť na adrese: `http://nbviewer.ipython.org/github/sulir/treepace/blob/master/doc/Tutorial.ipynb`.

2 Inštalácia

V tejto kapitole je uvedený postup inštalovania knižnice pre používateľa, ktorý ju nechce upravovať, iba používať jej verejné API.

2.1 Požiadavky

Vyžadovaný je Python $\geq$ 3.2; ideálne však 3.4, keďže od tejto verzie je v inštalátore pre operačný systém Windows zahrnutý balíčkovací systém, ktorý výrazne zjednoduší inštaláciu knižnice.

Medzi závislosti knižnice Treepace patria balíky Pythonu:

- setuptools,
- parsimonious, verzia 0.5.

Tie však v prípade použitia balíčkovacieho systému nie je potrebné sťahovať manuálne.

Je potrebné mať aktívne pripojenie na internet kvôli stiahnutiu samotnej knižnice a potrebných závislostí.

2.2 Postup inštalácie

Do príkazového riadku zadáme:

```
py -m pip install treepace
```

v prípade OS Windows, resp.

```
pip install treepace
```

v prípade OS Linux.

Tým sa knižnica Treepace nainštaluje do systémového adresára. Bude pripravená na importovanie v ľubovoľnom skripte.

3 Používanie

V tejto kapitole uvedieme najnutnejšie základy práce s knižnicou Treepace.

3.1 Import

Knižnicu importujeme príkazom:

```
from treepace import *
```

3.2 Použitie

Následne môžeme používať dostupné triedy a metódy, napr.:

```
tree.replace('{book < name} < {[re.match(r".*\d+$", _)]}',
    '$1 < [str.upper($2)]')
```

V znázornenom príklade sa názvy všetkých kníh končiace sa číslicou zmenia na veľké písmená.

Plný zoznam dostupných tried a funkcií je možné nájsť v systémovej príručke.

3.3 Spustenie demo aplikácií

Demo aplikácie sa pod OS Windows spustia z príkazového riadku:

```
py -m treepace.examples.document
```

resp.

```
py -m treepace.examples.gui
```

Technická univerzita v Košiciach
Fakulta elektrotechniky a informatiky

Návrh a implementácia jazyka stromových transformácií

Diplomová práca – Príloha C

Systémová príručka

2014

Bc. Matúš Sulír

Obsah

1 Popis knižnice

Treepace je knižnica určená na vyhľadávanie v údajových štruktúrach stromového typu, nahrádzanie častí stromov a ich transformáciu. Ponúka stručnú syntax a možnosť transformovať stromy, ktorých uzlami sú ľubovoľné objekty.

Táto systémová príručka je určená pre tých, ktorí chcú modifikovať samotnú knižnicu Treepace a pridať do nej novú funkcionalitu, prípadne odstrániť nájdené chyby, ale i jej bežným používateľom.

Knižnica je napísaná v jazyku Python a využíva objektovo-orientovaný prístup. Vyskytujú sa v nej aj prvky funkcionálneho programovania. Pre niektoré jej časti sú napísané tzv. unit testy.

2 Inštalácia

V tejto kapitole je uvedený postup inštalovania knižnice z pohľadu vývojára – teda v prípade, že ju chceme ďalej upravovať, nie iba používať ju v nezmenenej podobe.

2.1 Požiadavky

Vyžadovaný je Python $\geq$ 3.2; ideálne však 3.4, keďže od tejto verzie je v inštalátore pre operačný systém Windows zahrnutý balíčkovací systém, ktorý výrazne zjednoduší inštaláciu knižnice.

Medzi závislosti knižnice Treepace patria balíky Pythonu:

- setuptools,
- parsimonious, verzia 0.5.

Tie však v prípade použitia balíčkovacieho systému nie je potrebné sťahovať manuálne.

Odporúča sa tiež nainštalovať:

- Eclipse (prípadne iné vývojové prostredie alebo textový editor),
- Git.

Je potrebné mať aktívne pripojenie na internet kvôli stiahnutiu samotnej knižnice a potrebných závislostí.

2.2 Postup inštalácie

Najprv je vhodné stiahnuť najnovšiu verziu z Githubu:

```
git clone git://github.com/sulir/treepace.git
```

Prejdeme do priečinka obsahujúceho knižnicu a zadáme:

```
setup.py develop
```

Tým je knižnica Treepace pripravená a importovanie v ľubovoľnom skripte. Knižnica bude nainštalovaná priamo v lokálnom priečinku (tam, kde bola uložená počas zadávania inštalačného príkazu) – nebude kopírovaná do systémového. Na rozdiel od klasickej inštalácie, zmeny zdrojových kódov v lokálnom priečinku sa prejavia na správaní sa knižnice. Nesmieme ju však presúvať do iného adresára.

3 Popis tried a metód

V tejto časti sa nachádza zoznam verejných tried, metód a konštánt spolu s ich popismi v anglickom jazyku.

File base.py

Basic tree interface and behavior.

Class TreeBase(EqualityMixin, ReprMixin)

An abstract class containing the interface and implementation common for both a tree and a subtree.

- **`def root(self)`**

  Return the current root node.

- **`def traverse(self, node=lambda node: [], down=lambda: [], right=lambda: [], up=lambda: [])`**

  Traverse the tree in a pre-order manner with direction signaling.

  For each visited node or direction, execute the supplied function and generate the items which it returned. For example, given the tree 'a (b c)', the resulting sequence is node(a), down(), node(b), right(), node(c), up().

- **`def preorder(self)`**

  Return a generator for pre-order tree traversal.

- **`def node(self, value)`**

Return the first node with the given value (using string comparison).

- **`def leaves(self)`**

  Return all leaves of the subtree.

- **`def inner(self)`**

  Return all non-leaf nodes (including the root).

File build.py

A replacement tree constructing virtual machine.

Class BuildMachine(ReprMixin)

A machine which builds a tree which will be used as a replacement during the transformation.

- **`def __init__(self, match, instructions, variables)`**

  Initialize the VM with the default state.

- **`def __str__(self)`**

  Return the machine state in a form of a string.

- **`def build(self)`**

  Execute all instructions and return the built tree.

File compiler.py

A parser and instruction generator for transformation rule strings.

Class Compiler

A compiler from rule, pattern and replacement strings to instructions.

- **`def compile_pattern(pattern)`**

  Parse the pattern and return an instruction list.

- **`def compile_replacement(replacement)`**

  Parse the replacement and return an instruction list.

- **`def compile_rule(rule)`**

  Parse the rule and return two instruction lists – searching instructions and replacing instructions.

Class InstructionGenerator(NodeVisitor)

A base class with common behavior for post-order visitors which generate a list of virtual machine instructions from an AST.

- **`def __init__(self)`**

  Initialize the instruction list and a level counter.

- **`def generic_visit(self, node, visited_children)`**

  Just continue with the traversal.

- **`def visit_child(self, node, visited_children)`**

  Add the instruction 'REL child'.

- **`def visit_next_sibling(self, node, visited_children)`**

  Add the instruction 'REL next_sib'.

Class SearchGenerator(InstructionGenerator)

A generator of tree-searching instructions.

- **`def __init__(self)`**

  Initialize the group counters.

- **`def visit_pattern(self, node, visited_children)`**

  Return the generated instruction list (at the top of the AST).

- **`def visit_any(self, node, visited_children)`**

  Add an instruction which matches any node.

- **`def visit_constant(self, node, visited_children)`**

  Add an instruction which matches the constant.

- **`def visit_python_code(self, node, visited_children)`**

  Add an instruction which matches the predicate.

- **`def visit_reference_num(self, node, visited_children)`**

  Add a back-referencing instruction.

- **`def visit_group_start(self, node, visited_children)`**

  Add a group-starting instruction and adjust the counters.

- **`def visit_group_end(self, node, visited_children)`**

  Add a group-ending instruction and adjust the counter.

- **`def visit_sibling(self, node, visited_children)`**

Add the instruction 'REL sibling'.

- **`def visit_parent_any(self, node, visited_children)`**

  The 'parent' relation followed by an implicit 'any' pattern.

Class BuildGenerator(InstructionGenerator)

A generator of instructions which build a replacement tree.

- **`def visit_replacement(self, node, visited_children)`**

  Return the generated instruction list (at the top of the AST).

- **`def visit_constant(self, node, visited_children)`**

  Add an instruction which appends a node with a constant value to the tree.

- **`def visit_python_code(self, node, visited_children)`**

  Add an instruction which appends a dynamically generated node to the tree.

- **`def visit_reference_num(self, node, visited_children)`**

  Add a back-referencing instruction.

- **`def visit_parent_any(self, node, visited_children)`**

  Add an instruction which navigates up in the tree being built.

Class CompileError(Exception)

Raised when a non-parser related error occurs during compilation.

File formats.py

Support for importing trees from various formats (tabulated text, XML, ...) and exporting them.

In contrast to resource-mapping, importing/exporting loads and saves trees to their external representation on demand, not after every change.

Class IndentedText

A text where the indentation level determines the node level.

- **`def __init__(self, indent=' ')`**

  The indentation string is used only for exporting; it is automatically recognized during the import.

- **`def load_tree(self, string, node_class)`**

  Create a tree from the tab- or space-indented string.

- **`def save_tree(self, tree)`**

  Create a space- or tab-indented string from the tree.

Class ParenText

A text starting with a root node, followed by children enclosed in parentheses and siblings divided by spaces.

- **`def load_tree(self, string, node_class)`**

  Create a tree from the parenthesized string.

- **`def save_tree(self, tree)`**

Create a parenthesized string from the tree.

Class XmlText

A string containing an XML document.

- `def load_tree(self, string, node_class)`

  Create a tree from the XML string.

- `def save_tree(self, tree)`

  Create an XML string from the tree.

Class InvalidFormatError(Exception)

Raised when the imported string is invalid.

File instructions.py

Virtual machine instructions.

Class Instruction(EqualityMixin, ReprMixin)

Instructions should be immutable.

- `def _compile_code(self, expression, instr_vars)`

  Compile and save the given Python code.

- `def _evaluate_code(self, machine_vars, match, node=None)`

  Evaluate the saved code in an environment containing auxiliary functions, variables from the VM and the instruction object, matched groups and (optionally) the given node.

Class Find(Instruction)

An instruction searching for nodes which are in the currently set relationship with the context node and match the predicate.

- **`def __init__(self, expression, **instr_vars)`**

  Save the expression in a textual and compiled version.

- **`def execute(self, branch)`**

  Find the nodes and return a list of branches which should replace the supplied branch.

- **`def __str__(self)`**

  Return the string representation of the instruction.

Class SetRelation(Instruction)

An instruction which sets the relation to be used for next search or node adding.

- **`def __init__(self, relation)`**

  The given relation argument should be a class.

- **`def execute(self, branch_or_vm)`**

  Set the current relation of the branch / virtual machine.

- **`def __str__(self)`**

  Return the string representation of the instruction.

Class GroupStart(Instruction)

An instruction used to mark a numbered group start.

- **`def __init__(self, number)`**

  Save the group number as an instruction argument.

- **`def execute(self, branch)`**

  Add the corresponding group number and subtrees to the machine.

- **`def __str__(self)`**

  Return the string representation of the instruction.

Class GroupEnd(Instruction)

An instruction marking a numbered group end.

- **`def __init__(self, number)`**

  Save the group number as an instruction argument.

- **`def execute(self, branch)`**

  Remove the group number from the set of current group numbers.

- **`def __str__(self)`**

  Return the string representation of the instruction.

Class SearchReference(Instruction)

A back-reference to a numbered group in a search pattern.

- **`def __init__(self, number)`**

  Save the group number as an instruction argument.

- **`def execute(self, branch)`**

Prepend instructions which will search for a subtree same as the given group's subtree.

- **`def __str__(self)`**

  Return the string representation of the instruction.

Class AddNode(Instruction)

An instruction which adds the node to the correct position in the tree being built.

- **`def __init__(self, expression, **instr_vars)`**

  Save the expression in a textual and compiled version.

- **`def execute(self, vm)`**

  Create a new node, add it to the tree (or create a tree if it does not yet exist) and set it as the context node.

- **`def __str__(self)`**

  Return the string representation of the instruction.

Class AddReference(Instruction)

A back-reference to a numbered group in a replacement.

- **`def __init__(self, number)`**

  Save the group number as an instruction argument.

- **`def execute(self, vm)`**

  Add the given group's subtree to the tree and set its root as the context node.

  This may not seem intuitive when the 'child' relation is used immediately after

the back-reference, but it is consistent.

- **`def __str__(self)`**

  Return the string representation of the instruction.

Class GoToParent(Instruction)

An instruction for navigation in the tree being built.

- **`def execute(self, vm)`**

  Change the context node to the current context node's parent.

- **`def __str__(self)`**

  Return the string representation of the instruction.

File nodes.py

The basic node implementation, followed by custom node types with specific behavior.

Class Node(ReprMixin)

An in-memory node node with references to children and a parent.

The constructor, the 'value' property setter and the methods 'insert_child' and 'delete' are recommended to be overridden in specialized classes.

- **`def __init__(self, value, children=[])`**

  Initialize a new tree node.

  The optional child node list will be shallow-copied.

- **`def value(self)`**

  Return this node's value.

- **`def value(self, _value)`**

  Set the value of this node – a string, a map or any other object.

- **`def parent(self)`**

  Return the parent node.

- **`def children(self)`**

  Return a shallow-copied tuple containing the child nodes.

- **`def add_child(self, child)`**

  Add a child node to the end.

- **`def insert_child(self, child, index)`**

  Insert a child node at the specified index.

- **`def detach(self)`**

  Delete the node (it must not be a root).

- **`def index(self)`**

  Return a zero-based order of this node among its siblings.

- **`def level(self)`**

  Return this node's vertical level; the root node has a level of 0.

- **`def path(self)`**

Return a list of nodes from the root to this node.

- **`def str_path(self)`**

  Return a slash-separated path from the root node to this node.

- **`def replace_by(self, node)`**

  Replace the node by an another node (including children).

- **`def __str__(self)`**

  Return a string representation of the node's value.

Class LogNode(Node)

A tree node which writes human-readable information about all changes to a file (or standard output).

File relations.py

Tree node relations.

Class Child

A child relation.

- **`def search(self, node)`**

  Return an iterable containing this node's children.

- **`def build(self, context, node)`**

  Add the given node as a first child of the context node.

Class Sibling

All siblings (excluding the node itself).

- `def search(self, node)`

  Return an iterable containing the siblings of this children.

Class NextSibling

An immediately following sibling.

- `def search(self, node)`

  Return a list with one element (the next sibling) or an empty list.

- `def build(self, context, node)`

  Insert the given node just after the context node.

Class Parent

A parent relation.

- `def search(self, node)`

  Return a one-element list with the node's parent or an empty list.

Class Descendant

A descendant or the node itself – used when searching anywhere in the tree.

- `def search(self, node)`

  Return an iterable with all node's descendants in a pre-order manner.

Class Identic

An identity relation, used when searching exactly from the root.

- `def search(self, node)`

  Return a list with the given node.

File replace.py

Tree replacing strategies.

Class ReplaceStrategy

A tree replacing strategy is an algorithm for replacement of a subtree with an another tree which is applicable only if some condition is met.

- `def __init__(self, old, new)`

  Set an old subtree and a new tree.

- `def all_strategies()`

  Return a list of all strategies in the order they should be tried.

Class SameShape(ReplaceStrategy)

Replacement of a subtree by a tree with the same shape.

- `def test(self)`

  Traverse and compare the trees, ignoring the values.

- `def apply(self)`

  Set the subtree node values to the values of the tree.

Class ToOneNode(ReplaceStrategy)

Replacement of a subtree by one node.

- `def test(self)`

  The new tree must consist of only one node.

- `def apply(self)`

  The replacement node child list will consist of all children of the original subtree leaves.

Class NoConnectedLeaves(ReplaceStrategy)

Replacement of a subtree which does not have any children outside the subtree.

- `def test(self)`

  "The subtree must not have 'connected leaves' and its inner nodes must not have children outside the subtree.

- `def apply(self)`

  The subtree root node is replaced by the new tree root node.

Class LeafCountStrategy(ReplaceStrategy)

An abstract class representing replacement of a subtree by a tree whose leaf count is the same as the count of the subtree's leaves or 'connected leaves'.

- `def test(self)`

  In addition, non-leaf subtree nodes must not have children outside of the subtree.

- `def apply(self)`

Children of the subtree leaves become the children of the tree leaves, then the roots are replaced.

Class SameLeafCount(LeafCountStrategy)

Replacement of a subtree by a tree with the same leaf count.

Class SameConnectedCount(LeafCountStrategy)

Replacement of a subtree by a tree whose leaf count is the same as the count of the subtree's 'connected leaves'.

Class ReplaceError(Exception)

Raised when the replacement cannot be accomplished.

File search.py

A tree-searching virtual machine and searching branch implementation.

Class SearchMachine(ReprMixin)

A tree-searching virtual machine.

- `def __init__(self, node, instructions, variables, relation=Descendant)`

  Initialize the VM with the default state.

- `def search(self)`

  Execute all instructions and return the search results.

- `def __str__(self)`

Return the machine state in a form of a string.

Class SearchBranch(ReprMixin)

The search process can 'divide' itself into multiple branches.

- **`def __init__(self, node, instructions, vm, relation)`**

  Each branch is represented by a set of current group numbers, a match object (a subtree list containing current results), a context node, a current relation and an instruction list.

- **`def copy(self)`**

  Return a copy of this branch which can be modified without affecting the original branch.

- **`def __str__(self)`**

  Return the branch information as a string.

Class Match(ReprMixin)

A match is a list of groups; each group is one subtree.

- **`def __init__(self, groups)`**

  Initialize a match with a list of subtrees.

- **`def group(self, number=0)`**

  Return the given group; group 0 is the whole match.

- **`def groups(self)`**

  Return the list of all groups.

- **`def copy(self)`**

  Return a list of copies of all subtrees.

- **`def check_disjoint(matches)`**

  Raise ReplaceError if there exists a node which is present in at least two matches from the given list of matches.

- **`def __str__(self)`**

  Return a string containing all groups (subtrees).

File trees.py

The main tree class and a subtree implementation.

Class Tree(TreeBase)

A general tree which can contain any types of nodes.

- **`def __init__(self, root)`**

  Initialize the tree with a root node which can never be deleted (only replaced).

- **`def root(self, _root)`**

  Set a new root node.

- **`def load(cls, string, fmt=ParenText, node_class=Node)`**

  Create a new tree by importing it from a string in a given format.

- **`def save(self, fmt)`**

  Export the tree to a string in a given format.

- **def search(self, pattern, **variables)**

  Search for a given pattern anywhere in the tree and return a list of matches.

- **def match(self, pattern, **variables)**

  Search for a given pattern from the root node and return a list of matches.

- **def fullmatch(self, pattern, **variables)**

  If the tree matches the pattern from the root to the leaves, return a list of matches, otherwise return an empty list.

- **def replace(self, pattern, replacement, **variables)**

  Replace each found subtree with a new subtree.

  The replacement can be a string containing back-references or a callback function returning the new tree.

- **def transform(self, program, **variables)**

  Execute the transformation program which can contain multiple rules in the form: pattern -> replacement.

  Each rule is executed while its pattern matches. In addition, the whole rule list is looped until no rule matches.

- **def copy(self)**

  Shallow-copy the tree.

- **def __str__(self)**

  Return a parenthesized-text representation of this tree.

- **def __eq__(self, other)**

Values of all tree nodes are compared.

Class Subtree(TreeBase)

A subtree is a connected part of a tree with one root node and one or more leaves.

Each subtree node reference points to the same object as the node reference in the main tree.

- **`def __init__(self, nodes=[])`**

  Initialize the subtree with a (possibly empty) list of nodes.

- **`def add_node(self, node)`**

  Add the node to the subtree.

  It must be connected to some node currently present in the subtree.

- **`def remove_node(self, node)`**

  Remove the given leaf node from the subtree.

- **`def nodes(self)`**

  The returned node set should not be modified.

- **`def connected_leaves(self)`**

  Return leaves of the subtree which have children in the main tree.

- **`def to_tree(self)`**

  Shallow-copy node values into a new tree (with new nodes).

- **`def replace_by(self, tree)`**

Try available replacement strategies and raise an exception if neither of them is applicable.

- `def copy(self)`

  Return a shallow copy of the subtree (a new node set is created, but the node references point to the same nodes).

- `def __str__(self)`

  Return a text representation of the subtree (as if it was a tree).

Class SubtreeError(Exception)

Raised when a subtree cannot be modified in a given way.

File utils.py

Utility functions and mix-in classes.

Class EqualityMixin

Equality and inequality operator overloading based on attributes.

- `def __eq__(self, other)`

  All instance attributes are compared for equality.

- `def __ne__(self, other)`

  Just a negation of the equality result.

Class ReprMixin

Default string and debugging representations of objects.

- **`def __str__(self)`**

  Ideally, subclasses should supply their own implementations.

- **`def __repr__(self)`**

  The debug representation is automatically derived from the string representation by adding a class name and a unique identifier.

Technická univerzita v Košiciach
Fakulta elektrotechniky a informatiky

Návrh a implementácia jazyka stromových transformácií

Diplomová práca – Príloha D

Článok v anglickom jazyku

2014 **Bc. Matúš Sulír**

Treepace: Tree Transformation Language

Matúš Sulír and Slavomír Šimoňák

Department of Computers and Informatics
Technical University of Košice, Letná 9, 042 00 Košice, Slovakia

Abstract. Treepace is a new library and domain specific language for tree pattern matching and replacing, implemented in Python. Selected concepts resemble the well-known regular expressions API. The language is terse as it is possible to write the whole transformation in one row. Objects of any type can be used as node values. Node class inheritance allows for mapping to external objects, e.g. GUI components.

Key words: tree, transformation, pattern, replacement, language

1 Introduction

One of the most used languages to address parts of trees is XPath [1]. It consists of location steps [2]; each location step contains an axis and a node test [3]. While it is a succinct and expressive language, it does not support modification of trees. There exist languages like XSLT [4] and XQuery Update [5] for this purpose. The former not fully preserve the simplicity and terseness of XPath. The latter is better in this regard, but the shortcomings of XML persist. The nodes of XML documents internally contain only strings, not arbitrary objects. In addition, it is difficult to react on node changes because the only output is a resulting tree.

Two more candidates are not general enough – JastAdd, based on Rewritable reference attributed grammars [6] is specialized for AST transformation and Tsurgeon for linguistic applications [7].

2 The application programming interface

The basic building block of trees in Treepace is an object of class `Node`. A tree (of class `Tree`) has a reference to the root node. When matching against the tree, subtrees (`Subtree`) are returned. Each subtree is defined by a set of node references pointing to the main tree.

Trees can be manually constructed or loaded from a format like XML, parenthesized or indented text. After a tree is loaded in the memory, we can perform operations like searching an replacing. The API (application programming interface) of Treepace library was designed to be conceptually similar to the standard Python regular expression library. The comparison is in Table 1. However, Treepace modifies the original tree while regular expression replacement returns a new string.

Table 1. An analogy between regex and Treepace API calls

Action	Python regular expression API call example	Treepace API call example
search anywhere	`re.search('pattern', text)`	`tree.search('pattern')`
search from the start (root)	`re.match('pattern', text)`	`tree.match('pattern')`
match the whole text / tree	`re.fullmatch('pattern', text)`	`tree.fullmatch('pattern')`
replace a part of the text / tree	`re.sub('pattern', 'replacement', text)`	`tree.replace('pattern', 'replacement')`
transform a tree (replace in a loop while matches are found)	–	`tree.transform('pattern -> replacement')`

Values of nodes in our library can be of any type – from simple strings and numbers through structured data (e.g. associative arrays) to complicated objects bound to external entities like open file descriptors or network sockets. Thus, the transformations can perform actions on these objects according to patterns – for example, delete all files matching a condition.

Furtheremore, it is possible to inherit from the basic node class. This is useful to add various side effects to each node change. In Treepace, there are two examples included – a logging node and a GUI node (see Fig. 1 for a class diagram). The former writes an information about each changed node to the standard output. The latter displays a visualization of the transformation process in a GUI window – see Fig. 2. We can say that each node in the graphical component is mapped to a node in the memory. Every transformation consists of a sequence of these elementary operations:

- Create a node.
- Change the value of a node.
- Insert a node at the given position of the child list of a particular node.
- Detach a node from its parent.

Each class inherited from `Node` have to override just the mentioned methods to achieve mapping.

3 The transformation language

The transformation in Treepace consists of two parts: a pattern and a replacement.

3.1 Pattern

The pattern is used for searching – both standalone and as a part of replacing or transformation. It consists of node tests separated by relations – child (`<`),

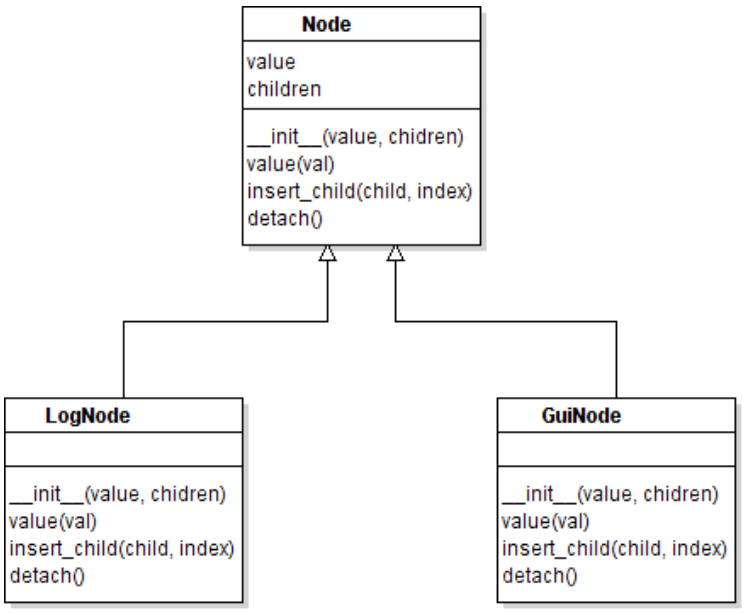


Fig. 1. Node class inheritance, adding side effects to tree transformation

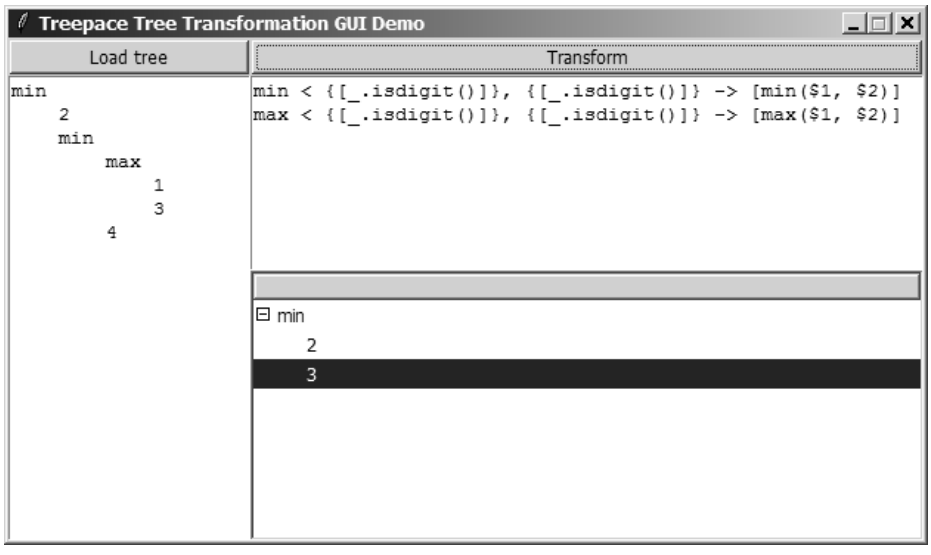


Fig. 2. GUI-mapped tree nodes during transformation

sibling (`&`), next sibling (`,`) and parent (`>`, followed by an implicit any-matching node test).

The most basic pattern, matching one arbitrary node is a dot (`.`). A literal text matches a node whose string representation is equal to it. The most powerful node test is a predicate enclosed in square brackets, which can contain any Python code. The code is evaluated in an environment containing a reference to the currently tested node and variables supplied via keyword arguments of the API method.

Parts of the pattern can be enclosed in braces to form a group. The groups are numbered from one and can be nested. The whole pattern implicitly becomes a group number 0. It is possible to back-reference a group both in the pattern and a replacement using the notation `$n` where `n` is the group number. This behavior strengthens the similarity between Treepace and regular expression implementations in many general-purpose languages.

3.2 Replacement

The replacement part of a rule consists of nodes separated by relations which they should form. Relations supported in the replacement are: child, next sibling and parent.

Each node in the replacement can be a literal, a back-reference or an arbitrary Python expression. In the last case, the code is evaluated and its result is assigned to the node value.

4 Execution

The execution of a transformation begins by parsing the rule and conversion of an AST (abstract syntax tree) to a list of instructions.

4.1 Searching

The searching instructions are executed on a tree-searching virtual machine, which consists of a list of search branches. Each branch is a quintuple:

$$(groups, matches, node, rel, instrs)$$

After execution, the resulting subtrees are in the *matches* field.

4.2 Building a replacement

Instructions for building a replacement tree for each single match are executed on a machine which is a quintuple:

$$(match, instrs, node, rel, tree)$$

The replacement is then stored in the field *tree*.

4.3 Replacing

Finally, each matched subtree is substituted by its replacement. The library contains a list of five replacing strategies, each of which contains a test whether it can be unambiguously applied and the application algorithm itself. They are tried from the highest priority to the lowest one. For example, the highest priority strategy is applicable if the match and replacement are of the same shape and its algorithm is to assign all node values of the old subtree to the new tree. If no strategy can be applied, an exception is raised.

5 Example

Fig. 3 displays a structure of a tree extracted from an XML document and a resulting tree after execution of the transformation from Fig. 4.

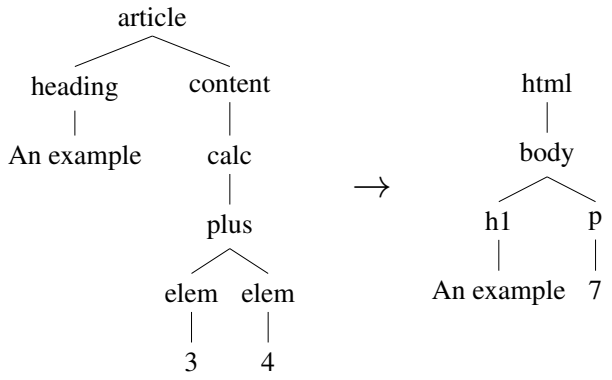


Fig. 3. A document transformation

```
tree.transform('''
  article -> html < body
  heading -> h1
  content -> p
  calc < plus < elem<{.}>, elem<{.}> -> [text(num($1) + num($2))]
''')
```

Fig. 4. The source code containing a Treepace transformation

6 Conclusion

Our goal was to create a terse, string-embedded language for tree transformations where the node values can be arbitrary objects and nodes can react to changes which occur during transformation. The language is not limited to a particular application domain. The ability to react on node changes has an advantage that tree-resource mapping like a simple tree visualization can be implemented with minimum effort.

The disadvantage of Treepace is currently a limited number of replacing strategies. However, it is possible to use only the pattern matching part of the library and perform the replacement manually.

References

1. CLARK, James et al.: *XML Path Language (XPath)* [online]. 1999. Available: `http://www.w3.org/TR/xpath`.
2. GENEVÈS, Pierre: *Course: The XPath Language* [online]. Grenoble: University of Grenoble, 2014. Available: `http://wam.inrialpes.fr/courses/PG-MoSIG13/xpath.pdf`.
3. GOTTLOB, Georg – KOCH, Christoph – PICHLER, Reinhard: *XPath processing in a nutshell.* In: ACM SIGMOD Record, 32.2. ACM, 2003. pp. 21–27.
4. CLARK, James et al.: *XSL Transformations (XSLT)* [online]. 1999. Available: `http://www.w3.org/TR/xslt`.
5. ROBIE, Jonathan et al.: *XQuery update facility 1.0* [online]. 2011. Available: `http://www.w3.org/TR/xquery-update-10/`.
6. EKMAN, Torbjörn – HEDIN, Görel. *Rewritable reference attributed grammars.* In: ECOOP 2004 – Object-Oriented Programming. Springer Berlin Heidelberg, 2004. pp. 147–171.
7. LEVY, Roger – ANDREW, Galen. *Tregex and Tsurgeon: tools for querying and manipulating tree data structures.* In: Proceedings of the fifth international conference on Language Resources and Evaluation. 2006. pp. 2231–2234.