

Technická univerzita v Košiciach

Fakulta Elektotechniky a Informatiky

Zvýrazňovanie syntaxe na základe definície meta-modelu

Matej Gagyí

Bakalárska práca

2015

Technická univerzita v Košiciach
Fakulta Elektotechniky a Informatiky
Katedra Počítačov a Informatiky

Zvýrazňovanie syntaxe na základe definície meta-modelu

Bakalárska práca

Matej Gagyi

Vedúci bakalárskej práce:

Ing. Sergej Chodarev, PhD.

Konzultant bakalárskej práce:

Košice 2015

Analytický list

Autor:	Matej Gagyí
Názov práce:	Zvýrazňovanie syntaxe na základe definície meta-modelu
Podnázov práce:	
Jazyk práce:	slovenský, anglický, nemecký
Typ práce:	Bakalárska práca
Počet strán:	0 , 50
Akademický titul:	Bakalár, Magister, Magister umenia, Inžinier, Inžinier architekt
Univerzita:	Technická univerzita v Košiciach
Fakulta:	Fakulta Elektrotechniky a Informatiky (FEI)
Katedra:	Katedra Počítačov a Informatiky ()
Študijný odbor:	Informatika
Študijný program:	Aplikovaná Informatika
Mesto:	Košice
Vedúci DP:	Ing. Sergej Chodarev, PhD.
Konzultanti DP:	
Dátum odovzdania:	29. Mája. 2015
Dátum obhajoby:	23.-30. Júna 2015
Kľúčové slová:	parser, YajCo, java, syntax, editor, jazyky
Kategória Konspekt:	Technika, technológie, inžinierstvo
Citovanie práce:	Gagyí, Matej: Zvýrazňovanie syntaxe na základe definície meta-modelu. Bakalárska práca. Košice: Technická univerzita v Košiciach, Fakulta Elektrotechniky a Informatiky, 2015. 0 s.
Názov práce v AJ:	Syntax highliting based on meta-model definition
Podnázov práce v AJ:	
Kľúčové slová v AJ:	parser, YajCo, java, syntax, editor, languages

Abstrakt v SJ

V tejto práci som sa zaoberal automatizáciou zvýrazňovania syntaxe pre jazyky modelované v nástroji YajCo. Hlavným cieľom práce bolo nájsť metódy pre analýzu gramatiky jazyka a generovanie konfigurácie pre rôzne nástroje pracujúce so zdrojovými kódmi. Ďalším cieľom bolo diskutovať pokročilejšie metódy analýzy a načrtnúť námet na nadväzujúce práce. Syntetická časť zhrňuje gramatické elementy používané rôznych systémoch zvýrazňovania syntaxe a analyzuje vzorku skutočných programovacích jazykov. Implementovali sme sadu jednoduchých pravidiel rozpoznávajúcich kľúčové vlastnosti rôznych elementov gramatiky a výstup pre editori Kate a Gedit. K záveru práce rozoberám metódy analýzy regulárnych výrazov, metódu prevodu regulárnych výrazov na Deterministický Konečný Automat a operácie, ktoré nám tato reprezentácia regulárneho jazyka umožňuje vykonávať.

Abstrakt v AJ

In this paper I focused on automating syntax highlighting for languages modeled in the tool YajCo. Main goal of the paper was to find methods suitable for grammar analysis and generating configuration for different tools used to work with source code. Another goal was to discuss advanced methods of analysis and draw a possible topic for following works. Synthetic part of the paper summarizes different elements of grammar used in syntax highlighting systems and analyses their features in a sample of real-world programming languages. We implemented a set of simple rules for categorizing grammar elements and an output for editors Kate and Gedit. I am closing the paper with description of methods suitable for analysis of regular expressions, a method for translating regular expressions into Deterministic Finite Automata and operation applicable to such a representation of the regular language.

ZADANIE BAKALÁRSKEJ PRÁCE

Študijný odbor: **9.2.9 Aplikovaná informatika**

Študijný program: **Aplikovaná informatika**

Názov práce:

Zvýrazňovanie syntaxe na základe definície meta-modelu

Syntax Highlighting based on Meta-Model Definition

Študent:

Matej Gagy

Školiteľ:

Ing. Sergej Chodarev, PhD.

Školiace pracovisko:

Katedra počítačov a informatiky

Konzultant práce:

Pracovisko konzultanta:

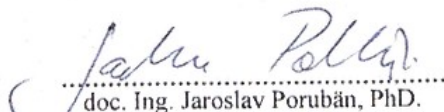
Pokyny na vypracovanie bakalárskej práce:

1. Analyzovať spôsoby špecifikácie jazykov pre zvýrazňovanie syntaxe v rôznych editoroch.
2. Analyzovať generátor syntaktických analyzátorov YAJCo a spôsob špecifikácie jazyka, ktorý sa v ňom používa.
3. Navrhnuť nástroj, ktorý bude schopný na základe špecifikácie jazyka pre YAJCo vygenerovať špecifikáciu zvýrazňovania syntaxe tohto jazyka pre niektoré editory.
4. Implementovať navrhnutý nástroj a overiť implementáciu na niekoľkých jazykoch.
5. Zhodnotiť riešenie a navrhnuť potenciálne rozšírenia špecifikácie jazykov používanej v YAJCo pre zlepšenie zvýrazňovania syntaxe.
6. Vypracovať dokumentáciu podľa pokynov vedúceho práce.


Jazyk, v ktorom sa práca vypracuje: slovenský

Termín pre odovzdanie práce: 29.05.2015

Dátum zadania bakalárskej práce: 31.10.2014


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




prof. Ing. Liberios Vokorokos, PhD.
dekan fakulty

Čestné vyhlásenie

Vyhlasujem, že som celú bakalársku prácu vypracoval/a samostatne s použitím uvedenej odbornej literatúry.

Autori metodických príručiek (pozri Katuščák [7], Gonda [6]) o záverečných prácach sa nazdávajú, že takéto vyhlásenie je zbytočné, nakoľko povinnosť vypracovať záverečnú prácu samostatne, vyplýva študentovi zo zákona a na autora práce sa vzťahuje autorský zákon.

Košice, 29. Mája.

.....

vlastnoručný podpis

Pod'akovanie

Rád by som poďakoval vedúcemu práce *Ing. Sergej Chodarev, PhD.* Za umožnenie tohto projektu, odbornú pomoc s realizáciou projektu a pripomienkovaním práce.

Predhovor

Problematika programovacích jazykov je predmetom mnohých výskumov. Väčšina prác sa však zameriava na implementáciu samotného jazyka ako na podporu a integráciu nových jazykov do zaužívaných nástrojov.

Stúdium programovacích jazykov a súvidiacich tém bola vždy moja ambícia. V minulosti som založil a dokončil už dva projekty spojené s podporou doménovo špecifických jazykov v IDE prostrediach. Skúsenosti so systémami ANTLR v3 a Bison boli hlavným motivátorom pre výber tejto témy.

Mojou ambíciou je ukázať, že vývoj nového jazyka môže viesť k jeho integrácii s existujúcimi nástrojmi implicitne, definíciou syntaxe.

Obsah

Zoznam obrázkov.....	10
Zoznam tabuliek.....	11
Zoznam symbolov a skratiek.....	12
Slovník termínov.....	13
Úvod.....	15
1 Formulácia úlohy.....	16
2 Analýza.....	17
2.1 Použité technológie.....	17
2.1.1 YajCo.....	17
2.1.2 Java.....	17
2.1.3 Maven.....	18
2.2 Postup riešenia zadania.....	18
2.2.1 Podpora syntaxe v editoroch.....	18
2.2.1.1 Výber editorov.....	18
2.2.2 Nástroj YajCo.....	19
2.2.2.1 Model jazyka YajCo.....	20
2.2.2.1.1 YajCoModelElement.....	20
2.2.2.1.2 Language.....	20
2.2.2.1.3 Concept.....	20
2.2.2.1.4 SkipDef.....	21
2.2.2.1.5 TokenDef.....	21
2.2.2.1.6 Triedy Abstraktného Syntaxu.....	22
2.2.2.1.7 Triedy Konkrétneho Syntaxu.....	23
3 Syntetická časť.....	24
3.1 Model generátora zvýrazňovania syntaxe.....	24
3.1.1 Elementy gramatiky.....	24
3.1.2 Regulárne výrazy v nástroji YajCo.....	24
3.2 Implementácia generátora syntaxe.....	25
3.2.1 Vstupný bod.....	25
3.2.1.1 Diskusia o zlepšení HighlightingGenerátora.....	26
3.2.2 LanguageVisitor, LanaguageAcceptor a SimpleModel.....	26

3.2.3 Generátor syntaxe, VelocityBackend a EditorStrategy.....	26
3.3 Pravidlá analýzy syntaxe.....	27
3.3.1 Kľúčové slová.....	27
3.3.2 Konštantné hodnoty.....	28
3.3.3 Operátori a výrazy.....	28
3.3.4 Komentáre a biele znaky.....	29
3.4 Analýza regulárnych výrazov.....	29
3.4.1 Nedeterministický Konečný Automat.....	29
3.4.1.1 Znak.....	30
3.4.1.2 Sekvencia.....	30
3.4.1.3 Alternatívy.....	31
3.4.1.4 Kvantifikácia.....	31
3.4.1.5 Transformácia ϵ -NKA na NKA.....	32
3.4.2 Deterministický Konečný Automat.....	34
3.5 Vynechané časti práce.....	35
3.5.1 Editor Notepad++.....	36
3.5.2 Editor CodeMirror.....	36
3.5.3 Editor Vim.....	38
3.5.4 Knížnica Pygments.....	38
3.5.5 Analyzátor regulárnych výrazov.....	38
4 Analýza skutočných jazykov.....	39
4.1 C.....	39
4.2 C++.....	39
4.3 Java.....	40
4.4 Ruby.....	40
4.5 SmallTalk.....	40
4.6 Python.....	41
4.7 Haskell.....	41
4.8 C#.....	42
4.9 Go.....	43
4.10 PHP.....	43
4.11 SQL – SQLite.....	44

5 Záver.....	45
Zoznam použitej literatúry.....	46
Prílohy.....	47

Zoznam obrázkov

Obr. 1: Fragment ε-NFA reprezentujúci sekvenciu.....	31
Obr. 2: Fragment ε-NFA reprezentujúci reláciu alternativity.....	32
Obr. 3: Fragment ε-NFA reprezentujúci kvantifikáciu (Kleene Star). Jednoduchý prípad, kedy máme opakovanie znaku.....	32
Obr. 4: Diagram ε-NKA ekvivalentný ukázkovému regulárnemu výrazu. .	33
Obr. 5: Diagram NFA.....	35
Obr. 6: Diagram DFA.....	36

Zoznam tabuliek

Tab. 1 Cieľové editori prvej implementácie.....	20
Tab. 2 Znaký používané v jazyku BrinFuck, všetky ostatné znaky sú považované za komentár.....	22
Tab. 3: Tabuľka stavových prechodov ϵ-NKA.....	34
Tab. 4: Tabuľková reprezentácia stavového automatu po prevedení do NKA formy.....	34
Tab. 5: Tabuľka stavov DKA, ktorú som vytvoril z pôvodného -NFA.....	36
Tab. 6: Zoznam kľúčových slov jazyka C. Počet: 32.....	40
Tab. 7: Zoznam kľúčových slov jazyka C++. Počet: 50.....	40
Tab. 8: Zoznam kľúčových slov jazyka Java. Počet: 50.....	41
Tab. 9: Zoznam kľúčových slov jazyka Ruby. Počet: 42.....	41
Tab. 10: Zoznam kľúčových slov jazyka SmallTalk. Počet: 6.....	41
Tab. 11: Zoznam kľúčových slov jazyka Python. Počet: 33.....	42
Tab. 12: Zoznam kľúčových slov jazyka Haskell. Počet: 55.....	42
Tab. 13: Zoznam kľúčových slov jazyka C#. Počet: 77.....	43
Tab. 14: Zoznam kľúčových slov jazyka Go. Počet: 25.....	44
Tab. 15: Zoznam kľúčových slov jazyka PHP. Počet: 67.....	44
Tab. 16: Zoznam kľúčových slov jazyka SQL, dialekt SQLite v3. Počet: 126.....	45

Zoznam symbolov a skratiek

API	Aplikačné programové rozhranie (ang.: Application Programming Interface)
AST	Abstraktný strom syntaxu (ang.: Abstract Syntax tree)
DI	Injektáž závislostí (ang.: Dependency Injection)
DKA	Deterministický Konečný Automat
DSL	Doménovo Špecifický Jazyk (ang.: Domain Specific Language)
GNU	General Public License
GUI	Grafické užívateľské rozhranie (ang.: Graphical User Interface)
IDE	Integrované vývojové prostredie (ang.: Integrated Development Environment)
NKA	Nedeterministický Konečný Automat
OS	Operačný Systém
POM	Project Object Model
SQL	Structured Query Language

Slovník termínov

GPL je druh softvérovej licencie, ktorá zaručuje užívateľom prístup k zdrojovým kódom aplikácie

Textový Editor je počítačový program alebo komponenta programu umožňujúca užívateľovi tvoriť a uprovať textové dáta vo forme čistého textu (ang. Plain text). Textový editor neumožňuje pracovať s prezentačnou formou textu. V tejto práci budeme uvažovať jedine editori vhodné na prácu s programovým zdrojovým kódom.

Zdrojový kód - Programové inštrukcie, ktoré tvoria logickú jednotku alebo celý softvérový systém reprezentované vo forme čitateľnej užívateľom. Zdrojový kód je vo väčšine prípadov prezentovaný užívateľovi vo forme čistého textu textovým editorom alebo textovým prehliadačom¹. Spomenuté programy môžu do zdrojového textu vložiť štylistické typografické prvky, ktoré uľahčujú užívateľovi čítanie a orientáciu v zdrojovom texte. Táto funkcionálnosť sa nazýva zvýrazňovanie syntaxe (ang. Syntax Highlighting).

Programovací jazyk - Formálny umelý jazyk, ktorý umožňuje formalizáciu programového návrhu do zdrojového textu².

Doménovo špecifický jazyk je programovací jazyk dedikovaný pre riešenie problémov v špecifickej doméne (ďalej len DSL). Použitelnosť DSL jazykov je obvykle niche, ale objavujú sa často zaradzované k skriptovacím jazykom vo všetkých

¹ Zdrojový kód môže byť užívateľovi prezentovaný grafickou formou. Patria sem nástroje ako Android App Designer, JetBrains Metaprogramming system a iné.

²Pravidlá transformácie návrhu do zdrojového kódu vyplývajú zo sémantiky jazyka, vždy závisia na paradigmatách programovacieho jazyka a programátor ich zvyčajne intuitívne odvodzuje počas písania kódu. Pravidlá pre organizáciu a štruktúrovanie samotného zdrojového kódu sa nazýva syntax. Syntaktické a lexikálne pravidlá tvoria Gramatiku, pridaním sémantiky do gramatiky dostávame programovací jazyk.

druhoch aplikácii od webu, cez finančné systémy až po hry a aj pri tvorbe samotných programovacích jazykov.

Syntax je v informatike súbor prepisovacích pravidiel, ktoré generujú všetky možné dokumenty daného jazyka. Syntax je množina a je súčasťou gramatiky jazyka. Každé pravidlo je kombinácia lexikálnych symbolov

Prepisovacie pravidlo je najmenšia jednotka, s ktorou pracuje syntaktický analyzátor a teda najmenší komponent definície jazyka. Definuje, aká postupnosť lexikálnych jednotiek je validný fragment jazyka.

Zvýrazňovanie syntaxe je funkcia IDE, keď editor mení font textu podľa syntaktického významu daného textu a tým uľahčuje orientáciu v zdrojovom kóde.

Syntaktický analyzátor je softvérová komponenta, ktorá dokáže prečítať informácie z dokumentov, ktoré sú považované za korektne štruktúrovaný dokument alebo fragment dokumentu písaný v danom programovacom jazyku. Výstup syntaktického analyzátora je AST.

Abstraktný Syntaktický Strom (ang.: Abstract Syntax Tree, alebo **AST**) je stromová dátová štruktúra presne popisujúca celý zdrojový kód, kde jednotlivé uzly stromu sú generované podľa hierarchie prepisovacích pravidiel syntaxu jazyka.

Generátor syntaktických analyzátorov je nástroj na jednoduchú tvorbu syntaktických analyzátorov. Transformuje lexikálne a syntaktické pravidla na zdrojový kód syntaktického analyzátora použiteľného integrovateľnú a preložiteľnú do kompletných programov. Pravidlá gramatiky sú zvyčajne formalizované dedikovaným deklaračným programovacím jazykom.

Úvod

Programovacie jazyky sú nedielnou súčasťou sveta informatiky. Vyvinúť programovací jazyk je často sám o sebe nesmierne zložitý problém. Programovací jazyk však nie je tvorený len kompilátorom, patria k nemu aj podporné vývojové nástroje. Jedným z najdôležitejších je editor s podporou zvýrazňovania syntaxe jazyka.

Takmer každý editor má vlastnú implementáciu zvýrazňovania syntaxe a inú sadu podporovaných elementov gramatiky. To znamená, že pre každý editor sme nútený znova definovať veľkú časť gramatiky jazyka.

Automatizácia tohto procesu, aj keď len čiastočná, by umožnila rapidnejší rozvoj programovacích jazykov a aktuálnejšie vývojové nástroje. Problematika je však široká, rôzne implementácie zvýrazňovania syntaxe sú navzájom nekompatibilné. Rovnako sú navzájom nekompatibilné aj generátory syntaktických analyzátorov.

Generátory syntaktických analyzátorov sú takmer ultimativným nástrojom v rukách vývojárov programovacích jazykov a doménovo špecifických jazykov. Sú tak univerzálne, že mnoho z nich je použiteľná aj pri modelovaní sieťových protokolov.

YajCo je nástroj, ktorý je schopný zastrešiť rôzne generátory ako Beaver a JavaCC bez zavádzania nového DSL jazyka. Gramatika je implementovaná pomocou anotácií v kóde Java tried. Zdrojový kód nástroja YajCo je modulárny. Zároveň implementuje celý model, ktorý potrebujeme pre analýzu gramatiky.

Táto práca je pokus o čiastočnú automatizáciu procesu tvorby konfiguračných súborov pre rozličné editory. Podarilo sa nám implementovať 2 editory a diskutovať o metódy vhodné pre vývoj sofistikovanejšej implementácie.

1 Formulácia úlohy

1. Analyzovať spôsoby špecifikácie jazykov pre zvýrazňovanie syntaxe v rôznych editoroch.
2. Analyzovať generátor syntaktických analyzátorov YajCo a spôsob špecifikácie jazyka, ktorý sa v ňom používa.
3. Navrhnuť nástroj, ktorý bude schopný na základe špecifikácie jazyka pre YajCo vygenerovať špecifikáciu zvýrazňovanie syntaxe tohto jazyka pre niektoré editory.
4. Implementovať navrhnutý nástroj a overiť implementáciu na niekoľkých jazykoch.
5. Zhodnotiť riešenie a navrhnuť potenciálne rozšírenia špecifikácie jazykov používanej v YajCo pre zlepšenie zvýrazňovania syntaxe.
6. Vypracovať dokumentáciu podľa pokynov vedúceho práce.

2 Analýza

2.1 Použité technológie

Pri vývoji nástroja automatizovaného zvýrazňovania syntaxe, som mal čiastočne definovanú sadu nástrojov a technológií, ktoré bolo nutné použiť. V tejto kapitole budeme diskutovať každú použitú technológiu a jej význam v projekte a prípadné alternatívy.

2.1.1 YajCo

YajCo je generátor syntaktických analyzátorov. Vstupom pre YajCo je hierarchia Java tried, gramatika jazyka je definovaná pomocou anotácií, ktoré YajCo ponúka. Pri preklade tried nástroj YajCo spracuje anotované triedy a vygeneruje výsledný syntaktický analyzátor. V súčasnosti existujú backendové moduly (adaptéri v terminológii design paternov) pre nástroje Beaver a JavaCC. YajCo bol vyvinutý na Technickej Univerzite v Košiciach.

Nástroj YajCo je navrhnutý modulárne a umožňuje vytvoriť ďalšie backendové moduly bez zmeny samotného nástroja. Nové moduli nemusia implementovať adapter pre generátor syntaktických analyzátorov. To môžeme využiť pre náš účel a vytvoriť backend modul, ktorý bude transformovať popis gramatiky jazyka na pravidlá zvýrazňovania syntaxe pre textové editory.

YajCo je v podstate len ďalšia vrstva nad samotnými DSL jazykmi generátorov syntaktických analyzátorov. Zjednocuje a integruje rôzne generátory syntaktických analyzátorov pod jednotný interface. Touto vlastnosťou YajCo rozširuje rodinu generátorov syntaktických analyzátorov viac vertikálne ako horizontálne. Alternatíva k tomuto nástroju v OpenSource komunite zatiaľ neexistovala a proprietarne produkty s podobnou funkcionalitou sa na trhu nepresadili.

2.1.2 Java

Výber programovacieho jazyka sa podriadi výberu nástroja YajCo. YajCo je napísaný v jazyku Java a na vstupe akceptuje hierarchiu tried definovaných v jazyku Java. Nie je teda dôvod na tomto projekte používať iný jazyk ako Java.

Jednou z výhod jazyku Java je rigidný hierarchický objektový model umožňujúci čistú prácu s reflexiou (Analyza programovej štruktúry počas behu programu). To umožňuje tvoriť programy a najrôznejšie programové knižnice a udržať ich ľahko integrovateľné. Univerzálnosť jazyka Java môže byť príčinou ďalšej výhody jazyka - enormná veľkosť komunity Java programátorov a ľahká dostupnosť programových knižníc.

Spomeňme anotácie v jazyku Java, ktoré ležia v srdci nástroja YajCo. Anotácie umožňujú pridávať do hierarchickej štruktúry programu deklaratívne metadáta. Tieto môžu byť dostupné počas behu programu, alebo počas prekladu programu. Na základe metadát z anotovaných tried a ich polí YajCo generuje syntaktické pravidlá pre backend moduli.

2.1.3 Maven

Rovnako ako pri výbere jazyka, aj pri buildovacom systéme som sa rozhodl kôli lepšej integrácii využiť tie isté nástroje ako využíva YajCo.

2.2 Postup riešenia zadania

V tejto časti budeme hovoriť o tom, ako prebiehalo vypracovanie zadania tejto práce a s akými problémami som sa pri vývoji nástroja Syntaxer stretol.

2.2.1 Podpora syntaxe v editoroch

Prvým krokom pri riešení problému bolo, že som sa pozrel na funkcie zvýrazňovania syntaxe u rôznych editorov a porovnal ich s informáciami poskytnutými nástrojom YajCo.

2.2.1.1 Výber editorov

Pri výbere cieľových editorov prvej implementácie som sa riadil podľa dostupnosťou informácií o konfiguračnom formáte, jednoduchosťou formátu a veľkosťou užívateľskej bázy editorov. Chceli sme pokryť programy pre OS Windows, OS GNU/Linux a Webové platformy. Editori a kľúčové faktory ich výberu sú uvedené v tabuľke Tab. 1 Cieľové editory prvej implementácie.

Tab. 1 Cieľové editory prvej implementácie

Editor	Platforma	Licencia
Notepad++	Windows	Freeware
Kate	GNU/Linux	GPLv3
Gedit	GNU/Linux	GPLv3
Vim	GNU/Linux	GPLv3
Pygments	Python	GPLv3
CodeMirror	Web	Public domain

Editory Notepad++, CodeMirror, Vim Pygments boli nakoniec z implementácie vypustené. Dôvody tohoto rozhodnutia si podrobnejšie rozoberieme v 3.5 Vynechané časti práce.

Implementované sú teda formáty pre editory Kate a Gedit. Oba editory disponujú excelentnou dokumentáciou k syntax highlighting enginu, ktorý implementujú a tiež excelentnou znovupoužiteľnosťou – Každý program pre prostredie Gnome a KDE využívajúci komponentu GSourceView alebo KatePart bude s generovanými konfiguračnými súbormi pracovať automaticky.

2.2.2 Nástroj YajCo

Gramatika implementovaného jazyka je vstupom nástroja YajCo. Gramatika je popísaná anotovanou hierarchiou tried. Vstupný bod nástroja YajCo je podtrieda AnnotationProcessor, YajCo preto beží počas kompilácie Java tried cieľového jazyka.

AnnotationProcessor má počas kompilácie úlohu vykonávať reflexívne operácie nad kompilovanými triedami a meniť ich bytecode, prípadne generovať nové triedy, ktoré vstúpajú do kompilácie ako dynamický generovaný kód v čase kompilácie.

Anotované triedy, ktorými popisujeme cieľovú gramatiku však nie sú reprezentáciou modelu Yajca, model je implementovaný triedami v package YajCo.model. Tieto triedy sú inšanciovane a referencované počas reflexívnej analýzy gramatiky. Naplnený model je následne predaný backend modulu, ktorý je nakonfigurovaný v Project Object Model súbore (pom.xml, ďalej len POM) na spracovanie.

2.2.2.1 Model jazyka YajCo

Rozoberme si triedy modelu YajCo a akú úlohu hrajú v modeli.

2.2.2.1.1 YajCoModelElement

Trieda YajCoModelElement, ako sám názov napovedá, je abstraktná trieda, ktorá je predkom všetkých ostatných tried tvoriacich model jazyka. Takéto tried zvyčajne obsahujú len tie metódy, ktoré sú spoločné pre každý element modelu.

V tomto prípade ide o jedinú metódu getSourceElement(). Elementy modelu sú organizaované do stromu, ak vynecháme sémantickú vrstvu, ktorý môžeme vďaka tejto metóde traverzovať smerom k koreňu. Pre traverz smerom k listom stromu musí traverzujúci algoritmus poznať konkrétne typy elementov a polia týchto typov.

Príklad: Ak máme koreň modelu, musíme vedieť ako je jeho trieda implementovaná aby som mohol traverzovať túto úroveň stromu.

2.2.2.1.2 Language

Koreňom modelu jazyka je inštancia triedy Language a jeho predok je null hodnota. V každom modeli je len jediná inštancia. Potomkovia inštancie Language sú inštancie triedy TokenDef a SkipDef, ktoré majú veľmi podobnú implementáciu ale diametrálne odlišnú sémantiku v modeli jazyka, a trieda Concept.

V samotnom koreňovom uzle nájdeme tiež pole typu String, ktoré ho hodnota je názvom jazyka a pol properties. Tieto polia tu však netreba komentovať.

2.2.2.1.3 Concept

V praxi môžeme považovať za hlavnú časť jazyka v modeli YajCo množinu inštancií triedy Concept. Táto trieda reprezentuje jednu triedu v modeli cieľového jazyka. Trieda patrí do modelu jazyka v prípade, že obsahuje aspoň jednu anotáciu nástroja YajCo.

V terminológii klasických generátorov syntaktických analyzátorov, kde jednotka gramatiky je prepisovacie pravidlo, Concept je súbor syntaktických pravidiel, ktoré sú v relácii alternatív.

2.2.2.1.4 SkipDef

Dve ostatné polia triedy Language sú “skoky” a tokeny. Obe triedy definujú regulárnym výrazom symbol na úrovni lexera. Inštancie triedy SkipDef reprezentujú taký druh slov v jazyku, ktoré svojou postupnosťou znakov ani prítomnosťou v dokumente nemajú žiaden vplyv na výsledok spracovania dokumentu.

Za takéto slová môžeme považovať komentáre, ale v niektorých prípadoch biele znaky.

Príklad: Jazyk BrainFuck ignoruje všetky znaky ASCII abecedy okrem znakov v Tab. 2. Znak používaný v jazyku BrinFuck, všetky ostatné znaky sú považované za komentár:

Tab. 2 Znak používaný v jazyku BrinFuck, všetky ostatné znaky sú považované za komentár

,	+	<	[
.	-	>]

Takýto druh komentárov môžeme popísať jedinou inštanciou SkipDef a to s nasledujúcim regulárnym výrazom:

`[^,.\+<>\\[\]]`

2.2.2.1.5 TokenDef

Inštancie triedy TokenDef definujú symboly jazyka na úrovni lexera vo forme regulárnych výrazov a priradzujú im názov. Pomocou názvov potom môžeme referencovať slová z iných častí modelu cieľového jazyka.

Definovanie symbolov jazyka nie je povinné. Vyhľadanie skutočnej hodnoty terminálneho symbolu, z ktorých skladáme predpisy syntaxe jazyka, prebieha nasledovne:

1. Konštantný textový reťazec použitý v predpise syntaxe (napríklad pomocou anotácie @Before) vyľadáme v mape preddefinovaných symbolov.
2. V prípade, že záznam sa nachádza v mape, predpis (regulárny) pre rozpoznanie terminálneho symbolu bude hodnota priradená danému kľúču v mape. Tieto sú

často znovupoužiteľné kľúčové slová (), primitívne typy, operátori a dekoračné prvky jazyka ako sú zátvorky a oddeľovače.

3. V prípade že záznam sa v mape nenachádza, konštantný reťazec nie je preložený a využije sa ako predpis symbolu.

Hore uvedený postup umožňuje flexibilitu lexikálnej analýzy a však robí jej definíciu náchylnú na skryté chyby (zabudnuté definovanie konštantných názvov operátorov).

2.2.2.1.6 Triedy Abstraktného Syntaxu

Rozoberme si triedy Property, PropertyPattern, Pattern a Identifier a ich interakcie ako celok. Tieto triedy modelujú symboli jazyka, ktorých reťazec ovplyvňuje sémantický význam analyzovaného fragmentu dokumentu a je potrebné mať k tomuto reťazcu prístup z AST. Jedná sa najčastejšie o identifikátory premenných a funkcií, a iné pomenované elementy jazyka. Táto časť modelu YajCo sa v terminológii nástroja nazýva abstraktný syntax.

anotácia @Identifier, ktorá sa používa a úrovni členských premenných tried modelovaného jazyka, berie jeden pomenovaný parameter unique, ktorého hodnota určuje menný priestor identifikátora. V modeli jazyka existuje jedna inštancia triedy Identifier pre každú anotovanú premennú v hierarchii tried modelujúcich cieľovú gramatiku. Modelovanie abstraktného syntaxu umožňuje nástroju YajCo automatizovať časť sémantickej kontroly a to kontrolu tabuľky symbolov voči referenciám. Inými slovami, umožňujú identifikovať nedeklarované premenné a funkcie.

Identifikátor má slabú reláciu voči konkrétnym reťazcom symbolov. Reťazec identifikátora v AST je nastavený klientským kódom v konštruktore, alebo vo factory metode modelovaného Conceptu. Vďaka tomu je možné počas konštrukcie AST identifikátory pozmeniť, alebo dynamicky počítať. Táto stratégia vnáša istú flexibilitu do systému menných priestorov nástroja YajCo. Napríklad je možné implementovať jazyky s dynamickým vyhodnotením identifikátorov, ako napríklad jazyk Ruby.

Hierarchický model abstraktného syntaxu je nasledovný. Pre každú členskú premennú, v triedach modelu jazyka bude object triedy Concept referencovať jednu inštanciu triedy Property. Object Property môže obsahovať v modeli jazyka zoznam

objektov dediacich od abstraktnej triedy `PropertyPattern`. Jedinou implementáciou v nástroji YajCo je trieda `Identifier`. Ak je členská premenná anotovaná ako `@Identifier`, objekt `Property`, ktorý ju reprezentuje bude referencovať jeden objekt `Identifier` a pole `unique` bude obsahovať menný priestor identifikátor.

Táto informácia je užitočná z pohľadu tejto práce pre highliting definovaných premenných a funkcií. Ako však rozoberieme v syntetickej časti práce, problém transformovať túto informáciu na výstupný formát generátora syntax highlitingu, ktorý je valídnym konfiguračným súborom cieľových editorov.

2.2.2.1.7 Triedy Konkrétneho Syntaxu

Konkrétny Syntax je logická časť modelu jazyka, v ktorej nájdeme samotné prepisovacie pravidlá gramatiky a terminálne symboly, ktoré neboli definované pomocou anotácie `@TokenDef`. Konkrétny syntax je modelovaný triedami `Notation`, `NotationPart`, `NotationPartPattern` a ich podtriedami. Trieda `NotationPattern` je využívaná jedine implementáciou factory metód a v tejto práci ju môžeme ignorovať.

Pre každý konštruktor v modeli jazyka (alebo factory metódu) bude objekt triedy `Concept` referencovať jednu inštancciu `Notation`. Pre každý parameter konštruktoru bude objekt `Notation` referencovať jednu inštancciu triedy `NotationPart`. `NotationPart` rozširuje triedu `PatternSupport` a interaguje s triedou `NotationPartPattern` rovnako ako som vysvetlil v prípade tried `Property`, `PropertyPattern` a `Pattern`.

Informácia modelovaná Konkrétnym Syntaxom je dodstupná iba počas kompilácie prekladača a neovplyvňuje sémantickú vrstvu jazyka. Pre problém automatizovania zvýrazňovania syntaxe je táto informácia kľúčová. V syntetickej časti tejto práce budeme diskutovať aké informácie v modeli Konkrétneho Syntaxu môžeme nájsť a akými metódami môžeme tieto informácie analyzovať.

3 Syntetická časť

Jadrom ďalších kapitol je analýza a syntéza vedúca k riešeniu problému.

3.1 Model generátora zvýrazňovania syntaxe

V tejto kapitole sa chcem venovať hlavne štruktúre dát, s ktorými náš nástroj Syntaxer pracuje.

3.1.1 Elementy gramatiky

Popíšme teda elementy gramatiky, ktoré sú z pohľadu zvýrazňovanie syntaxe zaujímavé:

- Kľúčové slová
- Čísllice, reťazce a iné konštantné hodnoty
- Operátori a výrazy
- Komentáre
- Deklarácie, referencie a volania premenných a funkcií
- Makrá a iné špeciálne konštrukty jazyka

Z definície gramatiky nie je problém vygenerovať syntaktický analyzátor, ktorý dokáže rozoznať jednotlivé koncepty gramatiky a vytvoriť príslušnú štruktúrovanú reprezentáciu programu, zatiaľ čo sa ľahko spamätá aj z chýb v programe.

Vyextrahovať však zo špecifikácie gramatiky hore spomenuté elementy gramatiky však nie je triviálna úloha. Špecifikácia gramatiky totiž nemodeluje jednotlivé elementy gramatiky tak ako ich modelujú cieľové editori a ich konfiguračné súbory.

3.1.2 Regulárne výrazy v nástroji YajCo

V prípade nástroja YajCo je situácia ešte o čosi komplikovanejšia, všetky lexikálne jednotky jazyka sú definované pomocou regulárnych výrazov. Nemáme teda vždy k dispozícii samotný reťazec symbolu, ale prepisovacie pravidlo podľa ktorého môžeme generovať všetky možné reťazce vyhovujúce predpisu. Regulárny výraz môže generovať nekonečnú množinu slov, preto nemôžeme generovať túto množinu aby sme ju analyzovali priamo. Musíme dedikovať analytickú metódu, ktorá nám umožní dopytovať sa na vnútornú štruktúru regulárneho výrazu.

Príklad: Vo väčšine jazykov nachádzame kľúčové slová, ktoré sú tvorené jediným slovom bez nealfabetických znakov. Pri rozpoznávaní kľúčových slov v gramatike by som sa dopytovali na nasledovnú informáciu: Sú všetky slová generované predpisom zložené z písmen abecedy (a znakov _ a *, vid. 4. Analýza skutočných jazykov)

Dopytovací systém vhodný pre naše potreby môžeme založiť na teórii stavových automatov a budeme o ňom diskutovať v tejto práci. Implementácia takéhoto systému je však mimo rozsah zadania práce. Zároveň by sa jednalo o veľmi silný nástroj a preto verím, že nadväzujúce práce by sa mohli venovať práve tomuto zaujímavému problému.

3.2 Implementácia generатора syntaxe

Nástroj YajCo je modulárny a umožňuje pridávať nové backendy, generátor zvýrazňovania syntaxe je implementovaný ako takýto plugin. Je spúšťaný procesorom anotácií pri kompilácii zdrojových kódov cieľového jazyka.

Rozoberieme si podrobne architektúru nástroja Syntaxer, akým spôsobom ho umožňujeme používať spoločne s ľubovoľným jazykom a kľúčové faktori, ktoré viedli ku konečnej implementácii. Bol som totiž viazaný API kontraktom nástroja YajCo a ten je zasviazaný na kontrakt Java API pre spracovanie anotácií.

3.2.1 Vstupný bod

Vstupným bodom nástroja Syntaxer je trieda HighlightingGenerator. Táto trieda je pomerne jednoduchá a rozkladá svoje zodpovednosti medzi viaceré členské triedy, ktoré si rozoberieme v ďalšom odseku.

HighlightingGenerator inštanciuje všetky komponenty potrebné k analýze gramatiky: Analyzátor gramatiky, model jazyka a ich prepojenie a tiež backend. Backend sa skladá z dvoch častí: prvá komponenta sú editoru špecifické pravidlá post-procesovania modelu zvýrazňovania, druhá je na editore nezávislá a stará sa o vyhodnotenie predlohy konfiguračného súboru na základe post-processovaného modelu a persistenciu výsledného prúdu.

Pri invokácii dostane HighlightingGenerátor ako parameter model jazyka zostavený nástrojom YajCo, ktorého topológiu som diskutoval v 2.2.1. Model jazyka YajCo.

3.2.1.1 Diskusia o zlepšení HighlightingGenerátora

Syntaxer nemá žiadne ďalšie pohyblivé časti, celá logika je teda pevne zadefinovaná v zdrojovom kóde a HighlightingGenerátor je zodpovedným za inicializáciu programu. Na druhej strane, topológia kódu a zvolené návrhové vzory boli vybrané tak, aby každá časť bola nahraditeľná bez zásahu do ostatných častí.

HighlightingGenerátor je jediný obmedzujúci faktor modularity nástroja Syntaxer. Zavedením Injektáže Závislostí (ang. Dependency Injection, ďalej len DI) by zjednodušila znovupoužiteľnosť a rozšíriteľnosť nástroja YajCo. Konfigurácia Syntaxera by sa v takom prípade mohla nachádzať mimo zdrojových kódov.

DI sme nepoužili, pretože nástroj YajCo, ktorý je naším framework API, žiaden DI framework nevyužíva. Výberom a testovaním DI frameworku v čase spracovania anotácií (v čase kompilácie cieľového jazyka) by nepomohlo dosiahnuť iel' tejto práce.

3.2.2 LanguageVisitor, LanaguageAcceptor a SimpleModel

Model jazyka nástroja YajCo je stromová štruktúra. Problémy reprezentované stromovou štruktúrou

LanguageVisitor má za úlohu analyzovať gramatiku a rozpoznať niektoré koncepty podľa zadefinovaných pravidiel. Napríklad ak sa Token začína na písmeno, tak ho považuje za kľúčové slovo. Pravidlá analýzy syntaxe si rozoberieme v kapitole 3.3 Pravidlá analýzy syntaxe.

3.2.3 Generátor syntaxe, VelocityBackend a EditorStrategy

Posledným článkom pri generovaní syntaxe je z rozanalyzovaný model jazyka vytvoriť výslednú definíciu zvýrazňovania syntaxe. Túto úlohu vykonáva trieda VelocityBackend. Jediná metóda write() berie ako parameter naplnený model jazyka, tento predá template processoru Velocity, ten naplní preddefinovanú šablónu údajmi z rozanalyzovaného modelu a VelocityBackend zabezpečí spoľahlivý zápis na disk, alebo inú formu výstupu (napríklad výpísanie nespracované výsledky analýzy.)

3.3 Pravidlá analýzy syntaxe

Rozoberme si jednotlivé pravidlá zakódované do nástroja Syntaxer. Je nutné si uvedomiť, že pravidlá ktoré nasledujú, nie sú všeobecne platné pre všetky jazyky, ktoré môžeme uvažovať.

3.3.1 Kľúčové slová

V analytickej časti tejto práce som už v skratke o rozpoznávaní kľúčových slov hovoril. V 4. Analýza skutočných jazykov nájdeme tabuľky kľúčových slov vybraných jazykov. Z uvedených tabuliek si môžeme všimnúť hneď niekoľko vlastností:

1. Všetky kľúčové slová sa skladajú z veľkých a malých znakov abecedy, podtrhovníka `_` a v niektorých jazykoch z ďalších znakov³
2. Veľké a malé písmená sa nemiešajú
3. Kľúčové slová nie sú dynamické, takže poznáme ich presný reťazec

V implementácii som sa rozhodl využiť fakty 1. a 3. Fakt 2. je viac menej náhoda, aj keď v našej vzorke prominentná.

Takto môžeme vysloviť nasledujúce tvrdenie o regulárnom výraze slova, ktoré predstavuje v jazyku kľúčové slovo:

Ak regulárny výraz popisuje kľúčové slovo, pravdepodobne neobsahuje žiadne špeciálne znaky, čísla, kvantifikátori a iné operátori, okrem podtrhovníka.

³Spomeňme, že v niektorých jazykoch ako je Haskell je pojem kľúčového slova trochu širší a kľúčové slovo môže obsahovať akýkoľvek znak. Podobne je to aj s jazykom Ruby, kde kľúčové slová sú v skutočnosti len volania funkcií so špeciálnym syntaxom, takže kľúčové slová sa riadia rovnakými pravidlami ako identifikátori jazyka Ruby a môžu ako posledný znak obsahovať otáznik `?` Alebo výkričník `!`. Ďalej jazyk PHP definuje `exit()` a `echo()` so zátvorkou a variabilným počtom medzier pred a medzi zátvorkami.

3.3.2 Konštantné hodnoty

Nástroj YajCo má zabudované rozpoznávanie konštánt niektorých typov ako sú celé čísla a čísla s pohyblivým exponentom. Jedná sa o pravidlá natvrdo zabudované do kódu na úrovni lexera. Keďže lexer a jeho API je závislé na tom, ktorý backend YajCa použijeme ako generátor kódu, nemôžeme ovplyvniť ako sú pravidlá implementované v moduli JavaCC a Beaver.

Čísla majú však ďalšie nezdokumentované obmedzenie, musia byť reprezentované v desiatkovej číselnej sústave. Podľa našich testov, syntaktický analyzátor vygenerovaný nástrojom YajCo nerozoznáva čísla reprezentované v osmičkovej, ani hexadecimálnej číselnej sústave. Rozoznávajúce nie sú ani sufixi určujúce typ konštanty⁴.

Reťazce reprezentujú ďalší problém. Autor jazyka musí implementovať Concept, ktorý reťazce znakov implementuje a popisuje ich na úrovni lexikálneho analyzátoru. To môžeme dosiahnuť tým, že parametre Conceptu anotujeme anotáciou @Token, implementujeme interface algebraických výrazov v Concepte a ošetríme preklad špeciálnych znakov (ang.: escape characters) z reprezentácie v zdrojovom kóde na ich hodnotu v čase behu programu.

3.3.3 Operátori a výrazy

Pri rozpoznávaní operátorov som využil centrálnu myšlienku API nástroja YajCo a tou sú koncepty. Ako som spomínal, pre každú triedu v popise gramatiky je vytvorená jedna inštancia triedy Concept. Objekty triedy Concept majú zoznam predprogramovaných konceptových vzorov, ktoré sa na daný Concept vzťahujú. Conceptové vzory do konceptu pripájame jednou z nasledovných anotácií:

- Operátor
- Enum
- Parentheses

LanguageVisitor drží stav pre každý objekt triedy Concept, ktorý navštívil a pri rozpoznávaní jednotlivých tokenov, aplikuje rozličné pravidlá, podľa toho, či sa jedná o operátor alebo nie.

Ak sa jedná o operátor, ignorujeme potencionálne kľúčové slová a každý neprázdny operátor pridáme do zoznamu operátorov.

⁴V jazykoch C a Java hodnotu 1 typu long zapisujeme pomocou sufixu: 1L

Výrazy sa skladajú z členov (konštanty, premenné) a operátorov, preto sú implicitne implicitne pokryté z veľkej časti aj výrazy.

3.3.4 Komentáre a biele znaky

Toto pravidlo sa aplikuje iba na zoznam anotácií @SkipDef. Ak slovo bude akceptované regulárnym výrazom z niektorej anotácie @SkipDef, bude brané ako biely znak.

3.4 Analýza regulárnych výrazov

Systém pre kategorizáciu tokenov použitý v implementácii nástroja Syntaxer je veľmi obmedzený. Ignoruje fakt, že terminálne symboly sú definované ako regulárne výrazy a nemáme priamy prístup k reťazcom ktoré generujú.

Hľadal som spôsob, akým by som sa mohol dopytovať na kľúčové vlastnosti generovaných jazykov, ktoré sú typické pre rozličné elementy gramatiky z vybraných jazykov. Príkladom takéhoto dopytu môže byť ekvivalencia regulárnych výrazov alebo vyhodnotenie množiny všetkých možných znakov na n-tom mieste vo všetkých slovách v jazyku.

Aby som mohli riešiť dopyty nad regulárnym výrazom, musíme ho preložiť do chodnej formy. Z regulárneho výrazu môžeme parsovaním vytvoriť strom prepisovacích vnorených pravidiel, ktorého listy sú znaky abecedy.

Takýto strom môžeme zostaviť pomocou 3 typov uzlov: Sekvencia, Alternatíva a Kvantifikácia.

3.4.1 Nedeterministický Konečný Automat

Niektoré dotazy sú riešené na parse strome regulárneho výrazu, existuje však ekvivalentná reprezentácia, ktorá nám umožňuje riešiť viac úloh. Je ňou Nedeterministický Konečný Automat (ďalej len NKA).

Príklad: Pomocou parse stromu regulárneho výrazu môžeme zistiť, ktoré znaky sa nachádzajú na prvom mieste v generovaných slovách. Ak problém zovšeobecníme na n-tý znak v generovaných slovách, s touto reprezentáciou sa vnorené opakovania sa stanú zložitým implementačným problémom.

Transformácia regulárneho výrazu na NKA sa riadi niekoľkými pravidlami:

1. Každý list t.j. Znak abecedy sa stane hranou NKA

2. Každý nelistový uzol bude transformovaný na diskretný fragment NKA s jedným vstupným stavom a jedným konečným stavom
3. Žiadna hrana nesmie z vonku fragmentu nesmie vstupovať do iného ako vstupného stavu.
4. Žiadna hrana z vnútra fragmentu nesmie opúšťať fragment, okrem hrán konečného stavu.

Traverziou stromu môžeme vybudovať a správne vnoriť každý fragment v post-order poradí. Zdefinujme špeciálny typ NKA, ktorý budeme vytvárať.

Definícia: Nedeterministický konečný automat s ϵ -prechodmi je päťica

$$A = (K, \Sigma, \delta, q_0, F) \quad \text{kde}$$

- K je konečná množina stavov
- Σ je konečná vstupná abeceda
- $q_0 \in K$ je začiatkový stav
- $F \subseteq K$ je množina akceptačných (konečných) stavov
- $\delta : K \times \Sigma_\epsilon \rightarrow P(K)$ je prechodová funkcia

a zároveň

- $\epsilon \notin \Sigma$
- $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$

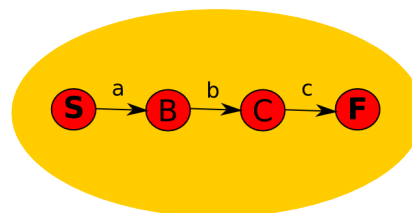
3.4.1.1 Znak

Na pozícii listov stromu sa nachádzajú znaky. Tie budeme modelovať ako primy prechod zo začiatkového stavu do konečného cez daný znak.

3.4.1.2 Sekvencia

Sekvencie je pomerne jednoduché namodelovať. Potomkovia sekvencie sa stanú prechodmi medzi lineárnou postupnosťou stavov, v poradí.

Ak je potomok nie je znakom abecedy, konečný stav predchádzajúceho potomka spojíme ϵ -prechodom so štartovacím stavom

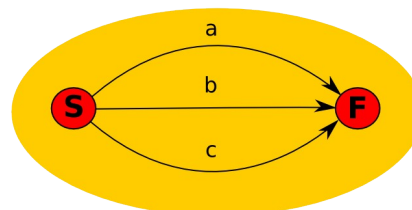


Obr. 1: Fragment ϵ -NFA reprezentujúci sekvenciu

spracovávaného potomka. Rovnako jeho konečný stav spojíme ε -prechodom so štartovacím stavom nasledujúceho potomka, alebo konečným stavom sekvencie.

3.4.1.3 Alternatívy

Začiatocný a konečný stav prepojíme prechodmi označenými znakmi v alternatíve. V prípade, že v alternatíve nájdeme iného potomka ako znak, spojíme začiatocný stav alternatívy so začiatocným stavom potomka ε -prechodom a to isté urobíme aj pre ich konečné stavy.



Obr. 2: Fragment ε -NFA

reprezentujúci reláciu alternatívy

3.4.1.4 Kvantifikácia

Kvantifikácia je odlišná od Sekvencie a Alternatívy tým, že má jediného potomka.

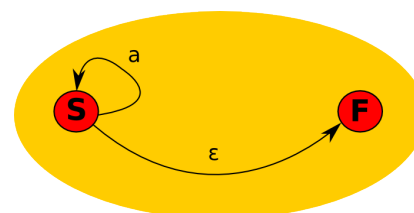
Pri transformácii kvantifikátorov na ε -NKA musíme brať do úvahy o aký typ kvantifikácie sa jedná. V regulárnych výrazoch rozšíreného unixového typu nachádzame 4 rôzne zápisy kvantifikátorov, všetky vo forme postfix operátora:

- $*$ prefix sa opakuje 0 a viackrát
- $+$ prefix sa opakuje aspoň raz
- $?$ prefix sa nachádza na danej pozícii raz alebo vôbec
- $\{n,m\}$ prefix sa opakuje minimálne n -krát a maximálne m -krát

Operátor $\{n, m\}$ diskutovať nebudeme, pretože sa používa zriedkavo. Čitateľ sa nad jeho transformáciou môže zamyslieť samostatne.

Vychádzame z toho, že začiatocné stavy a konečné stavy fragmentu a potomka sú prepojené ε -prechodmi. Ďalej nás budú zaujímať dve vlastnosti:

1. Môže byť potomok vynechaný? (objavovať sa 0-krát)
Ak áno, vložíme ε -prechod zo začiatocného do konečného stavu fragmentu
2. Môže sa potomok objavovať viac ako raz?
Ak áno, vložíme ε -prechod zo konečného do začiatocného stavu fragmentu



Obr. 3: Fragment ε -NFA

reprezentujúci kvantifikáciu (Kleene Star). Jednoduchý prípad, kedy máme opakovanie znaku.

V tomto kroku si musíme uvedomiť, že so zavádzaním kvantifikácie do nášho modelu, vytvárame slučku skladajúcu sa z ε -prechodov. Prichádzame tak o jeden predpoklad, ktorý by sme mohli uvažovať, ak by sme v algebre regulárnych výrazov nemali kvantifikáciu. Algoritmy, ktoré budú pracovať nad ε -NKA, ako napríklad transformácia na NKA, musia ošetriť tento okrajový prípad.

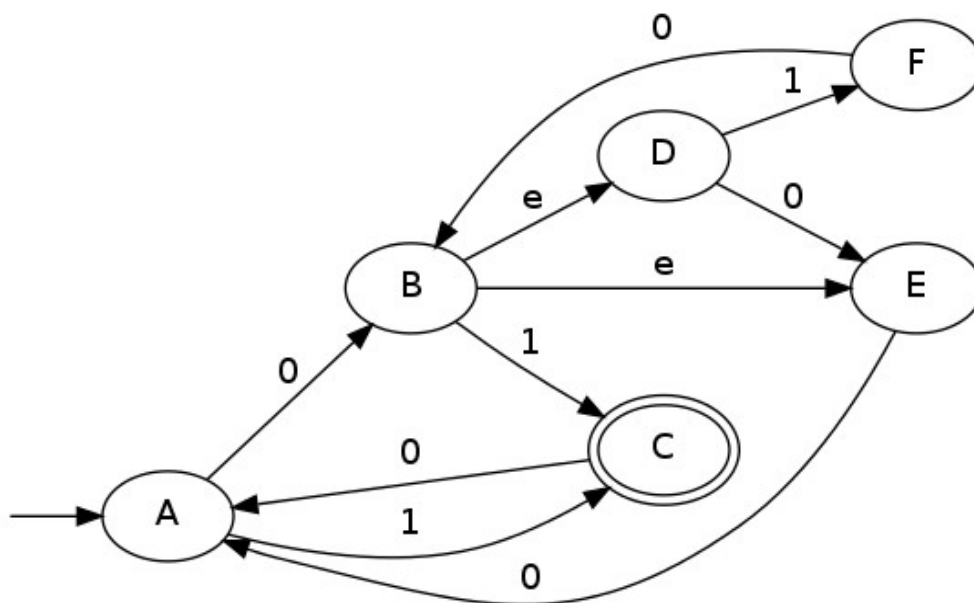
3.4.1.5 Transformácia ε -NKA na NKA

Vytvorenie ε -NKA je medzikrok k vytvoreniu NKA. Reprezentujme NKA ako tabuľku, ktorá modeluje prechody medzi stavmi. Budeme transformovať nasledovný regulárny výraz:

$$((0((1|00+1)00) * (100+1)0)1|1)0) * 0((1|00+1)00) * (1|00+1)0)1$$

Nakoľko celý konečný automat zostrojený podľa hore uvedených pravidiel by bol veľmi rozsiahly, v nasledujúcom obrázku a tabuľke som uviedol ekvivalentný ε -NFA, u ktorého som redukoval všetky spojitý ostrovy stavov prepojené medzi sebou len ε -prechodmi.

Obr. 4: Diagram ε -NKA ekvivalentný ukázkovému regulárnemu výrazu



Tab. 3: Tabuľka stavových prechodov ε -NKA

Stav	0	1	ε
A	B	C	\emptyset
B	\emptyset	C	{D, E}
C	A	\emptyset	\emptyset
D	E	F	\emptyset
E	A	\emptyset	\emptyset
F	B	\emptyset	\emptyset

Cieľom je odstrániť stĺpec ε .

Definícia: Nech $\delta^*(q_0, w)$ je množina všetkých stavov, ktoré môžeme dosiahnuť zo stavu q_0 nasledovaním cesty w . Tejto funkcii sa po anglicky hovorí „Closure function“.

Príklad:

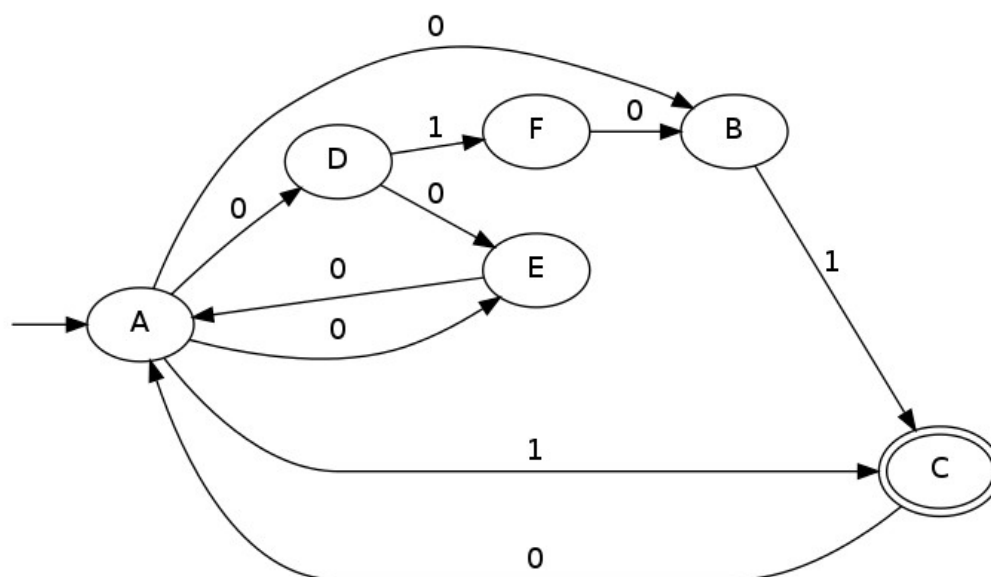
1. $\delta^*(A, \varepsilon) = \{A\}$
2. $\delta^*(A, 0) = \{B, D, E\}$
3. $\delta^*(A, 01) = \{C, F\}$

Tabuľku NKA zostavíme dosadením stavu q a prechodu x do funkcie $\delta^*(q, x)$:

Tab. 4: Tabuľková reprezentácia stavového automatu po prevedení do NKA formy

Stav	0	1
A	{B, D, E}	C
B	\emptyset	C
C	A	\emptyset
D	E	F
E	A	\emptyset
F	B	\emptyset

Obr. 5: Diagram NFA



3.4.2 Deterministický Konečný Automat

Prevodom na Deterministický Konečný Automat (ďalej len DKA) dostaneme k dispozícii možnosť testovať ekvivalenciu dvoch DKA a rôzne iné algoritmi vyvinuté nad DKA.

Prevod na DKA vychádza z ekvivalencie DKA a NKA. Začíname v začiatočnom stave NKA. Pre každý možný prechod v aktualnej kombinácii stavov môžeme použiť vytvoríme jeden prechod v DKA a cieľový stav bude reprezentovať kombináciu stavov, do ktorých sa simuláciou NKA dostaneme po čítaní daného znaku.

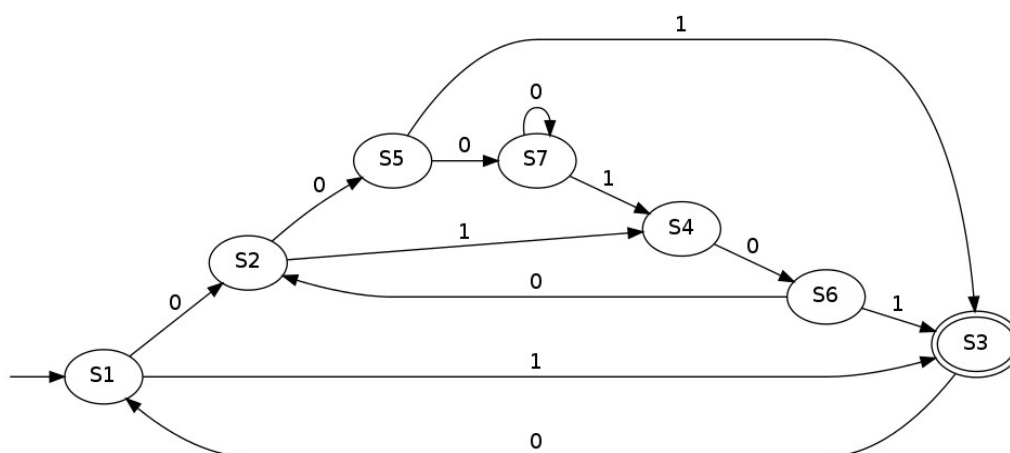
Toto vykonávame iteratívne, kým nevytvoríme tabuľku všetkých kombinácií stavov, do ktorých je možné sa dostať simuláciou NKA. Výsledná tabuľka je prekvapivo krátka:

Tab. 5: Tabuľka stavov DKA, ktorú som vytvoril z pôvodného -NFA

Stav	0	1
$S1 = \{ A \}$	$\{ B, D, E \} = S2$	$\{ C \} = S3$
$S2 = \{ B, D, E \}$	$\{ A, E \} = S5$	$\{ C, F \} = S4$
$S3 = \{ C \}$	$\{ A \} = S1$	\emptyset
$S4 = \{ C, F \}$	$\{ A, B \} = S6$	\emptyset
$S5 = \{ A, E \}$	$\{ A, B, D, E \} = S7$	$\{ C \} = S3$
$S6 = \{ A, B \}$	$\{ B, D, E \} = S2$	$\{ C \} = S3$
$S7 = \{ A, B, D, E \}$	$\{ A, B, D, E \} = S7$	$\{ C, F \} = S4$

Jednou z výhod, tejto reprezentácie je, že DKA je možné upraviť a opätovne previesť na regulárny výraz. Tým by sa otvorila možnosť robiť okrem dopytov nad regulárnymi výrazmi aj transformácie a optimalizácie regulárnych výrazov.

Obr. 6: Diagram DFA



3.5 Vynechané časti práce

Niektoré plánované komponenty práce nakoniec neboli zahrnuté do imlementácie Syntaxera. Zosumarizujme si všetky vynechané časti práce a rozoberme dôvody, ktoré viedli k týmto rozhodnutiam.

3.5.1 Editor Notepad++

Tento syntax zvýrazňujúci editor často nachádzame odišťlovaný na windowsovských klientských strojoch. Bol to tiež jediný editor pre platformu Windows, ktorým som sa plánoval zaoberať.

Rozbor konfiguračného formátu zvýrazňovania syntaxe Notepad++ bol zabrzdený slabou dokumentáciou tohto formátu. V nástroji nájdeme grafické užívateľské rozhranie (ďalej len GUI) určené pre tvorbu nových a dialógy GUI sú jedinou dokumentáciou, ktorá nám bola k dispozícii prostredníctvom internetu.

Bol som nútený pokúsiť sa o reverzný inžiniering výstupného formátu GUI rozhrania. Kedže to znamenalo, že nutnosť venovať viac času editoru Notepad++ ako ostatným editorom, Notepad++ sa prirodzene stal menej výhodným pre splnenie hlavného cieľa práce.

3.5.2 Editor CodeMirror

Jedná sa o webový editor zdrojových kódov implementovaný v jazyku JavaScript. Editor CodeMirror alebo jeho zdrojové kódy si našli uplatnenie v mnohých webovo orientovaných projektoch.

Konfigurácia tohto editoru sa nepodobá na žiaden iný editor, ktorému som sa venoval v rámci práce. Zatiaľ čo ostatné editori⁵ používajú deklaratívny prístup, pre CodeMirror musíme napísať lexovaciu funkciu v JavaScripte.

Definícia: Lexovacia funkcia je transformácia toku znakov na tok slov

$$L(\{c_1, c_2, c_3, \dots, c_n\}) \rightarrow \{w_1, w_2, \dots, w_m\}$$

API kontrakt Lexovacej funkcie v Javascripte vyzerá nasledovne:

- CodeMirror drží stav procesu zvýrazňovania syntaxe a to pozíciu vo vstupnom reťazci znakov, do ktorej prebehlo rozpoznávanie a pozíciu do ktorej prebehlo čítanie ďalšieho slova.
- Funkcia lexera token(next) berie jeden parameter a vráti reťazec reprezentujúci typ nasledujúceho slova.

⁵Tu nepočítame knižnicu Pygments pre jazyk Python. Pygments je konfigurovaný imperatívnym kódom napísaným v jazyku Python.

- Parameter `next()` je funkcia, ktorá prečíta a vráti nasledujúci znak neprečítaný znak vo vstupnom reťazci.
- Po skončení volania `token()` sa pozícia rozpoznávania nastaví na aktuálnu pozíciu nasledujúceho neprečítaného znaku. Volanie sa opakuje, kým ne je rozoznaná syntaxu všetkého kódu viditeľného na obrazovke.

Pri editore CodeMirror som narazil na rozsiahly, aj keď riešiteľný problém a to: YajCo používa na popis slov gramatiky regulárne výrazy, CodeMirror nám umožňuje čítať iba jediný znak a po jeho prečítaní stane sa súčasťou nasledovného slova⁶ ktoré musí byť vyhovieť popisu tokenu v gramatike. Jedným riešením je nasledovný algoritmus:

1. Prečítaj nasledujúci znak a pridaj ho na koniec aktuálneho reťazca.
2. V cýkle testuj všetky regulárne výrazy na celom aktuálnom reťazci.
3. Prvý výraz, ktorý úspešne akceptuje aktuálny reťazec definuje typ nasledovného slova
4. Pokiaľ neprečítaš celý vstup, vráť sa na krok 1

Aktuálne API nie je dostatočne flexibilné, aby bola takáto implementácia triviálna. JavaScript je pomerne pomalý jazyk, aj dnes v moderných virtuálnych strojoch. Prvé testy ukázali, že táto metóda môže byť veľmi pomalá. Pri 500 slovách v zdrojovom kóde a jazyku, ktorý definuje 100 rôznych tokenov, kde každé slovo na vstupe má 5 znakov, budeme vykonávať 250'000 dopytov na regulárne výrazy.

Jednoduchý test ukázal, že jedna invokácia lexera môže trvať viac ako 10 sekúnd. Lexer môže byť volaný pri zmene ľubovoľného znaku v zdrojovom kóde. To by bolo kontraproduktívne a riešenie nie je triviálne. Z toho dôvodu som sa rozhodl odložiť optimalizáciu generátora pre editor CodeMirror.

⁶CodeMirror umožňuje aj nahliadnuť na nasledujúci znak bez jeho prečítania. To nám dáva iba jeden znak lookahead, takže tvrdenie v texte nie je úplne presné, ale pre naše účely stačí.

3.5.3 Editor Vim

Editor Vim využíva ako konfiguráciu imperatívny DSL jazyk zabudovaný do programu Vim. Pravidlá sú však deklaratívne, takže je možné prispôbiť existujúce šablóny.

Tento editor je veľmi odlišný od ostatných editorov. Je nutné sa s takýmto editorom najprv naučiť pracovať. Našťastie dokumentácia je rozsiahla a užívateľská báza široká.

Z toho dôvodu som sa sústredil najprv na dokončenie aspoň tých editorov, ktoré som vedel spoľahlivo a rýchlo testovať.

3.5.4 Knižnica Pygments

Rovnako ako pri editore Vim, imperatívny systém konfigurácie je odlišný od deklaratívneho prístupu v mnohých iných editoroch a stál by príliš veľa času popri doladovaní zvyšku zdrojového kódu.

3.5.5 Analyzátor regulárnych výrazov

Táto komponenta zabrala najviac času zo všetkých. Nakoľko by to znamenalo, že môžeme vykonávať rôzne algoritmy nad regulárnym výrazom vo vhodnej a veľmi silnej reprezentácii.

Je nutné implementovať syntaktický analyzátor regulárnych výrazov. Tu sme sa pozerali na niektoré OpenSource implementácie, ale neuspeli sme v hľadaní takej, ktorá by bola ľahko integrovateľná s plánovanými algoritmami. Neskúmal som však žiadnu prácu urobenú priamo na univerzite, čo by však mohlo byť úspešné hľadanie.

Urobil som rozhodnutie napísať si vlastnú implementáciu syntaktického analyzátora regulárnych výrazov v Jave. Bolo by dokonca možné použiť samotný nástroj YajCo. Potrebujeme parsovať krátke regulárne výrazy podporované v Jave a vytvoriť jeho AST reprezentáciu a následne vykonať sériu transformácií, ktorými dospejeme k DFA.

Podľa pravidiel pre každý uzol AST vykonáme transformačnú operáciu a iteratívne naplníme celú tabuľku stavov ϵ -NKA. Tento automat môžeme simulovať a odpovedať na jednoduché dotazy, alebo môžeme ho previesť na DKA. DKA umožňuje odpovedať na dotazy typu ekvivalencia dvoch regulárnych výrazov.

4 Analýza skutočných jazykov

V tejto časti práce uvidíme zoznam kľúčových slov vybraných programovacích jazykov, zhromaždený pre účely analýzovania reprezentatívnej množiny existujúcich jazykov. Počet kľúčových slov každého jazyka je uvedený v popise každej tabuľky.

4.1 C

Tab. 6: Zoznam kľúčových slov jazyka C. Počet: 32

auto	do	goto	signed	unsigned
break	double	if	sizeof	void
case	else	int	static	volatile
char	enum	long	struct	while
const	extern	register	switch	
continue	float	return	typedef	
default	for	short	union	

4.2 C++

Tab. 7: Zoznam kľúčových slov jazyka C++. Počet: 50

abstract	boolean	byte	catch	class
continue	do	else	extends	finally
for	if	import	int	long
new	private	public	short	strictfp
switch	this	throws	try	volatile
asser	break	case	char	cons
default	double	enum	final	float
got	implements	instanceof	interface	native
package	protected	return	static	super
synchronized	throw	transient	void	while

4.3 Java

Tab. 8: Zoznam klúčových slov jazyka Java. Počet: 50

abstract	do	import	short	volatile
continue	if	public	try	const
for	private	throws	char	float
new	this	case	final	native
switch	break	enum	interface	super
assert	double	instanceof	static	while
default	implements	return	void	
goto	protected	transient	class	
package	throw	catch	finally	
synchronized	byte	extends	long	
boolean	else	int	strictfp	

4.4 Ruby

Tab. 9: Zoznam klúčových slov jazyka Ruby. Počet: 42

BEGIN	break	ensure	or	undef
END	case	FALSE	redo	unless
__ENCODING__	class	for	rescue	until
__END__	def	if	retry	when
__FILE__	defined?	in	return	while
__LINE__	do	module	self	yield
alias	else	next	super	
and	elsif	nil	then	
begin	end	not	TRUE	

4.5 SmallTalk

Tab. 10: Zoznam klúčových slov jazyka SmallTalk. Počet: 6

true	nil	self	super	thisContext
false				

4.6 Python

Tab. 11: Zoznam klíčových slov jazyka Python. Počet: 33

FALSE	for	while	elif	pass
class	lambda	and	if	break
finally	try	del	or	except
is	TRUE	global	yield	in
return	def	not	assert	raise
None	from	with	else	
continue	nonlocal	as	import	

4.7 Haskell

Tab. 12: Zoznam klíčových slov jazyka Haskell. Počet: 55

!	,	`	default	let, in
'	=	{, }	deriving	mdo
''	Err:510	{-, -}	deriving instance	module
-	>		do	newtype
--	?	~	forall	proc
-<	#	as	foreign	qualified
-<<	*	case, of	hiding	rec
->	@	class	if, then, else	type
::	[,]	data	import	type family
;	\	data family	Infix, infixl, infixr	type instance
<-	-	data instance	instance	where

4.8 C#

Tab. 13: Zoznam kľúčových slov jazyka C#. Počet: 77

abstract	do	in	protected	TRUE
as	double	int	public	try
base	else	interface	readonly	typeof
bool	enum	internal	ref	uint
break	event	is	return	ulong
byte	explicit	lock	sbyte	unchecked
case	extern	long	sealed	unsafe
catch	FALSE	namespace	short	ushort
char	finally	new	sizeof	using
checked	fixed	null	stackalloc	virtual
class	float	object	static	void
const	for	operator	string	volatile
continue	foreach	out	struct	while
decimal	goto	override	switch	
default	if	params	this	
delegate	implicit	private	throw	

4.9 Go

Tab. 14: Zoznam kľúčových slov jazyka Go. Počet:25

break	case	chan	const	continue
default	defer	else	fallthrough	for
func	go	goto	if	import
interface	map	package	range	return
select	struct	switch	type	var

4.10 PHP

Tab. 15: Zoznam kľúčových slov jazyka PHP. Počet: 67

__halt_compiler()	default	exit()	insteadof	static
abstract	die()	extends	interface	switch
and	do	final	isset()	throw
array()	echo	finally	lis)	trait
as	else	for	namespace	try
break	elseif	foreach	new	unset()
callable	empty()	function	or	use
case	enddeclare	global	print	var
catch	endfor	goto	private	while
class	endforeach	if	protected	xor
clone	endif	implements	public	yield
const	endswitch	include	require	
continue	endwhile	include_once	require_once	
declare	eval()	instanceof	return	

4.11 SQL – SQLite

ABORT	CREATE	FROM	NATURAL	ROLLBACK
ACTION	CROSS	FULL	NO	ROW
ADD	CURRENT_DATE	GLOB	NOT	SAVEPOINT
AFTER	CURRENT_TIME	GROUP	NOTNULL	SELECT
ALL	CURRENT_TIMESTAMP	HAVING	NULL	SET
ALTER	DATABASE	IF	OF	TABLE
ANALYZE	DEFAULT	IGNORE	OFFSET	TEMP
AND	DEFERRABLE	IMMEDIATE	ON	TEMPORARY
AS	DEFERRED	IN	OR	THEN
ASC	DELETE	INDEX	ORDER	TO
ATTACH	AUTOINCREMENT	INDEXED	OUTER	TRANSACTION
DESC	DETACH	INITIALLY	PLAN	TRIGGER
BEFORE	DISTINCT	INNER	PRAGMA	UNION
BEGIN	DROP	INSERT	PRIMARY	UNIQUE
BETWEEN	EACH	INSTEAD	QUERY	UPDATE
BY	ELSE	INTERSECT	RAISE	USING
CASCADE	END	INTO	RECURSIVE	VACUUM
CASE	REFERENCES	IS	ESCAPE	VALUES
CAST	EXCEPT	ISNULL	REGEXP	VIEW
CHECK	EXCLUSIVE	JOIN	REINDEX	VIRTUAL
COLLATE	EXISTS	KEY	RELEASE	WHEN
COLUMN	EXPLAIN	LEFT	RENAME	WHERE
COMMIT	FAIL	LIKE	REPLACE	WITH
CONFLICT	FOR	LIMIT	RESTRICT	
CONSTRAINT	FOREIGN	MATCH	RIGHT	

Tab. 16: Zoznam kľúčových slov jazyka SQL, dialekt SQLite v3. Počet: 126

5 Záver

V tejto práci som ukázal akým spôsobom súvysia definície gramatiky jazyka a konfigurácie systémov zvýrazňovania syntaxe. Ukázali sme tiež, že vykonávať zvýrazňovanie syntaxe na základe definície gramatiky nie je triviálna úloha.

Opísal som metódu spracovania regulárnych výrazov do reprezentácie deterministického konečného automatu. V práci som vysvetlil, prečo takáto reprezentácia nám má umožniť zvýšiť presnosť zvýrazňovania syntaxe. Tento systém som však neimplementoval, pretože ďaleko presahoval rozsah práce.

Z implementácie som tiež vynechal editori Notepad++ a CodeMirror a tiež nástroj Pigments. Tieto implementácie by mohli byť témou nadväzujúcich prác.

Zoznam použitej literatúry

- [1] ROVAN, Branislav – FORIŠEK, Michal: Formálne Jazyky a Automaty. Košice : TU-FEI, 2008. 88 str.
- [2] PORUBĀN, Jaroslav: Návrh a implementácia počítačových jazykov, Košice : TU-FEI, 2008. 124 str.
- [3] Dokumentácia editoru Gedit [online]. Dostupné na internete: <<https://wiki.gnome.org/action/show/Projects/GtkSourceView>>.
- [4] WRITING A SYNTAX HIGHLIGHTING FILE [online]. Dostupné na internete: <<http://kate-editor.org/2005/03/24/writing-a-syntax-highlighting-file/>>.
- [5] Dokumentácia editoru CodeMirror [online]. Dostupné na internete: <<http://codemirror.net/>>.
- [6] GONDA, Vladimír: Ako napísať a úspešne obhájiť diplomovú prácu. Bratislava : Elita, 2003. 124 s. : il. ISBN 80-8044-076-X
- [7] KATUŠČÁK, Dušan : Ako písať záverečné a kvalifikačné práce. Nitra: Enigma, 2004. 162 s. il. ISBN 80-89132-10-3

Prílohy

Príloha A: CD médium – diplomová práca v elektronickej podobe, prílohy v elektronickej podobe.