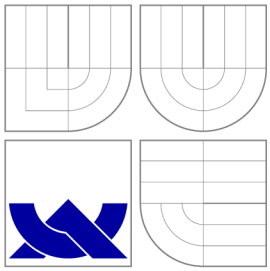
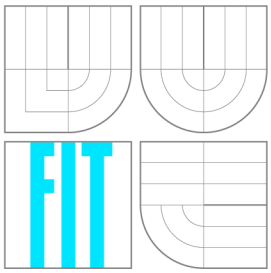


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

SYNTAKTICKÝ ANALYZÁTOR STYLOVÝCH PŘEDPISŮ CSS

CASCADING STYLE SHEETS PARSER

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. RADEK SEDLÁK

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. RADEK BURGET, Ph.D.

BRNO 2016

Zadání diplomové práce/18395/2015/xsedla90

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2015/2016

Zadání diplomové práce

Řešitel: **Sedlák Radek, Bc.**
Obor: Informační systémy
Téma: **Syntaktický analyzátor stylových předpisů CSS**
Cascading Style Sheets Parser
Kategorie: Překladače

Pokyny:

1. Prostudujte problematiku syntaktických analyzátorů a jejich generování na základě dodaného předpisu (gramatiky).
2. Seznamte se s existujícími generátory syntaktických analyzátorů na platformě Java. Zaměřte se zejména na nástroj ANTLR verze 3 a 4.
3. Seznamte se s projektem jStyleParser a syntaktickým analyzátorem, který využívá.
4. Navrhněte obdobný syntaktický analyzátor pro tento projekt s využitím generátoru ANTLRv4.
5. Implementujte navržený analyzátor.
6. Proveďte testování výsledného analyzátoru.
7. Zhodnoťte dosažené výsledky a navrhněte další možná rozšíření.

Literatura:

- T. Parr: The Definitive ANTLR 4 Reference, Pragmatic Bookshelf, 2013
- Dokumentace k projektu jStyleParser: http://cssbox.sourceforge.net/jstyleparser/documentation.php

Při obhajobě semestrální části projektu je požadováno:

- Body 1 až 4

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Burget Radek, Ing., Ph.D.**, UIFS FIT VUT
Datum zadání: 1. listopadu 2015
Datum odevzdání: 25. května 2016

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav informačních systémů
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Dušan Kolář
vedoucí ústavu

Abstrakt

Tato práce se zabývá aktualizací syntaktického analyzátoru pro projekt jStyleParser s využitím nástroje ANTLR 4. Projekt jStyleParser je analyzátor CSS napsaný v jazyce Java, slouží ke zpracování stylových předpisů zapsaných v jazyce CSS a převedení těchto předpisů do odpovídajích datových typů pro možnost další práce s těmito daty. Projekt také umožňuje přiřadit styly odpovídajícím elementům DOM v HTML dokumentu. V práci je nejprve popsána problematika syntaktických analyzátorů a jejich generování na základě daného předpisu (gramatiky), dále jsou popsány existující generátory na platformě Java se zaměřením na generátor ANTLR. Zbývající část se zabývá samostatným projektem jStyleParser – aktuálním stavem projektu a návrhem nového syntaktického analyzátoru. Po návrhu úprav je popsána implementace a testování správnosti implementace. V závěru jsou zhodnoceny dosažené výsledky a navrhnut další vývoj aplikace ve směru k CSS3.

Abstract

This thesis deals with upgrading of the parser for the jStyleParser project using ANTLR 4 tool. The jStyleParser project is a CSS parser and analyzer written in Java. It is used for processing cascading style sheets and their transformation to appropriate data types in order to allow further processing of this data. The project also allows to assign styles to corresponding elements of HTML document's DOM. The thesis first describes the topic of parsers and their generation based on the given rules (grammar). Further, there are described the existing generators on the Java platform with focus on the ANTLR generator. The remaining part is dedicated to the jStyleParser project — the current state of the project and the proposal of the the generator upgrade from ANTLR version 3 to 4. After the proposal, its implementation and testing is described. In the conclusion, the results are evaluated and further development of the application is discussed in the direction towards CSS3.

Klíčová slova

jStyleParesr, ANTLR, Java, syntaktický analyzátor CSS, CSSBox, CSS do Java struktury

Keywords

jStyleParesr, ANTLR, Java, CSS parser, CSSBox, CSS to Java structure

Citace

SEDLÁK, Radek. *Syntaktický analyzátor stylových předpisů CSS*. Brno, 2016. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Burget Radek.

Syntaktický analyzátor stylových předpisů CSS

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Radka Burgeta, Ph.D.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Radek Sedlák
19. května 2016

Poděkování

Tímto bych chtěl poděkovat svému vedoucímu Ing. Radkovi Burgetovi, Ph.D. za odborné vedení, rady a ochotu při tvorbě této práce. Dále bych chtěl poděkovat své rodině a přátelům za podporu během celého studia.

Obsah

Kapitola 1

Úvod

Tato diplomová práce se zabývá projektem jStyleParser. Tento projekt vznikl jako součást projektu CSSBox. CSSBox je (X)HTML/CSS zobrazovací engine napsaný v jazyce Java, jeho hlavním účelem je poskytnou všchny zpracovatelné informace o zobrazovaném dokumentu. Již zmiňovaný podprojekt jStyleParser slouží ke zpracování stylových předpisů zobrazovaného dokumentu zapsaných v jazyce CSS. Jelikož je celý projekt vyvíjen již od roku 2007, tak jStyleParser využívá starou verzi generátoru lexikálního a syntaktického analyzátoru a to ANTLR verze 3. Tato verze již není podporovaná a proto je vzhledem k dalšímu vývoji aplikace nezbytný přechod na novou verzi generátoru ANTLR. V době psaní této práce je nejnovější verze 4.5.3, kterou také nová verze aplikace bude využívat.

Cílem této práce je prostudovat a popsat problematiku synktaktických analyzátorů a jejich generování na základě dodaných předpisů (gramatik). Dále se zaměřit na existující generátory syntaktických analyzátorů na platformě Java, zejména prostudovat ANTLR ve verzi 3 a 4. Hlavní náplní této práce je prostudovat aktuální verzi projektu jStyleParser, po seznámení se s projektem navrhnout úpravy syntaktického analyzátoru tak, aby byl generovaný pomocí nástroje ANTLR verze 4.5.3. Tyto navržené změny do projektu implementovat a pomocí manuálních a automatických testů otestovat, zda jsou výsledky zpracování CSS nově vzniklé verze projektu stejné s výsledky stávajíc verze.

V kapitole 2 jsou popsány teoretické informace ohledně problematiky tvorby lexikálních a syntaktických analyzátorů, možnosti jejich automatického generování z předem dodaných gramatik a porovnání výhod a nevýhod analyzátorů generovaných oproti analyzátorům negenerovaným.

Popisem existujících generátorů na platformě Java, a to zejména generátorem ANTLR verze 3 a 4, se zabývá kapitola 3. V této kapitole je čtenář seznámen s tím, jaké generátory na platformě Java existují a jaké mají základní charakteristiky a vlastnosti, u generátorů je také uvedena poslední dostupná verze včetně data vydání. Popis generátoru ANTLR je rozdělen na části, pomocí kterých jsou podrobnější informace o generátoru rozděleny do logických bloků.

V kapitole 4 je čtenáři přiblížen základ jazyka CSS, který projekt jStyleParser zpracovává. Jsou zde popsány základní pojmy, které s jazykem CSS souvisí a pro další informace je čtenář odkázán na webové zdroje.

Kapitola 5 seznámí čtenáře s projektem jStyleParser. Přiblíží mu hlavní význam projektu jako dílčí část projektu CSSBox, dále popisuje strukturu projektu, metody a technologie, které byly v projektu využity. Jednou z důležitých podkapitol je bezesporu vstupní gramatika, ze které je vygenerován pomocí nástroje ANTLR verze 3 výsledný syntaktický analyzátor.

Kapitola 6 popisuje návrh nového generátoru s využitím ANTLR verze 4.5.3. V této kapitole je popsán způsob migrace z generátoru ANTLR verze 3 na verzi 4, rozdíly v gramatikách pro jednotlivé verze a metody, jakým jsou informace z průběhu parsování vstupních dat zpracovávány pro další práci. Dále jsou v kapitole také popsány změny tříd, které jsou ve výsledném parseru využívány. Tyto třídy slouží buď k rozšíření funkčnosti základních tříd poskytovaných knihovnou ANTLR, a nebo jsou to třídy, které zajišťují inicializaci, řízení a předání výsledků parseru dále.

Sedmá kapitola se zabývá samostatnou implementací navržených změn v aplikaci tak, aby aplikace používala pro zpracování CSS nástroj ANTLR verze 4.5.3. V kapitole jsou nejprve popsány modifikace, které byly prováděny v příslušných gramatikách a následně jsou popsány úpravy ve třídách, které souvisí se samostatným analyzátorem. Podkapitola 7.4 se zabývá implementací zpracování výsledků ze syntaktické analýzy. V této podkapitole jsou popsány dva způsoby, jakými bylo zpracování výsledků implementováno a detailně popsán způsob zpracování pomocí implementace návrhového vzoru `Visitor`, který se jevil jako lepší oproti vzoru `Listener`. V závěru kapitoly je popsán způsob zotavování z chyb vzniklých při analyzování vstupních CSS dat.

V kapitole 8 je popsán způsob, jakým bylo prováděno manuální a automatické testování vzniklé aplikace. Po popisu těchto dvou metod jsou shrnuty výsledky testování obsahující odkaz na službu Travis CI, kde je možné ověřit aktuální výsledky automatických testů aktuální verze programu dostupné v repozitáři ve službě GitHub.

V závěru této práce jsou zhodnoceny dosažené výsledky a nakonec je navržen další vývoj projektu směrem k CSS3 a sjednocení způsobu zpracování a zotavení chyb při analýze.

Kapitola 2

Problematika syntaktických analyzátorů

Tato kapitola se zabývá teoretickými informacemi o lexikální a syntaktické analýze strukturovaného textu na základě dodaného předpisu (gramatiky) a tvorbou automaticky generovaných analyzátorů. V první podkapitole jsou nejprve uvedeny základní pojmy a definice, které se v této problematice vyskytují a je nutné jim rozumět. Po základech je vysvětlena lexikální analýza, na kterou následně navazuje analýza syntaktická. V podkapitole 2.3 je popsán způsob generování syntaktických analyzátorů a jejich použití v aplikacích. Text kapitoly vychází z [14], [15] a [20].

2.1 Základní pojmy

V tomto teoretickém úvodu jsou stručně představeny základní pojmy z teorie formálních jazyků a překladačů, které je nutné znát pro pochopení problematiky syntaktických analyzátorů. Tento úvod si neklade za cíl poskytnout přesné a formální definicie, ale spíše čtenáři objasnit pojmy, které jsou dále v textu používány, z praktického hlediska. Pro hlubší pochopení problematiky, formální definice a pochopení všech souvislostí doporučuji prostudovat odbornou literaturu z oblasti *teoretické informatiky* a *formálních jazyků a překladačů*.

Abeceda je libovolná neprázdná konečná množina. Prvky abecedy nazýváme **symboly** [14]

Slovo nad abecedou V je konečná posloupnost symbolů z V. Prázdná posloupnost se nazývá prázdné slovo a značí se ϵ. [14]

Gramatika je čtveřice G=(N,T,P,S), kde

- N je abeceda neterminálů
- T je abeceda terminálů
- P je konečná množina pravidel ve tvaru `x` $\rightarrow$ `y`
- S je počáteční neterminál

Gramatiky jsou vhodným prostředkem pro definici syntaxe programovacích či jiných strukturovaných jazyků.[15]

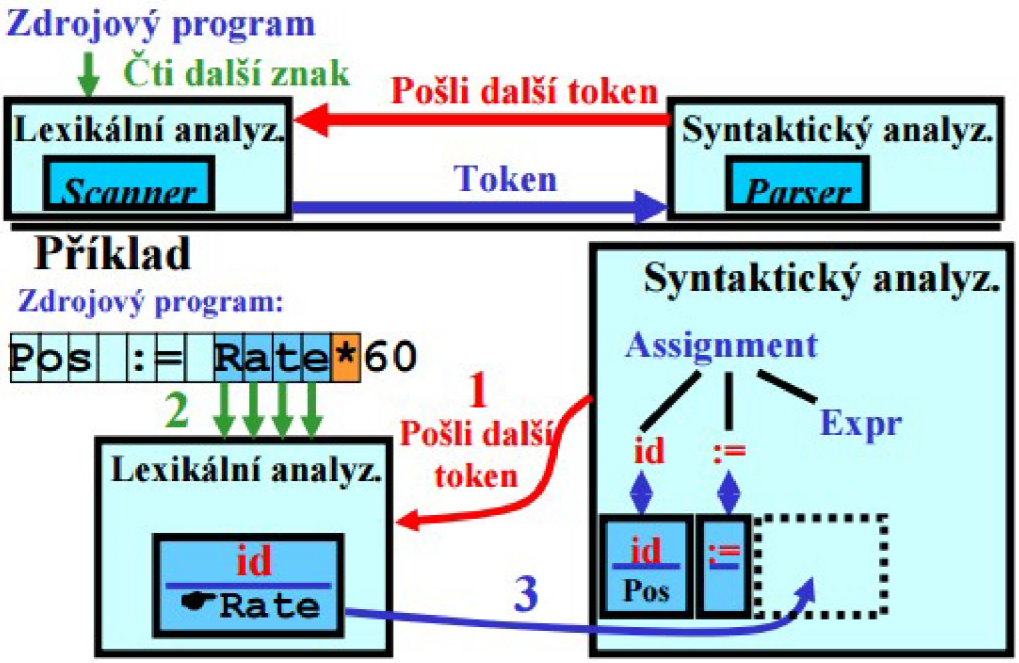
Lexém je logicky související posloupnost znaků jako je identifikátor, konstanta, klíčové slovo apod. [15]

Token je dvojice ve tvaru `< druh lexému, atribut >` . Slouží k uchování atributů jednotlivých lexémů – např. token pro identifikátor, který má jako atribut jeho název by vypadal takto: `< id, "identifikator1" >`. [15]

2.2 Lexikální a syntaktické analyzátory

Lexikální a syntaktické analyzátory jsou prvními dvěmi jednotkami tvořícími překladače jazyků. Při analýze a zpracování vstupních dat jsou tato data předána lexikálnímu analyzátoru, který tato data zpracuje a svůj výstup pošle na vstup syntaktického analyzátoru, jehož výstupem je derivační strom, který hierarchicky popisuje strukturu vstupních dat.

Na následujícím obrázku je schéma spolupráce lexikálního a syntaktického analyzátoru:

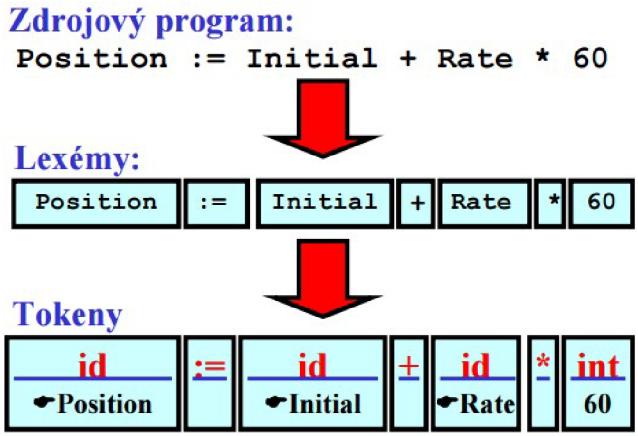


Obrázek 2.1: Spolupráce lexikálního a syntaktického analyzátoru. Převzato z [15]

2.2.1 Lexikální analýza

Lexikální analyzátor (lexer) slouží ke čtení zdrojového programu, který následně transformuje na řetězec lexikálních symbolů – lexémů, jsou to např. identifikátor, konstanta, číslo apod. U jednotlivých lexémů je ale třeba uchovávat i další informace – jako je např. název identifikátoru, hodnota konstanty atd. Z tohoto důvodu jsou lexémy reprezentovány v podobě tokenů (viz 2.1). Tyto zpracované tokeny jsou jednotlivé elementy zdrojového jazyka nesoucí informaci o typu lexému a jeho hodnotě (může být i prázdná).

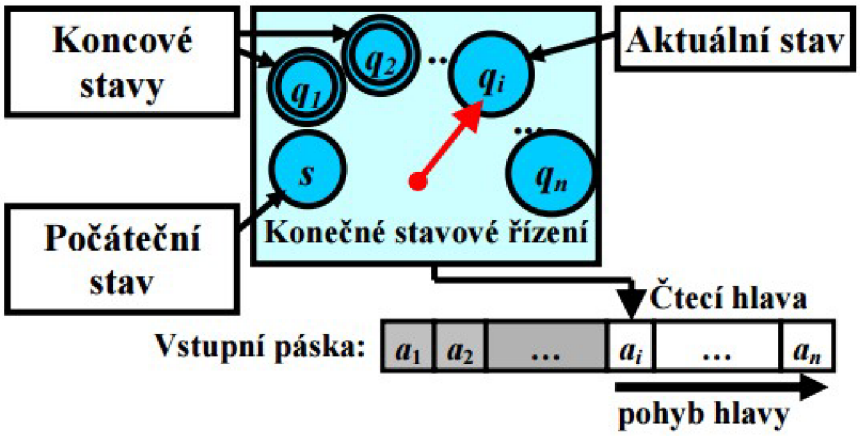
Na následujícím obrázku je vyobrazeno rozdělení zdrojového programu na lexémy a potom na tokeny:



Obrázek 2.2: Vztah mezi zdrojovým programem, lexémy a tokeny. Převzato z [15]

Při tvorbě lexikálního analyzátoru je třeba definovat pravidla, podle kterých budou jednotlivé lexémy zpracovávány – např. celé číslo může začínat znaménkem mínus a následně musí být tvořeno pouze z číslic. Jednoduchým nástrojem, kterým lze tato pravidla pro tvorbu lexémů definovat, jsou regulární výrazy (definice viz [15] kapitola 3.1). Jedním z prostředků, kterými se dá regulární výraz (RV) specifikovat je konečný automat (KA) (viz [15] kapitola 3.2). Jelikož je KA snadné implementovat, tak většinou tvoří základ pro realizaci lexikálního analýzátoru.

Na následujícím obrázku je ilustrace konečného automatu:



Obrázek 2.3: Ilustrace konečného automatu. Převzato z [15]

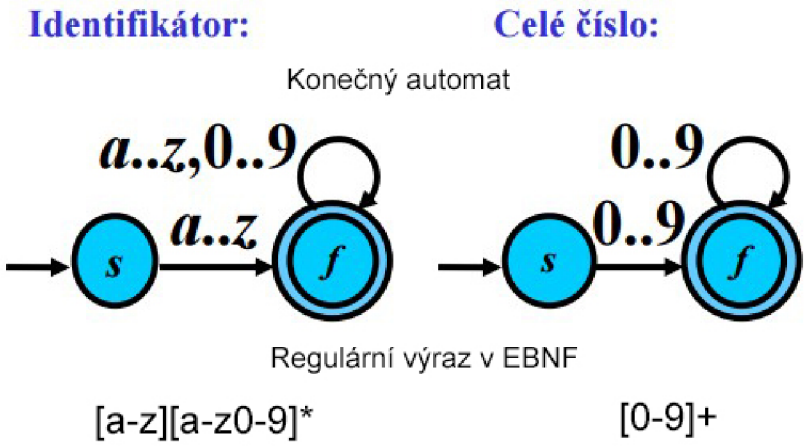
Základními modely pro lexikální analýzu jsou tedy regulární výrazy a konečné automaty.

Návrh lexikálního analyzátoru

Při návrhu lexikálního analyzátoru je nutné se rozhodnout, jakým způsobem bude scanner implementován. Buď je možné jej implementovat přímo ve zvoleném programovacím jazyce s využitím všech jeho prostředků (např. využít již vestavěných regulárních výrazů),

nebo ve zvoleném jazyce implementovat KA přijímající všechny lexémy přijímaného jazyka.Další možností je využít již existujícího automatizovaného nástroje pro vygenerování příslušného kódu. Po výběru metody implementace je nutné specifikovat pravidla pro vytvoření jednotlivých lexémů. Způsob popisu záleží na výběru implementace. Pro přímou implementaci záleží na výběru programovacího jazyka a jeho vlastností. Pro implementaci KA bude vhodné pro jednotlivé lexémy vytvořit odpovídající konečné automaty a tyto jednotlivé KA pak spojit do jednoho velkého. Pro automatické generování scanneru pomocí nějakého nástroje záleží především na specifikaci nástroje – ve většině případů takové nástroje pracují s regulárními výrazy, které mohou rozšiřovat o svoje specifické zápisy. Velmi často se reguálrní výrazy zapisují v Extended Backus-Naureově Formě (EBNF).

Příklad vytvoření KA a RV pro Identifikátor a a celé kladné číslo je znázorněn na následujícím obrázku:



Obrázek 2.4: KA a RV přijímající lexémy identifikátor a celé kladné číslo. Převzato z [15]

Specifikaci definujeme i pro lexémy, u kterých chceme, aby je lexikální analyzátor zpracoval, ale neposílal syntaktickému analyzátoru (např. mezery, komentáře a další).

Pokud používáme generátor, tak stačí specifikaci lexémů zapsat do strukutry, ze které generátor výsledný scanner vygeneruje. Jestliže implementujeme vlastní analyzátor, tak je třeba implementovat zpracování lexémů a převod na tokeny ručně. Lexikální analyzátor je většinou implementovaný jako komponenta, kterou používá syntaktický analyzátor. Jakmile syntaktický analyzátor potřebuje nový token, tak zavolá právě lexikální analyzátor a ten mu jej poskytne.

Při lexikální analýze také mohou vzniknout chyby, kdy posloupnost znaků ve vstupním souboru neodpovídá žádné specifikaci lexému. Takovéto chyby se nazývají chybami lexikálními. Lexikální analyzátor může u takovýchto chyb buď skončit s hláškou, že při lexikální analýze nastala chyba, nebo se může pokusit z této chyby zotavit. Možností zotavení chyb je několik:

- vypouštění znaků tak dlouho, dokud se analyzátoru nepodaří rozpoznat správný lexém
- vrácení speciálního terminálního symbolu a přenechání chyby na syntaktickém analyzátoru

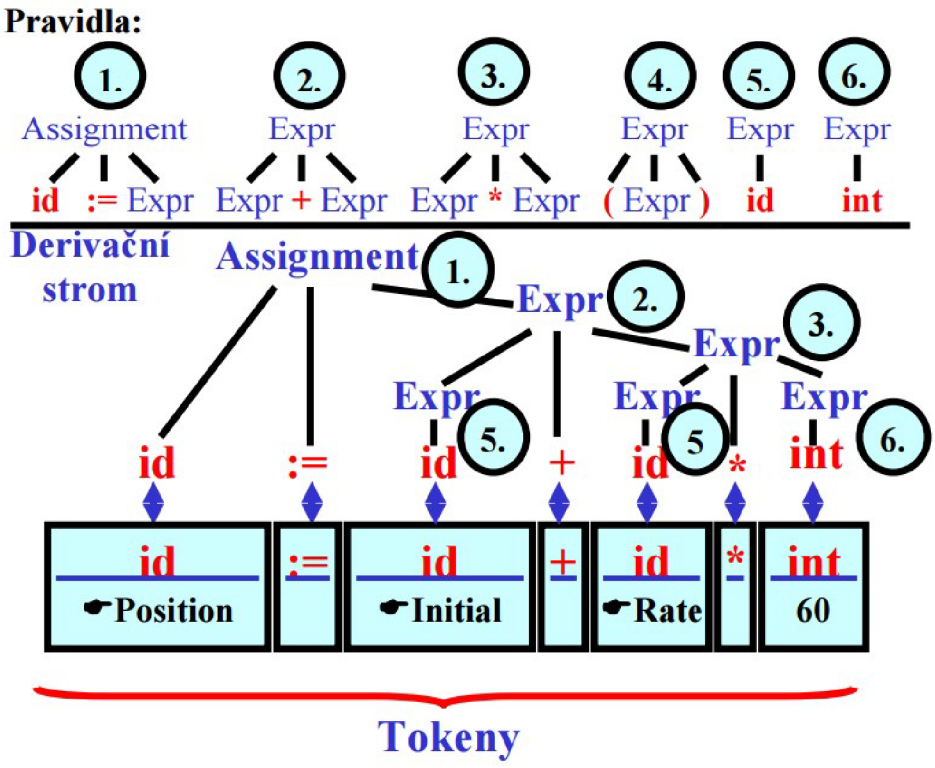
- náhrada nesprávného znaku správným
- vzájemná výměna dvou sousedních znaků

Většina pokročilých překladačů samozřejmě neskončí pouze s ohlášením lexikální chyby, ale využívá právě některého ze způsobů zotavení z chyby. Jestliže se zotavení z chyby povede, je možné zpracovat zbytek vstupních dat a případně tak odhalit další chyby.

2.2.2 Syntaktická analýza

Syntaktický analyzátor (parser) na základě vstupního řetězce tokenů zjišťuje, zda tento řetězec patří do zdrojového jazyka, tj. zdrojový program je syntakticky správný. Jestliže zdrojový program je syntakticky správný, pak je na základě pravidel sestaven derivační strom, který reprezentuje syntaktickou strukturu zdrojového programu.

Následující obrázek ilustruje tvorbu derivačního stromu z tokenů podle pravidel gramatiky:

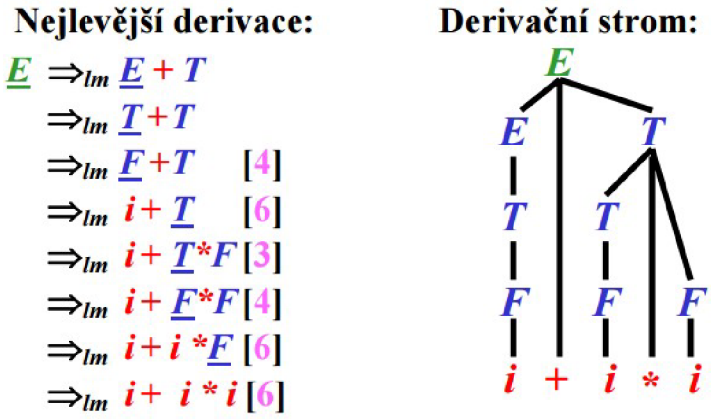


Obrázek 2.5: Konstrukce derivačního stromu z tokenů. Převzato z [15]

Pro popis syntaxe programovacích jazyků se nejčastěji používají bezkontextové gramatiky, které úzce souvisí s tzv. zásobníkovými automaty (ZA) a rozšířenými zásobníkovými automaty (RZA). Bezkontextové gramatiky a ZA (RZA) tvoří základní model pro syntaktickou analýzu. Platí, že jazyk je bezkontextový právě tehdy, když jej lze akceptovat zásobníkovým automatem.

Syntaktická analýza metodou shora dolů pro větu `w` patřící do analyzovaného jazyka `L` vede k nalezení posloupnosti pravidel z gramatiky použitých při levé derivaci věty `w`.

Obdobně funguje analýza metodou zdola nahoru, kdy je využito pravé derivace věty `w`. Uvažujme bezkontextovou gramatiku `G=(N,T,P,E)`, kde `N = E,F,T`, `T = i,+,*,(,)`, `P = 1: E → E+T, 2: E → T, 3: T → T*F, 4: T → F, 5: F → (E), 6: F → i` Následující obrázek demonstruje nejlevější derivaci a odpovídající derivační strom pro vstupní řetězec `i + i * i`.

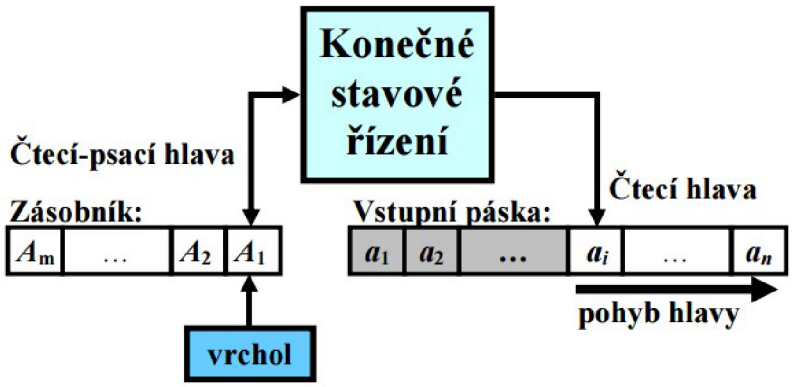


Obrázek 2.6: Nejlevější derivace a odpovídající derivační strom. Převzato z [15]

Jestliže lze pro řetězec více jak jeden derivační strom, pak hovoříme o tom, že gramatika jazyka je nejednoznačná. Takováto gramatika způsobuje potíže při tvorbě syntaktického analyzátoru. Syntaktický analyzátor musí umět bezpečně poznat, který derivační strom je pro řetězec správný. Automatické nástroje s tímto počítají a při zápisu gramatiky pro syntaktickou analýzu je možné u jednotlivých pravidel s výskytem nejednoznačnosti definovat prioritu zpracování těchto pravidel.

Zásobníkové automaty

Jak již bylo výše zmíněno, tak teoretickým modelem syntaktických analyzátorů jsou zásobníkové automaty. Zásobníkový automat je obdoba konečného automatu rozšířena o zásobníkovou paměť, jejíž vrchol ovlivňuje každý přechod mezi stavy automatu. Při každém přechodu v ZA je nahrazen jeden symbol (vrchol) zásobníku. V RZA lze nahradit celý řetězec symbolů na vrcholu. Ilustrace zásobníkového automatu:



Obrázek 2.7: Zásobníkový automat. Převzato z [15]

Pomocí zásobníkového automatu lze simulovat analýza shora dolů a pomocí rozšířeného ZA můžeme simulovat analýzu zdola nahoru.

Návrh syntaktického analyzátoru

Syntaktický analyzátor se snaží zjistit, zda zdrojový text tvoří větu odpovídající gramatice analyzovaného jazyka. Posloupnost lexikálních symbolů, které analyzátor zpracovává, získává z výsledků lexikálního analyzátoru. Pokud se při syntaktické analýze nepodaří vytvořit derivační strom, tak vzniká tzv. syntaktická chyba a většinou se analyzátor z této chyby snaží nějakým způsobem zotavit. Postup vytváření derivačního stromu závisí právě na použité metodě `shora dolů`, nebo `zdola nahoru`, čemuž odpovídají i gramatiky `LL` a `LR`. Ručně implementované analyzátory často používají LL gramatiky. LR gramatiky, popisující větší třídu jazyků, obvykle používají analyzátory, které jsou vytvářeny automatizovaně.

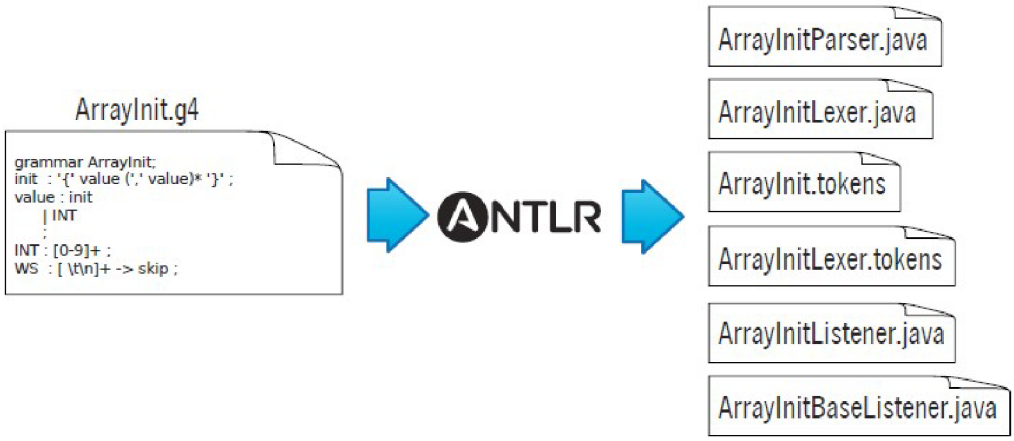
V praxi je obvykle od syntaktického analyzátoru požadováno více, než pouze informace o tom, zda je zdrojový program syntakticky správně. Proto je výstupem analyzátoru reprezentace, která nese další užitečné informace. Většinou se jedná o derivační strom, nebo posloupnost akcí, které vytváří vnitřní reprezentaci struktury zdrojových dat. Výsledky syntaktické analýzy jsou poté zpracovány, nebo předány sémantickému analyzátoru, který vyhodnocuje závislosti, které nelze popsat pomocí bezkontextových gramatik.

Stejně jako lexikální analyzátor, tak i syntaktickém analyzátor rozpoznává chyby specifické pro jeho fázi analýzy zdrojového programu. Způsob zotavení z těchto chyb závisí na metodě a technikách použitých při implementaci analyzátoru. Pokud analyzátor při chybě dokáže odhadnout, jak by měla pro vstupní data derivace správně vypadat, tak může zjistit, zda chybí nějaký prvek jazyka, nebo zda nějaký prvek přebývá. Z těchto informací poté může syntaktický analyzátor zvolit strategii zotavení.

2.3 Generování lexikálních a syntaktických analyzátorů

Generátory syntaktických analyzátorů jsou vlastně samy syntaktickými analyzátory. Jejich vstupními daty je ve většině případů gramatika zapsaná podle definovaných pravidel. Tato gramatika je předložena generátoru a ten z ní vygeneruje zdrojové kódy analyzátoru, který dokáže zadanou gramatiku zpracovat. Vygenerované zdrojové kódy následně programátor

použije ve svém projektu ke zpracování vstupního textu a provedení jeho lexikální a syntaktické analýzy. Na následujícím obrázku je blokové schéma toho, jak pracuje generátor syntaktických analyzátorů ANTLR. Na začátku je gramatika, která je pomocí nástroje ANTLR zpracována a ten na základě této gramatiky vygeneruje syntaktický analyzátor, který přijímá jazyk definovaný vstupní gramatikou:



Obrázek 2.8: Ilustrace chování nástroje ANTLR v.4. Převzato z [20]

2.3.1 Gramatiky pro generování analyzátorů

Většina existujících generátorů pro vygenerování ať už lexikálního či syntaktického analyzátoru používá gramatiku, jejíž podoba je vždy generátorem specifikovaná. Obecně se dá říct, že gramatika, ze které se analyzátor generuje je většinou definovaná pomocí regulárních výrazů s využitím rozšiřujících notací, pomocí kterých může uživatel definovat další různé akce, které se v průběhu analýzy následně vykonávají.

Gramatika pro scanner

Pro generování lexikálního analyzátoru je třeba popsat jednotlivé lexémy spadající do přijímaného jazyka a převést je na tokeny, které budou následně posílány syntaktickému analyzátoru. Následující příklad definuje gramatiku pro lexikální analyzátor generátoru ANTLR verze 4:

```
lexer grammar ExampleLexer;

ID : [a-zA-Z]+;
INT : '-'?[0-9]+ ;
EQ : '=';
WHITESPACE : [ \t\n]+ -> skip;
```

Tato gramatika definuje zpracování tokenů identifikátor (ID), celé číslo (INT) a bílé znaky (WHITESPACE). V gramatice je definováno pomocí `-> skip` to, že scanner nebude bílé znaky parseru vůbec posílat.

Gramatika pro parser

Gramatiky pro syntaktický analyzátor jsou oproti lexikální složitější. V gramatice je zapsána syntaxe zpracovávaného jazyka. Gramatika dále může obsahovat části kódu, které se vykonávají při zpracovávání jednotlivých pravidel gramatiky – např. počítání zpracovaných tokenů, vytváření pomocných proměnných apod. Některé gramatiky mohou obsahovat příkazy pro definování priority jednotlivých alternativ v pravidlech – může být vhodné u gramatik popisujících matematické výrazy apod. Příklad gramatiky pro syntaktický analyzátor generátoru ANTLR verze 4:

```
parser grammar ExampleParser;

options { tokenVocab=ExampleLexer;}

assign: ID EQ INT;
```

Výše uvedená gramatika demonstruje zápis pravidla `assign`, které slouží k přiřazení celého čísla do proměnné s identifikátorem `ID`.

2.3.2 Generování a použití analyzátoru

Jakmile jsou správně vytvořeny jednotlivé předpisy pro tvorbu analyzátoru, tak je nutné z těchto souborů vygenerovat zdrojové kódy analyzátoru. Pro vygenerování těchto zdrojových kódů slouží právě generátory, které stačí správným příkazem zavolat a předat jim zdrojové soubory s definicí analyzátorů. Takovýto nástroj zpracuje předložený vstupní soubor, zkontroluje jeho správnost a na základě uvedených pravidel vygeneruje zdrojové kódy analyzátoru.

Jakmile jsou kódy analyzátoru vygenerovány, tak již stačí tyto kódy použít v aplikaci a podle manuálu takto vygenerovaný analyzátor vstupních dat správně použít.

2.3.3 Výhody a nevýhody generovaných analyzátorů

Používání generovaných analyzátorů má velkou řadu výhod. Především je to rychlost, jakou lze parser vstupních dat vytvořit. Bez generátoru by programátor musel pro každý nový projekt vytvořit lexikální analyzátor pro vstupní gramatiku, poté na základě lexikálního analyzátoru vytvořit syntaktický analyzátor, následně vytvořit abstraktní syntaktický strom a poté teprve provádět další akce, jako je sémantická analýza, generování kódu, optimalizace apod.

S využitím generátoru se programátor musí naučit jakým způsobem se generátor používá a poté napsat gramatiku vstupního souboru, použít generátor, který vygeneruje lexikální, syntaktický, nebo oba analyzátory a poté se tvůrce aplikace může věnovat zpracování výsledků analýzy.

Další výhodou je bezesporu znovupoužitelnost již definovaných gramatik v dalších projektech.

Používání již hotového nástroje má také výhodu v tom, když na vývoji projektu pokračuje někdo jiný. Takový člověk se pouze naučí používat nástroj, který vě většině případů disponuje dokumentací, manuálem, či alespoň sadou příkladů s použitím nástroje. Nejpoužívanější nástroje rovněž mají vytvořenu programátorskou komunitu, ve které je možné diskutovat o případných problémech, nebo hledat u této komunity podporu.

Nevýhodou generovaných analyzátorů může být nutnost naučit se tyto analyzátory používat. Dále také vygenerovaný analyzátor může být méně efektivní, protože kód je generován strojově – manuálně naprogramovaný překladač může být efektivnější a rychlejší. Další nevýhodou u některých generátorů může být to, že části programového kódu je nutné definovat již v gramatice – gramatika i generovaný kód se poté stávají nepřehlednými, což výrazně zhoršuje možnost ladění a odhalování případných chyb ve fázi zpracování výsledků.

Kapitola 3

Existující generátory na platformě Java

Následující podkapitoly popisují základní charakteristiku existujících generátorů dostupných pro platformu Java. Generátor ANTLR je popsán ve verzích 3 (využíváná v projektu jStyleParser) a 4 (aktuální při tvorbě této práce). Jelikož je nástroj ANTLR součástí jádra projektu, tak jsou tyto dvě verze popsány podrobně. Text této kapitoly vychází z [19] a [20].

3.1 ANTLR

Tato podkapitola se zabývá podrobněji generátorem ANTLR, který je v projektu využíván. Jsou zde popisovány majoritní verze 3 a 4. Tyto dvě verze jsou si v základech velice podobné, ale verze 4 má také různá vylepšení a úpravy. Z tohoto důvodu je nejdříve popsán generátor ANTLR obecně a poté jsou uvedeny rozdílnosti jednotlivých verzí.

ANTLR (ANother Tool for Language Recognition), čili "Další nástroj pro rozpoznávání jazyka" je výkonný generátor syntaktických analyzátorů (dále parserů) pro čtení, zpracování, vykonávání, nebo překládání strukturovaného textu či binárních souborů. Je široce využíván k budování jazyků, nástrojů a frameworků. ANTLR ze zadané gramatiky generuje parser, který umí vytvářet a procházet parsovací stromy. [2]

Základní konstrukcí pro ANTLR je gramatika, která je v podstatě seznamem pravidel, které popisují strukturu konkrétního jazyka. Z těchto pravidel ANTLR vygeneruje syntaktický analyzátor s rekurzivním sestupem, který zpracovává věty jazyka. Jazyk může být buď programovací jazyk a nebo jednoduchý strukturovaný formát dat. Co je to za jazyk pro ANTLR není podstatné. Tento nástroj vytvoří analyzátor, který zjistí, zda vstupní řetězec spadá do definovaného jazyka (existuje pravidlo, které dokáže vstupní data popsat ?).

ANTLR 3 umožňuje generování analyzátorů pro lexikální, syntaktickou analýzu a analyzátor, který zpracovává abstraktní syntaktické stromy. Verze 4 je odlehčena od generování AST a namísto tohoto mechanismu je v projektu implementován návrhový vozr `Listener` a `Visitor`, pomocí kterých lze procházet výsledný parsovací strom. Gramatika pro ANTLR je zapsaná pomocí EBNF.

ANTLR verze 3 definuje 4 druhy gramatik: lexer, parser, tree a kombinovanou gramatiku pro lexer a parser dohromady. Všechny gramatiky mají stejnou základní strukturu, která je ilustrována na následující části kódu:

```
/** komentář */
typGramatiky grammar nazevGramatiky;
<< specifikace možností >>
<< specifikace tokenů >>
<< atributy >>
<< akce >>

pravidlo1 : ...  | ... | ... ;
pravidlo2 : ...  | ... | ... ;
```

Pořadí jednotlivých sekcí musí být v gramatice zachováno. Z definovaných gramatik ANTLR vygeneruje zdrojové kódy, jejichž názvy odpovídají jejich rolím. Z gramatik s názvem `A*.g` vygeneruje třídy `ALexer.java`, `AParser.java` a `ATreeParser.java`, které implementují jednotlivé fáze analyzátoru. Pro zpracování vstupu je v knihovně s nástrojem ANTLR definována třída `ANTLRInputStream` a `CommonTokenStream`. Tyto třídy slouží jako vstupy lexikální a syntaktické analýzy.

Ve výsledném programu je nejprve pomocí třídy `ANTLRInputStream` vytvořen vstupní zdroj dat. Následně je tento vstupní soubor předán lexikálnímu analyzátoru, který se inicializuje. Poté je lexikální analyzátor předán jako vstupní parametr syntaktickému analyzátoru. Syntaktický analyzátor postupně volá lexer, který mu postupně poskytuje zpracované tokeny. Vygenerovaný parser se následně snaží pomocí definovaných pravidel vytvořit strukturu derivačního stromu – `CommonTree`.

Ve verzi ANTLR 3 je fáze syntaktické analýzy a generování AST rozděleno do dvou částí. Nejprve se parser snaží zdrojový soubor zpracovat podle pravidel definovaných v gramatice pro parser a následně tato data převede podle definic přepisovacích pravidel v gramatice. Tato přepisovací pravidla slouží ke zjednodušení syntaxe jazyka, umožňují vypuštění některých tokenů, či zjednodušení struktury, která je následně pomocí `TreeParseru` zpracována. Jakmile je dokončena syntaktická analýza, tak její výsledek, který je zpracován jako `CommonTree`, je předložen poslednímu analyzátoru pro zpracování AST. Tento analyzátor vychází opět z gramatiky. Gramatika pro zpracování AST již definuje akce, které se starají o zpracování a manipulaci hodnot jednotlivých prvků vstupních dat. Zpracování stromu probíhá hned při procházení stromu. Funkce, které přísluší jednotlivým pravidlům obsahují uživatelem definovanou obsluhu uživatelských akcí. Tyto akce umožňují uživateli zpracovávat výsledky analýzy.

Nevýhodou zpracovávání výsledků analýzy, která je definovaná pomocí uživatelských akcí přímo v gramatice analyzátoru pro zpracování derivačních, nebo abstraktních syntaktických stromů je bezesporu fakt, že vygenerované zdrojové kódy obsahují části kódu, které slouží pro zpracování jazyka a části kódu, které jsou definované uživatelem. Pochopení uživatelských akcí, které vedou ke zpracování výsledků analýzy vstupních dat, může být proto velice obtížné, neboť je nutné procházet zdrojové kódy, které míchají rozpoznání struktury kódu a zároveň zpracování výsledků analýzy.

Verze 3 využívá pro parsování backtrackingu, který je náročný na ladění, protože se parser může rekurzivně zanořovat.

3.1.1 verze 4

Majoritní verze 4 generátoru ANTLR vyšla 21. ledna 2013 [3]. Při vzniku této práce je k dispozici verze 4.5.3.

ANTLR v.4 je o hodně jednodušší k naučení oproti předchozí verzi. Největší změna nastala v tom, že tato verze snižuje důraz na vkládání uživatelského kódu do gramatik a místo

toho upřednostňuje používání návrhových vzorů návštěvník a posluchač. Tyto nové mechanismy osvobozují gramatiky od aplikačního kódu, což vede ke zlepšení čitelnosti a menší frakturovanosti kódu napříč aplikací. Bez používání vnořených aplikačních kódu je mnohem snadnější používat gramatiky v jiných aplikacích. Vnořené akce jsou pořád podporovány, ale slouží spíše k definování obtížnějších konstrukcí.

Při používání této verze generátoru není nutné definovat gramatiku pro generování AST, jelikož ANTLR verze 4 již generuje derivační stromy a nástroje na jejich zpracování automaticky. Narozdíl od generování těchto nástrojů stačí definovat třídu, která reaguje na jednotlivé akce při syntaktické analýze. Realizace této třídy spočívá buď v implementaci návrhového vzoru `Visitor` nebo `Listener`. Tato rozhraní jsou pomocí nástroje vygenerována spolu s analyzátorem jazyka.

Narozdíl od strategie `LL(*)` ve verzi 3, verze 4 používá strategii `ALL(*)`, která je efektivnější.

Instalace nástroje ANTLR v. 4.5.3.

Instalace nástroje na platformách Windows, Linux a OS X probíhá vesměs podobně a to dle následujících kroků:

1. Pro instalaci nástroje ANTLR je nutné mít nainstalovanou JAVU ve verzi 1.6 a vyšší
2. Stáhnout si JAR balík (verze complete) aktuální verze ANTLRu
3. Cestu k balíku přidat do CLASSPATH proměnné systémového prostředí
4. Vytvořit spouštěcí skripty pro antlr4 a grun

Následuje příklad instalace ANTLR4 v prostředí OS Linux:

```
$ cd /usr/local/lib
$ wget http://www.antlr.org/download/antlr-4.5.3-complete.jar
$ export CLASSPATH=".:/usr/local/lib/antlr-4.5.3-complete.jar:$CLASSPATH"
$ alias antlr4='java -jar /usr/local/lib/antlr-4.5.3-complete.jar'
$ alias grun='java org.antlr.v4.gui.TestRig'
```

Nástroj ANTLR je též dostupný jako Maven plugin, takže je možné jej jednoduše začlenit do projektů využívajících tento nástroj na správu sestavování projektů.

3.2 Ostatní generátory

Následující podkapitoly stručně popisují další dostupné generátory na platformě Java. Jsou zde popsány ty generátory jak lexikálních, tak syntaktických analyzátorů, které jsou používány nejvíce.

3.2.1 JFlex

JFlex [10] je generátor lexikálních analyzátorů (scannerů). Vygenerované scannery jsou založeny na deterministických konečných automatech. Jsou rychlé, nevyužívají nákladného zpětného navracení.

JFlex je navržen tak, aby spolupracoval s LALR parser generátorem CUP 3.2.2 od Scotta Hudsona a Java modifikací Berkeleyeho Yacc BYacc/J od Boba Jamisona. JFlex

může být také použit dohromady s ostatními generátory parserů jako je např. ANTLR 3.1, nebo jako samostatný nástroj.

Nástroj je volně dostupný pod open-source BSD licencí. K dispozici je i jako maven plugin.

Poslední dostupná verze 1.6.1 byly vydána 16.3.2015.

3.2.2 CUP

CUP [8] je zkratka pro Construction of Useful Parsers, je to LALR parser generátor pro jazyk Java. Byl vyvinut C. S. Ananianem, F. Flannerym, D. Wangem, A. W. Appelem a M. Petterem. CUP implementuje standardní LALR(1) generování parserů. Hlavní rysy:

- LALR(1) parsing engine s precedencí symbolů
- Zotavování z chyb
- Možnost definovat předpoklady pro správné dokončení syntaxe při chybě
- Volitelné generování derivačního stromu
- Volitelný výstup v XML
- Možnost vlastního kódu akcí
- Vývoj gramatiky za pomoci pluginu pro Eclipse
- Otevřená licence

Poslední verze 0.11b byla vydána 1.10.2015.

3.2.3 BYacc/J

BYACC/J [5] je rozšíření Berkeleyeho v 1.8 YACC-kompatibilního parser generátoru. Standardní YACC zpracovává YACC zdrojové soubory a generuje z nich jeden nebo více zdrojových souborů jazyka C, které při správné kompilaci vygenerují parser gramatiky typu LALR. Toto je užitečné pro parsování výrazů, interaktivních příkazů a čtení souborů. V projektu BYacc/J byl přidán parametr "-J", který způsobí to, že BYacc namísto zdrojových kódů v jazyce C/C++ vygenere zdrojové kódy v jazyce Java.

Poslední verze 1.15 byla vydána 27.11.2008.

3.2.4 Grammatica

Grammatica [9] je C# a Java generátor parserů. Nad podobnými nástroji (jako je yacc a ANTLR) je lepší v tom, že vytváří správně komentovaný a čitelný zdrojový kód, má automatické zotazování z chyb ,detailní chybové zprávy a také podporu pro testování a ladění gramatiky bez nutnosti generování zdrojových kódů.

Poslední dostupná verze 1.6 byla vydána 17.5.2015.

3.2.5 Beaver

Beaver [4] je LALR(1) generátor parserů. Generátor zpracovává bezkontextové gramatiky a konvertuje je na Java třídy, které implementují parser pro jazyk popsaný gramatikou. Beaver akceptuje gramatické výrazy zapsané v Extended-Bacus-Naureově formě (EBNF).

Beaver generuje pouze syntaktický analyzátor, proto je nutné do projektu integrovat scanner, který zajistí lexikální analýzu. API, které projekt poskytuje, usnadňuje zapojit populární scannery – jako je JFlex (viz 3.2.1) a JLex.

Poslední dostupná verze je 0.9.11 vydaná dne 18.12.2012

3.2.6 SableCC

SableCC [12] je generátor, který generuje zcela objektově orientované frameworky pro stavbu překladačů, interpretů a ostatních textových parserů. Generované frameworky hlavně zahrnují intuitivní striktně typované abstraktní syntaktické stromy s podporou jejich procházení pomocí tree walkeru. SableCC také zachovává čistotu mezi strojně generovaným a uživatelským kódem, což vede ke kratšímu vývojářskému cyklu.

Poslední dostupná verze je 4-beta.4 vydaná dne 8.8.2013.

Kapitola 4

Jazyk CSS

V této části práce je čtenáři stručně představen jazyk CSS (Cascading style sheet), právě k jehož zpracování slouží projekt jStyleParser. V kapitole jsou popsány základní prvky jazyka, jejich význam a syntax. Text kapitoly vychází z [6] a [7].

4.1 Definice

CSS je jazyk, který popisuje vzhled strukturovaných dokumentů jako je HTML nebo XML. Pomocí jazyka se popisuje, jak mají být jednotlivé elementy vykresleny na obrazovce, papíru, nebo na jiných médiích. [7]

Oficiální definice CSS je dostupná na webu World Wide Web Consorcia (W3C) dostupná na adrese `https://www.w3.org/TR/CSS/#css`. Vývoj jazyka CSS není jako ostatní jazyky verzovaný, ale rozlišují se u něho úrovně, pomocí kterých se odlišuje podpora jednotlivých vlastností. Aktuálně jsou definovány 3 úrovně jazyka – CSS 1, CSS 2 a CSS 3. Novější úroveň vždy obsahuje vše z předchozí úrovně a rozšiřuje ji o další, nové vlastnosti.

4.2 Základní prvky jazyka, syntaxe

CSS dokument je série **kvalifikovaných pravidel** a **"at-pravidel"**. Kvalifikovaná pravidla jsou obvykle stylovací pravidla, která aplikují CSS vlastnosti na elementy. At-pravidla definují zvláštní zpracování pravidel nebo hodnot v CSS dokumentu.

Většina kvalifikovaných pravidel jsou pravidla stylovací. Stylovací pravidla na začátku mají selektor a poté následuje blok, který je obalený do složených ({}) závorek. **Selektor** specifikuje, na které elementy se budou deklarace definované ve složených závorkách aplikovat. Každá **deklarace** má jméno, následované znakem dvojtečka a hodnotu. Deklarace mají volitelnou vlastnost `!important`, která zvyšuje prioritu uvedené deklarace, tato vlastnost je ve výchozí hodnotě považována za nedefinovanou. Jednotlivé deklarace jsou odděleny pomocí středníku. Typické stylovací pravidlo může vypadat nějak takto:

```
p > a {
        color : blue;
}
```

V tomto pravidle je `"p > a"` selektor, který ve zdrojovém HTML dokumentu vybírá všechny `<a>` elementy, které jsou potomky elementu `<p>`. Řádek obsahující `"color:blue;"` je deklarace, která specifikuje, že barva textu elementů, které jsou pomocí selektoru vybrány,

bude modrá.

Jak již bylo výše uvedeno, tak zvláštními pravidly v CSS jazyce jsou at-pravidla (angl. at-rules). Všechna tato at-pravidla jsou rozdílná, ale jejich zápis je velice podobný. Začínají znakem '@' následovaným jménem pravidla. Některá at-pravidla jsou jednoduché příkazy ukončené středníkem, jiná začínají jménem a následuje blok uvozený složenými závorkami ({}) a jsou podobná kvalifikovaným pravidlům. Následující příklad ukazuje zápis at-pravidla import:

```
@import "my-styles.css";
```

at-pravidlo "@import"je jednoduché pravidlo, které slouží k importování jiného CSS dokumentu. Za názvem pravidla následuje buď řetězec, nebo URL, které definují cestu importovaného souboru. Další at-pravidla jsou například `@charset`, `@media`, `@font-face`, `@page` a další.

4.3 Použití CSS v HTML dokumentech

V HTML dokumentech může být CSS definováno třemi způsuby:

1. Externí styl – tento typ použití je nejčastější, CSS se do HTML dokumentu vloží pomocí URL odkazu na tento dokument v tagu `link`, př:

   ```
   <link rel="stylesheet" type="text/css" href="mystyle.css">
   ```

   soubor mystyle.css pak může vypadat takto:

   ```
   body {
       background-color: lightblue;
   }
   h1 {
       color: navy;
       margin-left: 20px;
   }
   ```

2. Interní (vložený) styl – tento typ CSS se nachází v těle HTML dokumentu a je ohraničen tagem `style`, př:

   ```
   <style>
   body {
        background-color: red;
   }
   </style>
   ```

3. Inline styl – tento CSS zápis se používá ke stylování konkrétních elementů dokumentu, CSS kód se vkládá do atributu style elementu, který chceme nastylovat např.:

   ```
   <span style="color:red" >text</span>
   ```

   V těle inline stylu může být pouze seznam deklarací. Nelze zde zapisovat klasická pravidla, protože selektor je striktně definovaný tím, že je zápis stylu v atributu tohotot elementu.

Jelikož je možné do dokumentů vkládat styly více způsoby, tak se určování toho, který styl bude mít nejvyšší prioritu a použije se pro zobrazení elementu, řídí následujícími pravidly (1 je s nejvyšší prioritou, 3 s nejmenší):

1. Inline styly (uvnitř HTML prvku)
2. Externí a interní styly
3. Styly definované v prohlížeči jako základní

Pro podrobnější informace ohledně jazyka CSS doporučuji prostudovat http://www.w3schools.com/css/ a http://www.w3.org/Style/CSS/

Kapitola 5

Projekt jStyleParser

Tato kapitola popisuje, k čemu projekt jStyleParser slouží a jaké jsou v aktuální verzi použité technologie, nástroje a knihovny. Dále je čtenáři přiblížena struktura celého projektu a způsob využití projektu jako nástroje pro zpracování CSS dat s výsledkem získání odpovídající reprezentace stylů v datových typech jazyka Java. V popisu projektu je kladen důraz na používání nástroje ANTLR, jímž se celá tato práce zabývá. Celá kapitola vychází z informací dostupných na webu projektu (viz [17]) a dále ze zdrojových kódů aplikace.

5.1 O projektu

jStyleParser je parser s vlastním aplikačním rozhraním napsaný v jazyce Java. Toto aplikační rozhraní umožňuje efektivní zpracování CSS v Javě a mapování hodnot na datové typy jazyka Java. Aplikace zpracovává předpisy podle specifikace W3C CSS 2.1 a část CSS 3. Zpracování chyb v CSS probíhá rovněž podle jeho specifikace [18].

Knihovna jStyleParser je součástí projektu CSSBox. Projekt CSSBox je (X)HTML/CSS zobrazovací engine napsaný v jazyce Java, jeho hlavním účelem je poskytnou všechny zpracovatelné informace o zobrazenovaném dokumentu [16]. CSSBox využívá jStyleParser pro získání objektové reprezentace všech CSS dat, které aktuálně zobrazovaný dokument obsahuje a jejich přiřazení elementům DOM stromu. jStyleParser je tedy nástroj, který analyzuje vstupní CSS data pomocí syntaktického analyzátoru vygenerovaného nástrojem ANTLR a tato data transformuje do reprezentace pomocí datových struktur jazyka JAVA a dále je dokáže zpracovanému DOM stromu efektivně přiřadit.

5.2 Použité technologie

Celý projekt je napsaný v jazyce Java jako knihovna, kterou je možné použít v dalších projektech a je postaven na správci projektů Maven. Pro zpracování výpisů aplikace je využita knihovna logback, testování aplikace je vyřešon za pomoci knihovny jUnit a hamcrest. Jako generátor lexikálního a syntaktického analyzátoru z gramatik pro jazyk CSS je v aktuální verzi využito nástroje ANTLR verze 3.5.2, jehož aktualizací v projektu se zabývá tato práce. Dále jsou v projektu použité pomocné knihovny unbescape, xerces, xml-apis a nekohtml.

5.3 Zdrojové kódy a struktura projektu

Celý projekt je vyvíjen pomocí verzovacího nástorje Git a je dostupný v repozitáři ve službě GitHub na webové adrese https://github.com/radkovo/jStyleParser. Zdrojové kódy projektu se nachází v adresáři src, který obsahuje dva adresáře a to main a test. Jak je již z názvů adresářů patrné, tak adresář main obsahuje zdrojové kódy aplikace a adresář test obsahuje zdrojové kódy a data jednotlivých jUnit testů sloužících ke kontrole správnosti aplikace. Adresář main dále osahuje ve složce antlr3 zdrojové kódy gramatik pro jednotlivé fáze analyzátoru. Zdrojové kódy programu jsou rozděleny do následujících balíků:

- cz.vutbr.web.css
- cz.vutbr.web.csskit
- cz.vutbr.web.domasign
- org.fit.net

Balík cz.vutbr.web.css

Tento balík obsahuje třídu `CSSFactory`, jež je vstupní bod knihovny jStyleParser. Dále jsou v balíku obsaženy rozhraní datových typů, pomocí jejich výchozí implementace v balíku `cz.vutbr.web.csskit` je následně mapována vstupní data jazyka CSS na odpovídající datové typy. V balíku se nachází ještě rozhraní `CSSProperty`, které poskytuje základ pro CSS vlastnosti. Implementováním tohoto rozhraní mohou být přidány nové CSS vlastnosti.

Balík cz.vutbr.web.csskit

V balíku se nachází základní implementace rozhraní z balíku `cz.vutbr.web.css`. Používání tříd z tohoto balíku může být změněno zaregistrováním jiných implementací pomocí volání metod `register*` v třídě `CSSFactory`.

Uvnitř tohoto balíku se také nachází balík `antlr`, který obsahuje třídy analyzátoru vygenerovaného nástrojem ANTLR verze 3. Vygenerované třídy slouží k analýze CSS vstupu a jeho mapování do struktur definovaných v balíku `cz.vutbr.web.css`.

Balík cz.vutbr.web.domasing

Balík mimo jiné poskytuje třídu `Analyzer`, které umí třídit CSS deklarace, klasifikovat je napříč CSS médii a přiřadit je DOM elementům. K přiřazování CSS deklarací elementům DOM slouží třída `DeclarationTransformer`.

Balík org.fit.net

Tento balík obsahuje pomocné třídy pro zajištění komunikace po síti při stahování externích CSS souborů umístěných na internetu.

5.4 Využití nástroje ANTLR

Zdrojové soubory gramatik pro lexikální a syntaktickou analýzu a gramatika pro generování abstraktního syntaktického stromu se nachází ve složce `src/main/atnlr3`. Z těchto tří souborů jsou při překladu programu generovány příslušné třídy jazyka Java – DefaultCSSLexer

– lexikální analyzátor, DefaultCSSParser – syntaktický analyzátor a DefaultCSSTreeParser – generátor abstraktiního syntaktického stromu. Všechny generované soubory jsou vygenerovány do balíku `cz.vutbr.web.csskit.antlr`, ve kterém jsou také definovány třídy, které slouží pro celý proces parsování.

5.4.1 Gramatika pro lexikální analýzu

Tato gramatika se nachází v souboru `CSSLexer.g`. Na začátku je sekce *imaginárních* tokenů, které jsou poté použity v syntaktické analýze na straně přepisovacích pravidel. Příklad použití imaginárních tokenů pro zajištění generování kombinátoru mezi dvěma CSS selektory:

CSSLexer.g

```
tokens{
   ADJACENT;
   PRECEDING;
   CHILD;
   DESCENDANT;
}
```

CSSParser.g

```
combinator
    : GREATER S* -> CHILD
    | PLUS S* -> ADJACENT
    | TILDE S* -> PRECEDING
    | S -> DESCENDANT
    ;
```

Po sekci s tokeny následuje sekce `@members`. V této sekci jsou definovány všechny pomocné proměnné a objekty, které jsou využity při lexikální analýze. Část této sekce je pro ilustraci zobrazena zde:

```
@members {
    // třída pro logování informací
    private org.slf4j.Logger log;

    // počet aktuálně zpracovaných tokenů
    protected int tokencnt = 0;

    //továrna na vytváření tokenů
    protected cz.vutbr.web.csskit.antlr.CSSTokenFactory tf;

    //metoda pro inicializaci lexeru
    public void init() {
    }
}
```

V sekci members je také definována metoda init, kterou je nezbytné volat ihned po vytvoření instance třídy `CSSLexer`. Další částí akce `@members` jsou metody, které přepisují metody třídy `Lexer`, která je součástí knihovny ANTLR – jsou to metody pro vytvoření a získání dalšího tokenu (tyto metody obalují klasické tokeny kontextuálními informacemi z důvodu uchování těchto informací pro další práci s nimi) a metody pro zotavení z chyb. Následující kód popisuje definici jednotlivých tokenů tak, jak je uvedeno v gramatice jazyka CSS. Příklad definice tokenu pro token `FONTFACE`:

```
FONTFACE
   : '@font-face' ;
```

U některých pravidel pro tvorbu tokenů jsou definovány části kódu, pomocí kterých je uchovávána informace o aktuálně zpracovávaném tokenu. Jsou to především závorky, u kterých je pomocí těctho akcí zajištěno pamatování aktuálního stavu párování závorek v třídě `CSSLexerState`.

Poslední část gramatiky je sekce s fragmenty, tyto fragmenty jsou využity v definicích tokenů, nejsou ale dostupné v gramatice pro syntaktickou analýzu. Příklad definice tokenu PERCENTAGE, který slouží ke zpracování procentuální hodnoty s využitím fragmentu NUMBER_MACR:

```
PERCENTAGE
   : NUMBER_MACR '%' ;

fragment NUMBER_MACR
   : ('0'..'9')+ | (('0'..'9')* '.' ('0'..'9')+) ;
```

Zpracování pravidla CHARSET

Speciálním případem v lexikální analýze je zpracování pravidla CHARSET. Toto pravidlo se může vyskytovat pouze na začátku dokumentu a specifikuje znakovou sadu zpracovávaného dokumentu. V případě, že lexikální analyzátor zpracovává token `'@charset'`, tak již v průběhu lexikální analýzy je nutné změnit znakovou sadu zdrojového souboru. Po programové stránce je tento problém vyřešen tak, že vstupní stream souboru, o který se obecně stará třída z knihovny ANTLR, je zapouzdřen ve třídě, která umožňuje změnu znakové sady při procesu lexikální analýzy.

5.4.2 Gramatika pro syntaktickou analýzu

V souboru `CSSParser.g` je gramatika pro syntaktický analyzátor. Tato gramatika využívá pro zajištění tokenů gramatiku CSSLexer, výstup gramatiky parseru je nastaven na AST (abstraktní syntaktický strom) a hloubka dopředného vyhledávání k je nastavena na hodnotu 2. V gramatice se nachází akce `@members`, ve které se, stejně jako v předchozí gramatice, nachází funkce pro inicializaci parseru a funkce pro zotavení z chyb. Po akci `@members` následuje výčet pravidel gramatiky, tak aby odpovídala jazyku CSS. V gramatice jsou definována i pravidla, která nepopisují správnou konstrukci jazyka, ale slouží pro zpracování chybných tokenů na vstupu – jsou to pravidla např. `nostatement`, `noprop`, `norule` apod. Gramatika pro parser neodpovídá úplně přesně definici gramatiky pro CSS úroveň 2.1, ale obsahuje již některé prvky syntaxe pro úroveň CSS3, což může mít za následek jiné zpracování výsledků v některých specifických případech oproti ostatním parserům se striktním dodržením CSS 2.1.

V této gramatice je také pomocí `catch` bloků definováno chování v případě zachycení vyjímky typu `RecognitionException`. Tyto vyjímky jsou parserem zachyceny v případě vzniku syntaktické chyby. Pomocí definice těla bloků, zachytávajících tyto vyjímky, je pomocí třídy `CSSTreeNodeRecovery` vyřešeno zotavování z těchto chyb.

5.4.3 Gramatika pro konstrukci abstraktního syntaktického stromu

Gramatika pro konstrukci AST slouží k tomu, aby se z přepisovacích pravidel definovaných v gramatice pro parser vygeneroval správný syntaktický strom a zároveň se při konstrukci tohoto stromu volaly akce, které vytváří výslednou datovou reprezentaci vstupních dat. V gramatice jsou jednotlivým pravidlům gramatiky přiřazeny návratové hodnoty, pomocí kterých je datová reprezentace vytvořena. Pravidla se mezi sebou dle gramatiky volají navzájem a uchovávají výstupy volaných pravidel v pomocných proměnných, které jsou do sebe následně zapouzdřovány a postupně navraceny až do nejvyšší úrovně. Kód programu, který zpracovává mezivýsledky v jednoltivých pravidlech je definován také v této gramatice a to u jednotlivých částí v blocích, které jsou obaleny pomocí složených závorek . Po zpracování celého vstupu jsou tak v objektu AST uchovány všechny potřebné informace a poté stačí na tomto stromu jen zavolat metodu, která vrací požadovaná data.

Třída, která je z této gramatiky obsahuje jak programově vygenerovaný kód, který se stará o parsování, tak právě uživalsky definovaný kód. Protože jsou zdrojové kódy parsování a zpracování dat v jednom souboru, tak je těžké se v tomto souboru orientovat při ladění aplikace.

5.4.4 Pomocné třídy analyzátoru

Vedle gramatik, ze kterých jsou definovány zdrojové kódy výsledného parseru, se v balíku antlr nachází pomocné třídy, které jsou v generovaném parseru využívány, nebo slouží pro práci s výsledky parseru.

K vytváření vlastních tokenů v lexeru slouží továrna `CSSTokenFactory`, pro pracování se vstupními daty je k dispozici třída `CSSInputStream`. Pro uchovávání aktuálního stavu lexikální analýzy slouží třída `CSSLexerState` a pro zotavení z lexikálních chyb existuje třída `CSSTokenRecovery`. Pro parser a parser AST se zde nachází nástroj pro čtení výrazů – `CSSExpressionsReader`, pro zotavení z chyb slouží třída `CSSTreeNodeRecovery`. Pro práci s parsovacím stromem slouží třída `Treeutil` a pro zpracování výsledků je používána třída `SimplePreparator`.

Pro vytvoření parseru a lexeru, jejich inicializaci a zpracování vstupních dat slouží třída `CSSParserFactory`, která slouží jako vstupní balíku antlr.

Kapitola 6

Návrh aplikace s využitím ANTLR verze 4

V této části práce je popsán návrh přechodu ze stávající verze ANTLR 3 na verzi ANTLR 4. Při návrhu je vycházeno z toho, že po přechodu na novou verzi generátoru musí být zachována stejná funkčnost, jako ve verzi předchozí. První podkapitola se zabývá úpravou struktury projektu – změnou jména balíku antlr na antlr4. Dále následuje popis úprav gramatik, ze kterých je pomocí nástroje ANTLR v.4 generován celý analyzátor – gramatika pro lexikální a syntaktickou analýzu a gramatika pro konstrukci parsovacího stromu a zpracování výsledků analýzy. Po těchto kapitolách je popsán návrh úpravy pro zpracovávání chyb a zotavování se z nich. V poslední podkapitole jsou uvedeny úpravy tříd, které souvisí s celým analyzátorem a v průběhu analýzy jsou využívány.

6.1 Změna balíku antlr na antlr4

Jelikož se původní verzi projektu všechny třídy, které souvisí s tvorbou lexikálního a syntaktického analyzátoru nachází v balíku `cz.vutbr.web.csskit.antlr` a celý projekt ANLTR při přechodu z verze 3 na verzi 4 změnil v interní struktuře balík `antlr` na `antlr.v4`, tak nová verze projektu jStyleParser bude pro analyzátor z důvodu lepší přehlednosti používat balík `antlr4`. Vnější rozhraní projektu a metody pro získání výsledků extrakce dat ze vstupního souboru zajišťuje třída `CSSParser`, jejíž chování se navenek vůbec nezmění, takže by v jiných projektech, které jStyleParser využívají, při aktualizaci z verze 1.23 na verzi 2.0 měla být zachována zpětná kompatibilita a nebude potřeba provádět v kódu žádné úpravy. Změna balíku antlr na antlr4 je tedy pouze v rámci interní struktury projektu a na venek se nijak neprojeví.

6.2 Lexikální analýza

Prvním krokem při tvorbě nového analyzátoru je vytvoření gramatiky pro vygenerování lexikálního analyzátoru. Gramatika bude vycházet z původní gramatiky pro verzi 3, ale musí být kompatibilní s ANTLR v.4. Lexikální analyzátor využívá pomocné třídy `CSSLexerState`, `CSSTokenFactory` a `CSSTokenRecovery`, které bude potřeba upravit tak, aby byly opět kompatibilní s novou verzí generátoru. V souvislosti se změnou třídy `CSSTokenFactory` je nutné provést změny také ve třídách `CSSToken` a `CSSInputStream` z důvodu odlišnosti

některých parametrů a chování v nové verzi ANTLR. Tyto úpravy jsou popsány v kapitole 6.6. V původní gramatice se také používá třída `CSSExpressionReader`, která slouží ke zpracování CSS expressions, které již nejsou podporované a proto bude v nové gramatice tato třída odstraněna. V případě potřeby bude do gramatiky přidána nová funkčnost tak, aby bylo zajištěno stejných výsledků, jako ve stávající verzi.

6.3 Syntaktická analýza

Po úpravě gramatiky pro lexikální analýzu následuje úprava gramatiky pro syntaktický analyzátor. Tato úprava spočívá především v odstranění přepisovacích pravidel a úpravě ošetření chyb vzniklých při syntaktické analýze. Při odstraňování přepisovacích pravidel musí být alternativy jednotlivých pravidel vedoucí ke zneplatnění pravidla, ošetřeny také při zpracování výsledků analýzy. Úprava ošetření chyb vyvolaných při analýze spočívá v zachytávání vyjímek typu `RecognitionException` a definici chování vedoucího ke správnému zotavení chyby. Chování při zachycení vyjímky bude obdobné jako u stávající verze, ale budou využity možnosti nové verze generátoru.

6.4 Tvorba syntaktického stromu a zpracování výsledků analýzy

Po syntaktické analýze je třeba zpracovat výsledný parsovací strom a následně ze získaných hodnot vytvořit datovou strukturu, která odpovídá zpracovanému CSS. V aktuální verzi projektu existuje gramatika `CSSTreeParser.g`, z které se vygeneruje analyzátor pro výstupní `CommonTree` a při průchodu těchto dat je vygenerována výsledná dataová struktura v jazyce Java. Tato gramatika definuje tvorbu parsovacího stromu a související akce, které jsou prováděny při průchodu tímto stromem. Tyto akce jsou části Java kódu a zpracovávají výsledek syntaktické analýzy. Jelikož jsou definovány v gramatice pro generátor a následně při vygenerování analyzátoru převedeny do třídy, která slouží pro tvorbu a průchod parsovacím stromem, tak jsou tyto části smíchány s řídícími příkazy vygenerovanými generátorem. Ladění těchto akcí je proto velice náročné a je nutné se zorientovat v programovém kódu, který vygeneroval nástroj ANTLR. V nové verzi projektu tato gramatika vůbec nebude použita a jednotlivé akce budou transformovány do tříd implementujících návrhový vzor `Visitor` a `Listener`, pomocí kterých bude probíhat zpracování a vyhodnocení výsledků analýzy zdrojových dat. Výsledky zpracování vstupních CSS dat v nové verzi musí být identické se strukturou z předchozí verze.

6.5 Zpracování a zotavení chyb

Jakmile bude dosaženo správného mapování hodnot na datové typy, tak jak již bylo výše zmíněno bude třeba zajistit obdobné zpracování chyb vzniklých při lexikální a syntaktické analýze vstupních dat. V nové verzi bude použito implementace rozhraní `ErrorStrategy`. Implementací tohoto rozhraní se dá vytvořit vlastní mechanismus, pomocí kterého se zpracovávají chyby vzniklé při lexikální a syntaktické analýze. Některé lexikální chyby jsou v projektu zpracovávány již na úrovni vygenerovaného lexikálního analyzátoru a stará se o to třída `CSSTokenRecovery`. Ve stávající gramatice je použita třída `CSSTreeNodeRecovery`, jejíž funkčnost bude transformována do již zmíněné implementace rozhraní `ErrorStrategy`.

Tyto třídy budou muset být také upraveny tak, aby jejich funkčnost odpovídala stavu před aktualizací nástroje ANTLR.

6.6 Úpravy tříd používaných při analýze

Tato podkapitola popisuje úpravy nejdůležitějších tříd, které budou v balíku antlr4 a úzce souvisí s generováním syntaktického analyzátoru a následným zpracováním výsledků analýzy.

6.6.1 CSSParserFactory

Jednou z nejdůležitějších tříd v balíku, který se stará o analýzu, je třída `CSSParserFactory`. Tato třída poskytuje metody pro zpracovnání zdrojových dat a odstiňuje vnitřní vytvoření analyzátoru. V této třídě bude třeba upravit právě ty metody, které se starají o tvorbu, inicializaci a získání výsledků z vygenerovaného analyzátoru. Při úpravách musí být dodržena veškerá funkčnost veřejných metod tak jak bylo dosud, aby přechod ze starší verze na novější neměl vliv na používání třídy `CSSParserFactory`.

6.6.2 CSSInputStream

Tato třída bude upravena tak, aby odpovídala nové definici třídy ANTLRInutStream a zachovala všechny stávající metody.

6.6.3 CSSTokenRecovery

Úpravy této třídy budou probíhat až podle nekompatibility používaných nových vlastností generátoru ANTLR v.4.

6.6.4 CSSTreeNodeRecovery

Jak již bylo zmíněno výše, tak metody z této třídy budou přesunuty a upraveny do třídy `CSSErrorRecovery`. Dále bude odstraněna třída `TreeUtil`, která je využívána touto třídou a po odstranění tedy bude zbytečná.

6.6.5 CSSExpressionsReader

Po domluvě s vedoucím této práce bude tato třída odstraněna, protože v CSS již není podporována.

Kapitola 7

Implementace

Tato kapitola obsahuje detailní popis implementace úprav projektu jStyleParser, který nyní využívá nejnovější verzi generátoru ANTLR – 4.5.3. V první části je nejprve popsán způsob, jakým byly upraveny a transformovány gramatiky z původní verze projektu. Po popisu transformací gramatik následuje podkapitola, ve které jsou popsány změny tříd v balíku `cz.vutbr.web.csskit.antlr4`. Tyto třídy slouží pro zpracování výsledků analýzy, nebo rozšiřují funkcionalitu tříd používaných nástrojem ANTLR. V podkapitole 7.4 je popsán způsob zpracování naparsovaných informací do datových struktur v jazyce Java. Při tvorbě nové verze projektu byly implementovány oba způsoby, kterými se dají výsledky analýzy zpracovat. Po následném zhodnocení obou metod bylo s vedoucím této práce domluveno, že se bude používat zpracování výsledků pomocí implementace návrhového vzoru `Visitor`. Toto řešení bylo vyhodnoceno jako čistější, přehlednější a při dalším vývoji projektu bude implementace zpracování nových CSS vlastností jednodušší. Poslední podkapitola přibližuje to, jakým způsobem se v nové verzi aplikace řeší zpracování chyb v průběhu lexikální a syntaktické analýzy. Také je popsán způsob následného zotavení z těchto chyb tak, aby byla zachována stejná funkčnost jako v přechozí verzi projektu.

7.1 Aktualizace nástroje ANTLR na verzi 4.5.3

Jelikož se na správu projektu jStyleParser používá nástroj pro správu projektů Maven, tak aktualizace knihovny nástroje ANTLR 4.5.3 znamenala změnu konfiguračního souboru nástroje Maven – soubor `pom.xml`. V tomto konfiguračním souboru byla provedena změna cesty pro nástroj ANTLR (`*/antlr3 na */antlr4`). Dále v konfiguraci pluginu pro ANTLR byla změněna verze z 3.5.2 na 4.5.3 a do sekce `properties` byla přidána položka `<antlr4.visitor>true</antlr4.visitor>`, která zajistí vygenerování rozhraní pro návrhový vzor `Visitor`, pomocí kterého je následně zpracován výsledek analýzy CSS dat.

7.2 Transformace gramatik

Dalším krokem při zahájení implementace a začlenění nové verze generátoru ANTLR do projektu byla analýza a tranformace gramatik, pomocí kterých jsou generátorem ANTLR vygenerovány třídy implementující lexikální a syntaktický analyzátor a následně nástroj na získávání naparsovaných informací.

Všechny gramatiky se v původním projektu nachází ve složce `src/main/antlr3` a k nim existují korespondující gramatiky s prefixem `Default` ve složce

`/cz/vutbr/web/csskit/antlr`. Soubory s prefixem obalují gramatiku bez prefixu a obsahují základní metody pro práci s vygenerovanými analyzátory. Do nové verze byly gramatiky sjednoceny do jednoho souboru a převedeny do nové složky: `src/main/antlr4/cz/vutbr/web/csskit/antlr4`. Změna složky pro gramatiku lexikální a syntakticku byla provedena z důvodu zanesení informace o tom, že se jedná právě o gramatiku pro ANTLR4 a dále hierarchickou strukturu složek, podle které nástroj ANTLR4 získá jméno balíku pro vygenerovaný parser a také proto, aby se zamezilo případným konfliktům s předchozí verzí. Předchozí gramatiky měly příponu `ġ`, avšak ANTLR4 vyžaduje gramatiky s příponou `ġ4`, takže nové gramatiky byly podle tohoto vzoru přejmenovány. Změna struktury souborů s gramatikami pro verzi 3 a nově pro verzi 4 vypadá takto:

```
        verze s ANTLR 3                          verze s ANTLR 4

src/main/antlr3/                          src/main/antlr4/
   - cz/vutbr/web/csskit/antlr/               - cz/vutbr/web/csskit/antlr/
      - DefaultCSSLexer.g                       - CSSLexer.g4
      - DefaultCSSParser.g                      - CSSParser.g4
      - DefaultCSSTreeParser.g
   - CSSLexer.g
   - CSSParser.g
   - CSSTreeParser.g
```

Následující 3 podkapitoly se zabývají transformací původních gramatik pro ANTLR 3 na ekvivalentní gramatiky pro ANTLR 4.

7.2.1 Gramatika pro lexikální analýzu

Tato gramatika se v původním projektu nachází ve dvou souborech: `CSSLexer.g` a `DefaultCSSLexer.g`. V novém projektu je gramatika spojena do jednoho souboru s názvem: `CSSLexer.g4`.

Soubor `CSSLexer.g4` obsahuje definice tokenů a fragmentů pro lexikální analýzu a dále jsou zde přepsány zděděné metody pro interní práci lexeru – nastavení zpracovávaného proudu dat, získání dalšího tokenu, obnovení při chybě a zpracování chybových zpráv. V gramatice je ještě metoda, která slouží pro inicializaci celé třídy a je nutné ji volat ihned po vytvoření instance lexikálního analyzátoru. Transformace této gramatiky do nové verze zahrnovala následující úpravy:

- změna zápisu uživatelských akcí @init a @after – v nové verzi generátoru se uživatelské akce píší přímo do těla definice pravidla, názorně ukázáno na následujícím obrázku u pravidla pro řetězec:

```
 ANTLR 3                                ANTLR 4

STRING                                STRING :
  @init{                                {
     //akce před zpracováním              //akce před zpracováním
  }                                     }
  @after{                               STRING_MACR
     //akce po zpracování               {
  }                                        //akce po zpracování
  : STRING_MACR;                        };
```

- změna zpracování pravidla `CHARSET` pomocí třídy `CSSToken`, která poskytnuje novou metodu pro získání hodnoty pravidla pro `@charset`, neboť při lexikální analýze v ANTLR v.4 nelze v průběhu analýzy přistupovat k jednotlivým fragmentům, ze kterých se token skládá, a tak bylo nutné název znakové sady získat z kontextu celého tokenu.
- bylo zavedeno nové pravidlo, které zpracovává neukončený řetězec pro případ neočekávaného ukončení zpracovávaných vstupních dat:

```
fragment UNCLOSED_STRING
: UNCLOSED_STRING_MACR

fragment UNCLOSED_STRING_MACR
: QUOT (STRING_CHAR | APOS {ls.aposOpen=false;} )*
| APOS (STRING_CHAR | QUOT {ls.quotOpen=false;} )* ;
```

- používání třídy `CSSExpressionReader` bylo odstraněno, protože výraz `expression()` již není podporovaný. Zpracování tokenu ale zůstalo kvůli zotavení z případného výskytu tohoto pravidla ve zdrojovém CSS souboru
- dále byly upraveny některé speciální příkazy ANTLR verze 3 tak, aby byl zachován jejich význam v zápisu pro ANTLR 4

7.2.2 Gramatika pro syntaktickou analýzu

Stejně jako gramatika pro lexikální analýzu je i gramatika pro syntaktickou analýzu v původní verzi projektu obsažena ve 2 souborech: `CSSParser.g` a `DefaultCSSParser.g`. V nové verzi projektu je gramatika obsažena v souboru `CSSParser.g4`. Při transformaci gramatiky pro syntaktickou analýzu byly provedeny tyto úpravy:

- všechny regulární výrazy, které pokrývají bílé znaky typu: `S!*` byly převedeny na `S*`. V generátoru ANTLR verze 3 slouží vykřičník jako speciální znak k tomu, aby token nebyl zahrnut do AST – verze 4 tento speciální znak nemá, takže tokeny obsahující bílé znaky jsou obsaženy v kontextu jednotlivých pravidel. Pro jejich odstranění vznikla v třídě `CSSParserVisitorImpl` metoda `filterSpaceTokens`.
- bylo provedeno odstranění přepisovacích pravidel z gramatiky. Přepisovací pravidla v ANTLR v.3 slouží ke konstrukci AST z parsovacích pravidel. Jelikož ANTLR v.4

nepracuje s AST, tak tato pravidla jsou z gramatiky pro parser odstraněna. Při odstraňování bylo třeba při následném zpracovávání výsledků analýzy u některých pravidel upravit chování tak, aby byla kompatibilita s původní verzí dodržena. Příklad přepisovacích pravidel v pravidle pro `combinator`(modře):

```
combinator
: GREATER S*  -> CHILD
| PLUS S* -> ADJACENT
| TILDE S* -> PRECEDING
| S -> DESCENDANT
;
```

- v gramatice bylo nutné upravit všechny bloky, v kterých se zachytávají vyjímky typu `RecognitionException`. Obsah těchto bloků byl nahrazen odpovídajícím kódem, který zajišťuje zotavení z chyb tak, aby jeho chování bylo totožné jako v původní verzi projektu. Místo původního přepisování výsledního parsovacího stromu byl do aktuálního kontextu pravidla přidán chybový token, jehož výskyt je potom při zpracování výsledků analýzy v třídě `CSSParserVisitorImpl` kontrolován.
- do gramatiky byla přidána nová metoda `getCSSErrorHandler`, která slouží pro získání objektu `CSSErrorStrategy`, který nahradil `CSSTreeNodeRecovery` a je využívána v přechozím bodě (zotavení z chyb).

7.2.3 Gramatika pro tvorbu parsovacího stromu

Gramatika se v původním projektu nachází ve dvou souborech: `CSSTreeParser.g` a `DefaultCSSTreeParser.g`.

V novém projektu se tato gramatika vůbec nevyskytuje, protože generátor ANTLR v.4 pro zpracování výsledků analýzy využívá návrhového vzoru `Listener` nebo `Visitor` a zpracování výsledků analýzy tedy znamená implementaci rozhraní některého ze zmíněných návrhových vzorů. Popis zpracování výsledků analýzy je popsán v samostatné podkapitole 7.4.

Přestože gramatika byla odstraněna, tak byla využita při implementaci zpracování výsledků analýzy, protože obsahovala uživatelsky definované metody, které v původním projektu při procházení parsovacího stromu sloužily.

7.3 Úpravy tříd, které rozšiřují funkčnost analyzátoru

V této podkapitole jsou popsány úpravy tříd, které se nachazí v balíku antlr4 a jsou využívané pro vytvoření a inicializaci parseru, nebo jsou při samostatné analýze využity.

7.3.1 CSSErrorStrategy

Třída `CSSErrorStrategy` implementuje rozhraní `ANTLRErrorStrategy`, které definuje metody pro zotavení z chyb vzniklých při syntaktické analýze. Do této třídy byly přidány metody z třídy `CSSTreeNodeRecovery`, důležité jsou především metody `consumeUntil` a `consumeUntilGreedy`. Tyto metody slouží k tomu, aby v případě chyby přeskočily ve vstupních datech všechny tokeny do té doby, dokud na vstupu není vyžadovaný token a parser

tedy může zpracovat další pravidlo správně. Přeskakování chyb je voláno v případě zachycené vyjímky `RecognitionException` v třídě `CSSParser`.

7.3.2 CSSErrorListener

Nová verze generátoru definuje rozhraní `ErrorListener`, které tato třída implementuje a je předávána lexikálnímu a syntaktickému analyzátoru. Pomocí této třídy je na chybový výstup odesílána informace o lexikálních a syntaktických chybách vzniklých při zpracování vstupních dat.

7.3.3 CSSExpressionReader

Třída sloužící ke zpracování dynamických vlastností v CSS známých jako *CSS expressions* byla zrušena a podpora těchto CSS vlastností byla zrušena, protože dle [1] nejsou *CSSExpressions* od prohlížeče `Internet Explorer 8` a novějších podporovány. Parser počítá s možným výskytem těchto pravidel, ale ignoruje je. Pro ověření správné funkčnosti parseru byl přidán test `GrammarRecoveryTest3.expressionRecovery`.

7.3.4 CSSInputStream

Třída `CSSInputStream` se stará o zpracování vstupních dat, obaluje základní třídu definovanou v generátoru ANTLR a poskytuje některé další funkce jako je např. změna znakové sady zpracovávaného souboru. Tato třída je využívána lexerem. Změna této třídy spočívala v tom, že nově rozšiřuje třídu `ANTLRInputStream` namísto původní implementace třídy `CharStream`. S touto změnou souvísí drobné změny při zpracování vstupního souboru tak, aby bylo dodrženo původní chování.

7.3.5 CSSLexerState

V této třídě byly pouze zjednodušeny některé výrazy jazyka Java.

7.3.6 CSSParserFactory

Ve třídě `CSSParserFactory` bylo nutné upravit metody, které se starají o vytvoření a inicializaci `CSSLexer`-u a `CSSParser`-u. Dále byly upraveny metody, které zajišťují předání vstupních dat těmto objektům a nakonec metody pro zpracování a vrácení výsledků analýzy. Jelikož byla pro zpracování implementována obě rozhraní, tak kód původně obsahoval obě varianty zpracování. Po konzultaci s vedoucím práce bylo ze zkušeností při vývoji a zřejmé přehlednosti kódu zpracování výsledků analýzy pomocí implementování návrhového vzoru `Visitor` zachována pouze tato metoda. Pokud by do budoucna bylo třeba využívat třídu `CSSParserListener`, tak je na to aplikace připravena a bylo vytvořeno rozhraní `CSSParserExtractor`, které je popsáno v následující podkpaitole.

7.3.7 CSSParserExtractor

Jak již bylo výše zmíněno, tak v nové verzi projektu bylo přidáno nové rozhraní, které v třídě `CSSParserFactory` umožňuje použít metodu zpracování výsledků jak pomocí třídy `CSSParserVisitor`, tak i pomocí třídy `CSSParserListener`. V projektu je použita pouze třída `CSSParserVisitor`, ale kdyby chtěl někdo používat `CSSParserListener`, tak stačí

v `CSSParserFactory` upravit metodu `parse` a `parseMediaQuery` a v nich začít používat `CSSParserListener` namísto aktuálního `CSSParserVisitor`.

7.3.8 CSSParserListenerImpl, CSSParserVisitorImpl

Tyto dvě třídy byly do nové verze přidány a jak jejich název napovídá, tak obsahují implementace jednotlivých rozhraní vygenerovaných nástrojem ANTRL. Tyto třídy slouží ke zpracování výsledků analyzátoru – tvorbě datové struktury reprezentující vstupní CSS data – a následnému poskytnuní zpracovaných výsledků. Detaily implementace zpracování výsledků analýzy je popsáno v podkapitole 7.4.

7.3.9 CSSToken

Tato třída rozšiřuje základní třídu `CommonToken`. V nové verzi byla rozšířena o metodu `extractCHARSET`, která se stará o extrakci znakové sady z pravidla `@charset` již ve fázi lexikální analýzy. Dále byl přidán nový typ tokenu – `UNCLOSED_STRING`. Protože se nepodařilo jinou cestou docílit správného zpracování neuzavřených řetězců při neočekávaně ukončeném vstupním souboru (zotavení z chyb typu `Unexpected EOF`).

7.3.10 CSSTokenFactory

V této třídě bylo upraveno předávání parametru `input` – změna parametru typu `CharStream` na `Pair<TokenSource, CharStream>` z důvodu předávání parametru při vytváření objektu `CSSToken`.

7.3.11 CSSTokenRecovery

V třídě zabývající se zpracováním tokenů pro lexer a zotavováním z lexikálních chyb byla modifikována metoda `nextToken`, která slouží pro získání následujícího tokenu z lexeru. Úprava byla provedena tak, aby reflektovala předchozí funkčnost a zároveň fungovala v generátoru ANTLR v.4.

7.3.12 CSSTreeNodeRecovery a TreeUtil

Třída `CSSTreeNodeRecovery`, sloužící ke zotevení z chyb při tvorbě AST generátoru ANTLR v.3, byla odstraněna spolu s pomocnou třídou pro práci s AST – `TreeUtil` a její metody byly transformovány do třídy `CSSErrorStrategy`. Pro další informace o třídě `CSSErrorStrategy` viz 7.3.1.

7.3.13 Preparator a SimplePreparator

Toto rozhraní a třída slouží k přípravě bloků kaskádových stylů při zpracovávání výsledků analýzy a nebyly v nové verzi nijak změněny.

7.4 Zpracování výsledků analýzy

Jak již bylo zmíněno výše, tak o zpracování výsledků analyzátoru vygenerovaného nástrojem ANTLR verze 4 se starají třídy, které implementují rozhraní návrhového vzoru `Listener` nebo `Visitor`. Tato rozhraní jsou stejně tak, jako analyzátor vygenerovány automaticky nástrojem ANTLR v.4. Po dohodě s vedoucím této práce bylo z repozitáře odstraněno

používání třídy `CSSParserListner` a bylo ponecháno pouze zpracování výsledků analýzy pomocí třídy `CSSParserVisitor` popsané v následující podkapitole. Odstranění listeneru bylo především z důvodu jeho (ne)přehlednosti a také proto, aby se v budoucnu při provádění úprav zpracování tyto změny nemusely implementovat na dvou místech. V podkapitole 7.4.2 je však implementace listeneru popsána i přes jeho odstranění. Čtenář tak může vidět rozdíly mezi provoláváním metod implementujících rozhraní a obě metody porovnat.

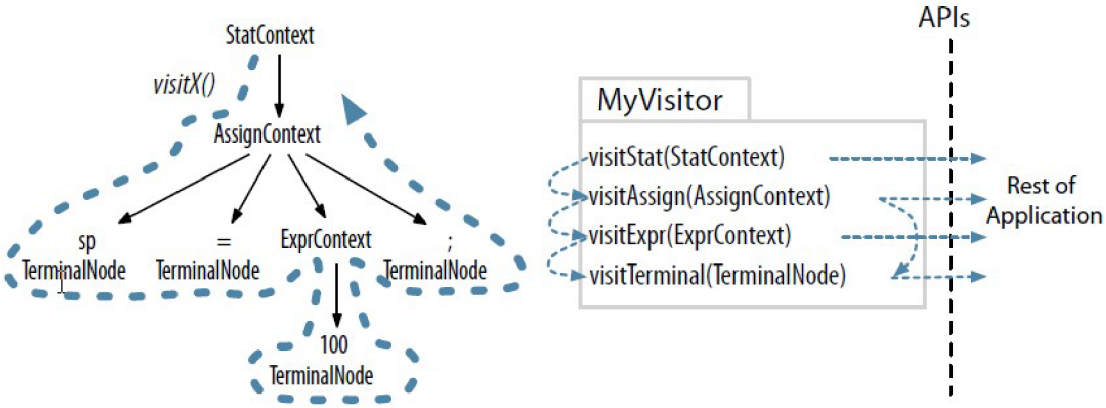
7.4.1 CSSParserVisitor

Z gramatiky pro parser je pomocí tohoto nástroje vygenerované rozhraní s následující podobou:

```
public interface CSSParserVisitor<T> extends ParseTreeVisitor<T> {
    // definice metod rozhraní, které odpovídají jednotlivým pravidlům
    // gramatiky parseru
    T visitStylesheet(CSSParser.StylesheetContext ctx);
    T visitStatement(CSSParser.StatementContext ctx);

    // a obdobně ostatní metody pro ostatní pravidla
    T visit.........(CSSParser........Context ctx);
}
```

Následující obrázek demonstruje jak funguje volání mezi jednotlivými metodami visit:



Obrázek 7.1: Ilustrace volání mezi metodami pro visitor. Převzato z [20]

V metodě `visitStat` je možno volat metodu `visitAssign` s kontextem pro pravidlo `assign` a tím pádem je možno získat návratovou hodnotu této metody a dále s ním pracovat, nebo vyhodnotit, zda není navrácený výsledek špatný. Na základě aktuálního stavu je také možno se rozhodnout, že hodnota celého vnořeného pravidla není třeba zjišťovat (např. program se nachází v chybovém stavu a mezivýsledek již není třeba vyhodnocovat, protože bude celé pravidlo zneplatněno).

Rozhraní je implementováno třídou `CSSParserVisitorImpl`. Na začátku definice třídy je sekce, která obsahuje privátní proměnné sloužící pro uchování zpracovaných dat. Pro získání zpracovaných dat z této třídy slouží metody `getImportMedia`, `getImportPaths`, `getRules` a `getMedia`. Třída také obsahuje definice pomocných metod, které slouží pro

práci s aktuálně zpracovávaným kontextem pravidla a jeho obsahem. Metody `extract*` jsou používané na nejrůznější extrakce a ošetření textu z kontextu jednotlivých pravidel. Pro zjištění, zda se v kontextu nachází token nesoucí informaci o syntaktické chybě, slouží metoda `ctxHasErrorNode`.

Při implementaci zpracování průchodu jednotlivými pravidly bylo využito částí kódů, které byly v původním projektu specifikovány v gramatice `CSSTreeParser.g`. Tyto kódy byly použity tak, aby při procházení programu bylo zřetelné, jak se mezi sebou provolávají jednotlivé metody, které jsou aplikované na příslušná pravidla, a jak se vytváří datová reprezentace. Při vytváření datové reprezentace jednotlivých pravidel jsou používány továrny `TermFactory`, `RuleFactory` a `Preparator`, jenž poskytují funkce pro vytváření odpovídajících datových struktur. Tyto objekty jsou pak shromažďovány v privátních proměnných, jak již bylo zmíněno. V kódu jsou rovněž zakomponovány pomocné výpisy, které jsou při vývoji velice užitečné – tyto výpisy jsou v programu implementovány pomocí služby `log4j`.

Narozdíl od předchozí verze projektu se již v této třídě pro zpracování výsledků parsování nenachází žádný strojově generovaný kód, takže je zpracování výsledků přehledné a pochopitelné.

V následující části kódu je demonstrováno zpracování pravidla `stylesheet` (vstupní pravidlo gramatiky), Pravidlo z gramatiky je definováno v komentáři před funkcí. Z všech alternativ je třeba zpracovávat pouze `statement`-y, kterých může být `0-n`. Význam jednotlivých částí kódu je popsán pomocí komentářů v kódu.

```
/**
  * Hlavní pravidlo pro CSS dokument
  * stylesheet: ( CDO | CDC  | S~| nostatement | statement )*
  */
@Override
public RuleList visitStylesheet(CSSParser.StylesheetContext ctx) {
    // výpis informace o~vstupu do metody
    logEnter("stylesheet: " + ctx.getText());

    // inicializace proměnné pro uchování všech zpracovaných pravidel
    this.rules = new RuleArrayList();

    // projde všechny definovaná pravidla a zpracuje je
    for (CSSParser.StatementContext stmt : ctx.statement()) {

        // zavolání visit metody na statement
        RuleBlock<?> s~= visitStatement(stmt);

        // statement je validní
        if (s~!= null) {
            // přidání do pravidel
            this.rules.add(s);
        }
    }

    // výpis zpracovaných hodnot
    log.debug("\n***\n{}\n***\n", this.rules);
    // výpis informace o~opuštění metody
    logLeave("stylesheet");

    //navrácení zpracovaných hodnot
    return this.rules;
}
```

7.4.2 CSSParserListener

Rozhraní pro listener vygenerované z gramatiky pomocí ANTLR v.4 má následující podobu:

```
public interface CSSParserListener extends ParseTreeListener {
    // definice metod rozhraní, které odpovídají jednotlivým pravidlům
    // gramatiky parseru
    void enterStylesheet(CSSParser.StylesheetContext ctx);
    void exitStylesheet(CSSParser.StylesheetContext ctx);

    // a obdobně ostatní metody pro ostatní pravidla
    void enter.........(CSSParser........Context ctx);
    void exit.........(CSSParser........Context ctx);
}
```

Následujcí obrázek demonstruje jak funguje volání mezi jednotlivými metodami:

Obrázek 7.2: Ilustrace volání metod třídy listener při průchodu walkeru. Převzato z [20]

Z obrázku je patrné, že jednotlivé metody jsou volány nezávisle na sobě a pokud potřebujeme, aby se předávaly výsledky zpracování mezi jednotlivými metodami, tak toto řešení není příliš vhodné.

Toto rozhraní bylo implementováno třídou `CSSParserListenerImpl`. Chování třídy bylo obdobné tak jako je tomu u třídy `CSSParserVisitorImp`. Jelikož ale v tomto návrhovém vzoru v jednotlivých metodách není dostupná informace, které metody byly volány před a které budou následovat, tak zpracování výsledků analýzy bylo značně komplikované – musely být vytvořeny pomoconé proměnné, do kterých se uchvovávaly mezivýsledky aktuálně zpracovávaných informací a na základě hodnot těchto pomocných proměnných bylo rozhodováno o zpracování dalších dat. Velkým problémem byla v tomto případě rekurze některých pravidel, protože musely být do pomocných proměnných vytvořeny zásobníky, což vedlo k velké nepřehlednosti při ladění programu.

7.5 Zotavování z chyb ve vstupních CSS datech

Tato podkapitola popisuje problematiku zotavování z chyb v datech, která jsou analyzátoru předložena a následně analyzátorem zpracována. Kapitola úzce souvisí s podka-

pitolou 7.3.1, jelikož pro zotavování chyb slouží právě třída `CSSErrorStrategy`, ve které jsou definovány metody volané při vzniku chyby syntaktické analýzy. Vzniklé chyby při analýze jsou generátorem ANTLR v.4 propagovány v programu v podobě vyjímek typu `RecognitionException`. Tyto vyjímky jsou zachytávány a zpracovávány ve vygenerované třídě `CSSParser`. Pro předefinování chování je tedy nutné zpracování a zachytávání těchto vyjímek předefinovat v gramatice pro parser – v tomto případě tedy gramatice `CSSParser` v souboru `CSSParser.g4`. Následující příklad demonstruje definici pravidla pro kombinovaný selektor včetně zachycení potencionální vyjímky a reagování na ni vypsáním chyby na odpovídající chybový výstup.

```
combined_selector
    : selector ((combinator) selector)*
    ;
    catch [RecognitionException re] {
        log.error("Recognition exception | combined_selector");
    }
```

Takovýmto způsobem lze v průběhu syntaktické analýzy reagovat na chyby, které vznikou kvůli tomu, že zpracovávaná data neodpovídají syntaktickým pravidlům. Druhým způsobem, kterým se aplikace zotavuje z chyb je to, že jsou v gramatice u jednotlivých pravidel zavedeny alternativy zahrnující neplatné hodnoty a následně jsou tyto alternativy při zpracování výsledků analýzy detekovány a pravidlo je vyhodnoceno jako chybné. Na následujícím příkladu je uvedno pravidlo pro `media_rule`, které má být správně pouze jako pravidlo `ruleset`, ale je definována i alternativa `atstatement`, která vede k invalidaci celého pravidla.

Definice alternativní cesty `atstatement`, která vede k invalidaci pravidla v gramatice:

```
media_rule
    : ruleset
    | atstatement //invalid statement
    ;
    catch [RecognitionException re] {
        log.error("Recognition exception | media_rule");
    }
```

Příklad následného zpracování chyby při zpracovávání výsledků analýzy ve třídě `CSSParserVisitorImpl`(zjednodušeně):

```
public RuleBlock<?> visitMedia_rule(CSSParser.Media_ruleContext ctx){
    if(ctx.atstatement() != null){
        return null;// media rule is invalid
    }
}
```

Při dalším vývoji aplikace by bylo vhodné zpracovávání a zotavení z chyb při analýze sjednotit. Z aktuálně používaných variant je určitě lepší zvolit variantu zachytávání vyjímek. Varianta s výčtem alternativ pravidel vedoucích k zachycení chyb může vést k opomenutí některých případů, které se mohou ve zdrojových datech vyskytnout, a navíc výskyt těchto pravidel v gramatice zvyšuje nečitelnost gramatiky, protože v gramatice se vyskytují pravidla, která nejsou definována v syntaxi jazyka CSS. Soubor s gramatikou tak obsahuje zbytečná pravidla, která by v gramatice nemusela vůbec existovat.

Kapitola 8

Testování

V této kapitole je popsán způsob testování správnosti implementace projektu jStyleParser s využitím nového generátoru ANTLR 4. První podkapitola se zabývá manuálními testy a druhá testy automatizovanými. Poslední podkapitola se zabývá vyhodnocením výsledků testů.

8.1 Manuální testování

Celá aplikace byla v průběhu implementování průběžně manuálně testována. Při manuálním testování byly také průběžně pouštěny automatické testy a výsledky těchto testů byly porovnávány s výsledky testů projektu s generátorem ANTLR v3. Manuální testování bylo prováděny nejvíce v případech dokončování implementace zpracování nového pravidla. Po implementování zpracování nového pravidla byly testy vedeny následujícím způsobem:

1. Spuštění automatických testů
2. Kontrola zda prochází všechny automatizované testy, které fungovaly do této implementace. Pokud některý test nefungoval, tak byla vyhledána příčina problému a tento problém opraven, poté návrat k 1
3. Detekce testů, které by měly být po implementaci aktuálního pravidla úspěšné
4. Pokud některý test nebyl úspěšný, byla vyhledána příčina problému a tento problém opraven, poté návrat k 1
5. Výsledky testů specifické pro nově implementované pravidlo byly manuálně porovnány s výsledky testů totožných pravidel v původní verzi
6. Pokud nebyly výsledky testů shodné, byla vyhledána příčina problému a tento problém opraven, poté návrat k 1

8.2 Automatizované testování pomocí JUnit

Jelikož je manuální testování takto rozsáhlého projektu nedostačující, tak projekt jStyleParser obsahuje ve svém repozitář ve složce `src/test/` sadu cca 120 automatizovaných testů. Tyto testy se snaží pokrýt všechny možné situace, které mohou při parsování a zpracování CSS nastat a mohou být strojově zkontrolovány. Pro testování je využit framework `JUnit`,

který právě slouží k psaní automatizovaných, opakovatelných testů[11]. Jednotlivé testy jsou seskupeny do tříd. Tyto třídy sdružují testy, které se zaměřují na testování podobných oblastí jazyka CSS, např. testování zotavení se z chyb při syntaktické analýze. V následujícím seznamu jsou vypsány všechny třídy testů, které jsou v projektu jStyleParser obsaženy, společně s krátkým popiskem testované oblasti.

- AdvancedCSSTest – obsahuje sadu testů, které zpracovávají celý HTML dokument, z něhož je poté analyzován vložený CSS soubor a následně prováděna kontrola přiřazení CSS pravidel jednotlivým elementům
- AnalyzerTest – tyto testy slouží k otestování funkčnosti třídy Analyzer, která slouží k analýze výsledků CSS Parseru
- CollectionSpeedTest – test slouží k otestování rychlosti kolekcí využitých v projektu – nepoužívá se
- DecoderTest – testuje zpracování komplexních vlastností – např. vlastnost `background`
- DOMAssignDirectTest – otestování správnosti přiřazování CSS vlastností jednotlivým elementům DOM. Do této sady testů byl přidán test `linkedStyleTest`, který kontroluje funkčnost zpracovávání CSS stylů vložených do HTML souborů pomocí elementu `link`.
- DOMAssignMediaTest – otestování správnosti přiřazování CSS vlastností jednotlivým elementům DOM pro konkrétní media query
- DomAssignTest – otestování správnosti přiřazování CSS vlastností jednotlivým elementům DOM a kontrola správnosti priority přepisování vlastností
- ElementMatcherTest – testování správnosti přiřazování CSS vlastností jednotlivým elementům DOM – knotrola správnosti vzhledem ke zpracování velkých a malých písmen v pravidlech
- ErrorTest – kontrola chování při zadání špatných pravidel – nefunguje správně
- EscapingTest – kontrola správného zpracování escape sekvencí
- FontFace(přidáno v průběhu implementace nové verze) – slouží pro kontrolu správného zpracování praivdel definujících styl písma
- FunctionsTest – testování správnosti zpracování funkcí v hodnotách deklarací
- GrammarRecovery1Test, GrammarRecovery2Test, GrammarRecovery3Test – otestování správného zotavení z chyb.
- ImportTest1 – testuje správnost vkládání a zpracovávání vnořených CSS dokumentů
- MediaTest – kontrola správnosti zpracování pravidel pro konkrétní media query
- NodeDataVariantTest – test pro datové typy uzlů – nepoužívá se
- PsudoClassTest – testování správnosti zpracování pseudo tříd
- SelectorTest – kontrola správnosti zpracování selektorů

- SimpleTest – obsahuje sadu základních testů – např. zpracování hodnoty barvy
- UAConformancyTest – kontrola správného zotavení z chyb popsaných v pravidlech pro zpracování chyb ve specifikaci CSS [13]
- ViewPortTest – (přidáno v průběhu implementace nové verze) – kontrola zpracování pravidla pro viewport

8.2.1 Pomocné třídy

V automatických testech jsou využity pomocné třídy nacházející se ve stejném adresáři jako samotné testy. Tyto třídy slouží především k manuálnímu vytváření CSS pravidel v definovaných datových typech tak, aby se takto manuálně vytvořené hodnoty daly porovnat s hodnotami získanými ze syntaktického analyzátoru.

- `DeclarationsUtil` – třída slouží pro tvorbu a opareace nad CSS deklaracemi
- `DOMSource` – třída je abstrakcí parseru, který je schopný získat DOM HTML dokumentu
- `ElementMap` – slouží pro mapování elementů podle identifikátorů, nebo názvů
- `SelectorsUtil` – slouží pro tvorbu a operace nad CSS selektory

8.2.2 Rozšíření testů

V rámci testování aplikace byly testy rozšířeny o případy, kterými se stávající testy nezabývaly dostatečně, nebo vůbec.

- `@font-face` – pravidlo @font-face, které definuje vzhled písma nebylo v původním projektu vůbec testováno, takže byla do projektu přidána třída `FontFaceTest`, která provádí základní kontrolu implementace tohoto pravidla
- `@viewport` – toto pravidlo nebylo v původní sadě testů také zahrnuto, proto byla přidána kontrola tohoto pravidla ve třídě `ViewPortTest`
- `GrammarRecovery1Test` – do sady byly přidány 4 testy na otestování správného zotavení při špatné hodnotě výrazu v deklaraci pravidla (hodnota je obalena pomocí `[]`, hodnota je obalena pomocí `()`, hodnota je obalena pomocí `{}`, hodnota je klíčové slovo např. `@media`)
- `GrammarRecovery3Test` – přidáno pravidlo pro zotavení při nalezení výrazu `expression`, který není v nové verzi podporován

8.3 Výsledky testování

Jak již bylo výše zmíněno, tak celá aplikace byla několikrát testována v průběhu vývoje a následně po dokončení implementace. Testování probíhalo jak manuálně, tak pomocí automatizovaných testů napsaných s využitím frameweorku JUnit. Všechny manuální testy, pomocí kterých byla testována nová verze, byly také zkoušeny na původní aplikaci a výsledky těchto testů byly porovnávány a vyhodnocovány tak, aby byla v nové verzi nástroje

zachována stejná funkčnost. Automatizované testy byly v průběhu tvorby aplikace spouštěny tak, aby byl co nejvíce využit jejich potenciál v rámci tvorby jednotlivých částí aplikace – především při implementaci zpracování jednotlivých pravidel gramatiky.

Po dokončení implementace byly všechny automatizované testy úspěšné a chování aplikace nebylo odlišné od předchozí verze. Při manuální kontrole také nebyly objeveny žádné neočekávané výsledky zpracování vstupních CSS dat.

Kontrola úspěšnosti kompilace celé aplikace pomocí Maven pluginu a výsledky automatických testů jsou dostupné ve službě Travis CI na adrese `https://travis-ci.org/sedlakr/jStyleParser`. Tato služba provádí při každé změně repozitáře automaticky kompilaci a spuštění automatizovaných testů nad celou aplikací.

Kapitola 9

Závěr

Cílem této práce bylo popsat problematiku syntaktických analyzátorů a jejich generování na základě dodaného předpisu (gramatiky), dále prostudovat již existující generátory na platformě Java, poté se seznámit s projektem jStyleParser a navrhnout, implementovat a otestovat obdobný analyzátor s využitím generátoru ANTLR verze 4.

Problematika syntaktických analyzátorů a jejich generování na základě dodaného předpisu (gramatiky) byla popsána v kapitole 2. Dále byl čtenář v kapitole 3 seznámen s již existujícími generátory na platformě Java, především s generátorem ANTLR ve verzi 3 a 4. Po tomto seznámení následuje kapitola, která čtenáři představí jazyk CSS a jeho základní prvky a vlastnosti.

Následně je v práci popsán aktuální stav projektu jStyleParser, nástroje, které využívá, struktura celého programu a význam jednotlivých částí. Při popisu projektu jsou v této kapitole popsány detailně jednotlivé gramatiky, ze kterých je následně celý syntaktický analyzátor generován.

Za představením projektu následuje kapitola popisující návrh úprav nového analyzátoru, pomocí kterých bude dosažen přechod ze zastaralé knihovny ANTLR verze 3 na aktuální verzi 4. Po tomto návrhu je v kapitole 7 popsána implementace nového analyzátoru projektu jStyleParser, který využívá novou verzi generátoru ANTLR – 4.5.3. Za kapitolou popisující implementaci navazuje kapitola 8 – Testování. V této kapitole jsou popsány způsoby testování funkčnosti nově vzniklé verze projektu jStyleParser a vyhodnoceny výsledky testů.

Jak již bylo zmíněno, výsledek této práce je nová verze projektu jStyleParser, která pro syntaktickou analýzu CSS a zpracování výsledků analýzy využívá nejnovější verzi generátoru ANTLR, konkrétně 4.5.3. Celá aplikace byla v průběhu vývoje testována jak manuálními testy, tak testy automatizovanými. Všechny prováděné testy byly vyhodnoceny úspěšně – viz 8.3.

Jelikož projekt jStyleParser aktuálně pracuje převážně s CSS ve verzi 2.1 a CSS verze 3 je podporována jen částečně, tak by bylo do budoucna vhodné aplikaci dále rozšířit především ve směru přidávání podpory pro nová pravidla z CSS 3. Dále by bylo vhodné v aplikaci změnit způsob zpracování chyb ve vstupních CSS datech s využitím potenciálu nové verze generátoru ANTLR. Aktuální stav a možné vylepšení jsou popsány v podkapitole 7.5.

Literatura

[1] About Dynamic Properties (Internet Explorer). [Online], [cit. 2016-05-10].
URL https://msdn.microsoft.com/en-us/library/ms537634(v=vs.85).aspx

[2] ANTLR. [Online], [cit. 2016-01-04].
URL http://www.antlr.org/

[3] ANTLR verze 4.0. [Online], [cit. 2016-01-04].
URL https://github.com/antlr/antlr4/tree/4.0

[4] Beaver - a LALR Parser Generator. [Online], [cit. 2016-01-04].
URL http://grammatica.percederberg.net/

[5] BYACC/J. [Online], [cit. 2016-01-04].
URL http://byaccj.sourceforge.net/

[6] CSS Syntax Module Level 3. [Online], [cit. 2016-01-05].
URL http://www.w3.org/TR/css-syntax-3/

[7] CSS Tutorial. [Online], [cit. 2016-01-05].
URL http://www.w3schools.com/css/

[8] CUP. [Online], [cit. 2016-01-04].
URL http://www2.cs.tum.edu/projects/cup/

[9] Grammatica :: Parser Generator. [Online], [cit. 2016-01-04].
URL http://beaver.sourceforge.net/

[10] JFlex - JFlex The Fast Scanner Generator for Java. [Online], [cit. 2016-01-04].
URL http://jflex.de/

[11] JUnit. [Online], [cit. 2016-04-27].
URL http://www.junit.org/

[12] SableCC. [Online], [cit. 2016-01-04].
URL http://www.sablecc.org

[13] Syntax and basic data types. [Online], [cit. 2016-04-28].
URL https://www.w3.org/TR/CSS21/syndata.html#parsing-errors

[14] Alexander MEDUNA, R. L.: Opora předmětu IFJ. [Online], [Verze 1.2006+rev. 2009-2015], [cit. 2016-01-05].
URL https://wis.fit.vutbr.cz/FIT/st/course-files-st.php/course/IFJ-IT/texts/OporaIFJ.pdf

[15] Alexander MEDUNA, R. L.: Opora předmětu VYPe. [Online], [Verze 1.2006], [cit. 2016-01-05].
URL https://wis.fit.vutbr.cz/FIT/st/course-files-st.php/course/VYP-IT/texts/OporaVYP.pdf

[16] Burget, R.: CSSBox - Java HTML rendering engine. [Online], [cit. 2016-01-04].
URL http://cssbox.sourceforge.net/

[17] Burget, R.: jStyleParser - Java CSS parser. [Online], [cit. 2016-01-04].
URL http://cssbox.sourceforge.net/jstyleparser/

[18] Burget, R.: Knihovna jStyleParser. [Online], [cit. 2016-01-04].
URL http://www.fit.vutbr.cz/~burgetr/prods.php?id=63

[19] Parr, T.: *The Definitive ANTLR Reference: Building Domain-Specific Languages (Pragmatic Programmers)*. Pragmatic Bookshelf, 2007, ISBN 0978739256.

[20] Parr, T.: *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2013, ISBN 1934356999.