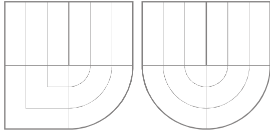
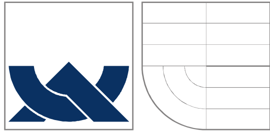
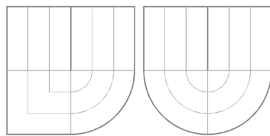
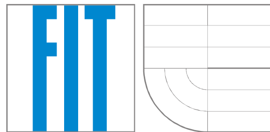



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

ZPRACOVÁNÍ PŘEDPISŮ CSS V JAZYCE JAVA

CSS OBJECT MODEL IN JAVA

DIPLOMOVÁ PRÁCE

AUTOR PRÁCE — Bc. Jan Švercl
AUTHOR

VEDOUCÍ PRÁCE — Ing. Radek Burget, Ph.D.
SUPERVISOR

BRNO 2008

Abstrakt

Tato práce se věnuje problematice manipulace s předpisy kaskádových stylů. Prvním cílem je vytvoření knihovny pro manipulaci s předpisy CSS – je navrhnuto objektové rozhraní předpisu CSS, pomocí nástroje JavaCC vygenerován syntaktický analyzátor, doplněna implementace rozhraní a vše spojeno do funkčního celku. Knihovna umožňuje načtení předpisu CSS a převedení do objektové reprezentace, editaci či následně export zpět do textového souboru.

Druhá část práce se zabývá implementací knihovny pro ohodnocení stromu dokumentu – každý (X)HTML dokument je tvořen stromem elementů, ke kterým jsou následně v předpisu CSS vyhledávána pravidla a ve správném pořadí přiřazovány jejich deklarace.

V závěru práce je popsána DEMO aplikace, která demonstruje možnosti obou knihoven a umožňuje provádění experimentů či testování. Ukázána je také technika profiling, umožňující vyhledat ve zdrojovém kódu výkonnostně náročné pasáže, které mohou být dále optimalizovány.

Klíčová slova

Java, CSS, předpis CSS, stylopis, kaskádový styl, gramatika, ohodnocení stromu dokumentu, JavaCC

Abstract

This thesis concerns itself with the problems of manipulation with Cascading Style Sheets. The first aim is to create a library for manipulation with Cascading Style Sheets – an object interface of Cascading Style Sheet is proposed, a parser is generated by means of an appliance of JavaCC, the implementation of interface is completed and as a whole connected to a functional unit. The library enables reading Cascading Style Sheet and its transfer into an object representation, editing and subsequently exporting back to the text file.

The second part of the thesis deals with the implementation of library for assign property values – every (X)HTML document is formed by the tree of elements to which the rules are consequently searched for in CSS and their declarations are asigned in the correct order.

In conclusion of the thesis the DEMO application, which illustrates the possibilities of both libraries and facilitates performing of experiments or testing, is described. Hereafter the technique of profiling, which enables to find out the efficiently demanding passages, which can be optimised further, in the source code, is shown.

Keywords

Java, CSS, Cascading Style Sheet, parser, grammar, assigning property values, JavaCC

Citace

ŠVERCL, J. *Zpracování předpisů CSS v jazyce Java*. Brno, 2008, diplomová práce, FIT VUT v Brně.

Zpracování předpisů CSS v jazyce Java

Prohlášení

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

……………………
Bc. Jan Švercl
1.5.2008

Poděkování

Na tomto místě bych chtěl poděkovat vedoucímu mé diplomové práce, Ing. Radku Burgetovi, Ph.D., za zájem, připomínky a čas, který mé práci věnoval.

© Bc. Jan Švercl, 2008.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

1 Úvod....4

2 Struktura předpisů CSS....6

2.1 Předpis CSS....6

2.2 Media....7

2.3 Pravidla....8

2.4 Selektory....8

2.5 Vlastnosti....10

2.6 Hodnoty....11

3 Ohodnocení stromu dokumentu předpisem CSS....13

3.1 Postup....13

3.2 Deklarace s vyšší prioritou - !IMPORTANT....14

3.3 Výpočet přesnosti selektorů....15

3.4 Přiřazené hodnoty....15

4 Definice objektového rozhraní....16

5 Vytvoření syntaktického analyzátoru – parseru....22

5.1 JavaCC....22

5.2 Zápis gramatiky pomocí EBNF....23

5.3 Vytvoření gramatiky pro nástroj JavaCC....24

5.4 Rozbor gramatiky pro JavaCC....24

5.5 Generování parseru....25

5.6 Příklad výstupu parseru....25

6 Knihovna pro práci s předpisy CSS - implementace....28

6.1 Načtení z externího zdroje....28

6.2 Editace....30

6.3 Export (dump)....31

6.4 Validace....31

6.5 Testování....31

6.6 Použití v projektu....32

6.7 Nedostatky a možnosti rozšíření....32

7 Implementace algoritmu pro ohodnocení stromu dokumentu....34

7.1 Vstupní data....34

7.2 Návrh struktury....34

7.3 Výběr pravidel pro daný element....37

7.4 Optimalizace – zrychlení výběru pravidel pro element....37

7.5 Popis funkce třídy NodeData – aplikace deklarace na element.....39
7.6 Výstup.....40
7.7 Omezení.....41
7.8 Možnosti rozšíření.....42
8 Ladění výkonu (profiling).....43
8.1 Ukázka optimalizace části kódu.....43
9 DEMO aplikace.....47
9.1 Návrh struktury.....47
9.2 JTidy.....47
9.3 Kompilace.....48
9.4 Ovládání.....48
9.5 Testovací data.....49
10 Závěr.....50
10.1 Vlastní přínos.....50
Literatura.....52
Seznam diagramů.....52

Seznam použitých zkratek

API – Application Programming Interface, aplikační rozhraní

CSS – Cascading Style Sheet, předpis kaskádových stylů

EBNF – Extended Backus-Naur Form, rozvinutá Backus-Naurova forma

HTML – HyperText Markup Language, značkovací jazyk pro hypertext

SAC – Simple API for CSS, Objektové rozhraní prezentované organizací W3C

URI – Uniform Resource Identifier, jednotný identifikátor zdroje

XHTML – eXtensible HyperText Markup Language, rozšířená verze HTML

1 Úvod

Předpisy CSS slouží k definici vzhledu elektronických dokumentů, v dnešní době převážně webových stránek v jazyce (X)HTML. Hlavním cílem při vzniku bylo oddělit prezentační vlastnosti od samotného obsahu dokumentu. Zvýšila se tak dostupnost a flexibilita, naopak snížila se velikost zdrojových kódů a přirozeně i jejich složitost. Oddělená definice může být pak sdílena i více dokumenty a umožňuje tak rychlé a snadné změny vzhledu i v rozsáhlých webových prezentacích.

Tato práce si klade za cíl vytvořit dva nástroje usnadňující práci s předpisy CSS v jazyce Java. Prvním z nich bude knihovna pro manipulaci se samotnými předpisy CSS – tedy jejich načtení a převedení do objektové reprezentace, editace či export zpět do textového souboru. Druhým nástrojem bude algoritmus zajišťující ohodnocení stromu dokumentu předpisem CSS – každý (X)HTML dokument je možné reprezentovat pomocí stromu elementů, ke kterým jsou přiřazovány jednotlivé deklarace ve správném pořadí. Na závěr bude vytvořena jednoduchá demonstrační aplikace, která bude oba nástroje obsahovat a umožní testovat funkčnost či provádět experimenty.

První kapitoly práce obsahují přirozeně teoretický úvod do dané problematiky. Na úplném začátku jsou popsány předpisy CSS, převážně z pohledu struktury jejich zápisu. Výklad se postupně věnuje všem stavebním prvkům předpisu CSS – od úplného vrcholu hierarchie tvořeného předpisem CSS až po pravidla, deklarace, vlastnosti a hodnoty. Druhou teoretickou kapitolou je popis ohodnocení stromu dokumentu – proces, při kterém jsou jednotlivé deklarace přiřazovány k elementům, kterým předávají informaci o změně některých prezentačních vlastností.

Prvním praktickým úkolem, diskutovaným v třetí kapitole, je vytvoření objektového rozhraní, které přesně kopíruje hierarchickou strukturu předpisu CSS. Objekty tříd, implementující toto rozhraní, pak umožní nést předpis CSS, pracovat s ním a případně provést export zpět do jeho textové reprezentace.

Jelikož největší část knihovny pro práci s předpisy CSS tvoří algoritmy pro načtení externího souboru, je ve čtvrté kapitole rozebrán postup vytvoření syntaktického analyzátoru (parseru). Ten je automaticky vygenerován nástrojem JavaCC z gramatiky jazyka a umožňuje sestavení syntaktického stromu ve formě Java objektů.

Následující dvě kapitoly se již věnují popisu konkrétní implementace obou knihoven. V textu jsou úmyslně vynechány nepodstatné věci a hlavním cílem je vysvětlit zajímavé části zdrojového kódu. Výklad je doplněn ilustračními diagramy a obrázky, které velkou měrou napomáhají v pochopení dané problematiky.

Jakmile byla dokončena implementace knihoven, bylo zapotřebí provést revizi a pokusit se maximalizovat jejich výkon. Sedmá kapitola se tedy zabývá technikou „profiling", určenou

k monitorování využití systémových prostředků a usnadňující optimalizaci. Na jednoduchém příkladu je zde představen problém, provedeno jeho vyřešení a demonstrovány dosažené výsledky.

Popis DEMO aplikace uzavírá výkladovou část této práce. Je zde ukázána převážně kompilace programu, jeho ovládání a popis experimentálních vstupních dat.

Závěr pak patří přirozeně zhodnocení celé práce z pohledu autora. Jsou zde popsány poznatky ze sběru teoretických informací, problémy při implementaci, zasazení projektu do širšího kontextu i dosažené výsledky. Nedílnou součástí závěru je také vlastní přínos autora a vize do budoucna.

Diplomová práce úzce navazuje na předchozí stejnojmenný semestrální projekt. V semestrálním projektu byla popsána z teoretického hlediska struktura předpisů CSS a postup při ohodnocení stromu dokumentu. Bylo také navrženo objektové rozhraní, které je dominantní součástí knihovny pro práci s předpisy CSS. Z těchto poznatků bylo vycházeno a slouží jako základ pro implementaci dalších, již zmíněných, částí.

Následující text, jakožto obsah diplomové práce, není určen pro četbu úplnými laiky. Od čtenáře očekává alespoň základní znalost problematiky a názvosloví CSS. Popisování základních informací by zbytečně prodlužovalo rozsah práce a nepřinášelo odborníkům na akademické úrovni žádné nové informace.

2 Struktura předpisů CSS

Tato kapitola si pokládá za cíl popsat strukturu předpisů CSS. Výklad bude proveden převážně pomocí přirozeného jazyka, pro získání kompletní a přesné specifikace je nutné navštívit web organizace W3C, kde jsou tyto dokumenty k dispozici.

Předpisy CSS mohou být zapisovány pomocí tří způsobů – v externím souboru, přímo v těle dokumentu či jednoduchým zápisem deklarací přímo v elementu. Následující text popisuje první možnost, tedy strukturu zápisu v externím souboru.

2.1 Předpis CSS

Mluvíme-li o struktuře předpisů CSS, je přirozené očekávat na úplném vrcholu hierarchie právě předpis CSS. Uvnitř předpisu CSS se pak již nacházejí jednotlivá pravidla, z nichž některé mají speciální význam, ostatní slouží přímo k definici vzhledu dokumentu.

Začíná-li pravidlo znakem '@', má určitý speciální význam a nazývá se jako tzv. At-pravidlo. At-pravidel je několik, jejich význam je pak následující:

- **@charset** – slouží k definici použité znakové sady. Je důležité umístit toto pravidlo na úplném začátku předpisu CSS, v opačném případě by mohlo dojít k zahájení načítání v jiném kódování. Jakákoli pravidla @charset jinde než na začátku dokumentu musí být ignorována. V praxi se často vynechává, jelikož běžné předpisy jsou tvořeny pouze prvními 128-mi znaky ASCII, které jsou pro všechna kódování stejná. Pokud pravidlo znakové sady chybí, měl by se jí snažit prohlížeč zjistit nějakým jiným způsobem, typicky například může být zadáno při volání `<link charset="">`, lze použít znakovou sadu odkazujícího dokumentu či v poslední řadě zvolit implicitní hodnotu UTF-8.
  zápis: pravidlo je uvozeno identifikátorem @charset a následuje bez mezery název znakové sady v jednoduchých, respektive dvojitých, uvozovkách, zakončený středníkem. Příklad:

  ```
  @charset"UTF-8";
  ```

  Jako název znakové sady lze použít jakýkoli alias definovaný organizací IANA [4].

- **@import** - umožňuje vkládání jiných předpisů a vytváření hierarchických struktur z více částí. Pravidlo @import musí být vloženo ihned za pravidlem @charset, případně být na začátku dokumentu (pokud @charset chybí) – není-li na tomto místě, musí být prohlížečem ignorováno. Při vkládání je také možné definovat určitá media, pro která má být vkládaný předpis použit.

zápis: pravidlo je uvozeno identifikátorem @import, po mezeře následuje odkaz na vkládaný předpis CSS a volitelný seznam médií oddělených čárkou. Odkaz může být zapsán buď jako řetězec uvozený v uvozovkách nebo pomocí funkční notace url(). Pravidlo je zakončeno středníkem. Příklad:

```
@import "styl.css";
@import url(styl.css) screen, tv;
```

- **@media** – umožňuje označit některá pravidla tak, aby se použila pouze při výstupu na konkrétní výstupní zařízení.
  zápis: pravidlo je uvozeno identifikátorem @media, po mezeře následuje název jednoho nebo několika médií oddělených čárkou. Pravidla jsou pak uzavřena ve složených závorkách. Příklad:

```
@media screen, tv {
  A { color: #FFF; }
}
```

- **@page** – umožňuje definovat některé typické vlastnosti pro zobrazení dokumentů na stránkovém výstupním zařízení (typicky tiskárna či slidy projektoru). Lze tak definovat například okraje, rozměry stránek či orientaci papíru.
  zápis: pravidlo je uvozeno identifikátorem @page, který může být volitelně doplněn selektorem pseudostrany. Následuje seznam deklarací uzavřený ve složených závorkách. Příklad:

```
@page:first {
  margin-top: 10cm;
}
```

Pořadí jednotlivých pravidel v předpisu CSS je důležité a nemůže být bez úmyslu autora zaměněno – ovlivnilo by to následující proces ohodnocování stromu dokumentu.

2.2 Media

Definicí pravidla @media je možné předepsat různé vlastnosti pro dokument v závislosti na tom, na jakém výstupním zařízení bude prezentován. Typické použití je vypsání společných pravidel pro všechna media přímo do předpisu CSS a následně doplnění odlišných pravidel dovnitř pravidel @media. Definice CSS definuje několik druhů médií jejichž identifikátory odpovídají názvu výstupního zařízení.

- **all** – deklarace jsou určena pro všechna výstupní zařízení
- **screen** – obrazovka běžného počítače
- **tv** – obrazovka televizoru
- **print** – tiskárna
- **projection** – prezentace na projektoru či tisk na průsvitky
- **aural** – předčítání dokumentu hlasovou čtečkou
- **braille** – vypsání obsahu na braillův řádek
- **embossed** – tisk na speciální tiskárně braillova písma
- **handheld** – kapesní počítač, typicky s malou obrazovkou
- **tty** – terminálové okno, umožňující vypisovat pouze znaky s pevnou šířkou.

2.3 Pravidla

Pravidla jsou základním stavebním kamenem předpisů CSS a nesou v sobě informace o tom, jak bude dokument prezentován. Začátek každého pravidla obsahuje jeden nebo několik selektorů, určujících na jaké elementy stromu dokumentu bude dané pravidlo aplikováno. Následuje jedna nebo několik deklarací, uzavřených ve složených závorkách a oddělených středníky. Deklarace se pak skládá z vlastnosti a jedné nebo několika hodnot. Příklad:

```
A {
  color: #FFF;
}
```

Z pohledu syntaxe mohou být pravidla umístěna buď přímo v předpisu CSS, nebo uvnitř at-pravidla @media.

2.4 Selektory

Selektory jsou vzory, které odpovídají určité skupině elementů ve stromu dokumentu. Lze pomocí nich popsat elementy, kterým je třeba předat deklarace uvnitř pravidla.

Selektory lze rozdělit do několika skupin podle toho, jakým způsobem vybírají odpovídající elementy. Níže je uveden jejich seznam spolu s praktickými ukázkami použití.

Univerzální selektor

Jedná se o základní selektor, který odpovídá všem elementům ve stromu dokumentu. Lze tak například pomocí `* {color: red;}` změnit barvu všech elementů na červenou. Velice často je tento selektor vynecháván, například zápis `*.msg` lze napsat i jednodušeji jako `.msg`.

Typový selektor

Vybírá elementy podle typu. Zápis je jednoduchý, selektor má stejný název jako typ vybíraného elementu. Například tento zápis `img {float: left;}` vybere všechny obrázky (elementy img) a nastaví jim vlastnost float.

Selektor pseudo-elementu

Umožňuje odkazovat se na elementy, které nejsou přímo v zápisu dokumentu, nicméně je možné je nějakým způsobem automaticky generovat. Například `p:first-line {color: #FF0000}` se odkazuje na pseudo-element first-line (první řádek) odstavce a nastavuje mu červenou barvu písma.

Selektor třídy

Umožňuje vybrat elementy, seskupené do určité třídy. Například `A.msg {color: red;}` změní barvu všem odkazům, které jsou ve třídě msg.

Selektor pseudo-třídy

Funguje velice podobně jako selektor třídy, nicméně využívá takzvaných pseudo-tříd – speciální třídy, jež jsou generovány typicky prohlížečem na základě nějakého stavu dokumentu. Zápis `A:hover {color: red;}` nastaví modrou barvu písma všem odkazům, nad kterými je aktuálně ukazatel myši.

Selektor ID

Vybírá element, jež odpovídá svým ID. Jelikož v každém dokumentu musí být všechna ID unikátní, odpovídá buď jednomu, nebo žádnému elementu. Například `#top {color: red;}` vybere element s ID = top a nastaví mu červenou barvu písma.

Selektor atributu

Umožňuje vybírat na základě hodnoty některého atributu elementu. Zápis je možná rozdělit do čtyř skupin:

- **atribut je definován** – odpovídá v případě, že daný atribut je v elementu definován, zápis `[atribut]`
- **atribut se rovná** – odpovídá elementu v případě že daný atribut je roven hodnotě, zápis `[atribut=hodn]`
- **atribut obsahuje** – v případě `[atribut~=hodn]` odpovídá, pokud hodnota atributu je seznam slov oddělených mezerou, kdy jedno z těchto slov je rovno `hodn`.
- **atribut začíná** – je-li dán selektor `[atribut|=hodn]`, jsou vybrány ty elementy, jejichž hodnota atributu se skládá ze slov oddělených svislou čárou a první z nich je rovno `hodn`.

Selektor následníka

Skládá se ze dvou jednoduchých selektorů oddělených mezerou, obecně A B. Vybírá všechny elementy B, které jsou uvnitř elementu A. Lze takto provádět tedy výběr nejen na základě vlastností elementu, ale i jeho kontextu. Je-li třeba vybrat všechny elementy typu SPAN uvnitř elementů DIV, použije se zápis `DIV SPAN {color: red;}`.

Selektor dítěte

Je velice podobný selektoru následníka pouze s tím rozdílem, že element A musí být přímým potomkem elementu B. Zápis `SPAN>A {color: red;}` pak vybírá všechny odkazy, které jsou přímo uvnitř elementu SPAN.

Selektor sourozenců

Umožňuje vybrat elementy mající stejného otce a ve stromu dokumentu za sebou bezprostředně následují. Je-li dán tento dokument:

```
<span>
  <A href="1">odkaz1</A>
  <A href="2">odkaz2</A>
</span>
```

pak zápis `A+A {color: red;}` změní barvu odkazu 2.

Z popisu selektorů vyplývá, že existuje několik jednoduchých selektorů a tři spojky <mezera>, „>“ a „+“, které je umožňují spojovat a vytvářet tak libovolně složité selektory závislé na kontextu dokumentu.

2.5 Vlastnosti

Vlastnosti určují, co se bude v odpovídajících elementech měnit. Jejich kompletní seznam je dostupný na webu organizace W3C (www.w3c.org), kde je u každé definována i množina možných hodnot[1]. Seznam vlastností se velice dynamicky mění postupně s vývojem předpisů CSS a u každé nové verze v něm lze očekávat změny.

1 Seznam vlastností pro CSS 2.1 dostupný online na http://www.w3.org/TR/CSS21/propidx.html (Prosinec 2007)

2.6 Hodnoty

Hodnoty lze rozdělit do několika skupin se společnými vlastnostmi:

Číselné hodnoty

Do této skupiny patří všechna čísla ať už celá nebo desetinná, doplněná volitelně znaménkem. Za číslem může dále následovat jednotka, která doplňuje jeho význam. Pomocí číselných hodnot se nejčastěji definují rozměry objektů dokumentu, lze ovšem zadávat i úhly, čas či frekvenci. V současné době patří mezi podporované jednotky tyto:

EM, EX, PX, CM, MM, IN, PT, PC, DEG, RAD, GRAD, MS, S, HZ, KHZ

Procenta

Každá vlastnost, která může obsahovat procenta, má jasně definováno, ke které jiné hodnotě je tato vztažena. Ta může být převzata z jiné vlastnosti nebo například z rodičovského elementu.

Barvy

Barvy jsou jednou z nejpoužívanějších hodnot v předpisech CSS. Lze jí definovat několika způsoby, kdy výsledkem je jedna a ta samá barva a záleží pouze na autorovi stránky, který upřednostní.

- **textovou reprezentací** – u základních barev lze použít anglické názvy základních barev (např. red). Ve specifikaci CSS je přesně uvedeno jaké barvy lze takto zadávat a jaké mají daná slovní vyjádření přesnou číselnou reprezentaci složek RGB.
- **hexadecimálním zápisem hodnot** – začíná znakem # (křížek) a následuje 3 nebo 6 hexadecimálních číslic, jež udávají RGB hodnoty barvy. Jsou-li použity pouze 3 hexadecimální číslice, dojde ke zdvojení každé z nich.
- **pomocí funkčního zápisu** – ten má tvar `rgb(255,0,0)` a přirozeně každé z čísel v rozmezí 0-255 určuje jednu ze složek RGB.

všechny zápisy, pokud reprezentují stejnou barvu, mají ekvivalentní význam, tedy:

```
red = #F00 = #FF0000 = rgb(255,0,0)
```

Identifikátory

Jedná se o textové řetězce, které mají definicí předpisů CSS stanovený přesný význam. Jedná se například o „none“, „inherit“, „bottom“, „dashed“, „left“ atd. Význam každého z identifikátorů pak záleží na vlastnosti, kde je použit.

Řetězce

Jednoduché řetězcové konstanty, uzavřené v jednoduchých či dvojitých uvozovkách. Dvojité uvozovky (resp. jednoduché) přirozeně nemohou být součástí takto uvozeného řetězce a proto je třeba použít tzv. escape sekvence.

V řetězcové konstantě nemůže být přímo vložen znak konce řádku, pokud je to třeba, musí být nahrazen escape sekvencí \A. Naopak pokud je třeba text zalomit, například z estetických důvodů, je nutné dát na konec každého řádku znak \ (zpětné lomítko).

URI

Zápis hodnot typu uri se provádí pomocí funkce, která má následující tvar: `url('index.html')` s tím, že uvozovky nejsou povinné. Odkazuje se tak na soubor index.html umístěný ve stejném adresáři jako dokument.

Zdroj: [1]

3 Ohodnocení stromu dokumentu předpisem CSS

Následující kapitola bude pojednávat o problému, kdy je dán určitý dokument ve formě stromu elementů a je třeba jej ohodnotit předpisem CSS. To znamená, že je třeba ke každému z elementů vyhledat odpovídající pravidla a předat deklarace uvnitř nich. V případě, že deklarace obsahují stejné nebo na sobě závislé vlastnosti, je důležité zajistit také jejich správné pořadí.

Výsledkem této operace je pak stejný strom dokumentu jako na vstupu, nicméně obohacený o definici jeho prezentačních vlastností. Takto ohodnocený strom je následně možné předat vykreslovacímu jádru prohlížeče a zobrazit například na obrazovce počítače.

V reálném nasazení se k ohodnocení stromu dokumentu používá několik předpisů CSS:

- **předpis webového prohlížeče** – obsahuje základní nastavení nutné pro prezentování každého dokumentu. Nese například informaci o tom, že nadpis H1 má větší písmo než ostatní text atd. Umožňuje-li prohlížeč zobrazovat stránky „bez stylu“, pak se jedná o zobrazení pomocí tohoto jednoduchého předpisu.
- **předpis uživatele** – dává možnost uživateli přizpůsobit si nějakým způsobem výchozí vzhled stránek. Provádí to buď volbou nějakého externího předpisu CSS či formou nastavení přímo v prohlížeči.
- **předpis autora** – obsahuje největší množství deklarací a ve většině případů poměrně zásadně mění vzhled dokumentu. Podle místa, kde se nachází jej lze rozdělit do 3 skupin:
  - **v externím předpisu** – pravidla jsou uložena v souboru odděleně a ten pak může být použit libovolným počtem dokumentů.
  - **uvnitř dokumentu** – předpis je vložen přímo ve zdrojovém kódu dokumentu, typicky na začátku.
  - **přímo v elementu** – deklarace jsou přímo vepsány v atributu style=““ konkrétního elementu.

3.1 Postup

Postup použitý při ohodnocení stromu dokumentu předpisem CSS je pevně stanovený definicí a skládá se z operací výběrů a řazení v následujících krocích:

- Pro každý element stromu se vyberou všechny deklarace, jejichž selektor odpovídá elementu a mající zvolený typ media.

- Vybrané deklarace se seřadí podle příslušnosti k danému předpisu a své důležitosti v následujícím pořadí. Pod pojmem důležitá deklarace se myslí taková, která obsahuje za hodnotou klíčové slovo !IMPORTANT, viz níže.
    1. Deklarace předpisu webového prohlížeče
    2. Deklarace předpisu uživatele
    3. Deklarace předpisu autora
    4. Důležité deklarace předpisu autora
    5. Důležité deklarace předpisu uživatele
- Deklarace ze stejného typu předpisu a se stejnou důležitostí se seřadí podle přesnosti svého selektoru (přesnost selektoru viz níže).
- Pokud ani seřazení podle přesnosti selektoru nemůže jednoznačně určit pořadí deklarací, aplikuje se jednoduše pořadí takové, jak byly jednotlivá pravidla zapsána po sobě v předpisu CSS.

Takto seřazené deklarace jsou připraveny pro ohodnocení daného elementu. Vyskytuje-li se více deklarací se stejnou vlastností, lze podle pořadí jednoznačně určit tu nejdůležitější.

3.2 Deklarace s vyšší prioritou - !IMPORTANT

Každá deklarace v pravidle může obsahovat také klíčové slovo !IMPORTANT, které dává dané deklaraci vyšší význam. Znamená to, že taková **deklarace nebude přepsána ostatními**.

Označování některých deklarací jako „důležité“ umožňuje vytvořit rovnováhu mezi předpisy uživatele a autora stránky. Za normálních okolností mají předpisy definované autorem stránky vyšší prioritu než předpisy uživatele, u důležitých deklarací je to však naopak. Existují-li dvě deklarace se shodnou prioritou v předpisu autora i uživatele označené jako !IMPORTANT, pak je použita hodnota nastavená uživatelem. Toto mohou využít uživatelé preferující nějaký specifický způsob zobrazování dokumentů.

Například deklarace definována v uživatelském předpisu:

```
* {
  color: #000000 !IMPORTANT;
}
```

zajistí, že se všechny texty zobrazí pouze černou barvou. Jakékoli definice barev autora stránek budou ignorovány (zastíněny touto deklarací).

3.3 Výpočet přesnosti selektorů

Aby bylo možné vypočítat prioritu každé z deklarací, je nutné vypočítat jak hodně je daný selektor přesný (specifický). Platí, že čím obecnější selektor, tím menší priorita a naopak. Výpočet je pak přesně daný definicí, kdy výsledkem jsou 4 číselné hodnoty a-b-c-d, pro které platí:

a - je-li deklarace zapsána přímo v dokumentu, pomocí atributu style=“.....“, pak a = 1, v ostatních případech a = 0

b - udává počet ID atributů v selektoru.

c - udává počet ostatních atributů a pseudo-tříd v selektoru

d - udává počet elementů a pseudo-elementů v selektoru

Je-li dán například selektor

```
A.red.message
```

pak a = 0 b = 0 c = 2 d = 1, tedy přesnost je 0, 0, 2, 1

Při řazení se pak využívá algoritmu schopného řadit podle více klíčů, kdy jednotlivé klíče jsou písmena a b c d. **Řadící metoda musí být stabilní**.

3.4 Přiřazené hodnoty

Jsou-li jednotlivé deklarace přiřazeny k elementům stromu dokumentu, dojde k přepočtům hodnot ve čtyřech fázích:

2. **specifické hodnoty (specified values)** – jedná se o hodnoty přímo získané z předpisu CSS tak, jak je zapsal autor dokumentu. Nejsou-li hodnoty zadány, použijí se výchozí.
3. **vypočítané hodnoty (computed values)** – Hlavním cílem této fáze je odstranění všech relativních hodnot a jejich nahrazení vypočítanými hodnotami absolutními. Jedná se převážně o relativně zadané délky (například v procentech) či velikosti písem (například identifikátor 'larger'). **Tyto hodnoty jsou využívány pro dědění.**

   Výpočet absolutní hodnoty není pro všechny vlastnosti shodný. Každá vlastnost obsahuje ve specifikaci popis, jakým způsobem se hodnoty získávají a co je k jejich výpočtu potřeba.
4. **použité hodnoty (used values)** – obsahují maximálně konkrétní hodnoty, které lze získat bez vykreslení dokumentu.
5. **aktuální hodnoty (actual values)** – jsou konkrétní hodnoty určené pro vykreslení dokumentu na daném zařízení. Pokud je například dán blokový element DIV s šířkou 50%, lze jeho absolutní velikost v pixelech získat až po rozložení všech elementů na obrazovce.

Zdroj: [1]

4 Definice objektového rozhraní

Objektové rozhraní popisuje objekty, pomocí kterých bude předpis CSS reprezentován a definuje metody umožňující modifikaci a přístup k vlastnostem. Je zcela přirozené, že toto rozhraní kopíruje výše popsanou strukturu předpisů CSS a jeho implementace umožní nést jakýkoli validní předpis verze CSS 2.1.

Na internetu lze nalézt několik podobných projektů, které se snaží o totéž. Jedná se převážně o definice na stránkách organizace W3C, kde jsou zveřejněna rozhraní umožňující reprezentovat předpisy CSS (např. Document Object Model CSS [2], SAC [3]). Žádné z těchto ale není vhodné použít v této práci, jelikož se jedná většinou o obecný přístup nezávislý na použitém jazyku.

Cílem tohoto návrhu je tedy vytvořit takové rozhraní, které bude přímo určené pro použití v jazyce Java a bude daný předpis CSS reprezentovat tak, aby práce s ním byla co nejrychlejší a nejsnazší.

Každé z následujících rozhraní definuje nutnou přítomnost metody `toString()`, která zajišťuje textovou reprezentaci daného objektu. Očekává se, že například metoda `toString()` u objektu třídy implementující rozhraní `Declaration` vrátí textovou reprezentaci dané deklarace a využije k tomu zavolání metody `toString()` nad svými selektory a vlastnostmi.

StyleSheet

Základní objekt reprezentující celý předpis CSS. AT-pravidla @charset a @import jsou uložena zvlášť pro snadnější přístup a zpracování. Metoda `toString()` vrací celý předpis CSS v textové podobě, jedná se tedy o export z objektové podoby například zpět do *.css souboru.

Metoda `check()` kontroluje konzistenci objektové reprezentace. Ve většině případů je volána právě na objektu StyleSheet, odkud jsou hierarchicky kontrolovány i další podobjekty.

```
String getCharset();
void setCharset(String charset);
List<ImportURI> getImportList();
List<Rule> getRulesList();
String toString();
void check() throws StylesheetNotValidException;
```

ImportURI

Reprezentuje AT-pravidlo @import. Umožňuje kromě URI definovat také media, pro která je importovaný předpis určen.

Parametr `path` metody `check()` slouží pro předávání pozice v předpisu CSS a lze jej použít v případě chyby pro vypsání místa výskytu.

```
String getUri();
void setUri(String uri);
List getMedia();
String toString();
void check(String path) throws StylesheetNotValidException;
```

Rule (abstract)

Abstraktní reprezentace pravidla – z něho jsou dále odvozena pravidla RuleMedia, RulePage a RuleSet. U metody `toString()` parametr `depth` určuje, kolik mezer se má vypsat před výpisem pravidla (o kolik je pravidlo odsazené od kraje – není důležité, pouze zlepšuje čitelnost.)

```
String toString(int depth);
void check(String path) throws StylesheetNotValidException;
```

RuleMedia (extends Rule)

Shromažďuje v sobě pravidla, která jsou určena pouze pro prezentaci dat na určitých výstupních zařízeních.

Metoda `check()` by měla zkontrolovat, že je zadáno alespoň jedno medium a že všechna media jsou ze seznamu podporovaných.

```
List<String> getMediaList();
List<RuleSet> getRulesList();
String toString(int depth);
void check(String path) throws StylesheetNotValidException;
```

RulePage (extends Rule)

Shromažďuje speciální deklarace, které slouží k popisu vzhledu na stránkových výstupních zařízeních.

```
String getPseudo();
void setPseudo(String pseudo);
List<Declaration> getDeclarationsList();
String toString(int depth);
void check(String path) throws StylesheetNotValidException;
```

RuleSet (extends Rule)

Běžné pravidlo nesoucí selektory a deklarace. Metoda `check()` by měla kontrolovat, že je k dispozici alespoň jeden selektor.

```
List<Selector> getSelectorsList();
List<Declaration> getDeclarationsList();
String toString(int depth);
void check(String path) throws StylesheetNotValidException;
```

Selector

Skládá se z jednoduchých selektorů, které jsou od sebe odděleny kombinátorem. Metoda `check()` by tedy měla zkontrolovat, že první jednoduchý selektor nemá žádný kombinátor a ostatní jej mají povinně. Nutná je přítomnost alespoň jednoho jednoduchého selektoru.

```
List<SimpleSelector> getSimpleSelectorsList();
String toString();
void check(String path) throws StylesheetNotValidException;
```

SimpleSelector

Jednoduchý selektor. Cílový element je možné popsat pomocí jeho typu (`firstItem`) a pomocí seznamu položek upřesňující třídy, ID atributy atd.

```
enum EnumCombinator { space, plus, greater }
SSType getFirstItem();
void setFirstItem(SSType firstItem);
List<SSItem> getItemsList();
EnumCombinator getCombinator();
void setCombinator(EnumCombinator combinator);
String toString();
```

SSType

Položka jednoduchého selektoru, která definuje typ elementu. V každém jednoduchém selektoru je maximálně jeden tento objekt, přístupný pomocí metod `getFirstItem()` a `setFirstItem()`.

```
String getValue();
void setValue(String value);
String toString();
```

SSItem (abstract)

Abstraktní reprezentace všech položek jednoduchého selektoru.

```
String toString();
```

SSItemAttrib (extends SSItem)

Položka jednoduchého selektoru vázaná na atribut elementu. Hodnota `Attrib` musí být zadána vždy, `Operator` je zadán pouze v případě, že se jedná selektor atributu „rovná se“, „obsahuje“ nebo „začíná“ (viz kapitola 2.4).

```
enum EnumOperator { equals, includes, dashmatch }
enum EnumValueType { ident, string }
String getAttrib();
void setAttrib(String attrib);
EnumOperator getOperator();
String getValue();
EnumValueType getValueType();
void setValue(EnumOperator op, String value, EnumValueType valueType);
String toString();
```

SSItemClass (extends SSItem)
Položka jednoduchého selektoru vázaná na třídu elementu.

```
String getValue();
void setValue(String value);
String toString();
```

SSItemID (extends SSItem)
Položka jednoduchého selektoru vázaná na ID elementu.

```
String getValue();
void setValue(String value);
String toString();
```

SSItemPseudo (extends SSItem)
Položka jednoduchého selektoru vázaná na pseudo-třídu elementu.

```
String getValue();
void setValue(String value);
String getFuncName();
void setFuncName(String funcName);
String toString();
```

Declaration
Deklarace skládající se z vlastnosti, seznamu hodnot a příznaku zvýšené důležitosti.

Metoda `check()` by měla zkontrolovat, zda je zadán nějaký řetězec reprezentující název vlastnosti (neměl by být prázdný). Dále je nutné zkontrolovat přítomnost alespoň jedné hodnoty, v případě výskytu více hodnot ověřit i správnost oddělujících operátorů.

```
boolean isImportant();
void setImportant(boolean important);
List<Term> getTermsList();
String getProperty();
void setProperty(String property);
String toString(int depth);
void check(String path) throws StylesheetNotValidException;
```

Term (abstract)
Abstraktní rozhraní zastřešující všechny typy hodnot deklarace.

```
enum EnumOperator { space, slash, comma }
EnumOperator getOperator();
void setOperator(EnumOperator operator);
String operator(String term);
```

TermColor (extends Term)
Hodnota barvy. Nastavení probíhá pomocí tří složek RGB, vrací standardní objekt třídy `java.awt.Color`.

```
void setValue(int r, int g, int b);
Color getValue();
String toString();
```

TermFunction (extends Term)

Hodnota definována pomocí funkčního zápisu. Skládá se z názvu funkce a seznamu hodnot jako jejich parametrů.

```
String getFunctionName();
void setFunctionName(String functionName);
List<Term> getTermsList();
String toString();
```

TermIdent (extends Term)

Hodnota identifikátoru.

```
String getValue();
void setValue(String value);
String toString();
```

TermNumber (extends Term)

Číselná hodnota, volitelně doplněná jednotkou. Implementovány musí být také funkce rozhodující o typu hodnoty (délka: číslo s jednotkou délky nebo 0, číslo a přirozené číslo: bez jednotky).

```
enum EnumUnit {px, em, ex, cm, mm, pt, pc, deg, rad, grad, ms, s, hz, khz}
Float getValue();
void setValue(Float value);
EnumUnit getUnit();
void setUnit(EnumUnit unit);
String toString();
boolean isLength();
boolean isNumber();
boolean isInteger();
```

TermPercent (extends Term)

Procentuální hodnota.

```
Float getValue();
void setValue(Float value);
String toString();
```

TermString (extends Term)

Řetězcová hodnota. Od identifikátoru se liší tím, že v předpisu CSS ohraničena uvozovkami.

```
String getValue();
void setValue(String value);
String toString();
```

TermUri (extends Term)

Hodnota URI.

```
String getUri();
void setUri(String uri);
String toString();
```

Pro lepší demonstraci uložení předpisu CSS do objektové struktury bude uveden jednoduchý příklad, který poukáže na všechny důležité vlastnosti. Mějme dán předpis CSS :

```
@charset “UTF-8“;
DIV IMG {
  border: #000000 1px solid;
}
@media screen, print {
  A:hover {
    color: #FF0000;
  }
}
```

pak výsledná struktura objektů tříd implementujících rozhraní bude vypadat následovně:

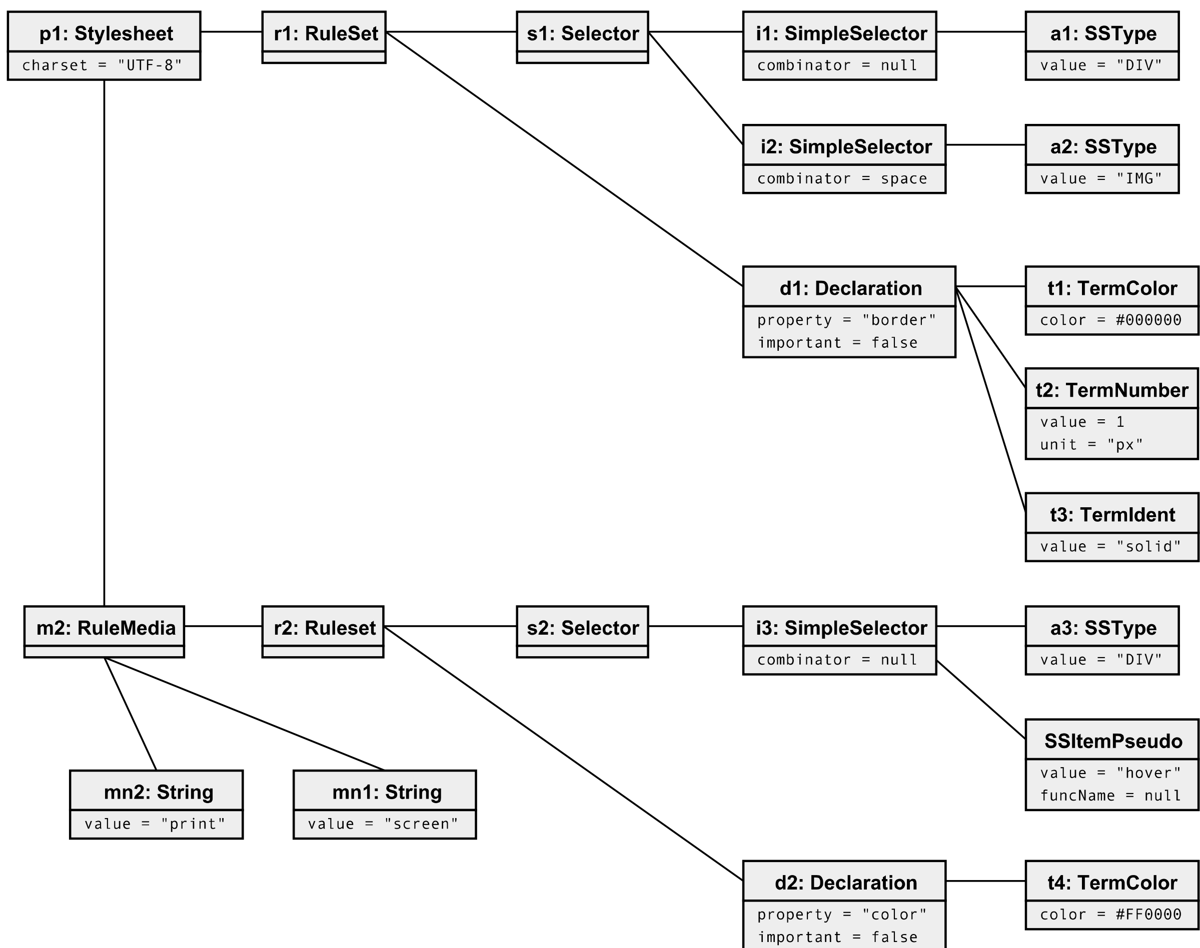


Diagram 1: Příklad objektové struktury (snapshot)

5 Vytvoření syntaktického analyzátoru – parseru

V této kapitole bude popsán postup při vytváření syntaktického analyzátoru, který bude součástí knihovny pro práci s předpisy CSS a bude zajišťovat jejich načítání.

Syntaktický analyzátor, v angličtině nazývaný také jako parser, je algoritmus, který umožňuje analyzovat vstupní řetězec. Výsledkem analýzy pak bývá rozhodnutí, zda daný vstup patří nebo nepatří do daného jazyka.

V případě knihovny pak bude parser nejenom rozhodovat o příslušnosti k jazyku, ale také vracet syntaktický strom pro další zpracování.

5.1 JavaCC

JavaCC (v angličtině Java Compiler Compiler) je soubor nástrojů pro automatické generování lexikálních a syntaktických analyzátorů. Na vstupu je vždy soubor s bezkontextovou gramatikou, ze kterého je vygenerován Java kód odpovídající parseru daného jazyka. Tento kód je možné zkompilovat pomocí běžného překladače Javy a ihned spustit.
JavaCC se skládá ze tří hlavních nástrojů pro práci s gramatikami:

- **JavaCC** – umožňuje vygenerovat algoritmus schopný rozhodnutí, zda daný vstup odpovídá jazyku generovanému gramatikou.
- **JJTree** – je rozšířený nástroj JavaCC, kdy vstup není pouze validován, nýbrž je rozkládán do syntaktického stromu a tento je na konci analýzy vrácen k dalšímu zpracování.
- **JJDoc** – nástroj umožňující převádět gramatiky z formátu JavaCC do běžné BNF formy.

Gramatiky, se kterými jsou schopny nástroje JavaCC pracovat jsou obecně typu LL(k). Tento typ gramatiky je ovšem nedeterministický a jeho použití má poměrně veliký vliv na výkonnost výsledného algoritmu. Z tohoto důvodu je umožněno vytvořit část analyzátoru jako LL(1) a pouze tam, kde je to nutné, použít LL(k). Je-li použita gramatika LL(k), je při generování parseru uživatel upozorněn na možné snížení výkonu.

Zápis gramatik je velice podobný notaci EBNF, která je stručně popsána níže. Kromě samotných přepisovacích pravidel obsahuje i několik speciálních značek umožňujících lépe specifikovat vstupní data a podobu výstupního algoritmu – parseru.

Celá sada nástrojů JavaCC je kompletně implementována pro prostředí virtuálního stroje jazyka Java 1.2 a vyšší.

5.2 Zápis gramatiky pomocí EBNF

Rozšířená Backus-Naurova forma (EBNF) je notace používaná pro formální zápis bezkontextových gramatik. Její výhoda je oproti například běžným přepisovacím pravidlům převážně ve snadné čitelnosti pro člověka. Zápis je jasný a přehledný.

Základními stavebními bloky této notace jsou terminální symboly reprezentující jednotlivé části zdrojového kódu jako například identifikátory, čísla, závorky atd. Z těchto terminálních symbolů jsou pak skládány symboly nonterminální.

```
cislo = "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
cislo_s_nulou = "0" | cislo;
```

V příkladu jsou dva nonterminály, kdy symbolu `cislo` odpovídá jakákoli číslice 1 – 9, symbol `cislo_s_nulou` pak reprezentuje číslice 0 – 9. Je zde také použit operátor svislé čáry, mající charakter OR („nebo“) sloužící pro výčet alternativ.

Je-li třeba reprezentovat určitou posloupnost symbolů, lze použít operátor čárka.

```
A = "1" | "2";
B = "3" | "4";
C = A , B;
```

Zde je ukázáno, že nonterminálnímu symbolu `C` odpovídá dvojice číslic, kdy první z nich bude 1 nebo 2 a druhá 3 nebo 4.

Některé symboly se mohou v jazyce nekonečně krát opakovat, což obsahuje i to že mohou být zcela vynechány. Tyto se pak vkládají do složených závorek:

```
A = "1" , { "2" } , "3";
```

Nonterminální symbol `A` odpovídá řetězci, který začíná číslicí „1“, následuje 0 – n číslic „2“ a je ukončen číslicí „3“.

Posledním vyjadřovacím prvkem je volitelný symbol, který je uzavřen do hranatých závorek. Znamená to, že tento symbol může, ale nemusí, být přítomen.

```
A = "1" , [ "2" ];
```

V tomto případě nonterminál `A` odpovídá buď řetězci „1“ nebo „12“ - číslice 2 na konci je volitelná a může být vynechána.

Těmito výše popsanými vyjadřovacími prvky lze popsat libovolnou bezkontextovou gramatiku. Jedná se o základní pravidla, ze kterých se odvodila spousta dalších notací. Tyto odvozené notace nemají vyšší vyjadřovací schopnost, pouze zjednodušují zápis a zvyšují přehlednost a čitelnost pro člověka. Jedna z nejznámějších modifikací je také definována jako standard ISO 14977.

5.3 Vytvoření gramatiky pro nástroj JavaCC

Při vytváření gramatiky pro nástroj JavaCC bylo vycházeno z předlohy, která je dostupná na webu W3C[2]. Vývojáři z této organizace tam prezentují, jak by mohla taková gramatika vypadat a upozorňují na některé důležité vlastnosti. V první části jsou popsány všechny terminální symboly pomocí regulárních výrazů tak, jak by je měl rozdělit lexikální analyzátor. V druhé části jsou pak jednotlivá přepisovací pravidla.

Po nastudování této gramatiky bylo třeba provést mírné korekce, jelikož využívá odlišnou formu zápisu. Například místo hranatých závorek pro volitelný symbol jsou použity závorky obyčejné následované otazníkem atd. Notace využívá stejných principů, které jsou jen odlišně zapsány.

Po přepsání formy zápisu a dodání několika nezbytných definic specifických pro JavaCC byla gramatika hotova a připravena pro použití.

Gramatika ve formátu pro nástroj JavaCC je uvedena v příloze A.

5.4 Rozbor gramatiky pro JavaCC

Soubor se zápisem gramatiky pro JavaCC obsahuje několik bloků, které mají specifický význam.

Hned za začátku se nachází sekce s nastavením některých parametrů. V tomto případě se nastavuje například ignorace velikosti písma či unicode kódování vstupu.

Následuje sekce s definicí API v jazyce Java. To je napsáno pomocí běžných Java metod, kdy tyto budou sloužit po vygenerování parseru jako hlavní vstupní bod. Těchto metod může být několik, kdy například mohou pokrývat požadavek předání vstupního řetězce různými způsoby (jako objekt třídy `String`, `InputStream` atd.).

Po definici rozhraní API lze již nalézt seznam všech terminálních symbolů uvozené klíčovým slovem TOKEN. Lexikální analyzátor z tohoto zápisu získá informaci o způsobu rozdělování vstupního řetězce na jednotlivé symboly, které budou předány na syntaktickou analýzu. Ze zdrojového souboru gramatiky je patrné, že se jedná o symboly jako závorky, identifikátory, řetězce, operátory atd.

2 http://www.w3.org/TR/CSS21/grammar.html

Hlavním blokem jsou bezesporu přepisovací pravidla. Ta jsou zapsána ve formě velice blízké EBNF, rozdíly jsou pouze v některých případech v jejich zápisu. Na vrcholu stojí opět nonterminální symbol StyleSheet, reprezentující celý předpis CSS, který je postupně rozepisován na pravidla, deklarace, vlastnosti a hodnoty a další.

V poslední části jsou vypsány některé terminální symboly, u nichž je při analýze a rozkladu na syntaktický strom důležitá jejich původní textová reprezentace (tzv. image). Parser, vytvořený pomocí JavaCC, standardně vytváří syntaktický strom obsahující pouze informaci o typu uzlu – tedy například uzel typu deklarace obsahuje dva uzly, jeden typu vlastnost, druhý typu hodnota. Pokud je nutné znát i konkrétní hodnoty, tedy v našem případě že se například jedná o vlastnost color a hodnotu #00000, je nutné toto dodefinovat.

Kompletní přehled struktury souborů gramatiky pro JavaCC je dostupný na webu autora.

5.5 Generování parseru

Při generování parseru se postupuje ve dvou krocích, kdy se postupně za sebou použijí nástroje JJTree a JavaCC.

```
svercl@local> jjtree CssParser.jjt
svercl@local> javacc CssParser.jj
```

První nástroj JJTree zpracuje gramatiku a vytvoří soubor CssParser.jj, který je vstupem pro JavaCC. JavaCC již vytvoří finální podobu parseru.

Používání parseru je velice snadné a přehledné. Objekt třídy `CssParser` poskytuje metodu `parse()`, jejímž parametrem je vstupní řetězec – ten je analyzován a v případě úspěchu vrácen v podobě stromu. Objektový strom reprezentující jednotlivé derivace je pak složen z objektů třídy `SimpleNode`. Práce se stromem se nijak neliší od běžných postupů – jsou k dispozici metody jako `getNumChildren()` či `getChild()` umožňující snadný pohyb po datové struktuře.

5.6 Příklad výstupu parseru

V následujícím objektovém diagramu je zobrazen výstup parseru po načtení jednoduchého předpisu CSS. Je zde patrné, že všechny objekty jsou třídy `SimpleNode` a tvoří navzájem stromovou strukturu.

Mějme dán předpis CSS:

```
A.msg {
  color: #000000;
}
```

pak pro tento vstup bude derivační strom vypadat následovně:

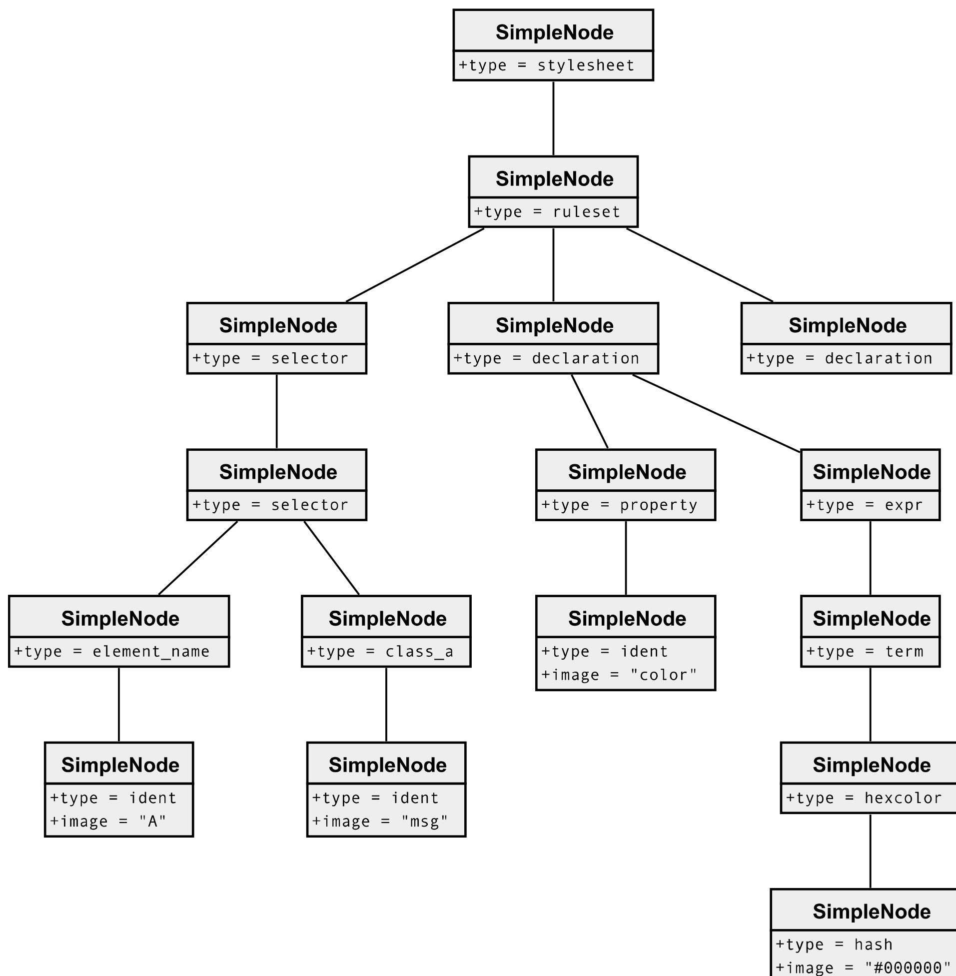


Diagram 2: Struktura objektů po načtení předpisu parserem

V diagramu je patrná asi největší nevýhoda takovéto struktury: všechny objekty jsou stejné třídy a všechny hodnoty pak reprezentovány jako řetězce typu `String`. Je to důsledek použití univerzálního nástroje, který parser automaticky vygeneruje. Další nevýhodou je také zbytečně velké množství objektů, přesně odpovídající přepisovacím pravidlům v gramatice.

Za zmínku také stojí vytvoření dvou uzlů typu „declaration“, což je způsobeno středníkem na konci první deklarace. Středník totiž neslouží k ukončování deklarací, nýbrž k jejich oddělování. Parser tedy předpokládá za středníkem další deklaraci, která je ale reprezentována pouze mezerou – tento jev se vyskytuje i v jiných programovacích jazycích a bývá obvykle nazván jako prázdný příkaz.

V této kapitole byl přiblížen způsob, jakým vznikla gramatika pro nástroj JavaCC. Z ní následně byl automaticky vygenerován parser, který analyzuje vstupní řetězec (v našem případě předpis CSS) a vytváří z něho stromovou strukturu. Tato stromová struktura je ovšem velmi obecná a manipulace s předpisy CSS by byla v takovéto podobě velice obtížná. Řešením tohoto problému se bude zabývat následující kapitola, kde bude popsán postup převedení této obecné stromové struktury do podoby definované objektovým rozhraním (popsaným v kapitole 4).

6 Knihovna pro práci s předpisy CSS - implementace

Knihovna poskytuje všechny operace potřebné pro komfortní manipulaci s předpisy CSS. Veliký důraz je kladen na snadnost začlenění do projektu a intuitivní použití.

Skládá se ze čtyř hlavních částí: třída `Engine`, objektové rozhraní, jeho implementace a automaticky vygenerovaný parser (pomocí JavaCC). Třída `Engine` má za úkol poskytovat metody pro načítání předpisů CSS z externího zdroje: tím může být například soubor, proud znaků (`InputStream`) či běžný řetězec (`String`). Objektové rozhraní je navrženo výše, zde musela být jeho implementace doplněna ještě o některé metody umožňující interní práci s daty z parseru.

Hlavní funkcionalitu knihovny lze rozdělit do pěti částí:

- **načtení** – předpis CSS na vstupu je načten parserem, vytvořen syntaktický strom a ten následně převeden do objektové podoby definované rozhraním. Výstupem této operace je objekt třídy `StyleSheetImpl` reprezentující vstupní předpis CSS.
- **vytvoření** – umožňuje vytvořit nový předpis CSS, se kterým budou prováděny další operace. Tato činnost se skládá pouze z vytvoření objektu třídy implementující rozhraní `StyleSheet` pomocí konstruktoru.
- **editace** – předpis, který byl načten nebo nově vytvořen, je možné libovolně upravovat. Lze přidávat či odstraňovat pravidla, upravovat deklarace či jejich hodnoty atd.
- **export** – složí k převodu z objektové reprezentace zpět na předpis CSS – například do souboru. Předpis, který je načten a následně vyexportován, musí být funkčně totožný (změní se pouze formátování, zmizí komentáře).
- **validace** – zkontroluje, zda daný předpis CSS má správný formát. Tato operace provádí pouze syntaktickou analýzu a proto odhalí pouze chyby tohoto charakteru. Sémantické vlastnosti nejsou kontrolovány - například zápis `color: 10px;` bude ze syntaktického hlediska správný, nicméně sémanticky je chybný.

6.1 Načtení z externího zdroje

Jedná se o nejsložitější operaci při manipulaci s předpisy CSS. K jejímu zahájení slouží statická metoda `process()` třídy `Engine`, vyžadující jako jediný svůj parametr vstupní předpis CSS.

Vstup, ať už předán v jakékoli formě, je převeden na objekt třídy `InputStream` a předán parseru. Ten provede lexikální a syntaktickou analýzu, jejímž výsledkem je syntaktický strom – objektová struktura takto načtených dat je naznačena v kapitole 5.6 na straně 25.

V druhé fázi je pak potřeba provést transformaci dat z univerzálního syntaktického stromu tvořeného objekty třídy `SimpleNode` do objektů implementujících rozhraní knihovny. Je při tom využito faktu, že tyto dvě struktury jsou velice podobné – jedná se vždy o strom, kde v kořenu je objekt nesoucí celý předpis CSS a postupně je přecházeno přes pravidla a deklarace k vlastnostem a hodnotám.

Algoritmy realizující převod jsou **obsaženy v konstruktorech** tříd implementujících objektového rozhraní – na začátku je zavolán konstruktor třídy `StyleSheetImpl`, kterému je předán jako parametr objekt třídy `SimpleNode`, kde `type = stylesheet`. Tento konstruktor si načte všechna data, která mu náleží (v tomto případě znakovou sadu - pravidlo @charset), a postupně volá konstruktory dalších tříd v závislosti na typu potomků objektu `SimpleNode`. Takto je zajištěn postupný průchod stromovou strukturou získanou z parseru a převod na jednotlivé objekty.

Po ukončení převodu je vrácen objekt třídy `StyleSheetImpl`, obsahující načtený předpis CSS.

Pro lepší pochopení je převod demonstrován na příkladu:

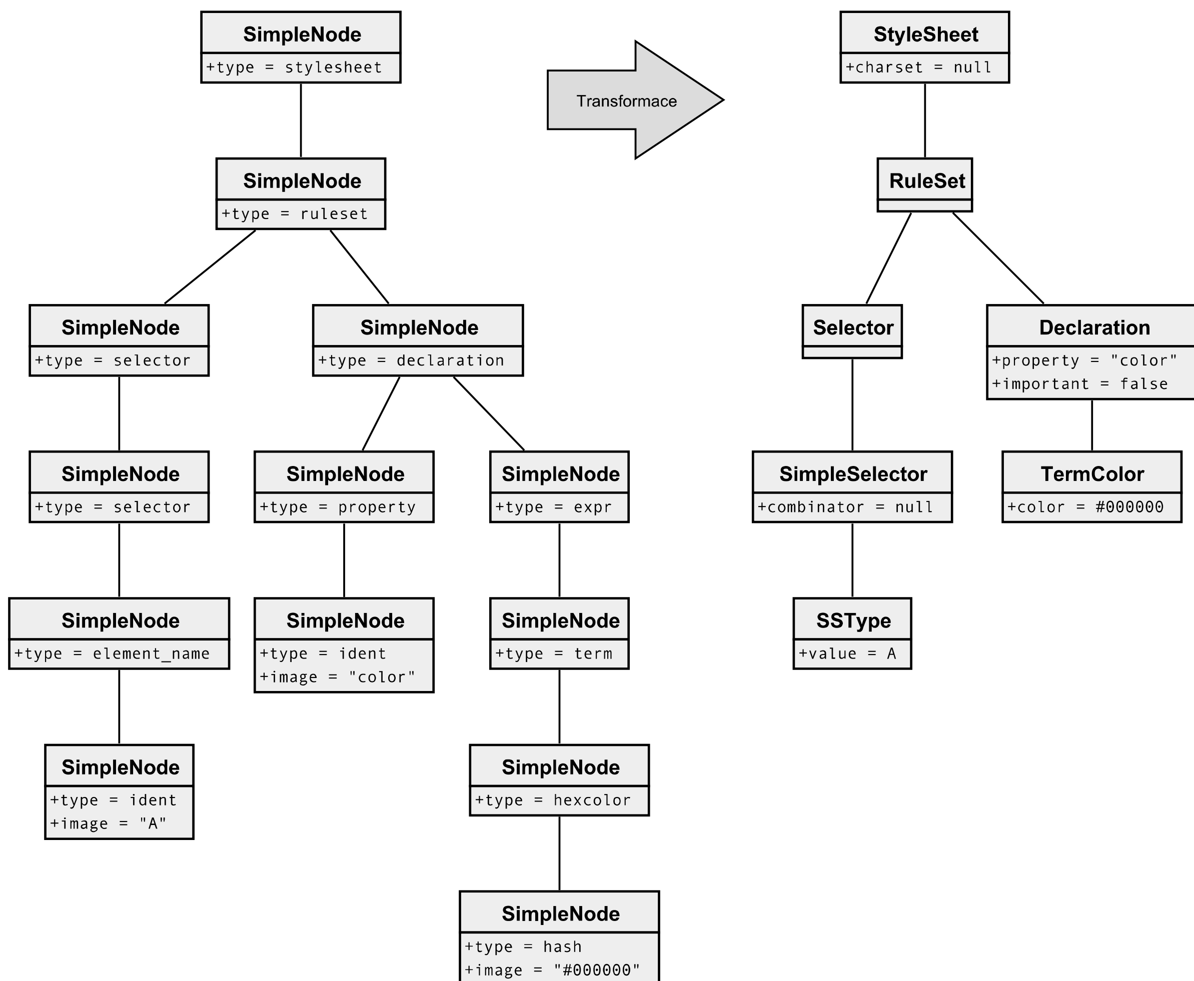


Diagram 3: Příklad převodu stromu do objektové reprezentace definované rozhraním

Vlevo je diagram objektů, získaný jako výsledek syntaktické analýzy z parseru, vpravo pak předpis CSS převedený do objektů tříd implementujících rozhraní.

6.2 Editace

Editace předpisu uloženého v objektové reprezentaci se provádí přirozeně pouhým vytvářením a odebíráním jednotlivých objektů. Chceme-li například přidat nové pravidlo s jednou deklarací, jednoduše vytvoříme objekty tříd jako RuleSetImpl, DeclarationImpl, SelectorImpl, SimpleSelectorImpl či Term<*>Impl a uspořádáme je dle jejich struktury do sebe.

Objekty jsou na sebe vázány pomocí typovaných objektů třídy ArrayList, které nesou vždy objekty níže v hierarchii. Je-li třeba pracovat s pravidly předpisu CSS, zavolá se nad objektem třídy StyleSheetImpl metoda GetRulesList(), která vrátí seznam pravidel (ArrayList<RuleSet>). S tímto seznamem lze libovolně pracovat – pravidla přidávat či

odebírat. Jak již bylo řečeno, seznam má definován třídu objektů, které může nést, což zaručuje že nebude obsahovat cokoli jiného. Obdobně pak funguje práce i s ostatními objekty v hierarchii.

Výše popsaná práce se stromovou strukturou je sice pohodlná pro programátora, nicméně přináší i drobné komplikace. Neprovádí se žádné kontroly prázdnosti seznamů, je tedy možné například vytvořit pravidlo, které nebude mít žádný selektor. Celý předpis CSS by se tak dostal do nekonzistentního stavu a mohlo by docházet k potížím při exportu. Řešením je metoda `check()`, kterou implementují všechny třídy objektového rozhraní a umožňují tak kontrolu konzistence. Přirozeně je tato operace prováděna postupně po objektech směrem níže v hierarchii.

6.3 Export (dump)

Nutnou součástí knihovny je i funkce pro export dat z objektové reprezentace zpět na předpis CSS.

Každá z tříd implementujících objektové rozhraní obsahuje metodu `toString()`, která vrací textovou reprezentaci daného objektu. Implementace těchto metod opět využívá hierarchie mezi objekty, kdy například při sestavování textové reprezentace objektu třídy `DeclarationImpl` se využijí textové výstupy z objektů tříd `Term<*>Impl`. Takto je zajištěno, že každý objekt zodpovídá za textový výpis sám sebe.

Samotný export celého předpisu CSS je pak proveden naprosto přirozenou a intuitivní cestou, tedy zavolání metody `toString()` nad objektem třídy `StyleSheetImpl`.

6.4 Validace

Slouží k analýze předpisu CSS a zjištění, zda je či není validní dle gramatiky knihovny. Funkčně se jedná v podstatě pouze o načtení předpisu parserem – pokud je vše v pořádku, skončí funkce bez návratové hodnoty. Při výskytu chyby je vyvolána výjimka s popisem chyby, která může být dále ošetřena.

6.5 Testování

Testování správné funkčnosti knihovny bylo prováděno v celé fázi vývoje a přirozeným cílem bylo zajistit bezproblémovou a hlavně správnou funkčnost.

Finální verzi knihovny bylo třeba podrobit důkladným testům, které měly za úkol odhalit případné nedostatky v implementaci. Jako ideální metoda se jeví práce s komplexními předpisy běžně užívanými na internetu.

K testování bylo náhodně vytipováno několik webových stránek s poměrně složitým designem (například portály idnes.cz, zive.cz, seznam.cz atd.). Hlavní stránky těchto webů pak byly pomocí prohlížeče uloženy na disk včetně všech obrázků, předpisů css a dalších podpůrných součástí.

Uložené předpisy CSS těchto stránek byly pak následně načteny pomocí knihovny do objektové reprezentace a z ní vyexportovány v textové podobě zpět do souboru. Touto operací došlo v předpisu CSS k několika změnám – například bylo změněno formátování, úplně zmizely komentáře, všechny definice barev byly převedeny na hexadecimální notaci atd. Podstatné ovšem bylo zachování funkčnosti tak, aby se jednotlivé webové stránky zobrazily totožně před i po konverzi. Tohoto stavu bylo úspěšně dosaženo ve všech testovaných případech.

6.6 Použití v projektu

Knihovna sama o sobě neobsahuje žádný kód, který by byl přímo spustitelný z příkazové řádky (chybí vstupní funkce `Main()`). Z toho důvodu se předpokládá, že její využití bude spíše jako součást nějakého jiného projektu, kde bude zastávat funkcionalitu pro práci s předpisy CSS.

Při návrhu byl kladen důraz na snadnost použití a proto lze vše ukázat na několika málo řádcích zdrojového kódu. Při použití se předpokládá, že třídy z balíku `cz.vutbr.web.css` a `cz.vutbr.web.cssKit` jsou dostupné v classpath.

```
. . .
String soubor = "/home/svercl/data/style.css";
StyleSheet predpis = Engine.parse(new File(soubor));
system.out.print(predpis.toString());
. . .
```

Demonstrovaný příklad ukazuje snadnost použití při práci s předpisy CSS v projektu. V prvním řádku je pouze definice vstupního souboru pro načtení. Druhý řádek pak provede načtení předpisu CSS do objektu třídy `StyleSheetImpl`. Nakonec je proveden export předpisu na standardní výstup.

6.7 Nedostatky a možnosti rozšíření

V této kapitole bude upozorněno na některé nedostatky, se kterými by měl programátor využívající knihovnu ve svém projektu počítat. Některé z nedostatků mohou být také vhodnou inspirací při případném rozšiřování projektu.

- **silná orientace na verzi CSS 2.1** – gramatika načítající předpisy CSS předpokládá, že daný soubor bude kompatibilní s verzí CSS 2.1 (v současné době aktuální standard). Z podobnosti verzí minulých a i dle výhledu do budoucna lze předpokládat, že problém by neměl nastat ani s předpisy jiných verzí, nicméně drobné komplikace nastat mohou. Například v CSS 1.0 bylo přípustné, aby byla délka zadána i pomocí palců (inches, jednotka in) – to je dle použité gramatiky nepřípustné a dojde k chybě.
- **náchylnost na chyby v předpisu CSS** – při načítání se předpokládá, že vstupní soubor neobsahuje syntaktické chyby – pokud ano, dojde k chybě a proces načítání se ukončí.

V praxi ale většina dnešních prohlížečů je poměrně benevolentní k chybám a i norma CSS definuje, že v některých případech se lze zotavit z chyb například ignorováním celého chybného pravidla.

7 Implementace algoritmu pro ohodnocení stromu dokumentu

Druhou částí této práce je implementace algoritmu, který bude schopen pro daný strom dokumentu přiřadit ke všem jeho elementům hodnoty z předpisu CSS. Takováto funkčnost může být vyžadována například jako součást nějakého webového prohlížeče implementovaného v jazyce Java.

Při implementaci bude využita knihovna pro práci s předpisy CSS a bude tak i částečně demonstrováno její nasazení. Důležitým požadavkem je také použití efektivních optimalizací pro zvýšení rychlosti, neboť proces ohodnocení může být pro rozsáhlé dokumenty poměrně výkonnostně náročný.

Výstup operace ohodnocení stromu dokumentu bude třeba nějakým způsobem převést do podoby, která je prezentovatelná a lze tak ověřit správnost. Nabízí buď možnost kompletního výpisu všech elementů včetně přiřazených hodnot či zpětný převod takto ohodnoceného dokumentu do HTML a vypsání vlastností jako inline stylu pomocí atributu `style=". . ."`. Druhá varianta umožní poměrně zajímavé testování podobné tomu v kapitole 6.5, kdy budou vybrány některé webové stránky a bude se testovat, zda jejich zobrazení je shodné s externím předpisem CSS jako při dosazení hodnot do inline stylů v konkrétních elementech.

7.1 Vstupní data

Jak již bylo naznačeno, vstupem budou dva typy dat:

- **strom dokumentu** – očekává se ve formě objektů tříd implementujících rozhraní ze standardního balíku `org.w3c.dom`. Ideálním prostředkem pro získání takovéto struktury je použití nástroje JTidy a načtení běžného HTML souboru. Více o JTidy níže.
- **předpis CSS** – ve formě objektů tříd implementujících objektové rozhraní prezentované v kapitole 4 této práce. Nabízí se možnost načtení takového předpisu z externího *.css souboru pomocí přiložené knihovny pro práci s předpisy CSS.

7.2 Návrh struktury

Implementace bude provedena pomocí tří základních tříd, obsahující veškerou funkcionalitu. Názvy a důležité metody jsou naznačeny v diagramu níže.

```
Controller
+process(d:Document,s:Stylesheet,media:String): HashMap<Element, NodeData>
+print(d:Document,data:HashMap<Element, NodeData>,os:OutputStream)
+printHTML(d:Document,data:HashMap<Element, NodeData>,os:OutputStream)
+printInlineStyle(d:Document,data:HashMap<Element, NodeData>,os:OutputStream)
```

```
Analyzer
#analyze(d:Document,s:StyleSheet,media:String): HashMap
-processElement(e:Element,assignedDeclarations:HashMap,
               mapElement:HashMap,mapClass:HashMap,
               mapID:HashMap,others:ArrayList)
-matchElementBySelector(e:Element,s:Selector): boolean
-matchElementBySimpleSelector(e:Element,ss:SimpleSelector): boolean
```

```
NodeData
+putDeclaration(d:Declaration)
```

Diagram 4: Třídy tvořící implementaci ohodnocení stromu dokumentu

NodeData

Objekty třídy `NodeData` jsou hlavním nositelem veškerých informací o jednom elementu. Při ohodnocování je pak ke **každému elementu vstupního dokumentu přiřazen jeden objekt třídy `NodeData`**.

Jedinou veřejnou metodou této třídy je `putDeclaration()`, jejímž vstupem je objekt třídy implementující rozhraní `Declaration`. Tato metoda převezme deklaraci a pokusí se jí zpracovat – pokud obsahuje správná data, uloží si je do své vnitřní struktury a přepíše případná data předcházející stejného charakteru. Více informací o funkčnosti této třídy lze nalézt v samostatné kapitole 7.5.

Analyzer

`Analyzer` je třída, mající za úkol nalézt pro každý element vstupního dokumentu pravidla, které se k němu (k danému elementu) vztahují. Tato pravidla jsou rozložena na jednotlivé deklarace a ty jsou pak následně seřazeny dle přesnosti jejich selektoru (viz kapitola 3.3). Výsledkem je tedy mapa (objekt třídy `HashMap`) obsahující pro každý element jeden seznam – tento seznam pak obsahuje všechny deklarace seřazené dle pořadí.

Třída obsahuje metody, které tuto činnost zajišťují. Proces je zahájen voláním metody `Analyze()`, na jejímž vstupu je dokument, předpis CSS a cílové medium. V této metodě se vstupní dokument prochází po jednotlivých elementech a pro každý je pak zavolána metoda `processElement()`. Zde se za pomoci `matchElementBySelector()` respektive `matchElementBySimpleSelector()` vyberou odpovídající pravidla.

Conroller

`Controller` je hlavní třídou, starající se o veškerou činnost uvnitř a komunikující s okolím. Pomocí metody `process()`, která má jako vstupní parametry dokument, předpis CSS a cílové medium, je zahájena operace ohodnocení.

V první části se zavolá metoda `Analyze()` třídy `Analyzer`, kdy se pro každý element zjistí seznam deklarací. V druhé části se pak pro každý element vytvoří objekt třídy `NodeData` a do něho se postupně „pumpují" deklarace ze seznamů (pomocí metody `putDeclaration()`).

Třída `Controller` má v sobě také několik metod zajišťujících výpis zpracovaných dat do prezentovatelné podoby. Může se jednat o textový či HTML výpis ohodnocených elementů či export stromu dokumentu zpět do HTML spolu s vlastnostmi zadanými jako inline styly.

Pro lepší demonstraci následuje diagram sekvence ukazující posloupnost jednotlivých akcí a kooperace objektů.

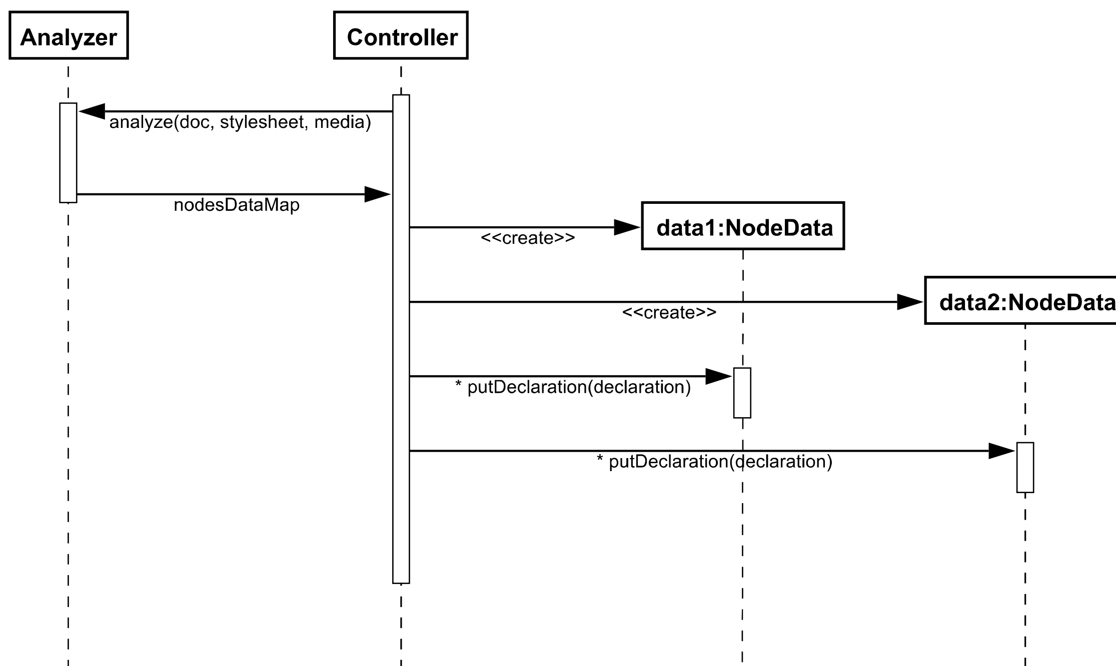


Diagram 5: Diagram sekvence ukazující návaznosti jednotlivých činností

V diagramu je patrné, že nejdříve je provedena analýza, která zjistí pro každý element všechny odpovídající deklarace. Ty jsou uloženy v objektech třídy `ArrayList` (po jednom pro každý element) v mapě třídy `HashMap`. Mapa má jako **indexy** právě **objekty elementů**.

V druhé fázi je pak pro každý element vytvořen jeden objekt třídy `NodeData`, který bude uchovávat vlastnosti elementu. Ihned po vytvoření jsou všechny vlastnosti nastaveny na výchozí hodnoty, které jsou následně modifikovány zpracováváním deklarací.

7.3 Výběr pravidel pro daný element

Základem pro výběr pravidel předpisu CSS pro daný element je rozhodnutí, zda daný element (případně i jeho kontext) odpovídá selektoru. Tuto funkcionalitu zajišťuje metoda `matchElementBySelector()` třídy `Analyzer`. Ta má na vstupu dva objekty, přirozeně element a selektor, a jejím výstupem je pak hodnota typu `boolean` reprezentující výsledek.

Druhou metodou usnadňující výběr pravidel předpisu CSS pro daný element je `matchElementBySimpleSelector()`. Tato metoda funguje obdobně jako předchozí uvedená s rozdílem, že na vstupu je místo selektoru pouze – jednoduchý selektor. Jak bylo naznačeno v kapitole 2.4 Každý selektor se skládá z jednoho či více jednoduchých selektorů. Vlastností jednoduchých selektorů je fakt, že je **lze vždy ověřovat vůči jednomu elementu** stromu dokumentu. Skládáním jednoduchých selektorů pomocí kombinátorů je pak možné pracovat i s kontextem elementu ve stromu dokumentu.

Implementace těchto vlastností využívá: metoda `matchElementBySimpleSelector()` pracuje s jedním elementem a jedním jednoduchým selektorem a pouze na základě shodnosti (pseudo)tříd, ID, názvu (pseudo)elementu a hodnot atributů rozhodne zda daný element odpovídá jednoduchému selektoru. Metoda `matchElementBySelector()` pak zajišťuje jen pohyb po stromu dokumentu a řeší zda se shoduje kontext elementu s vlastním selektorem.

7.4 Optimalizace – zrychlení výběru pravidel pro element

Ačkoli by bylo možné vyhledávat odpovídající pravidla pro dané elementy jednoduše pomocí cyklů, byl by takový postup velmi zdlouhavý, obzvlášť u složitějších dokumentů. Znamenalo by to provést kartézský součin element × selektor a všechny tyto dvojice ověřit pomocí metody `matchElementBySelector()`.

Vhodnější přístup je jistě rozdělit pravidla dle jejich selektorů do několika skupin, převážně dle příslušnosti ke třídě, ID a názvu elementu v selektoru. Při výběru je pak možné některé skupiny vyřadit přímo bez nutnosti ověřování selektorů a zvýšit tak rychlost ohodnocení.

V první řadě je třeba vytvořit třídu, jež bude v sobě uchovávat jedno pravidlo s deklaracemi (tedy objekt třídy implementující rozhraní `RuleSet`) spolu s **pořadovým číslem výskytu** v předpisu CSS. Pořadové číslo je důležité, jelikož **po rozdělení do jednotlivých skupin by zmizela informace o pořadí**, která hraje svou roli při následném ohodnocení. Třída má název `OrderedRule` a obsahuje v sobě pouze dva zmíněné objekty.

```
OrderedRule
-rule: RuleSet
-orderNum: int
+compareTo(o:Object): int
+equals(o:Object): boolean
+getRuleSet(): RuleSet
+getOrderNum(): int
```

Diagram 6: Třída pro uchování pravidel včetně jejich pořadového čísla

Jednotlivá pravidla opatřená pořadovým číslem lze pak rozdělovat do skupin. Existují čtyři hlavní skupiny, z nichž tři obsahují ještě další podskupiny.

1. **skupina className** – obsahuje podskupiny nazvané jmény tříd, vyskytující se v selektorech – pravidlo je zařazeno do podskupiny tehdy, vyskytuje-li se v jeho posledním jednoduchém selektoru daná třída. Je-li tříd v posledním jednoduchém selektoru více, v potaz se bere poslední z nich. Není-li v posledním jednoduchém selektoru žádná třída, pravidlo se vůbec do skupiny className nezařazuje.

   příklad: Pravidlo se selektorem `DIV.prvni A.prvni.cerveny` bude zařazeno do skupiny `className:cerveny`.
2. **skupina idName** – účel je shodný se skupinou className. Výjimkou je pouze práce s atributem ID místo třídy a fakt, že každý element může mít definováno pouze jedno unikátní ID (na rozdíl od tříd, kdy element může být součástí více tříd, v zápisu oddělených mezerou - <A class="prvni cerveny">).
3. **skupina elementName** – účel je shodný se skupinou idName.
4. **skupina ostatní** – obsahuje všechna ostatní pravidla nezařazená do skupin předchozích.

Při rozdělování do skupin samozřejmě může nastat situace, že se pravidlo díky svému selektoru dostane jak do některé z podskupin className tak i do například podskupiny idName. Do skupiny ostatní se pak zařazují všechna pravidla, která nejsou v žádné ze skupin předchozích.

Pro demonstraci procesu optimalizace výběru pravidel pro element následuje jednoduchý příklad:

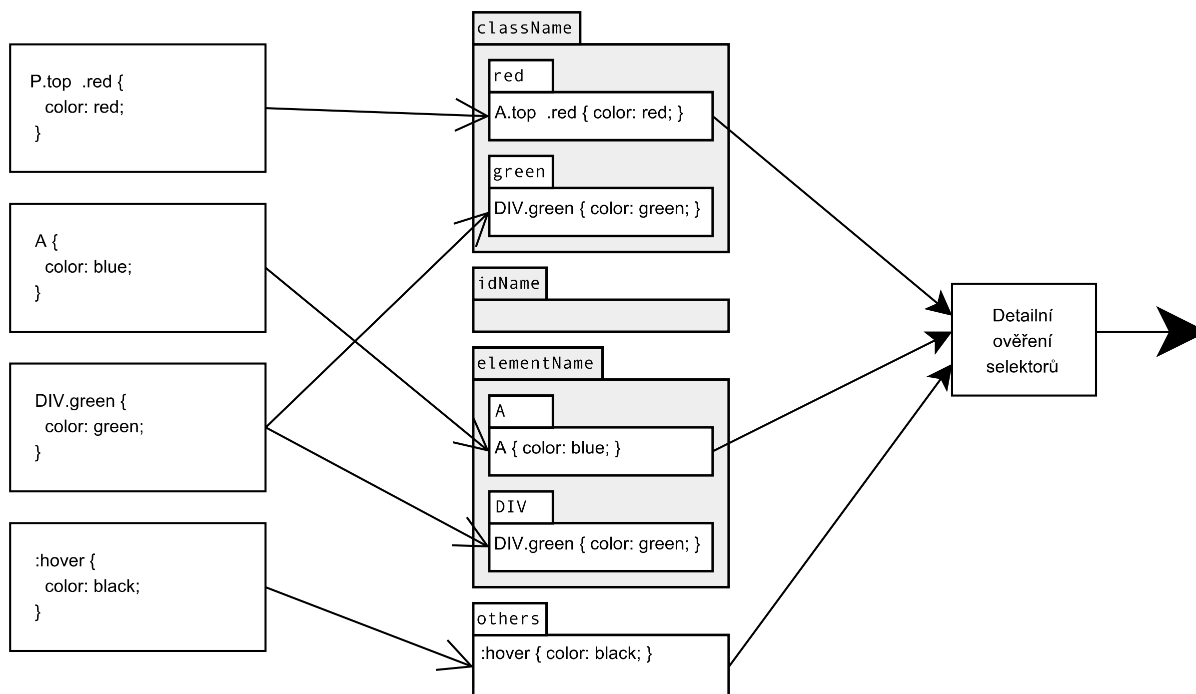


Diagram 7: Příklad optimalizace výběru pravidel pro konkrétní element

V diagramu je v levé části dán předpis CSS, rozdělený na jednotlivá pravidla. Tato pravidla se před začátkem ohodnocení stromu dokumentu rozmístí do skupin dle svého **posledního jednoduchého selektoru**. Takto předpřipravená data **se již v průběhu ohodnocení nemusí nějak měnit**.

Je-li pak potřeba vyhledat všechna pravidla pro konkrétní element (v tomto případě se jedná o `<A class='red' id='logo'>..</a>`), vyjmou se pravidla ze skupin `elementName:A`, `className:red`, `idName:logo` a samozřejmě (vždy) také ze skupiny `others` (ostatní). Tímto se zajistí užší výběr kandidátních pravidel, jejichž selektory jsou dále ověřovány podrobnou metodou `matchElementBySelector()`. Optimalizace tedy nevybírá přesnou množinu výsledných pravidel, ale pouze zúží počet kandidátů nutných prověřit komplexním způsobem.

7.5 Popis funkce třídy NodeData – aplikace deklarace na element

Při ohodnocení stromu dokumentu je třeba přiřadit ke každému elementu nějaký objekt, ve kterém se bude ono „ohodnocení“ shromažďovat – tímto jsou právě objekty třídy `NodeData`.

V sekvenčním diagramu na straně 36 již bylo naznačeno, že se vytváří pro každý element jeden objekt třídy `NodeData`, do kterého se pomocí metody `putDeclaration(...)` „pumpují“ jednotlivé deklarace a ty jsou následně zpracovány do formy hodnot proměnných. Proměnné jsou

deklarovány tak, aby pomocí jejich hodnot bylo možné popsat jakýkoli stav elementu přípustný dle normy CSS 2.1. Následující příklad ukazuje, jakým způsobem se uvnitř objektů třídy `NodeData` ukládá informace o horním odsazení (vlastnost padding-top).

```
. . .
public enum EnumPaddingWidth { length, percentage, inherit }
. . .

. . .
private EnumPaddingWidth paddingTopType = EnumPaddingWidth.length;
private TermPercent paddingTopPercentValue = null;
private TermNumber paddingTopNumberValue = new TermNumber(new Float(0));
. . .
```

Z příkladu je patrné, že vlastnost horní odsazení je uložena pomocí tří proměnných, výchozí stav je délka s hodnotou 0.

Pokud by došlo například ke zpracování deklarace `padding-top: 20%;`, změní se všechny tři proměnné: `paddingTopType` bude obsahovat hodnotu `“percentage“`, `paddingTopPercentValue` bude nastavena na 20% a poslední, `paddingTopNumberValue`, bude přirozeně nastavena na `null`.

Při dalším zpracování elementu, například vykreslení na obrazovku, je pak zjišťování vlastností z takto připravených dat triviální a velice nenáročné vzhledem s systémovým zdrojům počítače.

Pro zpracování jednotlivých deklarací existují metody, kdy **každá z nich je schopna zpracovat deklaraci s jednou konkrétní vlastností**. Například pro výše zmíněnou deklaraci s vlastností `padding-top` existuje metoda `processPaddingTop(...)`, která tuto deklaraci zpracuje a její hodnoty uloží do proměnných.

Výše zmiňovaná univerzální metoda `putDeclaration(...)` pak má za úkol pouze zkonvertovat jméno vlastnosti (například padding-top na procesPaddingTop) a pokusit se **vyhledat příslušnou obslužnou metodu**. Pokud se to podaří, deklarace je zpracována, pokud ne, je tato prohlášena za nepodporovanou.

7.6 Výstup

Výsledkem ohodnocení stromu dokumentu je mapa (objekt třídy `HashMap`), která má jako **klíče objekty elementů**, jako hodnoty pak objekty třídy `NodeData`. Při následné manipulaci je pak možné pro konkrétní element pouze vzít tento, použít jej jako klíč a z mapy získat jeho ohodnocení.

Výstupní data v takovémto formátu jsou ovšem bohužel velice obtížně prezentovatelná, nelze je snadno číst a hodnotit výsledky. Proto v sobě obsahuje třída `Controller` několik metod sloužících pro vypsání dat v různých formátech snadno čitelných pro člověka.

- **metoda** `print()` – vypíše na výstup celý strom dokumentu včetně všech vlastností v textové podobě. Textové reprezentace elementů jsou pod sebou a odsazeny takovým počtem mezer jako je jejich hloubka zanoření ve stromu dokumentu. Výpis je úplný - jelikož obsahuje u každého z elementů seznam všech vlastností, může být pro rozsáhlé dokumenty velice objemný.

  Poslední tři položky výpisu každého elementu obsahují seznam deklarací, který obsah ovlivnil. Nacházejí se tam správné (validní), chybné a nepodporované deklarace.
- **metoda** `printHTML()` – výstup je velice podobný výše popsanému – rozdíl je pouze ve způsobu formátování, kdy zde je použito jazyka HTML. Umožňuje to použít i některé další grafické vyjadřovací prostředky jako barva či řez písma, vzhled pozadí atd.
- **metoda** `printInlineStyleHTML()` – všechny vlastnosti uložené v objektu třídy `NodeData` jsou zpět přetransformovány do podoby inline stylu. Tyto jsou pak následně přidány do atributu `style=". . ."` každého z elementů a celý HTML dokument je vyexportován na výstup. Pomocí webového prohlížeče je pak možné porovnat výstupní HTML dokument s originálem a testovat tak funkčnost. *Pozn.: tato operace má některá svá omezení a její úspěch může záviset i na použitém webovém prohlížeči – například deklarace* `_margin: 10px;` *je chybná a bude knihovnou ignorována, kdežto prohlížeč MS IE jí normálně zpracuje jako* `margin: 10px;`.

7.7 Omezení

Je důležité upozornit případné uživatele či čtenáře na některé nedostatky implementace procesu ohodnocení stromu dokumentu. Případný uživatel by měl zvážit, zda ho při použití budou omezovat, a v případě potřeby dotyčné oblasti vyřešit.

- **předpis CSS pro ohodnocení musí být zadán explicitně** – při procesu ohodnocení stromu dokumentu se použije pouze předpis CSS, který je vstupem hlavní metody. Pro snadnější manipulaci by ale bylo vhodnější, kdyby se při analýze stromu vyhledaly všechny elementy `<link TYPE="text/css" . . .>` sloužící pro připojení předpisu CSS a s těmito pak bylo provedeno ohodnocení.
- **ohodnocení je prováděno pouze jedním předpisem, a to předpisem autora** – v kapitole 3 bylo uvedeno, že při ohodnocení stromu dokumentu hrají svojí roli 3 typy předpisů CSS – předpis webového prohlížeče, uživatele a autora. Pokud by tato funkcionalita byla vyžadována, je třeba zdrojový kód upravit.
- **nejsou podporovány všechny vlastnosti** – třída `NodeData` nemá implementovány metody pro zpracování vlastností pro media aural a paged. Jelikož je ovšem funkčnost velice podobná

zpracování ostatních vlastností, neměl by být problém tyto metody v krátkém čase implementovat.

7.8 Možnosti rozšíření

Pokud by bylo třeba projekt rozšířit, určitě se zde nabízí možnost pokračovat v implementaci takovým směrem, aby ve finále vznikl jednoduchý webový prohlížeč na platformě JAVA. V takovém případě by bylo třeba provést s dokumentem ještě minimálně následující kroky:

- **přepočty hodnot** – v kapitole 3.4 bylo naznačeno, že hodnoty se mohou při zpracování nacházet ve čtyřech fázích: specifické, vypočítané, použité a aktuální. Tyto výpočty jsou poměrně složité na implementaci, jelikož pro různé elementy i pro různé vlastnosti může být výpočet mírně odlišný. Problematický je převážně výpočet procentuálních délek elementů, které nemají pevně zadány rozměry svého rodiče. Hodnoty aktuální lze dokonce získat až po rozložení bloků na stránce či obrazovce.
- **kaskáda** – některé hodnoty, například písmo či barva, jsou děděny automaticky, u jiných je možné toto vyžádat explicitně pomocí identifikátoru `inherit`. Kaskádu lze provádět až po implementaci přepočtu hodnot, minimálně je třeba zajistit přepočet hodnot na „vypočítané".
- **vykreslení na obrazovku** – jedná se asi o nejobtížnější část, kdy se dokument změní ze své zdrojové podoby na jeho grafickou reprezentaci. Takovéto vykreslení musí brát v ohled spoustu aspektů a možností konkrétního systému, například instalovaná písma, vlastnosti zobrazovacího zařízení či preference konkrétního uživatele.

8 Ladění výkonu (profiling)

Ačkoli by se mohlo zdát, že naprogramováním aplikace a jejím otestováním po funkční stránce je práce na projektu hotova, není to tak. V projektu se mohou vyskytovat programátorské chyby, které sice nijak nenarušují správnost fungování algoritmů, ale mohou zbytečně zatěžovat systémové prostředky tam, kde to není nezbytně nutné. Nejvíce nebezpečné jsou pak přirozeně úseky kódu, které sice na první pohled nevypadají příliš složitě (z pohledu využití syst. prostředků), nicméně v důsledku jejich velmi častého provádění (například v cyklech) se může negativně projevit každá jinak zanedbatelná drobnost.

V dnešní době, kdy procesory a paměti dosahují poměrně vysokých výkonů za zanedbatelné ceny, by se mohlo zdát, že taková optimalizace je zbytečná. V některých případech asi ano. Nicméně stále existuje spousta aplikací, kde jsou požadavky na rychlost zpracování stále vysoké a každé zrychlení uživatelé ocení – a to je právě i případ tohoto projektu.

Pokud bude knihovna pro zpracování předpisů CSS a algoritmus pro ohodnocení stromu dokumentu využit v aplikaci určené pro práci s HTML dokumenty, bude určitě uživatel vyžadovat co nejlepší odezvy a pokud možno co nejrychlejší zpracování. Vzhledem k použití ne příliš rychlého jazyka JAVA je pak jistě nutné ladění výkonu nezanedbat.

Pro optimalizaci byl použit nástroj vývojového prostředí NetBeans Profiler. Ten pracuje tak, že do výsledného programu přikompiluje jisté značky, kdy jejich průchodem je schopen analyzovat doby trvání a počty volání jednotlivých metod. Výsledkem je pak strom, na jehož vrcholu je funkce `main()` s dobou trvání shodnou s dobou běhu programu, který se dále větví na jednotlivé volané metody. Lze tak na přehledných grafech vidět, provádění kterých metod je rychlé a bezproblémové, nebo kde naopak může být zdroj nějakých nevhodných konstrukcí. Při analýze hraje samozřejmě velikou roli zkušenost programátora – ne každá část programu zabírající větší množství systémových prostředků musí být nutně optimalizovatelná.

8.1 Ukázka optimalizace části kódu

Pro demonstraci a popis průběhu ladění výkonu byla vybrána jednoduchá metoda `methodName(String property)` třídy `NodeData`, která je velmi často využívána. Tato má za úkol transformovat název vlastnosti v předpisu css na metodu, která danou deklaraci zpracuje. Například:

```
border-color    ->    processBorderColor
```

Při implementaci byla tato metoda naprogramována nejdříve takto:

```
private String methodName(String property) {
    byte[] b = property.getBytes();
    String out = "process";
    boolean upper = true;
    for(byte chr : b) {
        String tmp = "" + (char)chr;
        if(tmp.matches("[a-zA-Z]")) {
            if(upper) {
                out += tmp.toUpperCase();
                upper = false;
            } else {
                out += tmp.toLowerCase();
            }
        } else if(tmp.matches("-")) {
            upper = true;
        }
    }
    return out;
}
```

Call Tree - Method	Time [%]	Time	Invocations
All threads		5014 ms (100%)	1
main		5014 ms (100%)	1
test.Main.**main** (String[])		5014 ms (100%)	1
cz.vutbr.web.css.Engine.**parse** (java.io.File)		2142 ms (42,7%)	1
cz.vutbr.web.domAssign.Controller.**process** (org.w3c.dom.Document, cz.vutbr.web.css.StyleSI		1976 ms (39,4%)	1
cz.vutbr.web.domAssign.Analyzer.**analyze** (org.w3c.dom.Document, cz.vutbr.web.css.Style		1042 ms (20,8%)	1
cz.vutbr.web.domAssign.NodeData.**putDeclaration** (cz.vutbr.web.css.Declaration)		671 ms (13,4%)	3627
cz.vutbr.web.domAssign.NodeData.**methodName** (String)		242 ms (4,8%)	3627
cz.vutbr.web.domAssign.NodeData.**processBorder** (cz.vutbr.web.css.Declaration)		131 ms (2,6%)	56
Self time		131 ms (2,6%)	3627
cz.vutbr.web.css.Declaration.**toString** (int)		89.4 ms (1,8%)	3627
cz.vutbr.web.domAssign.NodeData.**processBackground** (cz.vutbr.web.css.Declaration)		17.4 ms (0,3%)	243

Z analýzy je patrné, že metoda `methodName()` byla volána celkem 3627x a v celkovém čase běhu zabrala 242ms.

Jako první a evidentní problém je velmi časté vytváření objektu `tmp` třídy `String`. Ten se vytváří při každém průchodu cyklu, což je jistě velmi náročné a především zbytečné. Pokud má každá vlastnost průměrnou délku 20 znaků, znamená to, že se musí tento objekt vytvořit a následně odstranit 72540x. Následovala tedy první úprava:

```
    . . .
    String tmp = "";
    for(byte chr : b) {
        tmp = "" + (char)chr;
    . . .
```

Call Tree - Method	Time [%]	Time	Invocations
All threads		4928 ms (100%)	1
main		4928 ms (100%)	1
test.Main.**main** (String[])		4928 ms (100%)	1
cz.vutbr.web.css.Engine.**parse** (java.io.File)		2181 ms (44,3%)	1
cz.vutbr.web.domAssign.Controller.**process** (org.w3c.dom.Document, cz.vutbr.web.css.StyleSheet, String)		1787 ms (36,3%)	1
cz.vutbr.web.domAssign.Analyzer.**analyze** (org.w3c.dom.Document, cz.vutbr.web.css.StyleSheet, String)		904 ms (18,4%)	1
cz.vutbr.web.domAssign.NodeData.**putDeclaration** (cz.vutbr.web.css.Declaration)		634 ms (12,9%)	3627
Self time		167 ms (3,4%)	3627
cz.vutbr.web.domAssign.NodeData.**methodName** (String)		146 ms (3%)	3627
cz.vutbr.web.domAssign.NodeData.**processBorder** (cz.vutbr.web.css.Declaration)		127 ms (2,6%)	56
cz.vutbr.web.css.Declaration.**toString** (int)		91.5 ms (1,9%)	3627
cz.vutbr.web.domAssign.NodeData.**processMargin** (cz.vutbr.web.css.Declaration)		40.8 ms (0,8%)	170

Objekt byl deklarován jednou před cyklem a dále již byla pouze měněna jeho hodnota. Z analýzy je patrno, že se průběh zrychlil na 146ms.

Kód ale obsahuje stále velké množství obecných funkcí pracujících s řetězci. I toto lze odstranit a dovést optimalizaci do úspěšného konce:

```
private String methodName(String property) {
    byte[] b = property.getBytes();
    StringBuffer out = new StringBuffer("process");
    boolean upper = true;
    for(byte chr : b) {
        if((chr > 96 && chr < 123) || (chr > 64 && chr < 91)) {
            if(upper) {
                if(chr > 96) {
                    chr -= 32;
                }
                out.append((char)chr);
                upper = false;
            } else {
                if(chr < 91) {
                    chr += 32;
                }
                out.append((char)chr);
            }
        } else if(chr == 45) {
            upper = true;
        }
    }
    return out.toString();
}
```

Call Tree - Method	Time [%]	Time	Invocations
All threads		4799 ms (100%)	1
main		4799 ms (100%)	1
test.Main.**main** (String[])		4799 ms (100%)	1
cz.vutbr.web.css.Engine.**parse** (java.io.File)		2111 ms (44%)	1
cz.vutbr.web.domAssign.Controller.**process** (org.w3c.dom.Document, cz.vutbr.web.css.StyleSheet, String)		1841 ms (38,4%)	1
cz.vutbr.web.domAssign.Analyzer.**analyze** (org.w3c.dom.Document, cz.vutbr.web.css.StyleSheet, String)		1025 ms (21,4%)	1
cz.vutbr.web.domAssign.NodeData.**putDeclaration** (cz.vutbr.web.css.Declaration)		476 ms (9,9%)	3627
Self time		164 ms (3,4%)	3627
cz.vutbr.web.domAssign.NodeData.**processBorder** (cz.vutbr.web.css.Declaration)		128 ms (2,7%)	56
cz.vutbr.web.css.Declaration.**toString** (int)		78.9 ms (1,6%)	3627
cz.vutbr.web.domAssign.NodeData.**methodName** (String)		32.0 ms (0,7%)	3627
cz.vutbr.web.domAssign.NodeData.**processPadding** (cz.vutbr.web.css.Declaration)		14.3 ms (0,3%)	264

Po této úpravě je nárůst výkonu ještě více patrný. Doba zpracování se snížila téměř desetinásobně, což je jistě již nezanedbatelné.

Z procesu optimalizace je zřejmé, že každý algoritmus je možné napsat více způsoby, kdy se jedná vždy o nějakou rovnováhu mezi vysokou abstrakcí a rychlostí. Kritické funkce je možné napsat tak, aby jejich provedení bylo co nejrychlejší, nevýhoda je ovšem zápis ne nepodobný kódu v jazyce C. Na druhé straně více abstraktní kód plně využívající možností daného jazyka umožní snazší pochopení ostatními programátory a rychlejší úpravy.

V tomto duchu byl pak zdrojový kód knihovny pro zpracování předpisů CSS i algoritmu ohodnocení stromu dokumentu zrevidován a na místech, kde to bylo nutné, byly provedeny změny. Kritické, tedy často používané, metody byly upraveny tak, aby bylo jejich zpracování co nejvíce šetrné k systémovým zdrojům, naproti tomu obslužný kód zůstal maximálně abstraktní a přehledný.

Pozn.: Předchozí měření bylo provedeno při ohodnocení stromu dokumentu předpisem CSS. Aby se eliminovala statistická chyba, bylo provedeno měření vždy několik, vynechány extrémy a vybrán ten nejčastěji objevující se výsledek. Ačkoli ani takto nelze zaručit absolutní přesnost, výsledky mají svoji vypovídají hodnotu – zvýšení rychlosti je evidentní.

9 DEMO aplikace

Knihovna pro práci s předpisy CSS i algoritmus ohodnocení stromu dokumentu nemohou být spouštěny jako samostatné aplikace - neobsahují žádnou vstupní metodu, typicky nazývanou `Main()`. Jejich primární účel je být součástí rozsáhlejší aplikace a zajišťovat pouze funkcionalitu práce s předpisy CSS.

Demonstraci a testování funkčnosti je tedy třeba provádět pomocí DEMO aplikace, která může být spouštěna jako samostatná aplikace a zajišťuje vstup, výstup a komunikaci s uživatelem.

9.1 Návrh struktury

DEMO aplikace se bude skládat ze tří hlavních balíčků a jedné podpůrné knihovny:

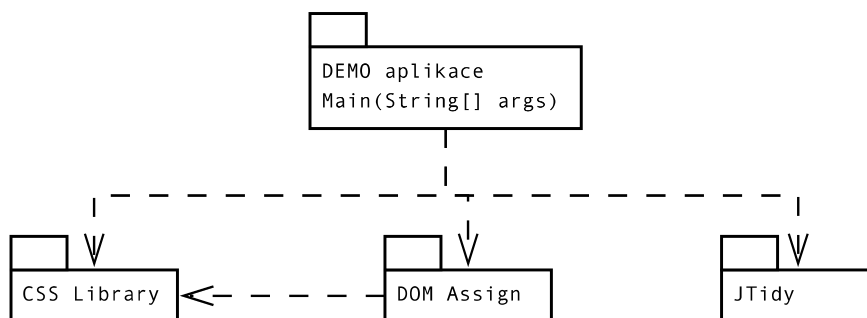


Diagram 8: Balíčky použité pro DEMO aplikaci

Z diagramu je patrné, že budou využity dva balíčky vypracované v rámci této práce (popsané v předchozích kapitolách) ke kterým přibude třetí, zajišťující vstup, výstup a jejich kooperaci.

Posledním použitým balíčkem bude pak nástroj JTidy, sloužící pro načtení vstupních (X)HTML dokumentů.

9.2 JTidy

JTidy je nástroj primárně určený pro kontrolu zdrojových kódů (X)HTML dokumentů a umožňující jejich přeformátování do vizuálně lepší podoby (ve smyslu zdrojového kódu). Navíc JTidy poskytuje DOM rozhraní k právě zpracovávanému dokumentu, což umožňuje využít jej jako univerzální parser (X)HTML dokumentů.

V tomto projektu bude JTidy využit právě jako nástroj pro načítání HTML dokumentů sloužících jako vstup při ohodnocení předpisem CSS. Jelikož JTidy vytváří DOM model přímo z tříd implementujících rozhraní `org.w3c.dom`, nebude třeba dalších konverzí.

9.3 Kompilace

Nutným předpokladem pro úspěšnou kompilaci je zajištění dostupnosti Java SDK 1.5.0 a Apache Ant 1.5.1 (nebo verze vyšší) v systému. Celý proces je pak řízen pomocí úkolů (tasks) definovaných v souboru build.xml.

Překlad lze zahájit příkazem:

```
svercl@local> ant all
```

v hlavním adresáři.

Je-li třeba přeložené soubory odstranit, lze to provést pomocí

```
svercl@local> ant clean
```

spuštění DEMO aplikace pak probíhá pomocí standardního příkazu

```
svercl@local> java -jar demo.jar
```

Při překladu jsou vytvořeny tři balíčky: `cssLib.jar`, `domAssign.jar` a `demo.jar`. Přirozeně balíček `cssLib.jar` obsahuje knihovnu pro práci s předpisy CSS, `domAssign.jar` umožňuje provádět operace ohodnocení stromu dokumentu. Obě tyto knihovny mohou být využity jako součásti většího projektu, kde budou zajišťovat operace pracující s předpisy CSS. Balíček `demo.jar` pak obsahuje samotnou DEMO aplikaci.

9.4 Ovládání

Ovládání DEMO aplikace je prováděno výhradně z příkazové řádky. Po spuštění či při chybném zadání parametrů je zobrazena následující nápověda:

```
Arguments:
-css <stylesheet> validate
-css <stylesheet> -html <html_doc> [-out <out_file>] assign_text
-css <stylesheet> -html <html_doc> [-out <out_file>] assign_html
-css <stylesheet> -html <html_doc> [-out <out_file>] assign_inline
```

Je patrné, že k dispozici je provedení čtyř různých operací. Jako argument je předán předpis CSS, dokument ve formátu HTML či případně i jméno výstupního souboru. Posledním argumentem je vždy název prováděné operace.

První operací je validace syntaxe předpisu CSS. To je provedeno jednoduše pomocí jeho načtení knihovnou a v případě že nedojde k žádným problémům, je vypsána hláška „Syntax OK“. V případě opačném je vypsáno chybové hlášení včetně informace o řádku a symbolech, které byly očekávány.

Zbylé tři operace provádějí ohodnocení stromu dokumentu. Jejími vstupy jsou přirozeně předpis CSS a HTML dokument, výstup pak záleží na zvolené metodě. Výstupní formát dat odpovídá metodám prezentovaným v kapitole 7.6. Dojde-li k chybě při načítání předpisu CSS, je chybové hlášení stejné jako při operaci validace, při špatném formátu souboru HTML dokumentu vypisuje varování či chyby přímo nástroj JTidy.

9.5 Testovací data

V adresáři `data` jsou připraveny tři webové stránky na kterých je možné si funkčnost ověřit. První, simple, je složena z několika elementů a vzhled je definován pomocí jednoduchého předpisu CSS. Ostatní dvě pak obsahují snímky větších veřejných webů na internetu – webové stránky organizace W3C a zpravodajský portál digiarena.cz.

Nejzajímavějším testováním je provedení ohodnocení stromu dokumentu a výpis zpět do formátu HTML (kdy styly jsou definovány pomocí inline notace v atributu `style=“. . .“`). Takto lze pak porovnávat vzhled stránky před a po ohodnocení a zjistit tak, zda byl proces úspěšný.

Dlužno podotknout, že stránky webu digiarena.cz musely být před procesem ohodnocení mírně „pročištěny“, jelikož obsahovaly spoustu nestandardních prvků. Nástroj JTidy si s takovým kódem sice dokáže poradit, nicméně některé elementy (například ty prázdné) automaticky zahodí a vzhled stránky by se pak mírně lišil.

Experiment lze provést například takto:

```
svercl@local> java -jar dist/demo.jar
-css data/digiarena/common.css
-html data/digiarena/default.html
-out data/digiarena/out.html
assign_inline
```

Tímto vznikne nový dokument out.html, vizuálně stejný jako zdroj (default.html), definice jeho vzhledu je ovšem provedena za pomoci inline stylů.

10 Závěr

V práci se podařilo dosáhnout všech cílů, které byly dány zadáním a popsány v úvodní kapitole. Byly vytvořeny dvě knihovny kopírující nejmodernější standardy a trendy v oblasti prezentování dokumentů na internetu. Textová část provede čtenáře od teoretického úvodu přes vysvětlení použitých programátorských technik až po závěrečnou fázi věnovanou optimalizaci.

Při implementaci knihovny pro práci s předpisy CSS velice pomohlo zorientování se v dané problematice a výběr vhodných nástrojů. Gramatika jazyka CSS je prezentována na webu organizace W3C, tuto stačilo pouze upravit a vygenerovat z ní pomocí nástroje JavaCC plně funkční syntaktický analyzátor. Převod dat ze syntaktického stromu do formy objektové reprezentace je pak triviální a bezproblémový. Díky tomuto postupu bylo dosaženo poměrně dobré stability a spolehlivosti současně s maximálním dodržením standardů CSS 2.1.

Práce na knihovně pro ohodnocení stromu dokumentu spočívala v podstatě na vyřešení dvou hlavních problémů, kterými jsou porovnání elementu se selektorem a extrakce dat z deklarací. U algoritmu pro vyhledávání odpovídajících selektorů k elementům bylo také nutné implementovat nějakou formu optimalizace, která by obešla nutnost porovnání všech elementů dokumentu se všemi selektory v předpisu CSS. Toto se podařilo úspěšně vyřešit a optimalizace poměrně značně zvýšila rychlost zpracování, převážně u rozsáhlejších dokumentů. Práce na ostatních částech knihovny již zahrnovala víceméně pouze obslužný kód, který řídí tok dat a vrací výsledek.

Po úspěšné implementaci následovala část optimalizace zdrojového kódu tak, aby se minimalizovala spotřeba systémových prostředků. Byl využit nástroj Profiler z vývojového prostředí NetBeans, díky kterému se podařilo odhalit výkonově problémové části kódu a provést jejich korekci. Bohužel bylo také zjištěno, že poměrně velkou část výpočetního času (téměř polovinu) zkonzumuje při načítání předpisu CSS automaticky vygenerovaný syntaktický analyzátor. Velkou měrou se také na zpracování podílí nástroj JTidy, který zajišťuje načtení a export (X)HTML dokumentů.

Pro implementaci bylo využito jazyka JAVA, konkrétně ve verzi 1.5. Snahou bylo vytvořit maximálně přehledný a jasný kód, doplněný vysvětlujícími komentáři. Ačkoli původně byly všechny komentáře napsány v českém jazyce, u knihovny pro zpracování předpisů CSS bylo nutné převést je do jazyka anglického. Tato úprava byla provedena z důvodu budoucího zveřejnění knihovny na internetu umožňující případné studium či použití programátory z celého světa.

10.1 Vlastní přínos

Tento projekt si kladl za cíl vytvoření několika nástrojů, které budou usnadňovat práci s předpisy CSS v prostředí jazyka JAVA. Většina kódu z této práce bude s největší pravděpodobností umístěna pod svobodnou licencí na internetu.

První takovou částí je definice gramatiky pro nástroj JavaCC. Autoři tohoto nástroje mají na svém webu k dispozici velké množství gramatik pro různé jazyky, předpisy CSS bohužel chybí. Rád bych tuto situaci změnil a gramatiku poskytl k volnému užití.

Vytvořené knihovny si kladou za cíl doplnit portfolio aktuálně dostupných nástrojů pro práci s předpisy CSS v jazyce JAVA. Na internetu jsou sice dostupné některé podobné projekty, nicméně jejich neaktuálnost či problémovost stála právě za vznikem zadání této diplomové práce. Pokud to bude jen trochu možné, rád bych svůj projekt i nadále ve svém volném čase zdokonaloval a odrážel do něho aktuální vývoj jazyka CSS.

Literatura

[1] BOS, B.; ÇELIK, T.; HICKSON, I.; LIE, H. W. *Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification* [online]. W3C, červenec 2007 [cit. 28. 12. 2007]. Přístup z URL http://www.w3.org/TR/CSS21/

[2] WILSON, C.; LE HÉGARET, P.; APPARAO, V. *Document Object Model CSS* [online]. W3C, listopad 2000 [cit. 8. 4. 2008]. Přístup z URL http://www.w3.org/TR/DOM-Level-2-Style/css.html

[3] BOS, B.; LE HÉGARET, P. SAC: *Simple API for CSS* [online]. W3C, červenec 2000 [cit. 8. 4. 2008]. Přístup z URL http://www.w3.org/TR/SAC/

[4] Organizace IANA: *CHARACTER SETS* [online]. květen 2007 [cit. 5. 1. 2008]. Přístup z URL http://www.iana.org/assignments/character-sets

[5] BERNERS-LEE T. et al. *Uniform Resource Identifier (URI): Generic Syntax* [online]. Leden 2005 [cit. 20. 1. 2008]. Přístup z URL http://www.ietf.org/rfc/rfc3986

[6] SPELL, B. *Java Programujeme profesionálně*. Praha: Computer Press, 2002. ISBN: 80-7226-667-5

[7] MEYER, E. *Eric Meyer o CSS - kompletní průvodce*. Praha: Zoner Press, 2007. ISBN: 978-8086815-64-0

Seznam diagramů

Diagram 1: Příklad objektové struktury (snapshot)....................21
Diagram 2: Struktura objektů po načtení předpisu parserem....................26
Diagram 3: Příklad převodu stromu do objektové reprezentace definované rozhraním....................30
Diagram 4: Třídy tvořící implementaci ohodnocení stromu dokumentu....................35
Diagram 5: Diagram sekvence ukazující návaznosti jednotlivých činností....................36
Diagram 6: Třída pro uchování pravidel včetně jejich pořadového čísla....................38
Diagram 7: Příklad optimalizace výběru pravidel pro konkrétní element....................39
Diagram 8: Balíčky použité pro DEMO aplikaci....................47

Příloha A – Gramatika předpisu CSS

```
/* Direktiva pro vynechání komentářů zapsaných pomocí lomítko-hvězdička */
SKIP :
{
  <SLASH_STAR_COMMENT: "/*" (~["*"])* ("*")+ (~["/","*"] (~["*"])* ("*")+)* "/" >
}

/* Popis terminálních symbolů pro lexikální analyzátor */
TOKEN :
{
  <#h : ["0"-"9", "a"-"f"]> |
  <#nonascii : ["\200"-"\377"]> |
  <#unicode : "\\" <h> (<h>)? (<h>)? (<h>)? (<h>)? (<h>)? ("\r\n"|[" ", "\t", "\r",
              "\n", "\f"])? > |
  <#escape : <unicode> | "\\" ~["\r", "\n", "\f", "0"-"9", "a"-"f"]> |
  <#nmstart : ["_", "a"-"z"] | <nonascii> | <escape>> |
  <#nmchar : ["_", "a"-"z", "0"-"9", "-"] | <nonascii> | <escape>> |
  <#string1 : "\"" (~["\n", "\r", "\f", "\""] | "\\" <nl> | <escape>)* "\""> |
  <#string2 : "'" (~["\n", "\r", "\f", "'"] | "\\" <nl> | <escape>)* "'"> |
  <#invalid1 : "\""(~["\n", "\r", "\f", "\""] | "\\" <nl> | <escape>)* > |
  <#invalid2  : "'"(~["\n", "\r", "\f", "'"] | "\\" <nl> | <escape>)* > |
  <#comment : "/*" (~["*"])* ("*")+ (~["/","*"] (~["*"])* ("*")+)* "/" > |
  <#ident : ("-")? <nmstart> (<nmchar>)* > |
  <#name : (<nmchar>)+ > |
  <#num : (["0"-"9"])+ | (["0"-"9"])* "." (["0"-"9"])+ > |
  <#string : <string1>|<string2> > |
  <#invalid : <invalid1>|<invalid2> > |
  <#url : ([";", "!", "#", "$", "%", "&", "*", "-", "~", ".", "/", ":", ",", "=",
           "?"] | <nmchar> )* > |
  <#ss : ([" ", "\t", "\r", "\n", "\f"])+ > |
  <#w : (<ss>)? > |
  <#nl : "\n" | "\r\n" | "\r" | "\f"> |

  <#A : "a" | ("\\0")? ("\\0")? ("\\0")? ("\\0")? ("41" | "61")  ("\r\n" | [" ",
        "\t", "\r", "\n", "\f"]) ? > |
  <#C : "c" | ("\\0")? ("\\0")? ("\\0")? ("\\0")? ("43" | "63")  ("\r\n" | [" ",
        "\t", "\r", "\n", "\f"]) ? > |
  <#D : "d" | ("\\0")? ("\\0")? ("\\0")? ("\\0")? ("44" | "64")  ("\r\n" | [" ",
        "\t", "\r", "\n", "\f"]) ? > |
  <#E : "e" | ("\\0")? ("\\0")? ("\\0")? ("\\0")? ("45" | "65")  ("\r\n" | [" ",
        "\t", "\r", "\n", "\f"]) ? > |
  <#G : "g" | ("\\0")? ("\\0")? ("\\0")? ("\\0")? ("47" | "67")  ("\r\n" | [" ",
        "\t", "\r", "\n", "\f"]) ? | "\\g"> |
  <#H : "h" | ("\\0")? ("\\0")? ("\\0")? ("\\0")? ("48" | "68")  ("\r\n" | [" ",
        "\t", "\r", "\n", "\f"]) ? | "\\h"> |
  <#I : "i" | ("\\0")? ("\\0")? ("\\0")? ("\\0")? ("49" | "69")  ("\r\n" | [" ",
        "\t", "\r", "\n", "\f"]) ? | "\\i"> |
  <#K : "k" | ("\\0")? ("\\0")? ("\\0")? ("\\0")? ("4b" | "6b")  ("\r\n" | [" ",
        "\t", "\r", "\n", "\f"]) ? | "\\k"> |
  <#M : "m" | ("\\0")? ("\\0")? ("\\0")? ("\\0")? ("4d" | "6d")  ("\r\n" | [" ",
        "\t", "\r", "\n", "\f"]) ? | "\\m"> |
  <#N : "n" | ("\\0")? ("\\0")? ("\\0")? ("\\0")? ("4e" | "6e")  ("\r\n" | [" ",
        "\t", "\r", "\n", "\f"]) ? | "\\n"> |
  <#O : "o" | ("\\0")? ("\\0")? ("\\0")? ("\\0")? ("51" | "71")  ("\r\n" | [" ",
        "\t", "\r", "\n", "\f"]) ? | "\\o"> |
  <#P : "p" | ("\\0")? ("\\0")? ("\\0")? ("\\0")? ("50" | "70")  ("\r\n" | [" ",
        "\t", "\r", "\n", "\f"]) ? | "\\p"> |
  <#R : "r" | ("\\0")? ("\\0")? ("\\0")? ("\\0")? ("52" | "72")  ("\r\n" | [" ",
        "\t", "\r", "\n", "\f"]) ? | "\\r"> |
  <#S : "s" | ("\\0")? ("\\0")? ("\\0")? ("\\0")? ("53" | "73")  ("\r\n" | [" ",
        "\t", "\r", "\n", "\f"]) ? | "\\s"> |
  <#T : "t" | ("\\0")? ("\\0")? ("\\0")? ("\\0")? ("54" | "74")  ("\r\n" | [" ",
        "\t", "\r", "\n", "\f"]) ? | "\\t"> |
```

```
  <#X : "x" | ("\\0")? ("\\0")? ("\\0")? ("\\0")? ("58" | "78")  ("\r\n" | [" ",
        "\t", "\r", "\n", "\f"]) ? | "\\x"> |
  <#Z : "z" | ("\\0")? ("\\0")? ("\\0")? ("\\0")? ("5a" | "7a")  ("\r\n" | [" ",
        "\t", "\r", "\n", "\f"]) ? | "\\z"> |
  <BLANK : <ss> > |
  <CDO : "<!--" > |
  <CDC : "-->" > |
  <EQUAL : "=" > |
  <INCLUDES : "~=" > |
  <DASHMATCH : "|=" > |
  <LBRACE : <w> "{" > |
  <PLUS : <w> "+" > |
  <MINUS : <w> "-" > |
  <GREATER : <w> ">" > |
  <COMMA : <w> "," > |
  <STRING : <string> > |
  <INVALID : <invalid> > |
  <IDENT : <ident> > |
  <HASH : "#" <name> > |
  <IMPORT_SYM : "@" <I> <M> <P> <O> <R> <T> > |
  <PAGE_SYM : "@" <P> <A> <G> <E> > |
  <MEDIA_SYM : "@" <M> <E> <D> <I> <A> > |
  <CHARSET_SYM : "@" <C> <H> <A> <R> <S> <E> <T> > |
  <IMPORTANT_SYM : "!" (<w>|<comment>)* <I> <M> <P> <O> <R> <T> <A> <N> <T> > |

  <EMS : <num> "em" > |
  <EXS : <num> "ex" > |
  <LENGTHPX : <num> "px" > |
  <LENGTHCM : <num> "cm" > |
  <LENGTHMM : <num> "mm" > |
  <LENGTHPT : <num> "pt" > |
  <LENGTHPC : <num> "pc" > |
  <ANGLEDEG : <num> "deg" > |
  <ANGLERAD : <num> "rad" > |
  <ANGLEGRAD : <num> "grad" > |
  <TIMEMS : <num> "ms" > |
  <TIMES : <num> "s" > |
  <FREQHZ : <num> "hz" > |
  <FREQKHZ : <num> "khz" > |

  <DIMENSION : <num> <ident> > |
  <PERCENTAGE : <num> "%" > |
  <NUMBER : <num> > |
  <URI : ("url(" <w> <string> <w> ")") | ("url("<w> <url> <w> ")") > |
  <FUNCTION : <ident> "(" >
}

/* Přepisovací pravidla */
SimpleNode Start() :
{}
{
  stylesheet() <EOF>
  { return jjtThis; }
}

void stylesheet() :
{}
{
  ( charset() )?
  (<BLANK> | <CDO> | <CDC>)* ( import_a() (<BLANK> | <CDO> | <CDC> )* )*
  ( ( ruleset() | media() | page() ) (<BLANK> | <CDO> | <CDC>)* )*
  /* Řešení bez lookahead, bohužel pomalejší */
  /*( ( ruleset() | media() | page() ) [<CDO> (<BLANK> | <CDO> | <CDC>)* | <CDC>
    (<BLANK> | <CDO> | <CDC>)*] )*  */
}
```

```
void import_a() :
{}
{
  <IMPORT_SYM> (<BLANK>)*
  [string() | uri()] (<BLANK>)* ( medium() (<COMMA> (<BLANK>)* medium())* )?
  ";" (<BLANK>)*
}

void media() :
{}
{
  <MEDIA_SYM> (<BLANK>)* medium() ( <COMMA> (<BLANK>)* medium() )* <LBRACE>
  (<BLANK>)* (ruleset())* "}" (<BLANK>)*
}

void medium() :
{}
{
  ident() (<BLANK>)*
}

void page() :
{}
{
  <PAGE_SYM> (<BLANK>)* (pseudo_page())? (<BLANK>)*
  <LBRACE> (<BLANK>)* declaration() ( ";" (<BLANK>)* declaration() )*
  "}" (<BLANK>)*
}

void pseudo_page() :
{}
{
  ":" ident()
}

void operator() :
{}
{
  ( slash() (<BLANK>)* | comma() (<BLANK>)* )?
}

void combinator() :
{}
{
  plus() (<BLANK>)* | greater() (<BLANK>)* | <BLANK>
}

void unary_operator() :
{}
{
  minus() | plus()
}

void property() :
{}
{
  ident() (<BLANK>)*
}

void ruleset() :
{}
{
  selector() ( <COMMA> (<BLANK>)* selector() )*
  <LBRACE> (<BLANK>)* declaration() ( ";" (<BLANK>)* declaration() )*
  "}" (<BLANK>)*
}
```

```
void selector() :
{}
{
  simple_selector() ( combinator() simple_selector() )*
}

void simple_selector() :
{}
{
  element_name() ( hash() | class_a() | attrib() | pseudo() )*
  | ( hash() | class_a() | attrib() | pseudo() )+
}

void class_a() :
{}
{
  "." ident()
}

void element_name() :
{}
{
  ident() | "*"
}

void attrib() :
{}
{
  "[" (<BLANK>)* ident() (<BLANK>)* [ (equal() | includes() | dashmatch() )
  (<BLANK>)*
  ( ident() | string() ) (<BLANK>)* ] "]"
}

void pseudo() :
{}
{
  ":" ( ident() | pfunction() )
}

void pfunction() :
{}
{
  function_begin() (<BLANK>)* [ident()] (<BLANK>)* ")"
}

void declaration() :
{}
{
  (property() ":" (<BLANK>)* expr() (prio())?)?
}

void prio() :
{}
{
  <IMPORTANT_SYM> (<BLANK>)*
}

void expr() :
{}
{
  term() ( operator() term() )*
}
```

```
void term() :
{}
{
  [ unary_operator() ](
  number() (<BLANK>)* | percentage() (<BLANK>)* | lengthpx() (<BLANK>)* |
  lengthcm() (<BLANK>)* | lengthmm() (<BLANK>)* | lengthpt() (<BLANK>)* |
  lengthpc() (<BLANK>)* | ems() (<BLANK>)* | exs() (<BLANK>)* |
  angledeg() (<BLANK>)* | anglerad() (<BLANK>)* | anglegrad() (<BLANK>)* |
  timems() (<BLANK>)* | times() (<BLANK>)* | freqhz() (<BLANK>)* |
  freqkhz() (<BLANK>)* ) | string() (<BLANK>)* | ident() (<BLANK>)* |
  uri() (<BLANK>)* | hexcolor() | function()
}

void function() :
{}
{
  function_begin() (<BLANK>)* expr() ")" (<BLANK>)*
}

void hexcolor() :
{}
{
  hash() (<BLANK>)*
}

void charset() :
{}
{
  <CHARSET_SYM> string() ";"
}

void equal() :
{}
{
  <EQUAL>
}

void includes() :
{}
{
  <INCLUDES>
}

void dashmatch() :
{}
{
  <DASHMATCH>
}

void comma() :
{}
{
  <COMMA>
}

void slash() :
{}
{
  "/"
}

/* Výpis terminálních symbolů, u kterých je třeba znát jejich obsah (tzv. image) */
void hash() :       { Token t; } {t=<HASH>        {jjtThis.setImage(t.image);}}
void ident() :      { Token t; } {t=<IDENT>       {jjtThis.setImage(t.image);}}
void string() :     { Token t; } {t=<STRING>      {jjtThis.setImage(t.image);}}
void uri() :        { Token t; } {t=<URI>         {jjtThis.setImage(t.image);}}
void number() :     { Token t; } {t=<NUMBER>      {jjtThis.setImage(t.image);}}
void percentage() : { Token t; } {t=<PERCENTAGE>  {jjtThis.setImage(t.image);}}
void lengthpx() :   { Token t; } {t=<LENGTHPX>    {jjtThis.setImage(t.image);}}
void lengthcm() :   { Token t; } {t=<LENGTHCM>    {jjtThis.setImage(t.image);}}
void lengthmm() :   { Token t; } {t=<LENGTHMM>    {jjtThis.setImage(t.image);}}
```

```
void lengthpt() :    { Token t; } {t=<LENGTHPT>    {jjtThis.setImage(t.image);}}
void lengthpc() :    { Token t; } {t=<LENGTHPC>    {jjtThis.setImage(t.image);}}
void ems() :         { Token t; } {t=<EMS>         {jjtThis.setImage(t.image);}}
void exs() :         { Token t; } {t=<EXS>         {jjtThis.setImage(t.image);}}
void angledeg() :    { Token t; } {t=<ANGLEDEG>    {jjtThis.setImage(t.image);}}
void anglerad() :    { Token t; } {t=<ANGLERAD>    {jjtThis.setImage(t.image);}}
void anglegrad() :   { Token t; } {t=<ANGLEGRAD>   {jjtThis.setImage(t.image);}}
void timems() :      { Token t; } {t=<TIMEMS>      {jjtThis.setImage(t.image);}}
void times() :       { Token t; } {t=<TIMES>       {jjtThis.setImage(t.image);}}
void freqhz() :      { Token t; } {t=<FREQHZ>      {jjtThis.setImage(t.image);}}
void freqkhz() :     { Token t; } {t=<FREQKHZ>     {jjtThis.setImage(t.image);}}
void plus() :        { Token t; } {t=<PLUS>        {jjtThis.setImage(t.image);}}
void minus() :       { Token t; } {t=<MINUS>       {jjtThis.setImage(t.image);}}
void greater():      { Token t; } {t=<GREATER>     {jjtThis.setImage(t.image);}}
void function_begin() : { Token t; } {t=<FUNCTION> {jjtThis.setImage(t.image);}}
```