Katedra informatiky
Přírodovědecká fakulta
Univerzita Palackého v Olomouci

DIPLOMOVÁ PRÁCE

Refaktorizace CSS kódu

2023

Jaromír Hradil

Vedoucí práce:
RNDr. Martin Trnečka, Ph.D.

Studijní program: Aplikovaná informatika,
Specializace: Vývoj software

Bibliografické údaje

Autor:	Jaromír Hradil
Název práce:	Refaktorizace CSS kódu
Typ práce:	diplomová práce
Pracoviště:	Katedra informatiky, Přírodovědecká fakulta, Univerzita Palackého v Olomouci
Rok obhajoby:	2023
Studijní program:	Aplikovaná informatika, Specializace: Vývoj software
Vedoucí práce:	RNDr. Martin Trnečka, Ph.D.
Počet stran:	57
Přílohy:	elektronická data v úložišti katedry informatiky
Jazyk práce:	český

Bibliographic info

Author:	Jaromír Hradil
Title:	CSS code refactoring
Thesis type:	master thesis
Department:	Department of Computer Science, Faculty of Science, Palacký University Olomouc
Year of defense:	2023
Study program:	Applied Computer Science, Specialization: Software Development
Supervisor:	RNDr. Martin Trnečka, Ph.D.
Page count:	57
Supplements:	electronic data in the storage of department of computer science
Thesis language:	Czech

Anotace

Tato práce se zabývá zkoumáním možností refaktorizace CSS kódu. Ta je prováděna pomocí několik různých technik, mezi které patří: utility třídy, shlukování CSS hodnot a vlastností a poznatky z formální konceptuální analýzy.

Synopsis

This thesis explores possibilities of CSS code refactoring. Refactoring is realized using several different techniques, including: utility classes, clustering of CSS properties and values and observations from formal concept analysis.

Klíčová slova: CSS, CSS refaktorizace, utility třídy, formální konceptuální analýza, atomické CSS

Keywords: CSS, CSS refactoring, utility classes, formal concept analysis, atomic CSS

Rád bych poděkoval mému vedoucímu doktoru Martinovi Trnečkovi za jeho cenné rady a postřehy.

Odevzdáním tohoto textu jeho autor/ka místopřísežně prohlašuje, že celou práci včetně příloh vypracoval/a samostatně a za použití pouze zdrojů citovaných v textu práce a uvedených v seznamu literatury.

Obsah

1 Úvod **7**

2 Teoretická část **8**
2.1 HTML . . . 8
2.2 CSS . . . 10
2.2.1 Selektory a jejich typy . . . 10
2.2.2 Deklarace . . . 12
2.2.3 Vkládání CSS do HTML . . . 13
2.2.4 Způsob aplikace pravidel . . . 14
2.2.5 At-pravidla . . . 17
2.2.6 Atomické CSS . . . 17
2.3 DOM . . . 18
2.4 Formální konceptuální analýza . . . 18

3 Použité technologie **20**
3.1 Jazyky JavaScript a TypeScript . . . 20
3.2 Puppeteer . . . 20
3.3 CSSTree . . . 21
3.3.1 @bramus/specificity . . . 21
3.3.2 @projectwallace/css-analyzer . . . 21
3.3.3 @projectwallace/css-code-quality . . . 22
3.3.4 csstree-validator . . . 22
3.4 js-beautify . . . 22
3.5 Color.js . . . 22
3.6 MDN Data . . . 23
3.7 Yargs . . . 23
3.8 GNU Octave . . . 23
3.9 GNU Wget . . . 23

4 Návrh **24**
4.1 Celkový náhled a spouštění . . . 24
4.2 Načtení a extrakce dat . . . 28
4.3 Zaznamenání vazeb . . . 30
4.4 Roztřídění a filtrace deklarací podle specifičností . . . 32
4.5 Zpracování finálních deklarací . . . 34
4.6 Optimilizace . . . 38
4.6.1 Shlukování CSS vlastností . . . 38
4.6.2 Shlukování CSS hodnot . . . 40
4.6.3 Optimalizace pomocí formálního konceptu . . . 41
4.7 Generování stylesheetu . . . 42
4.8 Vložení nového stylu do stránky . . . 44

5 Experimentální část a výsledky **45**
5.1 Tvorba datasetu 45
5.2 Experimenty 46

Závěr **51**

Conclusions **52**

A Obsah odevzdaných elektronických dat **53**

B Ukázky vizuální podoby stránky z datasetu a vygenerovaných výsledků **54**

Literatura **56**

1 Úvod

Jazyk CSS je jedním z klíčových elementů pro tvorbu webových stránek. Ačkoliv je většina jeho syntaxe a princip fungování jednoduchý, není snadné psát přehledný CSS kód.

Každý vývojář má své preference ohledně psaní CSS (celková organizace, způsob zápisu, ...). Tomuto faktu nenapomáhá ani skutečnost, že CSS nemá jednotnou podporu napříč prohlížeči (implementace CSS v jednom prohlížeči nemusí kompletně odpovídat implementaci v druhém) [1]. V některých případech se tak musí CSS kód přizpůsobit na cílené prohlížeče. Tyto faktory mají za následek to, že výsledný CSS kód bývá nepřehledný a špatně udržitelný.

Ve snaze zabránit tomuto jevu bylo vytvořeno mnoho postupů a nástrojů. Mezi populární postupy (*metodiky*) pro zajištění lepší čitelnosti a organizace CSS dle [1] patří: *BEM*, *ITCSS*, *atomické CSS*, nebo používání *CSS layer*.

Nástroje pro tvorbu CSS poskytují nad ním určitou úroveň abstrakce a celkově rozšiřují možnosti jeho zápisu. V [1] a [2] je uvedeno několik různých typů nástrojů: *preprocessor (SASS,*[1] *Less,*[2] *...)*, *postprocessor (PostCSS* [3]*)* či *CSS framework (Tailwind* [4]*)*.

Předmětem této práce je prozkoumat možnosti zlepšení organizace a čitelnosti CSS kódu pomocí automatizované refaktorizace. Tato refaktorizace je zaměřena na *statické webové stránky* [1]. Jde o typ stránky, která je doručena do prohlížeče jako jeden celek a není nutné zasílat další požadavky pro doplnění jejího obsahu. Ze statických stránek jsou uvažovány hlavně ty, co spoléhají nejvíce na funkcionalitu kombinace HTML a CSS. Je to z důvodu, aby se dal maximálně zachytit *aktuální stav* webové stránky (získání co nejvíce informací o aplikovaných CSS stylech). Pokud by stránka průběžně měnila svoji podobu, nebylo by možné plně zjistit bez simulace konkrétního scénáře (způsobující změnu vizualizace), zda budou příslušné styly vůbec použity. I CSS umožňuje dynamicky měnit vizualizaci stránky. Ze zmíněného důvodu se však uvažuje pouze malá podmnožina této funkcionality.

Ze zachyceného stavu stránky se získájí aplikovaná CSS pravidla. Ty jsou následně analyzována a podle nich je vygenerován nový (refaktorizovaný) styl. Tento styl je vytvořen pomocí aplikace refaktorizačních technik, které využívají tradiční (vycházející z CSS metodiky) i více experimentální přístup.

Je také je kladen důraz na to, aby se při vytváření refaktorizovaného stylu co nejvíce zachovala původní podoba webové stránky, a aby redundance výsledného kódu byla co možná nejmenší.

Práce je strukturovaná do kapitol a sekcí, kde je postupně rozebraná hlavní problematika a způsob jejího řešení.

[1] https://sass-lang.com/
[2] https://lesscss.org/
[3] https://postcss.org/
[4] https://tailwindcss.com/

2 Teoretická část

Tato kapitola definuje a popisuje základní termíny z oblasti webových technologií a formální konceptuální analýzy pro lepší pochopení dalších částí této práce. Pokud není uvedeno jinak, bylo v této kapitole čerpáno z [1]. Sekce 2.2.5 vychází z [1] a [3]. V sekci 2.4 bylo čerpáno z [4].

2.1 HTML

HTML (HyperText Markup Language) je značkovací jazyk vyvíjený konsorciem W3C.[5] Je určen pro definování vnitřní struktury obsahu webové stránky. Ta je tvořená pomocí HTML *elementů*.

Ty jsou základním stavebním prvkem syntaxe jazyka HTML. Každý má svou vlastní *značku* (*tag*) a je zapisován ve tvaru: `<tag>`. Značka elementu popisuje povahu jeho obsahu. Existují dva typy HTML elementů:

- *element*;
- *prázdný element*.

Forma zápisu standardního elemetu je:

```
<tag>element_content<tag/>
```

Prázdný element lze zapsat ve dvou formách:

```
<!--First form -->
<tag>
<!--Second form -->
<tag/>
```

Elementy je možné zanořovat, skládat za sebe a vkládat do nich obsah. Tak se vytváří ucelená podoba hierarchie struktury obsahu stránky. Ta má podobu stromu, kde každý element představuje stromový uzel a v něm zanořené elementy jsou jeho potomky. Tato struktura je přístupná pomocí programového rozhraní zvaného **DOM**, které je podrobněji vysvětleno v sekci 2.3. Příklad HTML a jeho struktury je možné vidět v ukázce 1.

[5] https://www.w3.org/

```
<html>
  <head>
      ...
  </head>
  </body>
     <h1>Example</h1>
     <div>
         ...
     </div>
  </body>
  ...
</html>
```

Zdrojový kód 1: Struktura HTML kódu

Každý element může obsahovat *atributy*. Ty v rámci konkrétního elementu mohou být:

- *povinné*;
- *doporučené*;
- *volitelné*.

Atributy a jejich hodnoty mohou přímo ovlivňovat podobu a chování stránky (nastavení zdroje dat, velikost obrázků, ...). Kromě výchozích atributů, HTML také umožňuje definovat vlastní.

Atribut má nejčastěji podobu dvojice ve tvaru: `klíč=hodnota`, kde klíčem je jeho jméno, kterému je přiřazena uvedená hodnota. U atributu bez hodnoty se uvádí pouze jeho jméno. Výjimkou je *binární (booleovský) atribut*. Tomu se přiřazuje hodnota *pravda*, pokud je uvedeno v elementu jeho jméno, jinak *nepravda*. Zápis atributů v HTML elementu by vypadal následovně:

```
<div id=``value1`` class=``value2`` width=100 readonly...>
```

2.2 CSS

CSS (Cascading Style Sheets) je jazyk také vyvíjený W3C konsorciem. Jak bylo zmíněno v sekci 2.1, HTML slouží pro vytvoření popisu struktury obsahu stránky. CSS se využívá pro nastavení její vizuální podoby.

Základním stavebním blokem jazyka CSS jsou **CSS pravidla**. CSS pravidlo tvoří několik částí:

- *selektor*;
- *deklarační blok*.

Selektory jsou podrobněji rozebrány v sekci 2.2.1. Co se týče deklaračního bloku, ten je tvořen jednotlivými **deklaracemi**. Ty jsou popsané v sekci 2.2.2. Způsob zápisu CSS pravidla je možné vidět v ukázce 2.

```
1 selector {
2     /*CSS declaration*/
3     css_property: css_value;
4 }
```

Zdrojový kód 2: CSS pravidlo

Pravidla jako celek pak tvoří popis stylu stránky.

2.2.1 Selektory a jejich typy

Selektor je textový řetězec ve specifickém tvaru, pomocí kterého se identifikuje přesné místo ve struktuře stránky, kde se má aplikovat obsah deklaračního bloku daného pravidla. Je úzce spjat se stromovou strukturou HTML elementů. Vyhledávání podle selektoru vždy začíná z kořenového elementu stránky.

V tabulce 1 je výčet základních typů CSS selektorů a jejich syntaxe. Lze je přímo použít pro výběr elementů nebo pro vytváření pokročilejší CSS konstrukce.

Typ selektor	Syntax
`id`	`#jméno_selektoru`
`class`	`.jméno_selektoru`
`element`	`jméno_elementu`

Tabulka 1: Základní typy CSS selektorů

Element selektor nevyžaduje žádné speciální podmínky ani specifickou syntaxi. Stačí u CSS pravidla použít značku HTML elementu jako selektor. Naproti tomu zbývající typy již vyžadují jisté podmínky pro jejich použití.

Jak bylo již zmíněno v sekci 2.1, každému HTML elementu je možné přiřadit atributy. V kontextu selektorů jsou důležité atributy `class` a `id`. Jejich hodnoty (jednotlivá slova oddělená mezerou) jsou jména selektorů. Na ty se dá odkazovat v CSS pravidlech a vybírat tak elementy, na které se ta pravidla aplikují. Typ selektoru je v tomto případě určen jménem atributu, u kterého je uveden. Podle toho se musí v selektoru CSS pravidla použít příslušná syntaxe z tabulky 1.

Základní typy selektorů je možné rozšířit a vytvářet pokročilejší. Ty umožňují širší formu selekce elementů. Tyto konstrukce jsou popsány v [5]. V rámci kontextu problematiky jsou důležité následující rozšiřující prvky:

- *pseudo-třídy*;
- *pseudo-elementy*;
- *atributové selektory*.

Pseudo-třída slouží pro selekci HTML elementů podle jejich umístění v dokumentu nebo určité vlastnosti. Zapisuje se ve tvaru: `:pseudo-class`. K ostatním selektorům je připojována zprava ve tvaru: `selector:pseudo-class`.

Příklady pseudo-tříd pro selekci podle vlastnosti elementu jsou: `:hover` (přejetí kurzorem myši nad elementem), `:active` (kliknutí kurzorem myši na element), a další.

Pro výběr podle umístění elementů lze použít: `:root` (kořenový element stránky), a podobně.

Existují i pseudo-třídy, které umožňují výběr elementů podle selektorů. Jsou to: `:is()`, `:has()`, `:not()` a `:where()`. Ty přijmají výčet selektorů oddělených čárkou, podle kterých se vybírají elementy. Zápis by vypadal následovně: `:pseudo-class(selector1, selector2, ...)`. Neumožňují selekci **pseudo-elementů** (vysvětleno níže) [5].

Pseudo-třídy `:is()`, `:has()` a `:not()` samy o sobě nemají žádnou *specifičnost* (vysvětlena v sekci 2.2.4). Tu jim určuje jeden z jejich selektorových argumentů, který má nejvyšší specifičnost [5].

`:where()` funguje na stejném principu jako `:is()` (umožňuje výběr jakýchkoliv HTML elementů podle uvedených selektorů). Nepřejímá však specifičnost ze svých argumentů. Má ji vždy nulovou. Díky této vlastnosti a podpoře napříč prohlížeči [6], je `:where()` používána při generování refaktorizovaného stylu (viz sekce 4.7).

Pomocí **pseudo-elementu** se vybírají prvky stránky, které nejsou reprezentované HTML elementy a nelze je proto pomocí jiných selektorů vybrat. Syntaxe pseudo-elementu má podobu: `::pseudo-element`. V kombinaci s dalším selektorem se umisťuje zprava za něj ve tvaru: `selector::pseudo-element`. Nicméně z důvodu zpětné kompatibility, se u vybraných pseudo-elementů (`::after`, `::before`, `::first-letter`, `::first-line`) podporuje i tvar: `:pseudo-element` [5]. Lze vidět, že v minulosti syntaxe pseudo-elementů odpovídala té, kterou nyní využívají pseudo-třídy. Tuto skutečnost nástroj u vybraných pseudo-elementů uvažuje a rozlišuje je o pseudo-tříd.

Příklady pseudo-elementů jsou: `::before` (selekce oblasti před elementem) nebo `::after` (selekce oblasti po elementu).

Atributové selektory, jak napovídá jejich název, slouží k výběru elementů podle jejich atributů a uvedených hodnot. Syntaxe atributových selektorů je následující: `[atribute_name=value]`. Znaménko = lze nahradit dalšími operátory pro jiný způsob výběru. Ty však v rámci této práce příliš důležité.
K ostatním selektorů se připojuje zprava ve formě:
`selector[atribute_name=value]`.

Jednotlivé selektory lze určitým způsobem uspořádat a vytvořit tak bližší kontext pro výběr elementů. V této práci nás budou zajímat hlavně dva způsoby.

Jedním z nich je *seznam selektorů* (*selector list*) [5]. Jeho konstrukce spočívá v naskládání jednotlivých selektorů za sebe. Ty jsou oddělené čárkou, jak je zobrazeno v ukázce 3.

```
1 selector1, selector2, selector3 {
2    ...
3 }
```

Zdrojový kód 3: CSS seznam selektorů

Tato konstrukce umožňuje každému selektoru v seznamu sdílet obsah příslušného deklaračního bloku a aplikovat jej na odpovídající elementy. Pokud je jeden selektor v seznamu nevalidní, pak je celé pravidlo neplatné [5].

Další konstrukcí je sekvence selektorů spojená pomocí *sestupného kombinátoru* (*descendant combinator*) [5]. Jednotlivé selektory jsou seřazené za sebou a oddělené mezerou (*sestupný kombinátor* nemá žádný znak). To je možné vidět v ukázce 4.

```
1 selector1 selector2 selector3 {
2    declaration_block
3 }
```

Zdrojový kód 4: Sekvence CSS selektorů se sestupnými kombinátory

Každý selektor z posloupnosti může vybírat jen z těch elementů, které jsou potomky elementů vybraných předchozím selektorem.

Nastává otázka, jakým způsobem aplikovat CSS pravidla, která nastavují stejnou vlastnost pro daný HTML element. Mechanismus pro řešení tohoto jevu je vysvětlen v sekci 2.2.4.

2.2.2 Deklarace

Deklarace sestává z CSS *vlastnosti* a její *hodnoty*. Zapisuje se ve tvaru:
`vlastnost: hodnota`. *Vlastnost* v kontextu CSS umožňuje nastavení podoby

specifické části/částí stránky pomocí přiřazené *hodnoty*. Celkový výčet CSS vlastností lze najít na [7].

Existují CSS vlastnosti, kterým se říká *zkratkové* (*shorthand properties*) [8]. Ty umožňují nastavení nekolika CSS stylů v jedné deklaraci. *Zkratkové vlastnosti* mají, až na výjimky, odlišný způsob zápisu.

Uvažujme CSS vlastnost `margin`. Ta nastavuje čtyři další: `margin-top`, `margin-bottom`, `margin-left` a `margin-right`. Uvažujme následující příklad:

```
margin: 0;
```

Uvedená deklarace je ekvivalentem pro:

```
margin-top: 0; margin-bottom: 0; margin-left: 0; margin-right: 0;
```

Některé zkratkové CSS vlastnosti jsou hierarchicky spřízněné s ostatními, protože mohou nastavovat stejné styly. Příkladem je vlastnost `border`.

```
border: solid
```

Tento zápis je ekvivalentem pro:

```
border-top-style: solid; border-bottom-style: solid;
border-left-style: solid; border-right-style: solid
```

`border-*-style` vlastnosti jsou také nastavitelné pomocí `border-top`, `border-bottom`, `border-left` a `border-right`. Tato hierarchická spřízněnost hraje roli v rezoluci aplikace aplikace stylů (více v sekci 2.2.4).

CSS umožňuje si nadefinovat *vlastní CSS vlastnost* (*custom property*) pomocí `--` prefixu, tedy: `--klíč: hodnota`. Na tu je se možné odkazovat ve výchozích css vlastnostech a získat tak její hodnotu pomocí funkce `var()` ve tvaru: `property: var(--custom-property)`.

2.2.3 Vkládání CSS do HTML

Existuje několik různých způsobů, jak lze vložit CSS kód do stránky tak, aby se jeho obsah aplikoval. Podle toho se odvozují následující typy CSS:

- *inline*;
- *vnořené*;
- *externí*.

Inline CSS je tvořeno pouze CSS deklaracemi. Ty se uvádí u HTML elementu, na který se mají aplikovat, pomocí atributu `style`. Inline CSS nemá

tedy klasickou podobu ve formě CSS pravidel. Jeho zápis vypadá následovně:

```
<div style``display: flex; flex-direction: row;``>
```

Vnořené CSS využívá standardní zápis CSS pravidel. Tento zápis se vkládá jako obsah do vymezeného HTML elemetu `<style>`:

```
<style>.nested-css{width: 0}...</style>
```

Externí CSS je uloženo v separátních souborech s příponou `*.css`, které se importují do dokumentu pomocí HTML elementu `<link>` ve tvaru:

```
<link rel=``stylesheet`` href=``path_to_CSS_file``>
```

Importovat lze jak lokální soubory, tak i ze síťových zdrojů.

Zmíněné typy CSS jsou důležité, jelikož hrají významnou roli v procesu rozhodování o pořadí aplikace CSS stylů. Uvažuje se tam, z jakého zdroje příslušné styly pocházejí. Tento proces je dále popsán v sekci 2.2.4.

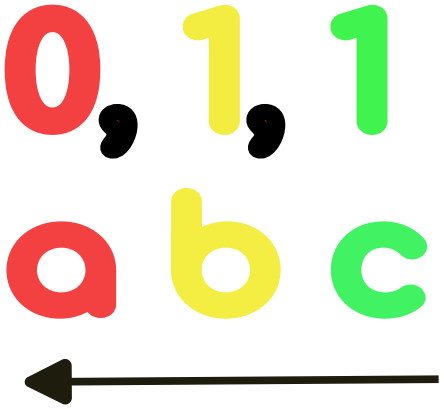
2.2.4 Způsob aplikace pravidel

Princip fungování CSS je postaven na dvou pilířích: *kaskáda* a *dedičnost*.

Kaskáda je způsob rezoluce aplikace jednotlivých CSS pravidel. Její fungování spočívá v jednoduchém způsobu. Pro každé CSS pravidlo, přesněji řečeno pro jeho selektor, je vypočítaná hodnota sestávající ze tří různých nezáporných celých čísel. Ta označují počet výskytů daných tříd selektorů u příslušného pravidla:

- třída **a** – **id** selektory
- třída **b** – **atributové** a **class** selektory, **pseudo-třídy**
- třída **c** – **element** selektory a **pseudo-elementy**

Tyto hodnoty se kolektivně označují jako **specifičnost**. Ta určuje konkrétnost a postavení daného selektoru mezi ostatními při rozhodování o aplikovaném stylu. Její hodnota roste směrem zprava-doleva. Třída **c** je nejméně specifická a třída **a** nejvíce, viz obrázek 1.



Obrázek 1: CSS specifičnost a směr jejího růstu

V případě více pravidel pro stejný HTML element má vždy přednost to s vyšší specifičností. Porovnávání se provádí lexikografickým uspořádáním. Může dojít ke kolizi pravidel. K té dojde, pokud mají dvě různá pravidla stejnou specifičnost. V tomto případě rozhoduje další mechanismus kaskády. Tím je *pořadí* pravidel. Přednost má vždy to pravidlo, které vyskytuje za tím druhým (kolidujícím).

Pro ukázku uvažujme dva příklady.

```
/* 1,1,0 */
#id1 .class1 {
    ...
}

/* 0,2,0 */
.class2 .class3 {
    ...
}
```

Zdrojový kód 5: CSS pravidla - rozdílná specifičnost

```
/* 0,3,0 */
.class1 .class2 .class3 {
    ...
}

/* 0,3,0 */
.class4 .class5 .class6 {
    ...
}
```

Zdrojový kód 6: CSS pravidla - stejná specifičnost

V příkladu 5 máme dvě pravidla. První obsahuje jeden **id** selektor a **class** selektor. Jeho specifičnost je **1,1,0**. V druhém pravidle jsou dva **class** selektory. Jeho specifičnost má hodnotou **0,2,0**. Zde se aplikuje první pravidlo, protože má vyšší specifičnost.

V příkladu 6 mají obě pravidla stejnou specifičnost - **0,2,0**. Proto bude rozhodovat jejich pořadí. Aplikováno bude druhé pravidlo.

Zde však koncept specifičnosti nekončí. V sekci 2.2.3 byly zmíněné typy CSS určené podle způsobů vkládání do stránky. Uvažujme **Inline** CSS. To má dle definovaných vlastností CSS vyšší přednost před zbylými typy bez ohledu na specifičnost. U zbylých CSS typů (**vnořené** a **externí**) je přednost určena specifičností a pořadím výskytu referencí na ně ve stránce, viz [9] a [10].

V kontextu kaskády je také nutné zmínit *důležitost*. Jedná se o způsob, jak lze obejít princip kaskády a zaručit, že se aplikuje přidělená hodnota dané CSS vlastnosti, pokud je validní. Tohoto lze dosáhnout tak, že k příslušné deklaraci přiřadíme `!important` příponu. Deklarace s příponou má tvar: `klíč:hodnota !important`. *Důležitost* umožňuje přepsat aplikovanou hodnotu bez ohledu na specifičnost jejího pravidla. Obecně se však nedoporučuje tuto příponu hojně používat, jelikož ji může poté přepsat jen další deklarace s `!important` příponou. Navíc musí dodržet princip kaskády (v rámci pravidel s `!important` příponou). Nastřádanost těchto přípon může tedy způsobit problém, pokud je zapotřebí změnit hodnoty v CSS kódu.

Dědičnost je další důležitou součástí CSS. Vymezuje a definuje přenos nastavených CSS vlastností z jednoho HTML elementu na další. Přenos je dán stromovou strukturou HTML (probíhá směrem z rodičovského uzlu na jeho potomky).

Ne všechny CSS vlastnosti se dědí se automaticky. Toto je možné obejít a řídit pomocí přímého nastavení klíčovým slovem `inherit`. Uvažujme příklad:

```
margin: inherit;
```

Je tam explicitně nastaveno, že `margin` má získat svoji hodnotu podle nastavených stylů v rodičovském uzlu. Pokud se chceme vyhnout dědičnosti, můžeme

nastavit danou CSS vlastnost na jinačí hodnotu. Co se týče pořadí aplikace zděděných hodnot a těch přímo uvedených, mají přímo uvedené hodnoty přednost nezávisle na tom, jakou specifičnost má pravidlo, ze kterého pochází děděná hodnota.

2.2.5 At-pravidla

At-pravidla (*At-rules*) jsou speciální druh CSS pravidel, které které umožňují nastavit způsoby, jak se má CSS chovat. Každé at-pravidlo začíná `@` identifikátorem. Z hlediska syntaxe jsou rozdělena na následující skupiny:

- *obecná*;
- *vnořená*.

Obecná at-pravidla jsou zapisována ve tvaru:

```
@at-rule_name (rule)
```

Příkladem těchto pravidel jsou: `@namespace`, `@charset` a `@import`. `@charset` pravidlo není oficiálním at-pravidlem [11]. Každé z těchto pravidel má jiný tvar. Mohou se nacházet vždy jen na začátku CSS stylesheetu.

Vnořená at-pravidla umožňují v rámci povolené syntaxe do jejich bloků vkládat další CSS kód (jen některá mohou obsahovat další vnořená at-pravidla). Jsou zapisována typicky ve tvaru uvedeném v ukázce 7.

```
1 @at_rule_name (rule) {
2     ...
3 }
```

Zdrojový kód 7: Vnořená at-pravidla

Celkový výčet vnořených at-pravidel je možné najít na [3]. Každé z nich má svou formu zápisu a účel. `@keyframes` je třeba použit pro specifikaci CSS animace.

Existují *podmíněná skupinová pravidla* (*conditional group rules*). Jsou to vnořená at-pravidla obsahující podmínku, která se průběžně vyhodnocuje a rozhoduje o aplikace obsahu jejich bloku. Patří mezi ně `@media`, `@supports` a `@document` (deprekováno). Jako jediná mohou obsahovat další vnořená at-pravidla.

2.2.6 Atomické CSS

V kapitole 1 byly zmíněny populární postupy pro psaní a organizaci CSS kódu. V této práci je důležitá metodika *atomického CSS*. Jde o způsob zápisu CSS

pravidel, které obsahují pouze jednu CSS deklaraci. Těmto pravidlům se říká *utility třídy*.

Ty jsou používané jako základní stavební blok při refaktorizaci (více v kapitole 4). V ukázce 8 je možné vidět příklad atomického CSS. Známým nástrojem pro psaní atomického CSS je framework *Tailwind*.

```
1
2  .utility-class1 {
3     display: flex
4  }
5
6  .utility-class2 {
7     width: 50%
8  }
9
10 .utility-class3:hover {
11    background-color: blue
12 }
```

Zdrojový kód 8: CSS utility třídy

2.3 DOM

DOM (Document Object Model) je programové rozhraní, které reprezentuje strukturu webové stránky. Umožňuje k ní přístup a vykonávat nad ní operace. Vymezení těchto operací je definováno v [12]. Ne všechny jsou však v prohlížečích implementovány. Proto se musí jejich dostupnost ověřit. Jak již bylo zmíněno v sekci 2.1, struktura HTML dokumentu má podobu stromu. Tuto strukturu používá DOM. DOM uzel je nadmnožinou HTML elementu. Kromě samotných HTML elementů, obsahuje i informace o vztazích rodič-potomek mezi jednotlivými elementy, jejich atributy, obsah, a další.

Přístup k těmto informacím je možný skrze programovací jazyk *JavaScript*, který je nedílnou součástí prohlížeče a webových technologií (podrobněji rozebrán v sekci 3.1). V rámci této práce jsou velmi důležité dvě metody. Obě jsou dostupné skrze JavaScriptový objekt `document`. Jsou to `document.querySelector()` a `document.querySelectorAll()`. Tyto metody umožňují získat odkazy na HTML elementy, pokud je lze pomocí příslušného selektoru vybrat [12].

2.4 Formální konceptuální analýza

Formální konceptuální analýza (zkráceně FCA) přímo nesouvisí s webovými technologiemi. V rámci práce jsou však poznatky z ní použity k realizaci optimalizační techniky (rozebráno v sekci 4.6.3).

Jedná se o metodu, která se zabývá zkoumáním vztahů mezi neprázdnou množinou *objektů* a neprázdnou množinou *atributů*. Reprezentaci vztahů mezi

objekty a atributy lze provést formou tabulky, kde hodnota uvedená v políčku tabulky značí, zda daný objekt má daný atribut.

Nechť X je množina objektů a Y množina atributů. Dále uvažujme binární relaci I mezi množinami X a Y. Trojici $\langle X, Y, I\rangle$ pak označujeme jako *formální kontext* (*formal context*).

U formálního kontextu jsou pro každou množinu A, kde $A \subseteq X$ a B, kde $B \subseteq Y$, definované *koncept tvořících operátory* (*concept-forming operators*) – $\uparrow\colon 2^X \to 2^Y$, $\downarrow\colon 2^Y \to 2^X$ tak, že:

$$A^{\uparrow} = \{y \in Y \mid \forall x \in A : \langle x, y\rangle \in I\}$$
$$B^{\downarrow} = \{x \in X \mid \forall y \in B : \langle x, y\rangle \in I\}$$

Z uvedených zápisů můžeme vidět, že $A^{\uparrow}$ je množina všech atributů z množiny Y, které sdílejí všechny objekty z množiny A. $B^{\downarrow}$ je množina všech objektů z množiny X, které sdílejí všechny atributy z množiny B.

Ve *formálním kontextu* je dvojice $\langle A, B\rangle$, kde platí, že $A^{\uparrow} = B$ a $B^{\downarrow} = A$, známá jako *formální koncept* (*formal concept*). *Formální koncept* si lze představit jako plochy obdélníku pokrývající odpovídající políčka v tabulce reprezentující relaci.

Množina všech *formálních konceptů* z *kontextu* $\langle X, Y, I\rangle$ se značí jako $B(X, Y, I)$:

$$B(X, Y, I) = \{\langle A, B\rangle \in 2^X \times 2^Y \mid A^{\uparrow} = B, B^{\downarrow} = A\}$$

Pokud mezi prvky množiny $B(X, Y, I)$ existuje uspořádání $\leq$, pak $\langle B(X, Y, I), \leq\rangle$ se nazývá *konceptuálním svazem* (*concept lattice*).

Naskytuje se otázka, jak najít optimální výběr *formálních kontextů* z $B(X, Y, I)$ tak, aby byl jejich počet co nejmenší a aby byly pokryty všechny dvojice z I. V této práci je využito dekompozice *binárních matic* a to konrétně algoritmu *GreConD* [13] pro rozklad matice $\mathcal{I}$ o rozměrech $n \times m$. Výsledkem rozkladu je *Booleovský produkt* binárních matic $\mathcal{A} \circ \mathcal{B}$. Ten je definován následovně:

$$(\mathcal{A} \circ \mathcal{B})_{ij} = \bigvee_{l=1}^{k} \mathcal{A}_{il} \cdot \mathcal{B}_{lj}$$

kde $\bigvee$ je výsledek funkce logického součtu.

Matice $\mathcal{A}$ má rozměry $n \times k$ a $\mathcal{B}$ má $k \times m$. *GreConD* se snaží o minimalizaci počtu k *faktorů* (*formálních konceptů*).

V optimalizaci budou množinu X představovat všechny uvažované HTML elementy, množinu Y pak aplikované deklarace (z uvažovaných elementů). Na reprezentaci jejich relace se pak použije použije *GreConD*, aby našel co nejvíce optimální shluky deklarací (formální kontexty), ze kterých se pak zkonstruují nová pravidla.

3 Použité technologie

Pro tvorbu nástroje byly vybrány technologie co nejvíce vybudované okolo webového prohlížeče, aby bylo možno získat co nejvíce přesné informace o stylu stránky a po zpracování vygenerovaný výstup vložit zpět.

3.1 Jazyky JavaScript a TypeScript

JavaScript [1] je dynamicky typovaný jazyk, který je jedním z pilířů pro tvorbu webových stránek. Pomocí něj lze měnit jejich podobu, jejich strukturu, přistupovat k DOMu a mnoho dalšího. Je součástí každého moderního prohlížeče, ale existuje i mimo ně (v této práci je použito běhové prostředí *NodeJS*[6]). Standard, který JavaScript implementuje, se nazývá *ECMAscript*.

K JavaScriptu existuje nádstavbový jazyk zvaný **TypeScript** [14], který vyvíjí americká společnost Microsoft. Tento jazyk přidává k JavaScriptu statické typování a obohacuje jeho syntaxi o další užitečné prvky. Díky statickým typovým kontrolám umožňuje předcházet problémům, které by byly zjištěny až za běhu. Není však schopen zabránit každému z nich. Sám o sobě nemá běhové prostředí. Místo toho je kompilován/transpilován na JavaScript. Díky této vlastnosti je možné převést čistě JavaScriptové aplikace do TypeScriptu. Tím je možné získat výhody, které TypeScript nabízí a zároveň zachovat kompatibilitu s existujícími systémy.

Téměr celý nástroj je napsán v TypeScriptu, včetně některých použitých technologií nebo alespoň poskytují definované typy pro jejich možnou integraci s TypeScriptem. Některé definicice typů nemají, proto jsou menší části nástroje realizovány také v JavaScriptu.

3.2 Puppeteer

Puppeteer[7] je open-source knihovna napsaná převážně v jazyce TypeScript. Je určena pro automatizovanou manipulaci a testování prohlížečů založených na **Chromiu**.[8]

Puppeteer pro komunikaci s prohlížečem využívá **DevTools Protocol**.[9] Jedná se o vysoko úrovňové API, pomocí kterého lze získavat z Chromia informace ohledně stavu prohlížeče, načtených stránkách, zpracovaných požadavcích apod. které se následně dají použít k profilování, odladění či testování prohlížeče.

Je možné získávat informace o aplikovaných CSS stylech přímo skrze DevTools Protocol. Nícméně se ukázalo, že DevTools Protocol neposkytuje informace o všech stylech, pokud nejsou aplikované. Příkladem jsou pravidla s pseudotřídami nebo pravidla obalená `@media` at-pravidly. Proto program testuje shodu selektorů individiuálně, jak je popsáno v sekci 4.3.

[6] https://nodejs.org
[7] https://pptr.dev/
[8] https://www.chromium.org/Home/
[9] https://chromedevtools.github.io/devtools-protocol/

Ve výchozím stavu Puppeteer pracuje s Chromiem v *headless módu*. V tomto módu činnosti se prohlížeč neotevře klasickým způsobem v podobě okna, ale pracuje jako proces v pozadí. Tento mód je možno vypnout při vytváření Puppeteer instance.

Skrze Puppeteer je možné napřímo provádět interakce s prohlížečem. Lze přistupovat ke všem funkcionalitám, které jsou dostupné skrze JavaScript v prohlížeči.

Nevyužívá se celá knihovna Puppeteer. Využívá se její podmnožina, která je dostupná pod názvem **puppeteer-core**.[10] Ta obsahuje veškerou funkcionalitu Puppeteeru, kromě přímého stáhnutí prohlížeče. Program používá puppeteer-core pro přenášení a získávání dat z prohlížeče, interakci s DOMem a přímou manipulaci s HTML dokumentem.

3.3 CSSTree

CSSTree[11] je open-source knihovna napsaná v jazyce JavaScript. Existují k ní definované typy[12] umožňující práce s Typescriptem. Tato knihovna slouží jako parser pro CSS s cílem poskytnout přístup k informacím získaných ze zpracovaného CSS vstupu. Tyto informace jsou dostuné skrze strukturu abstraktního syntaktického stromu. Podrobnosti o ní je možné najít zde [15]. Tu lze libovolně procházet, modifikovat a vygenerovat její textovou podobu.

CSSTree je používán téměř ve všech částech programu. Kromě toho je součástí i dalších použitých technologií.

3.3.1 @bramus/specificity

Pro rezoluci specifičností CSS pravidel je použita open-source knihovna **@bramus/specificity**.[13] Je napsaná v jazyce JavaScript se zabudovanými typy pro TypeScript. Funguje nad CSSTree. Podle informací získaných informací z něj pak spočítá specifičnosti selektorů. Podporuje i selektory z [5]. Umožňuje přímo porovnávat jednotlivé spočítané specifičnosti, což je hojně využíváno
u rezoluce specifičností pravidel.

3.3.2 @projectwallace/css-analyzer

@projectwallace/css-analyzer[14] je open-source knihovna napsaná v jazyce JavaScript se zabudovanými typy pro TypeScript. Také využívá CSSTree. Umožňuje analyzovat vložený CSS kód. Spočítá velikost vloženého kódu, počet unikátních deklarací, poočet unikátních selektorů atd. Kompletní výčet metrik je

[10] https://classic.yarnpkg.com/en/package/puppeteer-core
[11] https://github.com/csstree/csstree
[12] https://classic.yarnpkg.com/en/package/@types/css-tree
[13] https://github.com/bramus/specificity
[14] https://github.com/projectwallace/css-analyzer

možné najít zde [16]. Tato knihovna je používaná u experimentální části pro analýzu původního a refaktorovaného CSS.

3.3.3 @projectwallace/css-code-quality

@projectwallace/css-code-quality[15] je open-source knihovna vytvořená v JavaScriptu se zabudovanými typy pro TypeScript. Pomocí lze spočítat tří různá skóre analyzovaného CSS kódu:

- *Udržitelnost*
- *Komplexita*
- *Výkonnost*

Tyto skóre jsou pak použity v experimentální části pro ohodnocení původního a refaktorovaného CSS.

3.3.4 csstree-validator

csstree-validator[16] je JavaScriptová knihovnu pro validaci CSS částí zpracovaných CSSTree. Umožňuje validovat určité prvky (CSS deklarace, at-pravidla, ...) z interní reprezentaci CSSTree. Bohužel ne všechny nejnovější prvky CSS tam jsou zakomponované a validace je proto označí jako nevalidní, i když je prohlížeč podporuje. Proto bez přímého nastavení program pokročilejší formy validace (povolené hodnoty CSS vlastností, syntaxe at-pravidel) neprovádí a nechává ji na prohlížeči.

3.4 js-beautify

js-beautify[17] je open-source JavaScriptová knihovna. Existují k ní definice typů[18] pro TypeScript. Tuto knihovnu program používá pro formátování CSS kódů. Je použita také pro formátování CSS kódů v experimentální části.

3.5 Color.js

Color.js[19] je open-source JavaScriptová knihovna (se zabudovanými definicemi typů pro TypeScript). S touto knihovnou lze parsovat, reprezentovat a modifikovat barvy. Dokáže počítat jejich vzdálenost různými algoritmy, míchat je, alterovat jejich vlastnosti, převádět je mezi individuálními modely pro jejich reprezentaci a mnoho dalšího. Podporuje barvy z [17]. Tato knihovna je použita při optimalizaci shlukováním CSS hodnot.

[15] https://github.com/projectwallace/css-code-quality
[16] https://github.com/csstree/validator
[17] https://github.com/beautify-web/js-beautify
[18] https://www.npmjs.com/package/@types/js-beautify
[19] https://github.com/LeaVerou/color.js

3.6 MDN Data

MDN Data[20] je open-source souhrn infomací o webových technologiích. Je složen z kombinace JSON a Markdown souborů. MDN data jsou v rámci této práce používána jako zdroj informací mezi vztahy jednotlivých CSS vlastností (jestli je jedna zkratkovou vlastností pro druhou neno jestli neexistuje hierarchická spřízněnost). MDN data jsou mimo jiné zdrojem dat a informací pro oficialní *MDN webovou dokumentaci*[21] či CSSTree.

3.7 Yargs

Yargs[22] je open-source knihovna pro parsování vstupních argumentů z příkazového řádku. Umožňuje nastavit, typ parametru, výchozí hodnotu, zda je parametr povinný, jeho popis a mnoho dalších.

3.8 GNU Octave

GNU Octave[23] je open-source program využívaný k vědeckým výpočtům. Je téměř kompatibilní s Matlabem,[24] což umožňuje spouštění Matlab skriptů v GNU Octave. V rámci práce je využíván GNU Octave pro spouštění Matlab skriptu pro aplikaci optimalizace pomocí FCA (vysvětlena v 4.6.3).

3.9 GNU Wget

GNU Wget[25] je open-source program, pomocí kterého lze stahovat soubory z uvedených webových stránek. Často bývá součástí *Unix-like* operačních systémů jako nástroj příkazového řádku. Wget umožňuje širokou škálu nastavení jako například nastavení odkazů, aby byly spustitelné na lokálním stroji, aby komunikace probíhala skrze IPv4/IPv6 protokol a další. Program pomocí něj, pokud tak byl spuštěn, stáhne soubory stránky ze sítě, se kterými bude pak dále pracovat.

[20] https://github.com/mdn/data
[21] https://developer.mozilla.org
[22] https://yargs.js.org/
[23] https://octave.org/
[24] https://www.mathworks.com/products/matlab.html
[25] https://www.gnu.org/software/wget/

4 Návrh

Tato kapitola se zabývá rozborem návrhu jednotlivých částí implementovaného programu. Je vysvětleno, jak je nástroj navržen a jak pracuje.

4.1 Celkový náhled a spouštění

Program je navržen jako konzolová aplikace určená pro operační systém *Linux*. Chování programu lze řídit pomocí přepínačů, které je možné specifikovat jako vstupní argumenty z příkazové řádky. Celkový výčet rozpoznávaných přepínaču je uveden v tabulce 2.

Podle nastavených přepínačů a jejich hodnot, program vykoná příslušný způsob refaktorizace. Tu provede individuálně pro všechny nalezené HTML dokumenty ve specifikovaném adresáři. Každý dokument projde, vyextrahuje jeho CSS styly, ty zpracuje a podle nich vygeneruje nový CSS stylesheet (viz 4.7). Ten se bude jmenovat stejně jako daný dokument. Po ukončení refaktorizace jsou odebrány všechny uvažované původní CSS styly. Původní CSS stylesheety, které se nepodařilo načíst nebo nebyly přímo uvedené v dokumentu, program ponechá.

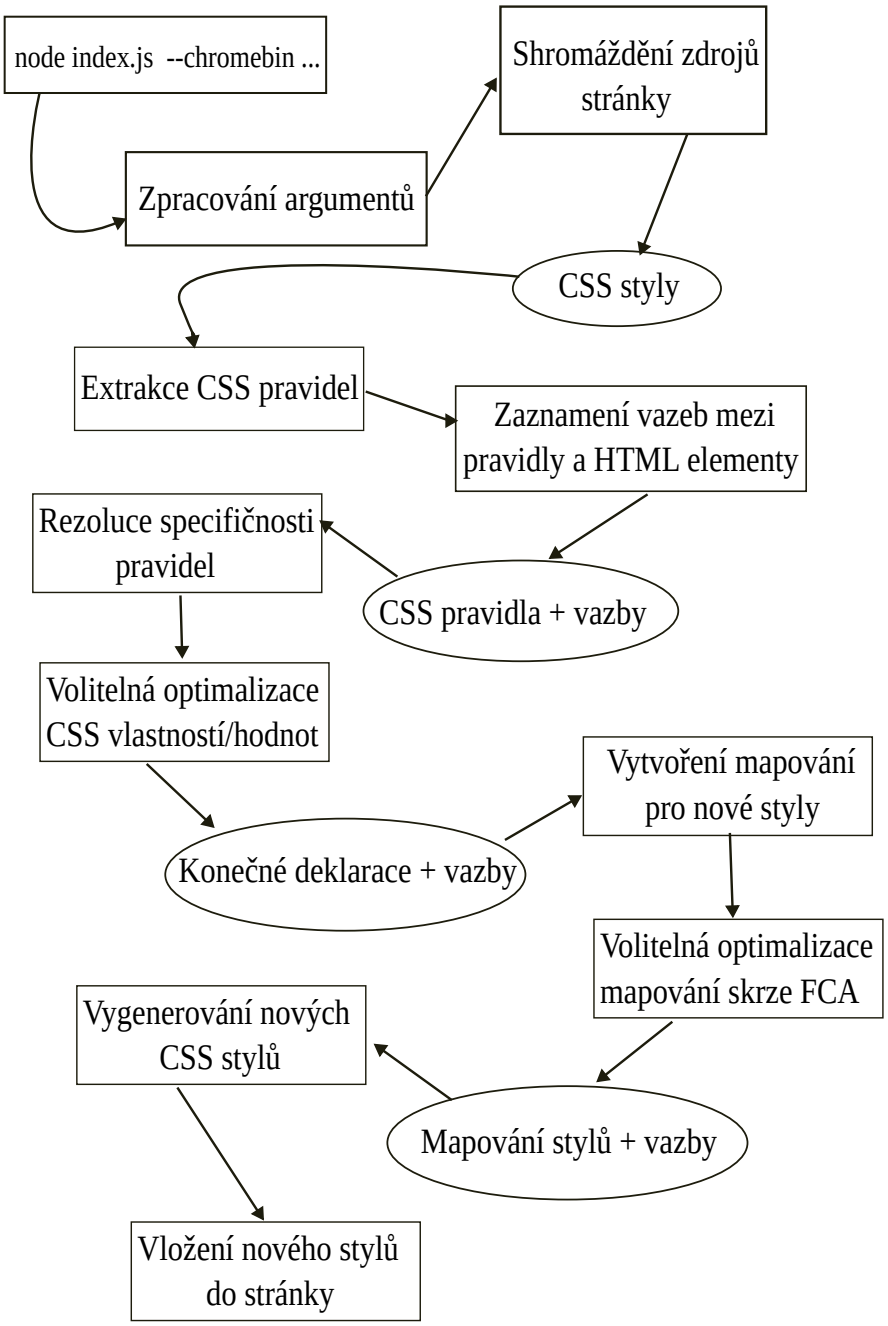
Zjednodušený pohled na architekturu a činnost programu je možné vidět na obrázku 2. Architektura je dále podrobněji rozebraná po částech v následujících sekcích této kapitoly.

Přepínač	Popis	Datový typ
−−inDir	Cesta ke stránce a jejím zdrojům (−−inDir nebo −−pageAddress musí být specifikovány).	řetězec
−−outDir	Cesta, kam program překopíruje zdroje stránky a provede refaktorizaci.	řetězec
−−chromeBin	Cesta k prohlížeči založeném na Chromiu.	řetězec

Přepínač	Popis	Datový typ
––pageAddress	Adresa webové stránky, odkud se stáhnou její zdroje (––inDir nebo ––pageAddress musí být specifikovány).	řetězec
––onlyRefactored	Ve finálním CSS stylesheetu budou pouze nová vygenerovaná CSS pravidla. (Vztahuje se pouze k pravidlům s deklaracemi)	boolean
––noOrderKeeping	Program nebude hlídat pořadí CSS pravidel jednotlivých uzlů (nebude dělat kontroly ani dodatečné generování dalších mapování pro dodržení pořadí).	boolean
––fcaOpt	Spustí optimalizaci vytvořených CSS mapování pomocí FCA.	boolean
––fcaDebugDir	Cesta, kam se po specifikaci uloží informace z FCA optimalizace (bez ––fcaOpt nic nedělá).	řetězec
––propOpt	Spouští optimalizaci podporovaných CSS vlastností.	boolean
––valOpt	Spouští optimalizaci hodnot podporovaných CSS hodnot.	boolean

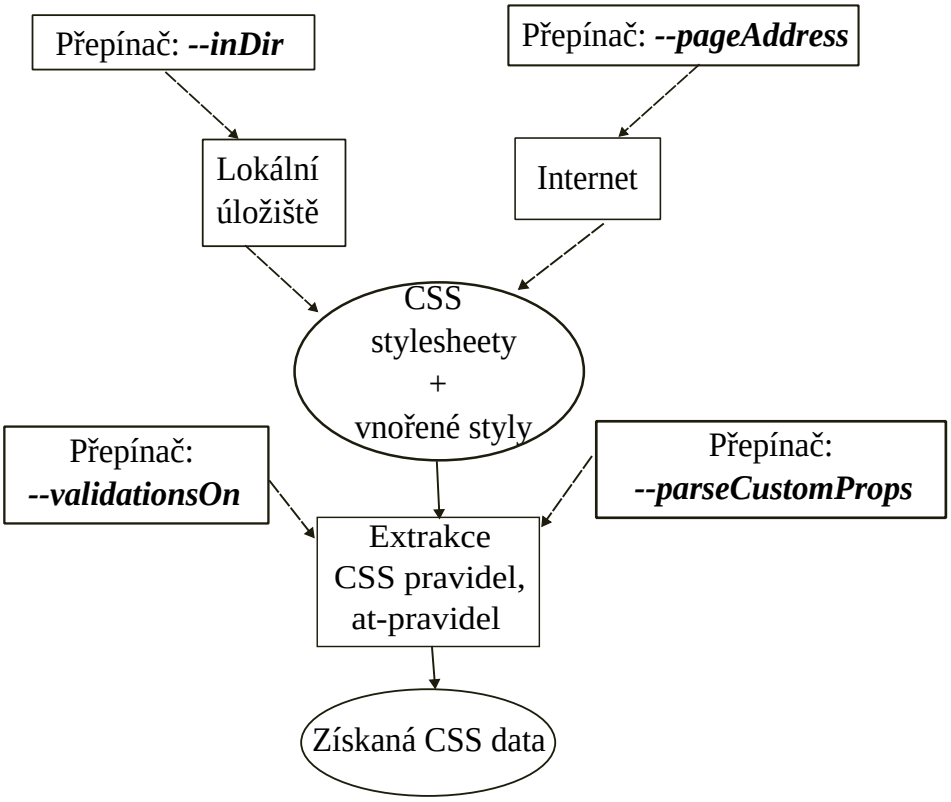
Přepínač	Popis	Datový typ
--maxSpaceDiff	Nastavuje maximální možný rozdíl mezi hodnotami podporovaných CSS vlastností pro nastavování rozložení stránky. Hodnota je v pixelech. (funguje pouze s --valOpt)	nezáporné číslo
--maxColorDiff	Nastavuje maximální možný rozdíl mezi hodnotami podporovaných CSS vlastností pro nastavování barev. Hodnota musí být v rozmezí 0-100 (rozmezí hodnot rozdílu barev, viz sekce 4.6.2). (funguje pouze s --valOpt)	nezáporné číslo
--validationsOn	Zapne validaci at-pravidel a CSS deklarací pomocí	boolean
--includeAllRules	Všechny styly (včetně nerozpoznaných) budou umístěny do finálního stylesheetu	boolean
--keepOutDirContents	Obsah složky specifikované přepínačem --outDir zůstane netknutý (pokud zdroje stránky neobsahují stejnomenné soubory)	boolean
--parseCustomProps	Pokud je specifikován, pak se budou uvažovat i *vlastní* CSS vlastnosti	boolean

Tabulka 2: Vstupní přepínáče programu a jejich vysvětlení



Obrázek 2: Zjednodušená architektura a princip fungování programu

4.2 Načtení a extrakce dat



Obrázek 3: Extrahování CSS stylů

Program nabízí dvě možnosti načítání zdrojů stránky. Online a lokální. V obou případech nejprve shromáždí a zduplikuje zdroje webové stránky (HTML a CSS soubory, obrázky, ...) a všechny je vloží do uvedené výstupní složky. Jakmile jsou data stránky zduplikována/stažena, program začne postupně procházet celý adresář a bude hledat HTML dokumenty. Pokud je našel, pak je jednotlivě pomocí *puppeteer-core* načte, vypne JavaScript stránky (aby neměnila svoji podobu) a přejde k načtení CSS stylesheet souborů a vnořených CSS stylů z HTML dokumentu. Stylesheety jsou vybírány podle toho, pokud na ně existuje odkaz v aktuálně zkoumaném dokumentu. Toto je realizováno skrze dotazování se DOMu na `<link>` a `<style>` elementy. Při hledání `<link>` elementů je používán i atributový selektor ve tvaru: `link[rel='stylesheet']`, protože `<link>` může importovat více typů zdrojů (obrázky, JavaScript soubory, ...).

U HTML elementů, které obsahují nebo odkazují na CSS, je možné nastavit `media` [18] atribut. Tento atribut nastavuje `@media` at-pravidlo obal okolo daného CSS kódu. Hodnota atributu se pak stane podmínkou tohoto at-pravidla pro aplikaci CSS stylu v jeho bloku. Nastavený `media` atribut může vypadat takto:

```
<link media=``(min-width: 360px)`` rel=``stylesheet``.../>
```

Pokud program `media` atributy najde, uloží je a nastaví je u odpovídající CSS pravidel.

Jednotlivé stylesheety a vnořené styly jsou popořadě načteny do paměti. Je nutno dodržet toto pořadí kvůli kaskádě CSS. Následně jsou individuálně poslány do extraktoru, který je pomocí parseru zpracuje a získá jejich reprezantace v podobě abstraktního syntaktického stromu.

V tomto stromě jsou následně vyhledána všechna CSS pravidla rozpoznaná parserem. Je ověřována i validita selektorů (je dodržována i vlastnost nevalidity u *seznamu selektorů*, viz sekce 2.2.1). Ve výchozím stavu parser nerozpoznává *vlastní* CSS vlastnosti. Aby je rozpoznal, je nutno programu specifikovat přepínač *−−parseCustomProps*.

Kromě nich jsou ještě extrahována CSS at-pravidla, která jsou rozdělena podle jejich syntaxe (vnořená a obecná, viz sekce 2.2.5).

Obecná at-pravidla se syntakticky liší od vnořených, proto se také extrahují zvlášť. Kontroluje se, zda se nachází na načátku CSS stylu.

Pro vnořená at-pravidla se uvažují následující případy podle jejich obsahu:

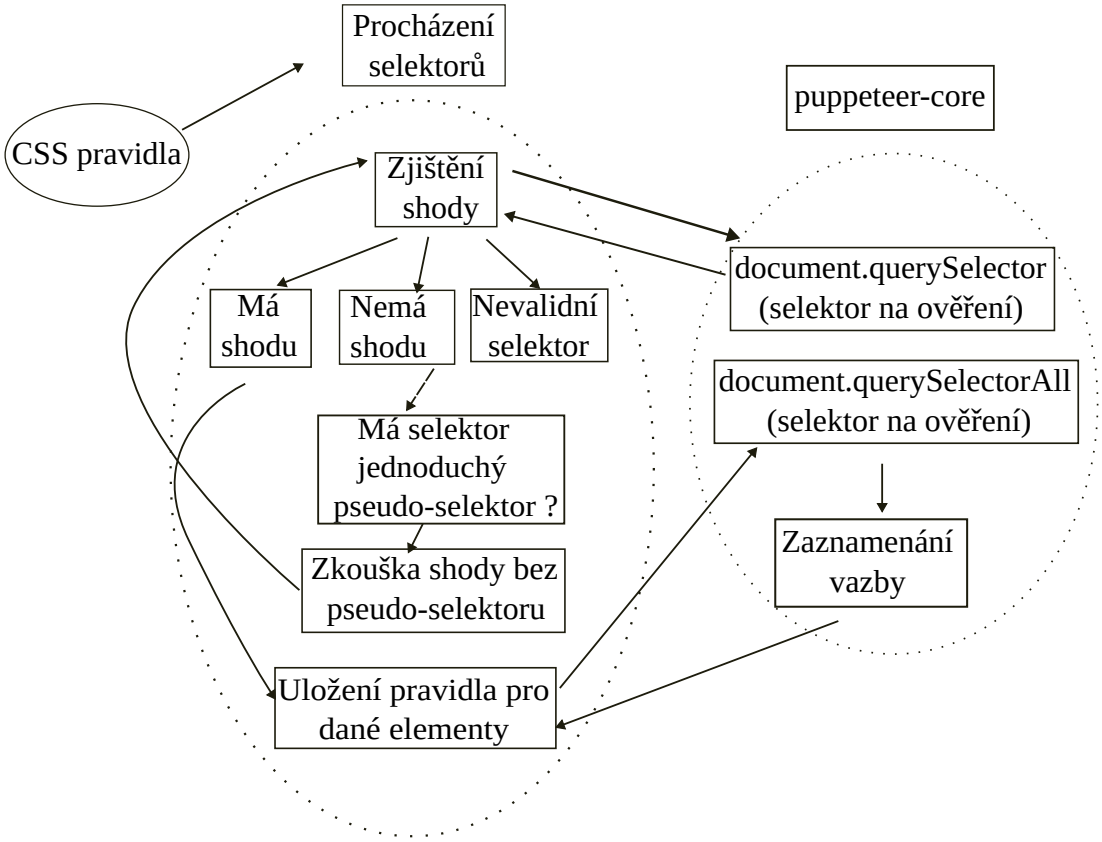
- *pouze deklarace (bez CSS selektoru)*;
- *CSS pravidla*.

At-pravidla pouze s deklaracemi se sbírají zvlášť. Ty *zbývající* se ukládají přímo u daných CSS pravidel, která jsou těmi at-pravidly obalena.

Uvažují se pouze ty pravidla a at-pravidla, který byly rozpoznány parserem. Program kontroluje, jestli byl parser schopen rozpoznat jejich základní tvar. U vnořených at-pravidel se kontrolují i *skupinově podmíněná pravidla* (viz sekce 2.2.5).

Ve standardním běhu se neprovádí pokročilejší validace povolené syntaxe (povolené hodnoty u dané CSS vlastnosti, syntaxe podmínek at-pravidel). Rozhodnutí o validitě je ponecháno prohlížeči. Pokud je specifikován přepínač *−−validationsOn*, provede se validace v rozsahu rozpoznávacích možností parseru a validátoru. Ne všechny nejnovější CSS prvky jsou podporovány (například rozšířené možnosti zápisu `@media` at-pravidel, viz [11]).

4.3 Zaznamenání vazeb



Obrázek 4: Proces vyhodnocování a zaznaměnávání vazeb

Jakmile jsou CSS pravidla zpracována, je zapotřebí zjistit, na jaké HTML elementy se aplikují. Pro získání této informace se program dotazuje DOMu. Používají se metody zmíněné v sekci 2.3: `document.querySelector()` a `document.querySelectorAll()`. Pomocí nich si program ověří, zda se daným selektorem dají vybrat HTML elementy. Celkově jsou určené kategorie:

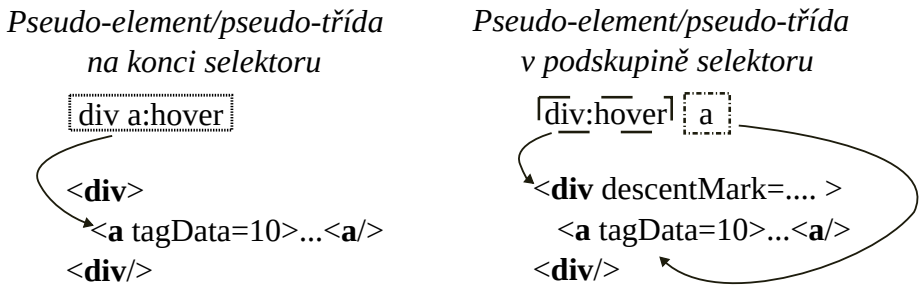
1. *selektor se shodou*;
2. *selektor bez shody*;
3. *nevalidní tvar selektoru*.

U *první kategorie* je jednoznačně možné pomocí selektoru získat seznam HTML elementů.

U *druhé* prohlížeč nevrátil žádné elementy odpovídající selektoru. V sekci 2.2.1 bylo zmíněno, že selektory mohou obsahovat pseudo-třídy a pseudo-elementy. Existuje tedy možnost, že výběr HTML elementů není možný z důvodu nenaplnění podmínek nutných k jeho selekci. Proto je selektor dále analyzován, zda neobsahuje jednoduchý (jednoslovní, neparametrický) selektor pseudo-elementu

nebo pseudo-třídy, který se nachází na konci analyzovaného selektoru. Pokud ano, program zkusí vybrat elementy selektorem bez pseudo-elementu/pseudo-třídy.

Pokud se pseudo-element/pseudo-třída vyskytují v podskupinách selektorové sekvence (často spojené *sestupným kombinátorem*, viz sekce 2.2.1), nabízí se otázka, zda nezkusit odebrat jednoduché pseudo-elementy/pseudo-třídy i z podskupin selektorové sekvence. Toto možné je, nicméně by se musely držet odkazy na jednotlivé HTML elementy z dané selektorové podskupiny a pro každou z nich by se pak musela vytvořit vazba pomocí *class* selektorů, které pak budou dohromady tvořit nový selektor u vygenerovaného pravidla. Na obrázku 5 je možné vidět ilustraci obou přístupů.



Obrázek 5: Kontrast zaznamenávání vazeb u pseudo-elementů/pseudo-tříd

Pro lepší jednoznačnost byla zvolena substituce pseudo-elementu/pseudo-třídy pouze na konci selektorů. Složitější kontrukce jsou ponechány na vyřešení uživateli.

U *třetí kategorie* není možná selekce. Tato situace nastává v případě, pokud selektor například obsahuje nepovolené znaky. Povolené znaky je možné najít v [5]. Existují však i pravidla, jejichž selektor je dle metod `document.querySelector()` a `document.querySelectorAll()` nevalidní, ale ve skutečnosti validní jsou. Příkladem jsou selektory užívané u at-pravidla `@keyframes` [19] (*0%*, *30%*, *100%*, ...).

Dalším příkladem jsou selektory obsahující *vendor prefix* prvky [20] (typické pro daný prohlížeč). U těchto typů selektorů nebyla možná selekce a proto jsou označeny jako nevalidní, nebo (v případě `@keyframes`) jako selektory bez shody a nechá se řešení na uživateli (jelikož nemusí být jasné, zda se někde jinde nepoužívá).

Výše popsané třídění se provádí pro individuální selektory. Pokud je v pravidle použit *seznam selektorů*, jsou jednolivé selektory rozděleny a u každého z nich je jednotlivě vyzkoušen a poznačen výsledek testu shody.

U elementů v HTML dokumentu se vazba poznačuje pomocí speciálně přidaného atributu *tagData*, který obsahuje celé nezaporné číslo představující index pozice v poli s uloženými CSS pravidly pro daný HTML element.

4.4 Roztřídění a filtrace deklarací podle specifičností

Po dokončení zaznamenávání vazeb se přejde k filtraci CSS deklarací aplikovaných na HTML elementy. K tomu využijeme specifičnost CSS selektorů daných pravidel. V rámci každého uzlu se provede filtrace jednotlivých deklarací a ponechají se pouze ty, které se aplikují dle CSS kaskády. V programu se řeší rezoluce specifičností v rozměrech:

- *ve stejném kontextu*;
- *napříč kontexty*.

Kontextem jsou zde míněny vlastnosti pravidla, které by mohly mít vliv na jeho aplikaci. Jedná se o selektor pseudo-elementu/pseudo-třídy a použitá at-pravidla.

Nejprve se provede rezoluce ve *stejných kontextech* (stejná CSS vlastnost, at-pravidla a pseudo-element/pseudo-třída). Program pro každý ze zaznačených HTML elementů načte na něj aplikovaná pravidla a spočítá specifičnost jejich selektorů. Tu si u příslušných pravidel uloží. Následně bude v rámci jednotlivých elementů popořadě (v načteném pořadí ze stylů) pravidla procházet a číst obsah jejich deklaračních bloků. Přečtené deklarace porovná s finálními (těmi, co se ve finále opravdu aplikují), které v rámci daného elementu zatím prošel. Pokud zjistí, že má již deklaraci se stejnou CSS vlastností v daném kontextu uloženou, porovná jejich specifičnost (specifičnost pravidel, ze kterých pochází) a `!important` přípony. Podle výsledku porovnání a případně ještě pravidla pořadí CSS kaskády pak přepíše/ponechá hodnotu dané vlastnosti. Pokud se ještě danou vlastnost nenalezl, uloží si její hodnotu automaticky.

Nyní má program pro každý element jeho finální deklarace. Nicméně se uvažovala pouze filtrace v rámci stejných kontextů. Může se však stát, že některé deklarace se nemusí aplikovat kvůli vlivu deklarací z jinačích kontextů (*zkratkové vlastnosti*, vyšší specifičnost, ...).

Proto se provádí i filtrace *napříč kontexty*. Kromě porovnávání stejných CSS vlastností se zde uvažují i hierarchicky spřízněné (viz 2.2.2).

Opět se budou procházet jednotlivé elementy a jejich finální deklarace. Nyní si představme důležité kategorie, které budou použity pro rozhodování o filtraci:

1. *deklarace bez pseudo-elementu/pseudo-třídy a at-pravidel*;
2. *deklarace s pseudo-elementem*/pseudo-třídou a bez at-pravidel*;
3. *1. a 2. kategorie s definovanými at-pravidly.*

U * platí výjimka. Ta je vysvětlena na konci této sekce.

Tyto kategorie určují, zda je možné finální deklarace podle té aktuální referenční filtrovat či nikoliv. U daného uzlu si program najde jeho finální deklarace a vytvoří se jejich kopii. Tu si uloží jako referenční deklarace. Ty budou použity při porovnání s těmi finálními. V zásadě podle aktuálních finálních deklarací se vytvoří jejich kopie a s tou jsou pak porovnány a určí se, zda není možné některé z nich vynechat.

Program iteruje napříč referenčními deklaracemi. V dané iteraci u referenční určí, do jaké ze zmíněných katergorií patří. Následně začne procházet všechny finální. Pokud je CSS vlastnost porovnávané finální deklarace stejná nebo hierarchicky spřízněná s vlastností referenční té referenšní, pak program zjistí, do jaké kategorie patří ta finální.

Následně podle kategorií referenční a porovnávané finální deklarace se rozhodne, zda je u porovnávané možné uvažovat o její filtraci. Rozhodovací proces je možné vidět v tabulce 3.

		Porovnávaná		
		1.	2.	3.
Referenční	1.	✓	✓	✓
	2.	×	✓	✓
	3.	×	×	✓

Tabulka 3: Rozhodování o možnosti filtrace podle kategorií referenční a porovnávané deklarace

Pokud program rozhodne, že ano, porovná se u daných deklarací jejich specifičnost, `!important` přípony, případně i pořadí kaskády. Navíc se zde podle určených kategorií případně porovnávají i pseudo-elementy/pseudo-třídy a at-pravidla.

Podle výsledku pak ponechá/vyfiltruje porovnávanou deklaraci. Zaroveň přeruší iterování mezi deklaracemi a začne celý proces (v rámci daného elementu) od začátku s aktualizovanými referenčními a finálními deklaracemi. Jakmile již

nedojde k žádné změně, výsledek se pro daný element uloží a pokračuje se zbývajícími elementů.

Tabulka 3 je sestavena podle následující logiky. Je taky vysvětleno, kdy se deklarace z daných kategorií mohou porovnávat a jakým způsobem, jelikož ten není ve všech možných scénářích stejný.

U *první kategorie* platí, že daná deklarace bude vždy prohlížečem uvažována při výpočtu finálního stylu. Pokud má deklarace z první kategorie vyšší specifičnost/příponu `!important` oproti těm, co závisejí na specifických podmínkách, pak platí, že ty ostatní se neaplikují. Existuje možnost nastavení precedence aplikace pravidel pomocí at-pravidla @layer [21]. Nicméně i zde platí, že pravidla deklarovaná mimo tato at-pravidla, aplikují se jako poslední bez ohledu na specifičnost. Pokud je tedy referenční deklarace z této kategorie, lze podle ní porovnávat všechny ostatní (se stejnou nebo hierarchicky spřízněnou vlastností), včetně deklarací z první kategorie.

Druhá kategorie je hodně podobná té první, nicméně je zde restrikce kvůli pseudo-elementu (zde platí výjimka vysvětlena níže)/pseudo-třídě, což činí deklaraci závislou na specifických vlastnostech. Její aplikace není jistá za všech okolností. Proto není porovnávána s deklaracemi z první kategorie. Porovnávání a rozhodování o filtraci se uvažují pouze deklarace se stejným pseudo-elementem/pseudo-třídou. Zbytek je stejný jako v přechozím případě.

Deklarace ze *třetí kategorie* se porovnávají jen s deklaracemi se stejnými at-pravidly. Zbytek porovnávání je stejný jako u první a druhé kategorie.

Je tu *výjimka* u deklarací s pseudo-elementy. Zde je možné uvažovat o filtraci pouze, pokud referenční i porovnávaná deklarace mají stejný pseudo-element. Poté se porovnání uskuteční podle zmíněné logiky. Toto je dáno skutečností, že pseudo-elementy se odkazují na části stránky nedostupné přes HTML elementy. Proto se musí řešit individuálně.

Na závěr jsou zbylé finální deklarace v rámci elementu seřazeny vzestupně podle specifičnosti od nejnižší po nejvyšší. Takto se zajistí pořadí CSS kaskády odpovídající tomu původnímu, protože nová pravidla u refaktorizovaného budou mít specifičnost s hodnotou **0,1,0**. Výjimku tvoří pravidla s pseudo-elementy, kde specifičnost bude **0,1,1** (více v sekci 4.7). Díky zmíněné separaci mezi prvky dosažitélne skrze pseudo-elementy a HTML elementy rozdíl ve specifičnosti nepředstavuje problém.

4.5 Zpracování finálních deklarací

Po roztřídení a filtraci je třeba připravit finálních deklarace pro závěrečný styl. Každou získanou deklaraci lze reprezentovat utility-třídu. Cílem je vygenerovat pro všechny HTML elementy mapování z finálních deklarací na utility-třídy s dodržením pořadí kaskády a co nejmenší redundancí výsledného kódu.

Nejjednoduším způsobem by bylo pro každou jednotlivou deklaraci vytvořit novou mapování s utility-třídou. U tohoto přístupu je problém s redundancí, jelikož deklarace by se s velkou pravděpodobností oběvovaly duplicitně a u tohoto

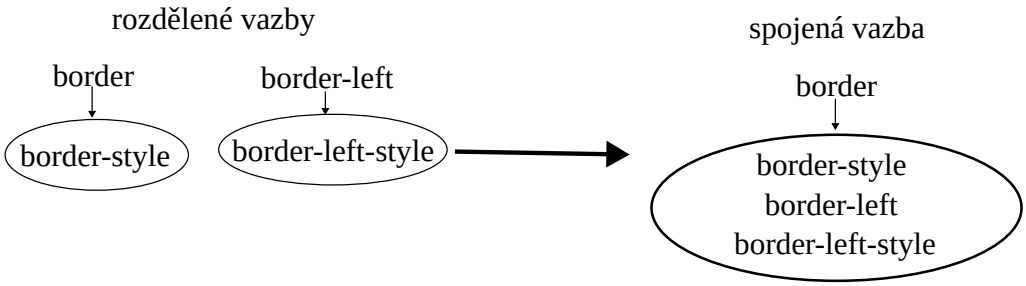
způsobu se neuvažuje žádná znovuvyužitelnost již vygenerovaných mapování.

Program se snaží co nejvíce o znovuvyužití již vygenerovaných mapování. Jelikož finální stylesheet bude jen jeden (pro každý HTML dokument) a pravidla mohou být obalena různými at-pravidly, jsou jednotlivá mapování ukládána do bloků. Blok je zde reprezentací prostorového regionu finálního stylesheetu. Každý blok má zaznamenaná at-pravidla, kterými jsou obalena všechna mapování z daného bloku. Bloky jsou vytvářeny postupně podle pořadí nálezu při procházení deklarací.

Zde také nastává problém, jelikož jednotlivá at-pravidla z bloků nemusí přesně odpovídat pořadí deklarací (seřazených vzestupně dle vyšší specifičnosti) u daného elementu.

Proto byl zvolen kompromis mezi znovuvyužitím a zachováním stylu. Program kontroluje pro deklarace se stejnou CSS vlastnostní nebo hierarchickou spřízněností v rámci daného elementu, jestli tam nedojde k možnému konfliktu v pořadí aplikace. Toto porovnání probíhá následujícím způsobem.

Program iteruje přes jednotlivé elementy. Nejprve identifikuje a uloží hierarchické vazby (vysvětleno v sekci 2.2.2) mezi CSS vlastnostmi z deklarací. Identifikuje se zde vazby mezi všemi vlastnostmi. Pokud se budou vyskytovat dvě vlastnosti, kde každá již bude mít nějakou vazbu a program zjistí, že jedna vlastnost může nastavit druhou, pak se on i jí podřízené vlastnosti (z daných deklarací) stanou podřízené té druhé. Příklad tohoto spojení lze vidět na obrázku 6.



Obrázek 6: Spojení spřízněných hierarchických vazeb do jedné

Toto spojení spřízněných vazeb pak napomáhá pro dodržení CSS kaskády.

Poté přejde k procházení jednotlivých deklarací. U každé CSS vlastnosti z deklarace program zjistí, do jaké z následujících kategorií spadá:

1. *unikátní*;
2. *duplikát*;
3. *s hierarchickou vazbou*;
4. *2. a 3. případ dohromady.*

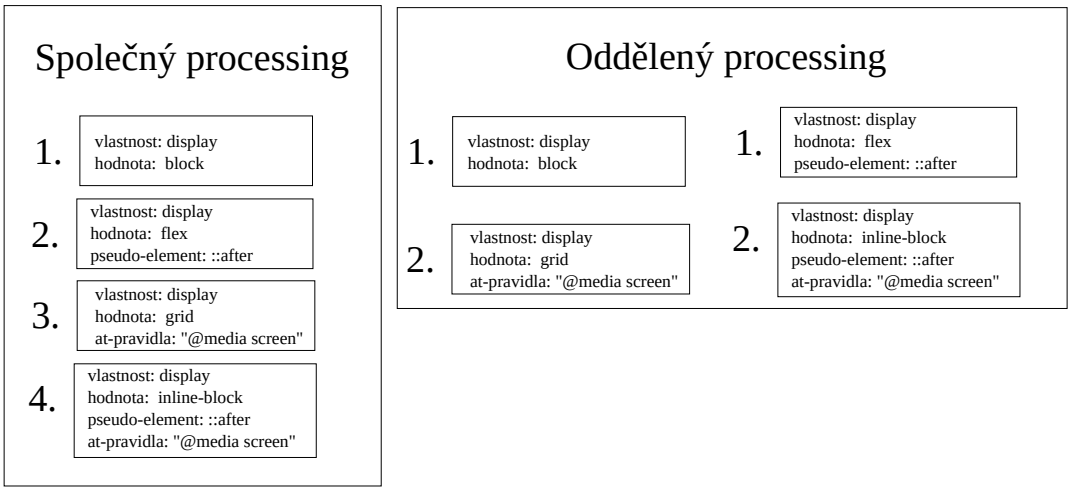
Unikátní vlastnost se v rámci elementu nachází jen jednou a nemá žádné hierarchické vazby. Není proto nutné řešit otázku možného konfliktu s dalšími deklaracemi. Pokud by byl specifikován přepínač *−−fcaOpt*, pak se tato deklarace uloží zlášť s indexem daného HTML elementu pro optimalizaci pomocí FCA (viz sekce 4.6.3).

Vlastnost označená jako *duplikát* se v deklaracích daného elementu vyskytuje vícekrát v jiných kontextech (různa at-pravidla, pseudo-třída). Aby se dodrželo jejich pořadí, program načte všechny všechny deklarace s danou vlastností v původním pořadí a poznačí si příslušnou vlastnost, aby ji dále neuvažoval (protože je zpracuje všechny najednou).

Pokud má vlastnost na sebe navázané další skrze *hierarchickou vazbou*, pak je všechny v původním pořadí načte a dále s nimi pracuje. Navázané vlastnosti jsou automaticky ignorovány, dokud nedojde řada na tu nadřazenou.

Poslední kategorie nastane pokud je daná vlastnost *duplikátem* a má zároveň *hierarchické vazby*.

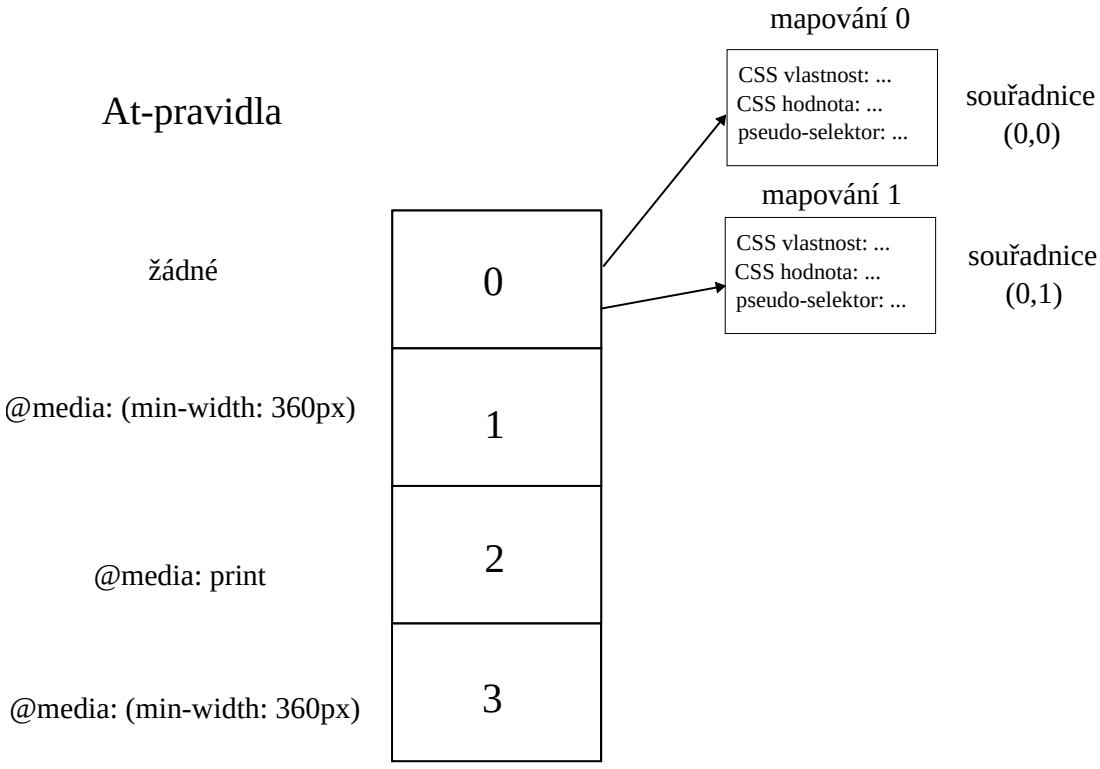
Je zde opět výjimka u deklarací s pseudo-elementy. Program je zpracovává zvlášť od ostatních deklarací, protože jejich oddělenost od HTML elementů umožňuje potenciální zmenšení počtu deklarací, u kterých se musí hlídat pořadí. Ukázku tohoto konceptu lze vidět na obrázku 7. Je možné vidět, že oddělené zpracování umožnilo získat dvě kratší posloupnosti, než jednu delší. Díky tomu je vetší šance, že se již taková posloupnost vyskytla a program by mohl využít existujících mapování. Deklarace se stejným pseudo-elementem se zpracovávají dohromady odděleně od těch s jiným pseudo-elementem.



Obrázek 7: Ukázka kontrastu společného a odděleného zpracovávání deklarací s pseudo-elementy a ostatních.

Program podle určené kategorie CSS vlastností vezme příslušné deklarace a vyhledá/vytvoří pro ně příslušná mapování, která odpovídají těmto deklaracím a jejich pořadí. Jak bylo zmíněno, všechna mapování jsou rozdělená do bloků podle at-pravidel. Tyto bloky se můžou opakovat (jinak by nebylo možné dodržet pořadí deklarací). Nejprve je zkontrolováno, kolik existujicích mapování je

možné u daných deklarací znovu použít. Existující mapování jsou organizovaná ve struktuře, kde lze přistoupit k danému mapování pomocí jeho *souřadnice*. Tato souřadnice má tvar: **(index at-rule bloku, index mapování)**. Souřadnice reprezentuje rozmístění bloků a v nich uložených mapování. Ukázka tohoto rozmístění je na obrázku 8.



Obrázek 8: Ukázka rozložení bloků a obsažená mapování.

Podle pořadí deklarací se pak zkontroluje a hledá posloupnost znovuvyužitelných mapování.

Pokud se zjistí, že počet znovuvyužitelných mapování neodpovídá počtu deklarací, pak je třeba vygenerovat nové, případně i nový blok pro hodnotu at-pravidla dané deklarace. Toto mapování je vygenerováno, případně i s novým blokem tak, aby se nacházelo za poslední souřadnici nalezeného znovuvyužitelného mapování.

Pokud by byla souřadnice posledního nalezeného mapování **(4,7)**, tedy čtvrtý blok a sedmé pravidlo v rámci toho bloku, pak by minimální souřadnice pro nové mapování by v rámci HTML elementu byla **(4,8)** (pokud by deklarace spadala do stejného bloku).

Jakmile je nové mapování přidáno, program se znovu pokusí o získání maximální znovuvyužitelné sekvence mapování. Toto je důležité, protože po přidaném mapování do daného bloku je možné, že se tím podařilo vytvořit sekvenci s dalšími již existujícími souřadnicemi v pořadí deklarací.

Jakmile je délka sekvence rovna počtu deklarací, jsou tyto souřadnice uloženy

pro daný HTML element.

Toto pokračuje, dokud každá deklarace nebude mít své mapování nebo nebude uložena pro optimalizaci.

Pokud by se programu specifikoval přepínač *--noOrderKeeping*, pak by se všechny CSS vlastnosti považovaly za *unikátní*. Toto by způsobilo, že by se snížila nutnost generování dodatečných mapování, tedy redukce velikosti finálního stylesheetu. Pokud ale jsou použity i vlastnosti z dalších kategorií, může to narušit pořadí jejich aplikace, tím pádem i styl stránky. Proto je tento přepínač vhodný spíše jen pro experimentální účely.

4.6 Optimilizace

Pokud se podíváme na obrázek architektury 2, můžeme vidět, že program umožňuje celkem tři typy optimalizace.

První dvě je možné použít po rezoluci specifičností, tedy získání finálních deklarací. Sem spadají optimalizace pomocí *shlukování*. Shlukování CSS vlastností a CSS hodnot.

Ta třetí je k dispozici po vytvoření mapování utility-tříd a/nebo získaní všech *unikátních* CSS vlastností (nezávislých na pořadí jiných, viz 4.5). Aplikuje poznatky z *formální konceptuální analýzi* pro tvorbu nových pravidel dle určených deklarací.

4.6.1 Shlukování CSS vlastností

Shlukování CSS vlastností spočívá ve spojení nekolika CSS vlastností do jedné, aby se tím ušetřil celkový počet deklarací. Podporovanými vlastnostmi pro shlukování jsou `margin` a `padding`.

Obě vlastnosti mají stejnou syntaxi. Proto je možné je rozložit až na čtyři individuální hodnoty, viz [8]. Toto je zároveň i podporovaná forma, se kterou program pracuje.

Pokud je specifikován přepínač *--propOpt*, pak jsou všechny `margin` a `padding` vlastnosti přítomné v daných HTML elementů převedeny do maximální rozšířené formy se čtyřmi hodnotami. Chybějící hodnoty jsou doplněny podle použité syntaxe, viz [8], [22] a [23].

Aby je mohl program převést, nesmí ani jedna z vlastností obsahovat *CSS globální hodnoty* - hodnoty, které můžou být uvedeny u každé CSS vlastnosti [9]. Dle [9] to jsou vlastnosti:

- *initial*;
- *inherit*;
- *revert*;
- *revert-layer*;
- *unset*.

Nemůže je použít, protože jejich povolená syntaxe je pouze jednoslovná, tedy ne v rozšířené formě. Pro představu, jak převod do rošířené formy vypadá, uvažujme následující deklaraci:

```
margin: 0;
```

ta bude převedena na:

```
margin: 0 0 0 0;
```

Oba zápisy jsou ekvivalentní.

Následně se pak v rámci elementu zjistí, jestli nejsou přítomny deklarace ve stejném kontextu (stejná at-pravidla + pseudo-element/pseudo-třída), které by se v rámci hierarchického vztahu u *zkratkových vlastností* mohly s nimi sloučit.

Pro **margin** to jsou:

- *margin-top*;
- *margin-bottom*;
- *margin-left*;
- *margin-right*.

Pro **padding** to jsou:

- *padding-top*;
- *padding-bottom*;
- *padding-left*;
- *padding-right*.

Pokud jsou přítomny, pak jsou shluknuty se svou zkratkovou vlastností. Shlukování bez přítomné zkratkové vlastnosti neproběhne, můsí tedy být přítomná v rámci aplikace deklarací po rezoluci specifičnosti. Pro ilustraci uvažujme:

```
margin: 0 0 0 0;
```

a

```
margin-top: 12px;
```

Při zapnuté optimalizaci CSS vlastností program z techto dvou deklarací vytvoří jednu:

```
margin: 12px 0 0 0;
```

která je ekvivalentní s těmi předchozími.

U slučovaných deklarací ze stejného kontextu se nemusí hlídat jejich pořadí, jelikož všechny deklarace již prošly rezolucí specifičností a filtrací.

4.6.2 Shlukování CSS hodnot

Shlukování CSS hodnot je zaměřeno na sdružování vybraných CSS vlastností spojených s nastavením barev a prostorového rozvržení stránky.

Podporovanými CSS vlastnostmi pro nastavení barvy jsou:

- *color*;
- *background-color*.

Podporované CSS vlastnosti pro nastavení prostorového rozvržení jsou:

- *margin*;
- *margin-top*;
- *margin-bottom*;
- *margin-left*;
- *margin-right*;
- *padding*;
- *padding-top*;
- *padding-bottom*;
- *padding-left*;
- *padding-right*;
- *height*;
- *width*;

U vlastností barev se ignorují hodnoty *transparent* a *currentcolor*. Ačkoliv *transparent* hodnota nemá žádnou barvu (její RGB reprezentace je (0 0 0 / 0%)) [17], pořád by se mohla smíchat s ostatními. Toto mělo za následek velké rozdíly v kontrastech, proto se nepoužívá při shlukování. *currentcolor* se neuvažuje, jelikož se nejedná o konkrétní hodnotu barvy, kterou by šlo přímo získat (nastavuje přejímání barvy z `color` vlastnosti elementu nebo podle CSS dědičnosti) [17].

U vlastností pro nastavení prostorového rozvržení se pracuje s hodnotami, které mají uvedené jako svojí jednotku *pixel*. Program je schopný uvažovat také *rem* [1], relativní jednotku, která nastavuje velikost podle nastavené hodnoty a velikosti písma kořenového elementu stránky. Program převádí *rem* jednotky na pixely pro jednodušší zpracování. Také platí, že vlastnosti `margin` a `padding` budou převedeny do rozšířené formy (přesně jako v sekci 4.6.1).

Narozdíl od optimalizace vlastností se tento proces neprovádí v rámci jednoho elementu, nýbrž mezi všemi, nehledě na kontexty jejich CSS deklarací. Program prochází všechny podporované CSS vlastnosti. Podle té, kterou momentálně uvažuje, vybere všechny možné deklarace se stejnou vlastností. Z těch pak nashromáždí všechny nastavené hodnoty.

Ty pak postupně zkoumá a provádí porovnání rozdílu jejich hodnot. Ty, které se vůči právě uvažované referenční hodnotě nacházejí v povoleném limitu, jsou u ní zaznamenány. Povolené limity rozdílu u obou typů vlastností jde nastavovat pomocí přepínačů *--maxSpaceDiff* a *--maxColorDiff*. U prostorových vlastností je vypočítán pro porovnání absolutní hodnota rozdíl obou hodnot, který se pak porovná s nastaveným limitem. U barev je proces více sofistikovanější. Program pro porovnání barev používá algoritmus *Delta E 2000* [24] implementovaný knihovnou *Color.js*.

Jakmile má nashromážděné shluky, zvolí z nich ten, ve kterém je nejvíce nashromážděných prvků. Z toho pak vybere tu hodnotu, jejíž rozdíl s tou referenční je minimální. Poté se přejde k vytváření nové hodnoty.

U prostorových vlastností se vypočítá aritmetický průměr referenční a druhé hodnoty. U `padding` a `margin` vlastností toto provádí dokonce tak, že uvažuje jednotlivé pozice z jejich rozšířené formy.

U barev je jejich míchání realizováno pomocí knihovny *Color.js*.

Po vytvoření je nová hodnota vložena se do všech deklarací, co obsahovali jednu ze dvou původních hodnot. Nehledí se zde na at-pravidla ani pseudo-elementy/pseudo-třídy. U `padding` a `margin` nahradí jednu ze čtyř hodnot.

Optimalizace se provádí, dokud je možné v rámci nastavených limitů ještě něco najít.

4.6.3 Optimalizace pomocí formálního konceptu

Deklarace, u kterých není nutné hlídat pořadí (viz 4.5), jsou při specifikovaném přepínači *--fcaOpt* poslány na optimalizaci pomocí *formálního konceptu*.

Program iteruje skrze všechny kombinace nalezených pseudo-tříd/pseudo-elementů a at-pravidel ze zaznamenaných deklarací. Podle dané kombinace projde záznamy a vybere ty elementy, které obsahují alespoň jednu

z deklarací odpovídající kontextové kombinaci. Jsou tak získány množiny *objektů* (elementů) a *atributů* (deklarací).

Ty jsou pak použity pro konstrukci tabulky, která reprezentuje jejich *relaci*. Sloupce tabulky korespondují s CSS deklaracemi a řádky odpovídají vybraným indexům HTML elementů. Dále je pak do dané buňky v tabulce zapsána hodnota: **0** – *neobsahuje deklaraci* a **1** – *obsahuje deklaraci*. Zkonstruovaná tabulka je zároveň *binární maticí*. Jakmile je kompletní, program ji uloží jako CSV soubor.

Následně pak pomocí *GNU/Octave* spustí převzatý skript,[26] který implementuje algoritmus *GreConD*. Skriptu je předána na vstupu matice z uloženého CSV souboru. Po dokončení výpočtu rozkladu si program uloží vyprodukované $\mathcal{A}$ a $\mathcal{B}$ matice. Z nich pak zkonstruuje nalezené *formální koncepty* a použije je pro vytvoření shluků CSS deklarací, které pak přiřadí příslušným elementům a uloží si je pro generování závěrečného stylu.

Při specifikovaném přepínači *--fcaDebugDir* program uloží ve specifikované složce JSON soubor s vyprodukovanými výstupy z každé iterace.

4.7 Generování stylesheetu

Po dokončení refaktorizace, je program připraven vygenerovat nový CSS styl. Nejprve jsou vygenerovány obecná at-pravidla a at-pravidla, která obsahují pouze deklarace. U těch se žádná refaktorizace neprovádí.

Podle zvolených přepínačů z tabulky 2 je pak vygenerována textová podoba korespondujících CSS pravidel pro mapování utility-tříd a/nebo FCA shluků. Pro každé pravidlo je vygenerován specifický selektor.

Jména jednotlivých selektorů jsou následující:

- *c-{hexadecimální index mapování}* – utility třídy
- *fca-{hexadecimální index mapování}* – fca shluky

Jednotlivé selektory jsou pak uloženy a s němi i vazby na HTML elementy, na které se aplikují. Výsledná specifičnost téměř u všech pravidel bude **0,1,0**. To samé platí i pro mapování, které obsahují selektor pseudo-třídy. Pro udržení specifičnosti na hodnotě **0,1,0** jsou dané pseudo-třídy zaobaleny do pseudo-třídy `:where(...)` ve tvaru:

```
.class_name:where(pseudo_class_selector)
```

Výjimku tvoří pravidla s pseudo-elementy. Ty není možné dle vložit do `:where()`. Z toho důvodu jsou vytvořena tradičním způsobem. Nová pravidla s pseudo-elementy budou mít specifičnost **0,1,1**. Opět to nevadí díky oddělěnosti HTML elementů a prvků z oddělené části stránky.

[26] https://github.com/martin-trnecka/matrix-factorization-algorithms/blob/master/GreConD.m

Generátor umožňuje také vygenerovat pravidla s neznámou shodou nebo nevalidními selektory. Pokud u se u těchto pravidel vyskytnul *seznam selektorů*, jsou z něj odebrány všechny selektory, u kterých zjištěna shoda. Celkové rozložení vygenerovaných částí nového stylesheetu v závěru je:

1. Obecná/pouze s deklaracemi at-pravidla
2. utility-třídy
3. FCA pravidla
4. Pravidla bez shody/s nevalidními selektory

Nový stylesheet je pak uložen do specifikovaného výstupního adresáře. Bude mít stejným název jako HTML dokument, pro který je určen.

Stojí za zmínku, že generátor při vytváření nových pravidel a jejich at-pravidel uvažuje sousednost podobných at-pravidel (vzniklých při tvorbě mapování). Pokud je lze spojit do obecnější pořadí, pak to generátor udělá. Pro příklad uvažujme následující bloky at-pravidel v příkladu 9:

```
/*Block 1*/
@media (min-width: 1200px){
    @media (max-width: 1400px){
        ...
    }
}

/*Block 2*/
@media (min-width: 1200px){
    @media (max-width: 1300px){
        ...
    }
}
```

Zdrojový kód 9: Rozdělené bloky at-pravidel

V ukázce 9 lze vidět, že tyto sousedící bloky at-pravidel mají společnou `@media (min-width: 1200px)` část. Generátor proto oba bloky spojí dohromady, jak je vyobrazeno v ukázce 10:

```
/*Common block*/
@media (min-width: 1200px){
    /*Block 1*/
    @media (max-width: 1400px){
        ...
    }
    /*Block 2*/
    @media (max-width: 1300px){
        ...
    }
}
```

Zdrojový kód 10: Spojené bloky at-pravidel

4.8 Vložení nového stylu do stránky

S novým vygenerovaným stylem a uloženými vazbami již zbývá pouze navázat nové CSS na korespondující HTML elementy. Odkazy na ně jsou zaznamenané pomocí *tagData* HTML atributů, jak bylo popsáno pří zaznamenávání vazeb shody v sekci 4.3. Zde jsou pak do atributu `<class>` uložena jména selektorů vygenerovaných pravidel.

Kromě tohoto program porovná původní hodnoty `<class>` a `<id>` atributů a odebere z nich selektory pravidel, které mají nalezenou shodu a nejsou obsaženy v žádném nevalidním/pravidle bez shody.

Nyní zbývá napojit odkaz na nový CSS stylesheet s refaktorovanými styly. Pokud HTML dokument obsahoval pouze vnořené styly, pak je vytvořen nový `<link>` element s odkazem na nový CSS stylesheet. Pokud se tam již existující `<link>` element vyskytoval, program jej vezme, odstraní jeho původní atributy a napojího ho na nový stylesheet.

Všechny přebytečné `<link>` a `<style>` elementy jsou odstraněny z dokumentu. Stylesheety, které se nenacházely na lokálním stroji a jsou referencované v dokumentu, pravděpodobně pochází ze síťového zdroje. Reference na ně jsou ponechány.

5 Experimentální část a výsledky

Tato kapitola se zaměřuje na prezentaci činnosti nástroje a vyhodnocení experimentů na datasetu vytvořeném z webových stránek.

5.1 Tvorba datasetu

V kapitole 1 bylo zmíněno, že program je zaměřen na statické webové stránky. Pro tvorbu datasetu bylo vybráno pět různých stránek. Byly vybrány následující:

- *www.ceskaposta.cz* (web České pošty);
- *www.inf.upol.cz* (web Katedry informatiky Univerzity Palackého);
- *www.nm.cz* (web Národního muzea);
- *www.npu.cz* (web Národního památkového ústavu);
- *www.npcs.cz* (web Národního parku České Švýcarsko).

Ze zmíněných adres byly dne 2. 8. 2023 staženy zdroje stránek pomocí programu *wget*. Použit byl tento příkaz (stejným stahuje program zdroje stránky při specifikovaném přepínači *--pageAddress*):

```
wget --restrict-file-names=unix \
    --no-directories \
    --adjust-extension \
    --page-requisites \
    --convert-links \
    --directory-prefix {PAGE_ADDRESS} {PAGE_ADDRESS}
```

kde `PAGE_ADDRESS` hodnota přesně odpovídala jménům položek z uvedeného výčtu stránek.

Wget převede stránky převede do formy, aby je šlo otevřít i lokálně. Není zachovaná původní funkčnost/podoba stránek v porovnání s tím, když by se otevřely v prohlížeči, nicméně pro demonstraci činnosti nástroje jsou postačující.

5.2 Experimenty

S kompletním datasetem nyní můžeme ukázat činnost programu. Celkově byly provedeny dva typy experimentů. U jednoho byl spuštěn nástroj se standardním během bez optimalizací. Ve druhém byla použita optimalizace formálním konceptem.

U prvního experimentu byl nad každou složkou se zdroji stránky spuštěn program s následujícími parametry:

```
node index.js  --inDir {WEB_PAGE_FOLDER} \
               --outDir {OUTPUT_FOLDER} \
               --chromeBin {PATH_TO_CHROMIUM_BROWSER} \
               --onlyRefactored \
               --parseCustomProps
```

Z příkazu jde vidět, že nebyly aplikovány žádné optimalizace. Dále pak, že ve finálním stylesheetu budou pouze nová vygenerovaná pravidla a to, že program bude uvažovat *vlastní* CSS pravidla.

U druhého experimentu byl nad každou složkou se zdroji stránky spuštěn program s následujícími parametry:

```
node index.js  --inDir {WEB_PAGE_FOLDER} \
               --outDir {OUTPUT_FOLDER} \
               --chromeBin {PATH_TO_CHROMIUM_BROWSER} \
               --onlyRefactored \
               --parseCustomProps \
               --fcaOpt \
               --fcaDebugDir {FCA_DEBUG_OUTPUT_FOLDER}
```

Tento příkaz je téměř identický s tím prvním, jen dodatečně používá FCA optimalizaci a ukládá její výstupy.

U obou experimentů program překopíroval původní obsah stránek do výsledkového adresáře, aby je šlo i po refaktorizaci načíst. Následně prošel a vygeneroval pro všechny HTML dokumenty (pokud měly nějáký validní CSS styl) nový stylesheet podle původních CSS stylů. Pokud potencionálně některý původní CSS stylesheet nebyl uveden v dokumentech, program ho ponecha v původní podobě u vygenerovaných výsledků.

Poté ještě provedl modifikace HTML dokumentů, kde zapsal příslušná jména selektorů nových pravidel do `class` atributů a odebral přebytečné hodnoty atributů a reference na zpracované CSS styly (viz sekce 4.8).

Následně se provedlo porovnání původního a výsledného CSS v rámci jednotlivých stránek. Pro analýzu byl vytvořen a použit skript, který za pomocí *puppeteer-core* prošel a extrahoval CSS kódy na analýzu. Princip získávání je stejný jako u implementovaného nástroje (externí i vnořené CSS) s vypnutým JavaScriptem stránky. Před analýzou byl každý CSS kód zformátován, aby se

nezkreslily výsledky (některá CSS z datasetu jsou komprimovaná a generované styly jsou formátované).

U druhého experimentu byly uloženy i výstupy optimalizace formálním konceptem.

Získané CSS kódy byly analyzovány (počet unikátních selektorů, různé specifičnosti, počet řádků kódu, ...) pomocí knihovny *@projectwallace/css-analyzer*. Výstupy analýzy byl uloženy jako JSON soubor pro každý CSS kód v jedné ze dvou podob:

1. `{STYLESHEET_NAME}_analysis.json`;
2. `style_element_{STYLE_ELEMENT_INDEX}_analysis.json`.

První tvar pojmenování je pro výstup analýzy CSS ze stylesheetu a ten druhý pro CSS ze `<style>` elementů. Zmíněné JSON soubory lze najít v adresáři experimentů, viz sekce A.

Poté byla použita knihovna *@projectwallace/css-code-quality*. Ta je schopná podle svých nastavených vnitřních limitů spočítat skóre tří veličin (*Komplexita*, *Udržitelnost* a *Výkonnost*). Podle popisu z oficiálního repozitáře[27] knihovny jsou tyto veličiny vysvětleny takto:

- *Komplexita* – složitost případných změn.
- *Udržitelnost* – čitelnost kódu.
- *Výkonnost* – případné následky na výkonnost.

Podle [25] jsou hodnoty skóre počítány jako $0 - 100$ bodů. Hodnoty těchto metrik byly spočítány pro každý CSS kód (původní i vygenerované) pro danou stránku. Je ze však problém, že CSS kódů může být více. Pokud by se všechny CSS kódy spojily v jeden a nad ním by se vypočítala skóre, mohlo by to zkreslit výsledky podle nastavených hodnot knihovny (mohlo by to vyústit i v nevalidní CSS). Aby se tomuto předešlo, jsou pro každý CSS kód spočítána skóre individuálně a zprůmerována podle rozsahu aktuálního a celkového CSS kódu takto:

$$adjustedScore = floor(\frac{calculatedScore \cdot styleSourceCodeLines}{totalSourceCodeLines})$$

Tento vzorec zohledňuje spočítáné skóre v rámci rozsahu daného CSS kódu a všech ostatních. Pokud by jedno CSS mělo malý rozsah a vysoká skóre a pak druhé, rozsáhlejší CSS kód by mělo nižší skóre, mohly by se tyto hodnoty během průměrování ztratit. Na konci jsou spočítaná skóre sečtena v rámci dané stránky. V tabulkách 4, 5 a 6 lze vidět výsledky spočtených skóre původní i vygenerovaného (z obou experimentů) CSS.

[27] https://github.com/projectwallace/css-code-quality#readme

Jde vidět, že v případě obou experimentů bylo u vygenerovaného CSS dosaženo vyšších výsledků. Nicméně je nutné zohlednit pár skutečností.

Některé části stránek (měnící se, využívající nepodporované konstrukce) nejsou zohledněny a proto není kompletně zachována původní podoba stránek.

Vysoká skóre *Udržitelnosti* ve vygenerovaných výsledcích nezohledňují jména nových selektorů. Kvůli velkému množství možných kombinací jsou generovány pojmenování tak, jak je popsáno v sekci 4.7. Tento styl nemusí být vhodný, jelikož neobsahuje infomaci o deklaraci/stylech, které obsahuje.

U obou experimentů je přítomné velké množství hodnot v `class` atributech v HTML dokumentech. Počet použitých utility-tříd/FCA pravidel se odvíjí od složitosti původního CSS. Čím méně je nutno hlídat kontexty původních pravidel a jejich pořadí, tím více úspornější vygenerovaný styl bude.

U expemetimentu s FCA optimalizací je v HTML počet odkazů o něco menší díky slučování deklarací. Toto by šlo dále lehce vylepšit použitím $--propOpt$ a $--valOpt$ přepínačů.

Když porovnáme skóre výsledků ze standardního běhu a běhu s FCA optimalizací, je vidět, že skóre FCA optimalizace jsou trochu nižší. Zde je potřeba zmínit, že síla FCA optimalizace je omezená podle množství kontextů CSS pravidel.

Aby se zachovala původní podoba stránky, je nutné v určitých případech (viz sekce 4.4) dodržet původní pořadí CSS deklarací. Kvůlit tomuto jsou na FCA optimalizaci posílány deklarace, u nichž není nutné hlídat pořadí. Ty zbylé jsou vytvořeny pomocí utility-tříd. Čím méně je nutno uvažovat pořadí a kontexty pravidel, tím lepší výsledky FCA optimalizace vrátí.

Přes zmíněné skutečnosti experimenty ukázaly, že implementovaný nástroj je z velké části schopen zachovat původní podobu stránek a to za použití pravidel se specifičnostmi s hodnotami **0,1,0** a **0,1,1**, viz ukázka vizuálních podoby stránky z datasetu a výsledku v sekci B.

	Původní CSS		
Stránka	Komplexita	Udržitelnost	Výkonnost
www.ceskaposta.cz	54	55	75
www.inf.upol.cz	96	99	85
www.nm.cz	73	76	71
www.npcs.cz	79	92	84
www.npu.cz	66	54	68

Tabulka 4: Výsledky spočtených skóre původního CSS stránek

	Vygenerované CSS – standardní běh		
Stránka	Komplexita	Udržitelnost	Výkonnost
www.ceskaposta.cz	95	95	95
www.inf.upol.cz	100	100	100
www.nm.cz	96	99	95
www.npcs.cz	100	100	90
www.npu.cz	99	100	95

Tabulka 5: Výsledky spočtených skóre vygenerovaného CSS stránek za standardního běhu

	Vygenerované CSS – FCA optimalizace		
Stránka	Komplexita	Udržitelnost	Výkonnost
www.ceskaposta.cz	95	94	90
www.inf.upol.cz	100	100	95
www.nm.cz	97	93	92
www.npcs.cz	100	98	85
www.npu.cz	100	95	89

Tabulka 6: Výsledky spočtených skóre vygenerovaného CSS stránek s FCA optimalizací

Závěr

V této práci byl ukázán jeden z možných způsobů automatizované refaktorizace CSS. Ačkoliv implementovaný nástroj není vhodný pro univerzální použití napříč všemi webovými technologiemi, experimenty ukázaly, že dokáže v rámci možností zachovat původní podobu stránek (statických) a zároveň vygenerovat CSS, které vykazuje lepší výsledky (v rámci použitých metrik při vyhodnocování). Zároveň je možné požít optimalizace FCA, která dokáže zredukovat počet odkazů v HTML dokumentu či optimalizace shlukováním pro další redukci CSS kódu.

Naskytují se možná vylepšení, mezi která patří:

- Mapování jmen generovaných selektorů na specifikovaná;
- Uvažování složitějších konstrukcí selektorů, viz sekce 4.3;
- Další možné optimalizace/způsoby generování CSS pravidel;
- Oveřování CSS podle nejnovějších standardů;
- Vylepšení výkonu.

Program je možné potencionálně rozšířit o další techniky refaktorizace CSS kódu, protože vnitřní reprezentace aplikovaných deklarací a zaznamené vazby na HTML elementy nejsou omezené pouze na použité v této práci.

Conclusions

This thesis showed one of the possible approaches using automatic refactoring of CSS. The implemented tool is not suitable for use across all web technologies however, the experiments showed the tool is capable of keeping within the bounds of possibilities the original visual form of website (static one). It can even generate a new CSS for this website which shows better results (within the bounds of metrics used during the experimental phase). It also allows using FCA optimization for reducing number of references in HTML document and/or clustering optimizations for further CSS code reduction.

There are possible options for optional upgrades.

- Mapping generated rules on customized selector names;
- Considering more complex selector constructions, viz section 4.3;
- Adding more optimization techniques or new refactoring approach;
- Checking CSS according to the newest standards;
- Performance improvement.

It's possible to add more refactoring techniques into the tool because inner representations of applied declarations and saved bindings to HTML elements aren't restricted to the techniques used in this thesis.

A Obsah odevzdaných elektronických dat

`css-refactor-transformer/`
: Adresář se zdrojovými kódy nástroje pro refaktorizaci.

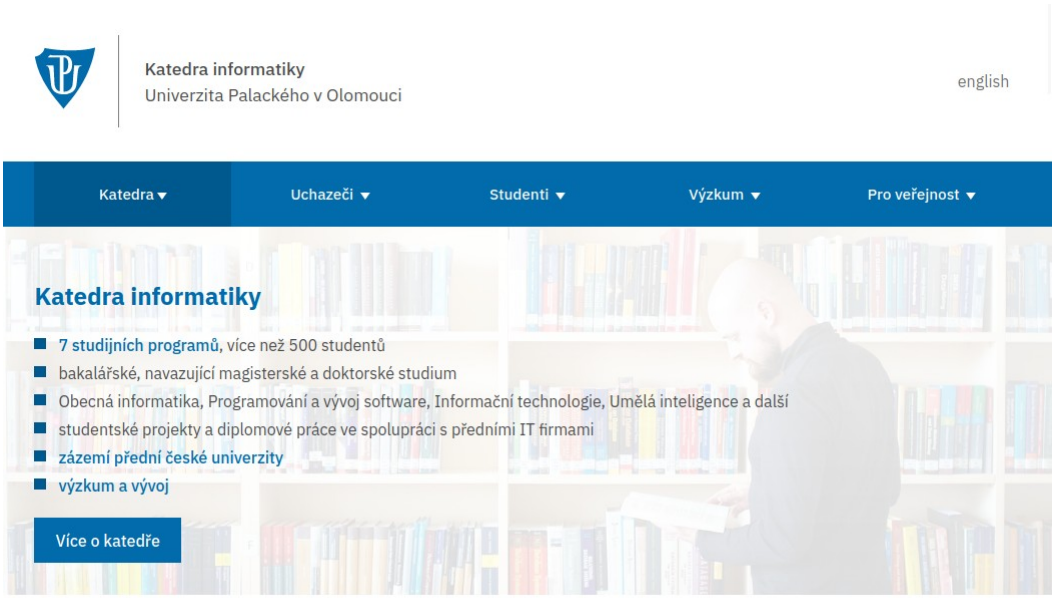
`css-result-evaluator/`
: Adresář se zdrojovými kódy skriptu použitým při analýze a vyhodnocování výsledků v experimentální části.

`data/`
: Adresář s použitým datasetem a vygenerovanými výstupy z experimentální části.

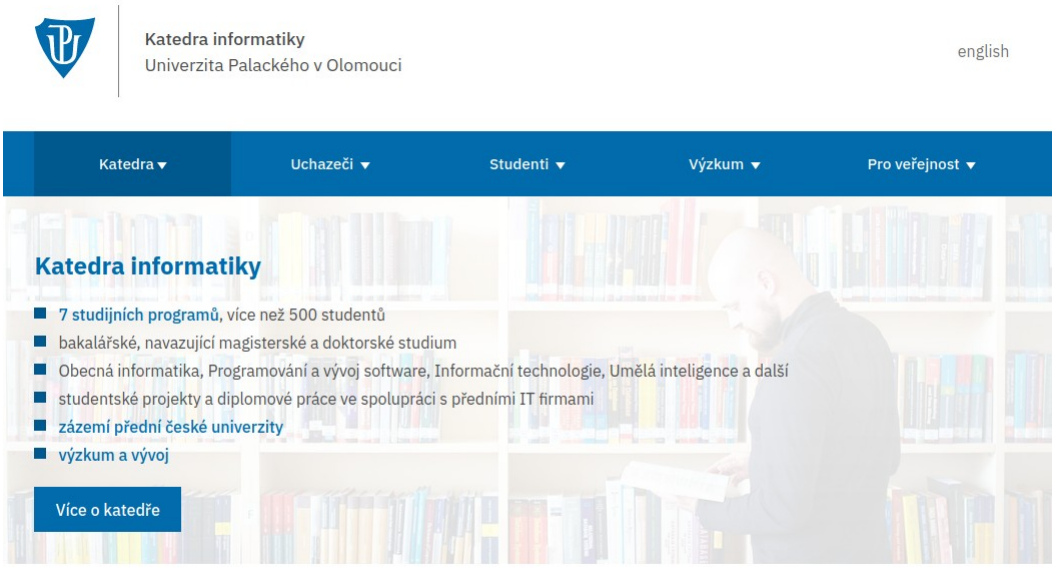
`text/`
: Adresář obsahující PDF soubor práce a zdrojové kódy v LaTeXu a potřebné soubory pro vygenerování textu práce.

`README.txt`
: Soubor s postupem pro zprovoznění nástroje a popisem struktury odevzdávaného adresáře.

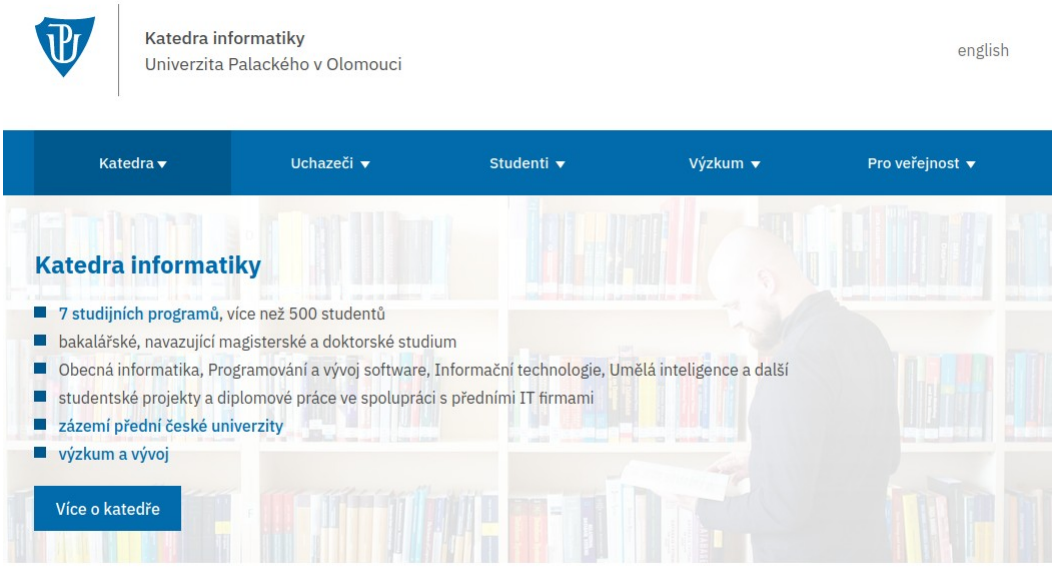
B Ukázky vizuální podoby stránky z datasetu a vygenerovaných výsledků



Obrázek 9: Ukázka načtené stránky `www.inf.upol.cz`, verze z datasetu.



Obrázek 10: Ukázka načtené stránky `www.inf.upol.cz`, verze z výsledku standardního běhu programu.



Obrázek 11: Ukázka načtené stránky `www.inf.upol.cz`, verze z výsledku běhu programu s FCA optimalizací.

Literatura

[1] RNDr. Trnečka M., PhD. *Tvorba webových stránek*. 2023. [Online, cit. 2023-06-13]. Dostupné z: https://www.dropbox.com/s/o37ans1111wctjo/skripta-web.pdf?dl=0.

[2] *Webové technologie, CSS preprocesory a postprocesory*. [Online, cit. 2023-07-18]. Dostupné z: https://www.dropbox.com/s/9h6uig6r4q76922/slidy.pdf?dl=0.

[3] *MDN Web Docs, At-rules*. [Online, cit. 2023-07-05]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/CSS/At-rule.

[4] Belohlávek, Radim. Introduction to formal concept analysis. *Palacky University, Department of Computer Science, Olomouc*. 2008, roč. 47.

[5] *Selectors Level 4*. [Online, cit. 2023-06-28]. Dostupné z: https://www.w3.org/TR/selectors-4/.

[6] *MDN Web Docs, :where()*. [Online, cit. 2023-07-18]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/CSS/:where.

[7] *W3C, List of CSS properties, both proposed and standard*. [Online, cit. 2023-07-06]. Dostupné z: https://www.w3.org/Style/CSS/all-properties.en.html.

[8] *MDN Web Docs, Shorthand properties*. [Online, cit. 2023-07-06]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/CSS/Shorthand_properties.

[9] *CSS Cascading and Inheritance Level 5*. [Online, cit. 2023-07-02]. Dostupné z: https://www.w3.org/TR/css-cascade-5/.

[10] *MDN Web Docs, Introducing the CSS Cascade*. [Online, cit. 2023-08-04]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/CSS/Specificity.

[11] *CSS Conditional Rules Module Level 3*. [Online, cit. 2023-07-05]. Dostupné z: https://www.w3.org/TR/css-conditional-3/.

[12] *DOM, Living Standard*. [Online, cit. 2023-06-28]. Dostupné z: https://dom.spec.whatwg.org/.

[13] Belohlavek, Radim; Vychodil, Vilem. Discovery of optimal factors in binary data via a novel method of matrix decomposition. *Journal of Computer and System Sciences*. 2010, roč. 76, s. 3–20. Dostupný také z: ⟨http://dx.doi.org/10.1016/j.jcss.2009.05.002⟩.

[14] Vanderkam, D. *Effective TypeScript: 62 Specific Ways to Improve Your TypeScript*. 2019. ISBN 9781492053712.

[15] *CSSTree AST format*. [Online, cit. 2023-06-28]. Dostupné z: https://github.com/csstree/csstree/blob/master/docs/ast.md.

[16] *Project Wallace, Metrics*. [Online, cit. 2023-08-02]. Dostupné z: https://www.projectwallace.com/docs/metrics.

[17] *CSS Color Module Level 4*. [Online, cit. 2023-07-29]. Dostupné z: https://www.w3.org/TR/css-color-4/.

[18] *HTML media Attribute*. [Online, cit. 2023-07-16]. Dostupné z: https://www.w3schools.com/TAGS/att_media.asp.

[19] *MDN Web Docs, @keyframes*. [Online, cit. 2023-07-09]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/CSS/@keyframes.

[20] *MDN Web Docs, Vendor Prefix*. [Online, cit. 2023-07-09]. Dostupné z: https://developer.mozilla.org/en-US/docs/Glossary/Vendor_Prefix.

[21] *MDN Web Docs, @layer*. [Online, cit. 2023-07-10]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/CSS/@layer.

[22] *MDN Web Docs, padding*. [Online, cit. 2023-07-10]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/CSS/padding.

[23] *MDN Web Docs, margin*. [Online, cit. 2023-07-10]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/CSS/margin.

[24] *Delta E 101*. [Online, cit. 2023-07-29]. Dostupné z: https://zschuessler.github.io/DeltaE/learn/.

[25] *Project Wallace, CSS Code Quality*. [Online, cit. 2023-08-02]. Dostupné z: https://content-project-wallace.vercel.app/css-code-quality.