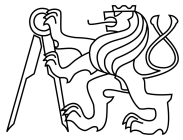
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ



ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	CSS preprocesor pro publikační systém Webgarden
Student:	Bc. Filip Vorel
Vedoucí:	Ing. Kamil Foltin
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce zimního semestru 2016/17

Pokyny pro vypracování

Vytvořte preprocesor pro kaskádové styly (CSS), který bude rozšiřovat funkcionalitu CSS. Zejména práci s různými proměnnými pro barvy, velikosti a další styly, makra pro automatizované vytváření bloků, práci s obrázky (např. barevné tónování) a další funkce. Cílem je zjednodušení a zrychlení vytváření CSS a jeho modifikace například pro různé barevné palety. Navrhněte meta-CSS jazyk a preprocesor, který bude umět vytvořit standardní CSS použitelné v běžných prohlížečích. Integrujte tento preprocesor do publikačního systému Webgarden.

Seznam odborné literatury

Dodá vedoucí práce.

L.S.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

prof. Ing. Pavel Tvrdík, CSc.
děkan

V Praze dne 18. prosince 2014

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra softwarového inženýrství

Diplomová práce

CSS preprocesor pro publikační systém Webgarden

Bc. Filip Vorel

Vedoucí práce: Ing. Kamil Foltin

9. ledna 2017

Poděkování

Děkuji své rodině za bezbřehou podporu, dále vedoucímu této práce Ing. Kamilu Foltinovi a kolegovi Ing. Lukášovi Alexandrovi za cenné rady.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 9. ledna 2017 .....

Odkaz na tuto práci

Vorel, Filip. *CSS preprocesor pro publikační systém Webgarden.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2017.

Abstrakt

Tato práce popisuje proces navržení a vnitřní funkcionalitu vytvořeného CSS preprocesoru a jeho implementaci do publikačního systému Webgarden.

Klíčová slova CSS preprocesor, Webgarden, PHP, HTML, SASS, LESS, Formální gramatika, OOP, TDD, Unit testování, Verzovací systém, Barevné modely, Návrhové vzory

Abstract

This thesis describes design and inner mechanism of created CSS preprocessor and its implementation in publishing system Webgarden.

Keywords CSS preprocessor, Webgarden, PHP, HTML, SASS, LESS, Formal grammar, OOP, TDD, Unit testing, Version control, Color models, Design Patterns

Obsah

Seznam obrázků

Úvod

Jazyk kaskádových stylů (CSS) je jedním ze základních kamenů tvorby webových stránek a slouží k popisu zobrazování jejich vzhledu. Mnohdy svou funkcionalitou nedostačuje, proto je často potřeba k vytvoření vzhledu i dalších nástrojů, jejichž využití tuto práci zpomaluje a komplikuje. Také díky svým predispozicím není standardní kód v jazyce CSS trvale udržitelný.

Z těchto důvodů vznikla pro webový projekt Webgarden potřeba vyvinout takovou aplikaci, která rozšíří jazyk CSS o nové vlastnosti a eliminuje potřebu využití dalších nástrojů.

Z této potřeby vychází nastavení cílů této diplomové práce, kterými jsou následující:

- Vytvořit teoretický návrh preprocesoru, který bude sestaven na míru požadavkům projektu Webgarden.
- Implementovat aplikaci s použitím stávajících technologií projektu Webgarden.
- Integrovat preprocesor do publikačního systému Webgarden.

V první kapitole práce představí problematiku jazyka "Cascading Style Sheets" neboli kaskádových stylů, jehož vlastnosti má navrhovaný preprocesor rozšiřovat. Druhá kapitola bude věnována výhodám využití preprocesoru a sepsání konkrétních požadavků na jeho funkce. Následující obsáhlá kapitola popisuje samotný proces návrhu preprocesoru a mechanismus jeho vnitřního fungování. V kapitole Implementace budou popsány stávající technologie využívané projektem Webgarden a následně bude přikročeno k samotné implementaci aplikace za využití těchto technologií. Pátá kapitola na komplexních příkladech předkládá výhody preprocesoru oproti jazyku CSS. Závěrečná kapitola bude věnována integraci hotového preprocesoru do publikačního systému Webgarden, ve kterém uživatelé budou moci výhody preprocesoru plně využít.

Finální navržený program bude nahrán na přiloženém CD, jako součást této práce.

Kapitola 1

CSS - Cascading Style Sheets

Cascading Style Sheets, neboli česky kaskádové styly, jsou jazykem pro popis zobrazovaní elementů dokumentů psaných ve značkovacích jazycích XHTML, HTML a XML. Dokumenty, psané v těchto značkovacích jazycích, slouží převážně pro zobrazování obsahu na internetu. Jazyk kaskádových stylů umožňuje odlišit zobrazení informací na obrazovce, při tisku či třeba v mluveném slovu[1].

Ukázka kódu 1.1: Nastaví velikost písma všech article elementů na 10 pixelů

```
article {
        font-size: 10px;
}
```

1.1 Využití

Jako mnoho ostatních standardů z oblasti webových stránek byl i tento standard navržen konsorciem W3C (agl. World Wide Web Consortium). Hlavním významem kaskádových stylů je možnost oddělit obsah dokumentu od jeho vzhledu. Standardní HTML toto oddělení plně neumožňuje, neboť vývoj byl značně ovlivněn vznikem hlavních internetových prohlížečů a jeho nedostatečným standardizováním[35]. Mnoho elementů jazyka HTML neslouží pouze ke strukturování obsahu ale i ke způsobu zobrazování.

Ukázka kódu 1.2: Následující text se zobrazí v tučném zvýraznění, aniž by byl pro strukturu dokumentu nějak významný.

```
<html>
  <b>CSS</b>
</html>
```

1.2 Syntaxe

Kaskádové styly se skládají ze seznamu pravidel. Každé pravidlo obsahuje sektor a blok deklarací. Blok deklarací následně obsahuje jednotlivé deklarace. Ty jsou složeny z identifikátoru následovaného dvojtečkou a hodnotou vlastnosti. Jednotlivé Deklarace dělí mezi sebou středník[17].

Ukázka kódu 1.3: Příklad CSS pravidla

```
header {
  background-color: black;
  font-size: 10px;
  padding: 5px !important;
}
```

Jako selektor pravidla bude určen header, mezi závorkami je určen blok deklarací. Např. background-color je deklarací s hodnotou black. Volitelně se v deklaraci může objevit sekvence !important, která za určitých podmínek zvýší význam deklarace.

1.2.1 Napojení kaskádových stylů na HTML stránku

Aby kaskádové styly mohla HTML stránka využívat, je nutné je do stránky zakomponovat. To se dá provést několika následujícími způsoby[17].

- Vložení je možné provést jako součást HTML souboru, přes element style.

  Ukázka kódu 1.4: Umístění do style elementu

  ```
  <style type="text/css">
  figure {
    width: 150px;
    height: 300px;
  }
  </style>
  ```

- Kaskádové styly se dají vložit přímo do HTML elementu pomocí atributu style.

  Ukázka kódu 1.5: Style atribut

  ```
  <div style="width: 150px; height: 300px;">
    Section with dimension 150x300px.
  </div>
  ```

- Kaskádové styly lze připojit jako externí soubor pomocí elementu link.

  Ukázka kódu 1.6: Element link

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

Připojit externí soubor s kaskádovými styly lze také pomocí hlavičky link.

Ukázka kódu 1.7: Hlavička pro externí soubor

```
Link: <style.css>; rel=stylesheet
```

V praxi se z důvodu oddělování závislostí nejvíce používá oddělení kaskádových stylů do externího souboru.

1.3 Verze

CSS má několik verzí. Každá staví na předchozí verzi a přidává nové funkcionality.

1.3.1 CSS verze 1

Zavádí celkovou syntaxi, použití selektorů, různé pseudotřídy a základní hodnoty s jejich jednotkami.

1.3.2 CSS verze 2

Tato verze přináší nové možnosti hlavně v pozicování elementů a možnosti v zobrazování toku textu oběma směry.

1.3.3 CSS verze 2.1

Tato verze opravuje chyby a odstraňuje některé nepoužívané vlastnosti předchozí verze. Snaží se o standardizaci celého CSS.

1.3.4 CSS verze 3

Zavádí se spolu se standardem HTML 5 a obsahuje nové vlastnosti jako např. animace, 2D a 3D transformace a nastavování míry neprůhlednosti prvků. Rozděluje standardizaci do několika tématicky příbuzných modulů, o těchto modulech se mluví jako o modulech úrovně 3.

1.3.5 CSS verze 4

V současné době neexistuje jednotný standard pro tuto verzi. O CSS4 se dá hovořit pouze u nových modulů (např. Flexbox) a u modulů, které rozšiřují moduly úrovně 3 na úroveň 4 (např. Selektory). Nově zavedené moduly se chápou jako moduly úrovně 1. Úrovně tedy nemusí odpovídat CSS verzím.

Následující práce zaručuje využití u standardizované CSS verze 3, měla by však být použitelná i u nově zavedených modulů a jejich vlastností.

1.4 Výhody

Jak již bylo zmíněno, hlavním přínosem použití jazyka CSS je možnost oddělení struktury dokumentu a stylu jeho zobrazení. To má značnou výhodu, pokud je potřeba celý systém zobrazování poupravit. Kaskádové styly je možné cachovat neboli ukládat do vyrovnávací paměti pro rychlejší odezvu prohlížeče. Při nastavování stylů stránky máme navíc možnost připravit zobrazování v určitých zařízeních, jako je například tiskárna[17].

1.5 Nevýhody

Nejednotná podpora CSS v hlavních prohlížečích způsobuje mnohdy komplikovanou práci při hledání ideálního vzhledu. CSS neumožňuje přistupovat k rodičovským prvkům, neumožňuje zavádět uživatelské proměnné a uživatelské funkce. Nedostatky jazyka CSS se snaží vyřešit navržený preprocesor.

Kapitola 2

Preprocesor

Preprocesorem se rozumí program zpracovávající vstupní data pro využití v jiném programu. V případě navrhovaného preprocesoru se bude jednat o předzpracování uživatelského pseudo-CSS kódu. Uživatelský pseudo-CSS kód bude představovat vstupní soubor ve formě textového řetězce. Vstupní textový řetězec bude napsán v syntaxi nově navrženého jazyku přímo pro preprocesor, tak aby rozšiřoval jazyk CSS o nové funkčnosti. Preprocesor pak tento pseudo-CSS kód převede na standardní kaskádové styly, použitelné pro zobrazování HTML stránek.

2.1 Existující nástroje

Nástrojů pro základní rozšíření jazyka CSS existuje celá řada. Jako výchozí bod pro funkcionalitu navrženého preprocesoru byly zvoleny společné vlastnosti dvou nejpoužívanějších preprocesorů - SASS a LESS[2].

2.2 Výhody nástrojů SASS a LESS oproti jazyku CSS

Liší se především syntaxí, ale oba dva podporují generování kódu z do sebe vnořených bloků, dědění vlastností, uživatelské proměnné, uživatelské funkce a funkce pro práci s barvami[3][4].

2.3 Nedostatky existujících nástrojů

Uvedené nástroje nedokážou provádět vyjímání vlastností. Vyjímáním vlastností se rozumí získání všech vlastností nějakého objektu, kromě zvolených. Například všechny vlastnosti objektu A kromě vlastností objektu B.

Dále zmíněné nástroje neumožňují přistupovat k obrázkům a provádět nad nimi základní úpravy, jako nastavení světlosti, změnu kontrastu a barvené tónování.

2.4 Požadavky na preprocesor

Po prozkoumání nabídky již existujících programů a vyslechnutí doporučení grafika webového projektu Webgarden, disponuje práce základem pro vytyčení cílů samotného preprocesoru. Těmto cílům jsou věnované následující kapitoly.

2.4.1 Jazyk rozšiřující CSS

Preprocesor by měl představit nový jazyk, který rozšíří jazyk CSS a umožní definovat následující vlastnosti.

2.4.1.1 Uživatelské proměnné

CSS neumí nadefinovat uživatelské proměnné, tím značně snižuje možnost lehce editovatelného kódu. Když se například rozhodneme změnit barvu, jsme nuceni ji změnit ve všech deklaracích kde je uvedena. Zavedením uživatelských proměnných se zvýší udržitelnost a variabilita kódu.

2.4.1.2 Uživatelské funkce

Další mechanismus, který ve standardním CSS chybí jsou uživatelské funkce s parametry. Navržený preprocesor proto zavádí tzv. mixiny, které představují programovou konstrukci podobnou matematickým funkcím. Na základě parametrů vstupu vytváří výstup.

2.4.1.3 Dědičnost

V objektově orientovaném programování se jedná o mechanismus předání stejných atributů a chování z jedné třídy na druhou. Standardní jazyk CSS dědičnost nepodporuje a nutí tak duplikovat definice pro každou třídu zvlášť.

2.4.1.4 Vnořování

Dalším mechanismem, který posiluje udržitelnost kódu, je vnořování. Vnořováním se rozumí organizování kódu do menších do sebe vnořených bloků kódu, kdy zanořenější blok získá vlastnosti od bloku nadřazeného.

2.4.1.5 Vyjímání vlastností

Tato operace by se dala chápat i jako rozdíl množin. Mnohdy je potřeba získat všechny vlastnosti množiny A, avšak kromě vlastností, jež náleží množině B. Navržený preprocesor toto řeší vyjímáním vlastností.

2.4.1.6 Barevné palety a barevné funkce

Největší podporu v hlavních prohlížecích má formát RGB (3.2.4.9), jedná se o aditivní způsob míchání barvy, jenž je tvořen třísložkovým formátem (red - červená, green - zelená, blue - modrá). Při tisku se používá model CMYK (3.2.4.9). Často je však potřeba upravovat jas a sytost barvy, k čemuž se tyto formáty nehodí. Blíže lidskému vnímání jsou modely HSV (3.2.4.9) a HSL (3.2.4.9)[36]. Jazyk CSS umožňuje pouze zobrazit barvy ve formátu RGB a od verze CSS3 i formát HSL[17]. Navržený preprocesor by měl umožňovat manipulaci s barvami ve všech zmíněných formátech, upravovat odstín, jas a sytost bez nutnosti dalšího nástroje. Z důvodu kompatibility by pak mělo být možné výslednou barvu převést na hexadecimální zápis RGB formátu a zajistit tak podporu ve všech prohlížečích.

2.4.1.7 Barevné tónování fotek a jiné manipulace s obrázky

Preprocesor by měl jednoduše umožňovat změnu tónu barvy obrázku bez použití externího nástroje a s využitím standardních knihoven PHP.

Kapitola 3

Návrh preprocesoru

Tato kapitola nejprve představí formální definice, se kterými návrh pracuje. Následně popíše jednotlivé fáze cyklu, ve kterém preprocesor operuje a ukáže detailněji vnitřní fungování algoritmů jednotlivých fází.

3.1 Formální definice

3.1.1 Formální jazyk

Formální jazyk v oblasti matematiky, informatiky a lingvistiky značí množinu slov nad určitou abecedou Σ. Abeceda Σ je konečná množina znaků. Slova reprezentují množinu všech konečných řetězců znaků nad abecedou Σ, značí se Σ^*. Formální jazyk může být definován různými způsoby, například:

- generován formální gramatikou,
- slovy vyhovujícími určitému regulárnímu výrazu,
- slovy akceptovatelnými nějakým automatem, například Turingovým strojem

3.1.2 Formální gramatika

Navržený jazyk preprocesoru bude generován formální gramatikou. Formální gramatika představuje seznam pravidel pro popis množin. Definuje jí čtveřice (N, Σ, P, S), kdy N představuje konečnou množinu neterminálních symbolů (neterminálů), Σ je konečná množina terminálních symbolů tak, že žádný symbol nepatří do N a N zároveň. P je konečná množina odvozovacích pravidel. Každé pravidlo je tvaru

$$(\Sigma \cup N)^* N (\Sigma \cup N)^* \longrightarrow (\Sigma \cup N)^*$$

a S je prvek z N nazývaný počáteční symbol. Symbolem ε se značí prázdný řetězec[5].

3.1.3 Syntaktický strom

Syntaktický strom představuje strukturu grafu, kdy uzly tvoří operátory a listy operandy. Vniká při syntaktické analýze textu vstupního souboru při fázi parsování (podkapitola 3.2.3) a reprezentuje významy jednotlivých pravidel navrženého jazyka[38].

3.2 Mechanismus fungování preprocesoru

Preprocesor vychází z mechanismu pro zpracování klasického CSS uvedeného v dosavadní pracovní verzi Syntax Module Level 3[1] a rozšiřuje ho především o zavedení dědičnosti, uživatelské proměnné, uživatelské funkce a práci s barvami a obrázky. Navrhuje nový jazyk pro definování těchto vlastností. Samotný mechanismus přeměny vstupního kódu na standardní CSS postupně prochází pět hlavních fází.

1. **Nastavení preprocesoru (podkapitola 3.2.1)**

   Provede nakonfigurování aplikace.

2. **Tokenizace (podkapitola 3.2.2)**

   Přemění textový vstup na menší významové bloky.

3. **Parsování (podkapitola 3.2.3)**

   Přemění významové bloky na syntaktický strom definovaný navrženým jazykem.

4. **Preprocesování (podkapitola 3.2.4)**

   Nad syntaktickým stromem se provede aplikace zavedených vlastností jako jsou dědičnost, uživatelské proměnné, uživatelské funkce.

5. **Výpis (podkapitola 3.2.5)**

   Program vrátí výstup v syntaxi jazyka CSS, který je použitelný pro webové stránky.

3.2.1 Nastavení preprocesoru

V preprocesoru by mělo být umožněno jednoduše volit prostředí, ve kterém se bude pracovat. Pokud bude například stát jako samostatná aplikace, bude moci přistupovat i k zápisu do souborového systému. Ve webovém prostředí bude tuto možnost z bezpečnostních důvodů možné zakázat. V případě zneužití webové stránky by totiž mohly být funkce pracující se souborovým

systémem použity k získání choulostivých informací či k poškození webových stránek samotných. Dále by měl preprocesor umožňovat konfigurovat v jakém formátu se budou vypisovat barvy do finálního výstupu. Tato konfigurace je potřebná z důvodu zpětné kompatibility se staršími prohlížeči.

3.2.2 Tokenizace

Tato fáze přeměňuje proud vstupních znaků na tokeny. Tokeny představují tématické celky s vnitřním významem. Slouží jako základ pro významová pravidla tvořená ve fázi parsování. Standardní jazyk CSS definuje tyto tokeny: <ident-token>, <function-token>, <at-keyword-token>, <hash-token>, <string-token>, <bad-string-token>, <url-token>, <bad-url-token>, <delim-token>, <number-token>, <percentage-token>, <dimension-token>, <include-match-token>, <dash-match-token>, <prefix-match-token>, <suffix-match-token>, <substring-match-token>, <column-token>, <whitespace-token>, <CDO-token>, <CDC-token>, <colon-token>, <semicolon-token>, <comma-token>, <[-token>, <]-token>, <(-token>, <)-token>, <{-token>, <important-token> a <}-token>[1].

Dále preprocesor zavádí tokeny <variable-token>, <mixin-token>, <class-token>, <require-token> , <excludeFrom-token>, <extends-token> a pro práci s různými barevnými paletami <color-token>.

Tokeny <ident-token>, <function-token>, <at-keyword-token>, <hash-token>, <string-token>, <url-token>, <number-token>, <percentage-token>, <dimension-token>, <variable-token>, <mixin-token> a <class-token> obsahují navíc vnitřní hodnotu, která se využívá v pozdější fázi manipulace. Například <percentage-token> může obsahovat číslo 90, reprezentuje tak jednotku míry devadesát procent.

K popsání mechanizmu tokenizace je nejprve nutné definovat slovní pojmy, se kterými algoritmus tokenizace pracuje, dále následuje popis samotných tokenů a závěr podkapitoly věnované tokenizaci patří algoritmu samotnému.

3.2.2.1 Definice pojmů

Následující sekce definuje pojmy pro popis algoritmu tokenizace dle [1] a základní funkcionality regulárních výrazů.

Pojmy algoritmizace

Algoritmus přeměny vstupního řetězce na proud tokenů pracuje s následujícími slovními definicemi.

- **Kódové písmeno**

  Unicode je znak v rozmezí 0 až $10FFFF_{16}$. Unicode je standard kódování pro zobrazování textových znaků.

- **Konzumace kódového písmena**

  Značí akci, při které se pohltí **aktuální kódové písmeno** za účelem vytvoření tokenu.

- **Následující kódové písmeno na vstupu**

  Představuje následující kódové písmeno ve vstupu, které ještě nebylo zkonzumováno.

- **Aktuální kódové písmeno na vstupu**

  Určuje poslední zkonzumované kódové písmeno.

- **Rekonzumování aktuálního kódové písmene na vstupu**

  Značí akci, při které se posune ukazatel na **aktuální kódové písmeno** o kódový znak dozadu tak, že při konzumaci bude znovu zkonzumován.

- **EOF**

  Představuje konceptuální kódové písmeno reprezentující konec vstupního souboru.

- **Číslice**

  Patří sem všechna kódové písmena v rozsahu 0-9 (U+0030 DIGIT ZERO až U+0039 DIGIT NINE (9))

- **Hexadecimální číslice**

  Kódové písmeno v rozsahu 0-9 či a-f bez rozdílu na velikosti písmen.

- **Písmeno**

  Do této skupiny spadají velká písmena v rozmezí U+0041 A až U+005A (Z), nebo malá písmena v rozmezí U+0061 (a) až U+007A (z).

- **Non-ASCII kódové písmeno**

  Patří sem kódová písmena jejichž hodnota je větší než U+0080.

- **Name-start kódové písmeno**

  Představují je písmena, Non-ASCII kódové písmena, nebo U+005F PODTRŽÍTKO (_).

- **Name kódové písmeno**

  V této skupině lze najít name-start kódové písmeno, číslici, nebo U+002D MÍNUS (-).

- **Non-printable kódové písmeno**

  Představuje netisknutelné kódové písmeno v rozsahu U+0000 (NULL) až U+0008 (BACKSPACE), nebo U+000B (ŘÁDKOVÁ TABULATURA), nebo kódové písmeno U+000E (SHIFT OUT) a U+001F (INFORMAČNÍ ODDELOVAČ), či U+007F (DELETE).

- **Nová řádka**

  Reprezentuje zalomení řádku.

- **Prázdný znak**

  Nová řádka, tabulatura či mezera.

- **Náhradní kódová písmena**

  Jakékoliv kódové písmeno spadající do rozsahu U+D800 až U+DFFF včetně.

- **Maximální kódové písmeno**

  Představuje největší kódové písmeno definované Unicodem - U+10FFFF.

- **Identifikátor**

  reprezentuje především <ident-token> ale tvoří i vnitřní hodnotu tokenů <function-token>, <at-keyword-token> a <hash-token>, vystupuje též jako jednotka u tokenu reprezentující určitou míru - <dimension-token>

Regulární výrazy

Regulární výrazy jsou mechanismem pro popis množin řetězců a tvoří regulární jazyk. Např. následující regulární výraz slouží pro získání hexadecimálních čísel.

Ukázka kódu 3.1: Regulární výraz reprezentující hexadecimální čísla

```
[0-9a-fA-F]*
```

Řetězec v závorkách určuje rozsah povolených podmnožin a hvězdička počet opakování. Existují tři možnosti počtu opakování. Znak "*" definuje libovolněkrát, znak "+" znamená více než jednou a znak "?" říká nula nebo jedenkrát. Zajímavým konstruktem syntaxe je **negative lookahead**, který určuje podmnožinu, jež řetězec nesmí splňovat[25]. Např. pro získání podmnožiny modelů traktorů, které nepatří do zvolené skupiny lze použít:

Ukázka kódu 3.2: Příklad **negative lookahead**

```
traktor (?!zetor)
```

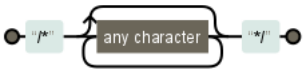
Preprocesor využívá regulárních výrazů hlavně z důvodu výpočetní rychlosti, neboť jsou podporovány nativně přímo v jazyce PHP.

3.2.2.2 Významy tokenů a jejich diagramy

Následující sekce popisuje jak tokeny, tak také jejich pomocné struktury[1]. Samotné tokeny, které jsou výsledkem vlastní tokenizace, končí postfixem -token. Pro lepší představu jsou tokeny vizualizovány jednoduchými diagramy vycházejícími z jejich regulárních výrazů[18]. Uvedené diagramy mají pouze ilustrativní charakter a přesný algoritmus získávání tokenů popisuje sekce 3.2.2.3. Pro zamezení nepřesností v důsledku překladu zůstávají názvy tokenů v anglickém jazyce.

- **comment-token (diagram 3.1)**

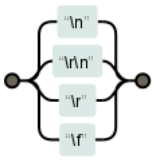
  Komentář je pomocná struktura, zpravidla používaná k popis kódu.



Obrázek 3.1: Diagram tokenu komentáře

- **newline (obr. 3.2)**

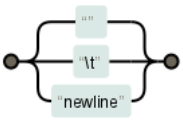
  Nová řádka reprezentovaná v různých operačních systémech.



Obrázek 3.2: Diagram nové řádky

- **whitespace (obr. 3.3)**

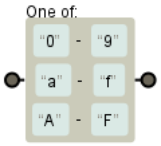
  Značí prázdný znak.



Obrázek 3.3: Diagram prázdného znaku

- **hex digit (obr. 3.4)**

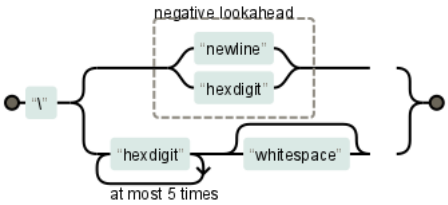
  Reprezentuje hexadecimální číslo.



Obrázek 3.4: Diagram hexadecimální číslice

- **escape (obr. 3.5)**

  Escapovaná sekvence začíná zpětným lomítkem.



Obrázek 3.5: Diagram escapované sekvence

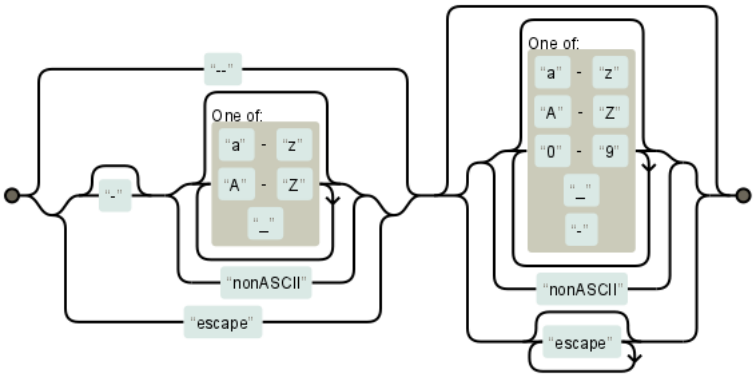
- **whitespace-token (obr. 3.6)**

  Reprezentuje libovolný počet prázdných znaků.

Obrázek 3.6: Diagram whitespace tokenu

- **ident-token (obr. 3.7)**

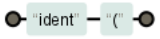
  Token, který obsahuje identifikátor např. pro jména elementů.



Obrázek 3.7: Diagram tokenu identifikátoru

- **function-token (obr. 3.8)**

  Tento token představuje entitu funkce v jazyce CSS, nejedná se však o plnohodnotné matematické funkce(tedy o funkce, které vracející výstup dle zadaných parametrů), ale spíše reprezentují konstrukce jednoduchých objektů jako např. `rgb(12, 58, 67)`.



Obrázek 3.8: Diagram tokenu funkce

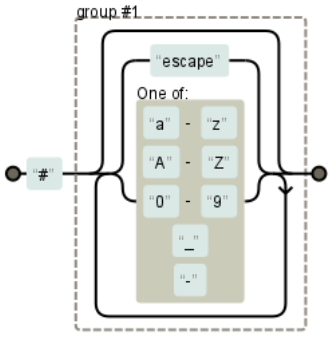
- **at-keyword-token (obr. 3.9)**

  Reprezentuje identifikátor pro pravidla začínající zavináčem.

Obrázek 3.9: Diagram at-keyword tokenu

- **hash-token (obr. 3.10)**

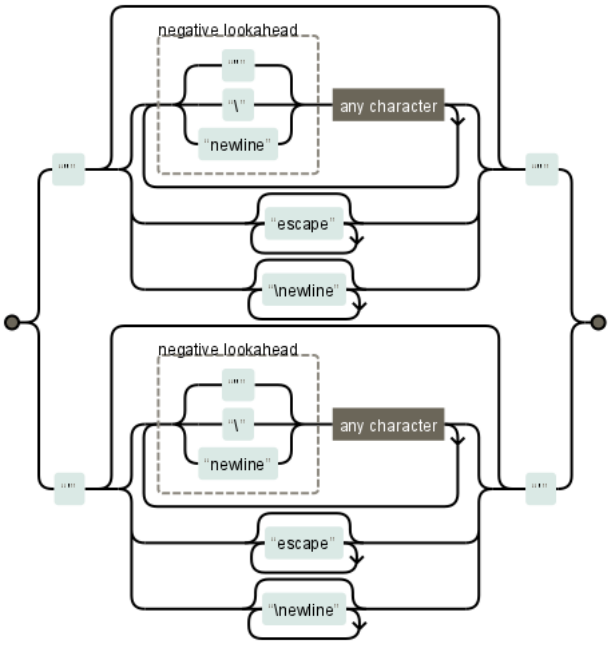
  Značí hashovací řetězec neboli sekvence začínající znakem "#", obsahující zpravidla hexadecimální zápis barev.



Obrázek 3.10: Diagram hash tokenu

- **string-token (obr. 3.11)**

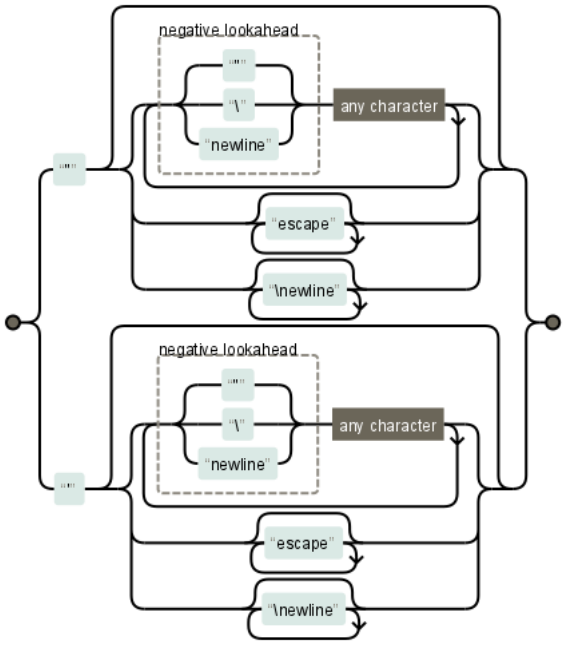
  Řetězec, který je uzavřený v uvozovkách nebo apostrofech.



Obrázek 3.11: Diagram reprezentace tokenu řetězce

- **badstring-token (obr. 3.12)**

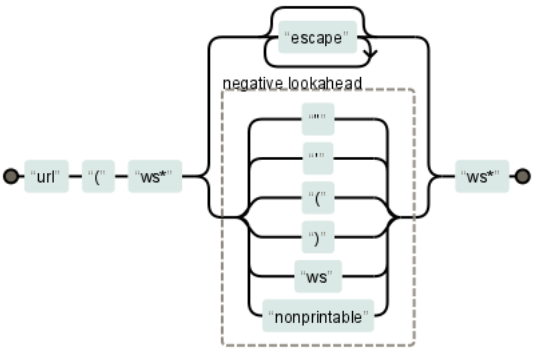
  Značí neúplný, neuzavřený řetězec.



Obrázek 3.12: Diagram tokenu neúplně zadaného řetězce

- **url-token (obr. 3.13)**

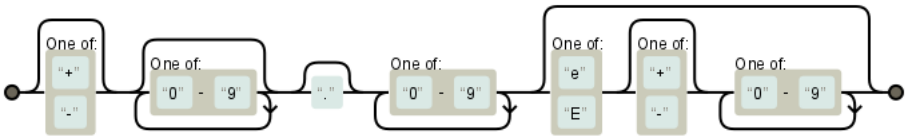
  Token, který určuje adresu k určitému souboru.



Obrázek 3.13: Diagram URL tokenu

- **number-token (obr. 3.14)**

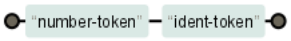
  Token, který reprezentuje libovolné číslo.



Obrázek 3.14: Diagram tokenu reprezentující číslo

- **dimension-token (obr. 3.15)**

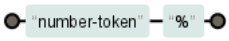
  Token znázorňující číslo s jednotkou, např. 50px.



Obrázek 3.15: Diagram tokenu, představující číslo se svojí jednotkou

- **percentage-token (obr. 3.16)**

  Čísla reprezentovaná ve formě procenta.



Obrázek 3.16: Diagram tokenu čísla v procentech

- **include-match-token**

  Token sloužící pro hledání shod řetězců je reprezentován sekvencí znaků "~=". Token jazyku CSS signalizuje, aby vybral všechny elementy HTML dokumentu, jejichž atribut obsahuje určený řetězec. Například "[title~=flower]" vybere všechny elementy "a", jejichž atribut "href" obsahuje slovo "flower".

- **dash-match-token**

  Tento token sloužící k hledání shod řetězců je reprezentován sekvencí znaků "|=". Token jazyku CSS signalizuje, aby vybral všechny HTML dokumentu, jejichž atribut začíná určeným řetězcem. Například "[lang|=en]" vybere všechny elementy, jejichž atribut "href" začíná na "en".

- **prefix-match-token**

  Další token sloužící k hledání shod řetězců je reprezentován sekvencí znaků "^=". Token jazyku CSS signalizuje, aby vybral všechny HTML dokumentu, jejichž atribut začíná určeným řetězcem. Například "a[href^="https"]" vybere všechny elementy "a", jejichž atribut "href" začíná na "https".

- **suffix-match-token**

  Tento token sloužící k hledání shod řetězců je reprezentován sekvencí znaků "$=". Token jazyku CSS signalizuje, aby vybral všechny HTML dokumentu, jejichž atribut začíná určeným řetězcem. Například "a[href$=".pdf"]" vybere všechny elementy "a", jejichž atribut "href" končí na ".pdf".

- **substring-match-token**

  Poslední token sloužící k hledání shod řetězců je reprezentován sekvencí znaků "*=". Token jazyku CSS signalizuje, aby vybral všechny HTML dokumentu, jejichž atribut obsahuje určený řetězec. Například "a[href*="school"]" vybere všechny elementy "a", jejichž atribut "href" obsahuje "school".

- **CDO-token**

  Značí sekvenci <!--.

- **CDC-token**

  Značí sekvenci -->.

- **EOF - End Of File**

  Konceptuální token, který značí konec vstupního řetězce.

- **important-token**

  Token, který signalizuje jazyku CSS zvyšenou důležitost deklarace, je reprezentován sekvencí "!important".

Pro své vnitřní fungování preprocesor zavádí následující tokeny.

- **variable-token (obr. 3.17)**

  Token, který představuje uživatelskou proměnnou.

Obrázek 3.17: Diagram tokenu uživatelské proměnné

- **mixin-token (obr. 3.18)**

  Tento token značí začátek definice uživatelské funkce ve formě mixinu.

Obrázek 3.18: Diagram tokenu definující mixin

- **extends-token**

  Token signalizující rozšíření třídy, je reprezentován sekvencí extends.

- **excludeFrom-token**

  Tento token označuje seznam pro mechanizmus vyjímání vlastností, je reprezentován sekvencí excludeFrom.

- **class-token (obr. 3.19)**

  Token, který reprezentuje definici třídy.

Obrázek 3.19: Diagram tokenu třídy

3.2.2.3 Algoritmus tokenizace

Algoritmus tokenizace převádí vstupní sekvenci kódových písmen na sekvenci tokenů. Algoritmus vychází z aktuální verze pracovního dokumentu k Syntax Module Level 3[1] jazyka CSS a rozšiřuje o získávání nových tokenů, potřebných k dosažení cílů preprocesoru z kapitoly 2.4.

Konzumace tokenu

Celý proces postupně čte textový vstup a konzumuje jednotlivá **vstupní kódová písmena**, porovnává je s následujícím seznamem a v případě shody zavolá tokenizační podoperace, které vracejí příslušné tokeny.

- **Začátek komentáře** - algoritmus provede tokenizační podoperaci **Konzumace komentáře** popsanou níže
- **Prázdný znak** - algoritmus zkonzumuje co možná nejvíce prázdných znaků a vrátí <whitespace-token>
- **Vstupní kódová písmena tvoří klíčové slovo "mixin"** - algoritmus vrátí <mixin-token> s vnitřní hodnotou nastavenou na jméno
- **Vstupní kódová písmena tvoří klíčové slovo "extends"** - algoritmus vrátí <extends-token>
- **Vstupní kódová písmena tvoří klíčové slovo "excludeFrom"** - algoritmus vrátí <extends-token>
- **U+002E TEČKA (.)** - následuje-li identifikátor, algoritmus vrátí <class-token> s vnitřní hodnotou nastavenou na hodnotu identifikátoru

- **U+0022 UVOZOVKA (")** - algoritmus provede tokenizační podoperaci **Konzumace řetězce**
- **U+0023 KŘÍŽEK (#)** - algoritmus vytvoří <hash-token> a do jeho hodnoty se uloží a zkonzumovaná hodnota
- **U+0024 ZNAK DOLARU ($)** - pokud následující sekvence znaků tvoří identifikátor algoritmus vrátí <variable-token> se jménem získaným jako hodnota identifikátoru; pokud po znaku dolaru následuje znak U+003D ROVNÁ SE (=) algoritmus vytvoří <suffix-match-token>, jinak vrátí <delim-token> s hodnotou nastavenou na **aktuální kódové písmeno**
- **U+0021 ZNAK VYKŘIČNÍKU (!)** - pokud následující sekvence znaků tvoří posloupnost important, vrátí algoritmus <important-token>; jinak algoritmus vrátí <delim-token> s hodnotou nastavenou na **aktuální kódové písmeno**
- **U+0027 APOSTROF (')** - provede tokenizační podoperaci **Konzumace řetězce**
- **U+0028 LEVÁ ZÁVORKA (()** - algoritmus vrátí <(-token>
- **U+0029 PRAVÁ ZÁVORKA ())** - algoritmus vrátí <)-token>
- **U+002A HVĚZDIČKA (*)** - pokud následuje znak U+003D ROVNÁ SE (=), vytvoří algoritmus <suffix-match-token>, jinak vrátí <delim-token> s hodnotou nastavenou na **aktuální kódové písmeno**
- **U+002B PLUS (+)** - pokud vstupní řetězec začíná číslem, algoritmus **rekonzumuje aktuální kódové písmeno** a provede podoperace **konzumace numerických tokenů**, v opačném případě vratí <delim-token> s hodnotou nastavenou na **aktuální kódové písmeno**
- **U+002C ČÁRKA (,)** - vrátí <comma-token>
- **U+002D MÍNUS (-)** - pokud vstupní řetězec začíná číslem, algoritmus **zkonzumuje numerický token** a vrátí ho; v případě, že následují dva znaky jsou U+002D MÍNUS U+003E ZNAMÉNKO VÍCE vrátí <CDC-token>; jinak zkontroluje zda nezačíná jako identifikátor a případně **zkonzumuje identifikátor** a vrátí ho; v jiných případech vratí <delim-token> s hodnotou nastavenou na **aktuální kódové písmeno**
- **U+002E TEČKA (.)** - pokud vstupní řetězec začíná číslem, algoritmus **zkonzumuje numerický token** a vrátí ho, v jiných

případech vratí <delim-token> s hodnotou nastavenou na **aktuální kódové písmeno**

- **U+003A DVOJTEČKA (:)** - algoritmus vrátí <colon-token>
- **U+003B STŘEDNÍK (;)** - algoritmus vrátí <semicolon-token>
- **U+003C ZNAMÉNKO MÉNĚ (<)** - v případě, že následují tři znaky jsou U+0021 VYKŘIČNÍK U+002D MÍNUS U+003E a MÍNUS U+003E, algoritmus vrátí algoritmus <CDO-token>; jinak vrátí <delim-token> s hodnotou nastavenou na **aktuální kódové písmeno**
- **U+0040 ZAVINÁČ (@)** - tvoří-li následující sekvence znaků posloupnost require, vrátí algoritmus <require-token>; následuje-li po zavináči identifikátor, algoritmus **zkonzumuje identifikátor** a vytvoří <at-keyword-token> s vnitřní hodnotou identifikátoru; v jiných případech vrátí <delim-token> s hodnotou nastavenou na **aktuální kódové písmeno**
- **U+005B LEVÁ HRANATÁ ZÁVORKA ([)** - algoritmus vrátí <[-token>
- **U+005C OBRÁCENÉ LOMÍTKO (\)** - začíná-li vstupní řetězec jako identifikátor, algoritmus **rekonzumuje aktuální kódové písmeno** a **zkonzumuje identifikátor**, jinak se jedná o parsovací chybu
- **U+005D PRAVÁ HRANATÁ ZÁVORKA (])** - vrátí <]-token>
- **U+005E ZNAK PŘÍZVUKU (^)** - pokud následuje znak U+003D ROVNÁ SE (=), algoritmus vytvoří <prefix-match-token>, jinak vrátí <delim-token> s hodnotou nastavenou na **aktuální kódové písmeno**
- **U+007B LEVÁ SLOŽENÁ ZÁVORKA ({)** - algoritmus vrátí <{-token>
- **U+007D PRAVÁ SLOŽENÁ ZÁVORKA (})** - algoritmus vrátí <}-token>
- **číslice** - algoritmus **rekonzumuje aktuální kódové písmeno** a **zkonzumuje numerický token**
- **name-start kódové písmeno** - algoritmus **rekonzumuje aktuální kódové písmeno** a provede podoperaci **zkonzumování identifikátoru**

- **U+007C VERTIKÁLNÍ ČÁRA (|)** - pokud následuje znak U+003D ROVNÁ SE (=), algoritmus vytvoří <dash-match-token>; jinak vrátí <delim-token> s hodnotou nastavenou na **aktuální kódové písmeno**
- **U+007E VLNOVKA (~)** - pokud následuje znak U+003D ROVNÁ SE (=), algoritmus vytvoří <include-match-token>, jinak vrátí <delim-token> s hodnotou nastavenou na **aktuální kódové písmeno**
- **EOF** - algoritmus vrátí <EOF-token>;
- **COKOLIV JINÉHO** - algoritmus vrátí <delim-token> s hodnotou nastavenou na **aktuální kódové písmeno**

Konzumace komentářů

Pokud jsou dvě následující kódová písmena lomítko "/" (U+002F) a hvězdička "*" (U+002A), algoritmus konzumuje **aktuální kódové písmeno** do dalšího nálezu hvězdičky "*" (U+002A) následované lomítkem "/" (U+002). Pokud v průběhu konzumace algoritmus narazí na kódové písmeno EOF, reprezentující konec souboru, jedná se parsovací chybu.

Konzumace numerických tokenů

Algoritmus nejprve **zkonzumuje číslo**, pokud následuje identifikátor, **zkonzumuje identifikátor** a vrátí <dimension-token> hodnotou čísla a jednotkou míry nastavenou na hodnotu identifikátoru; pokud následuje po zkonzumovaném čísle znak procenta, vrátí algoritmus <percentage-token>; v jiných případech se vrátí <number-token>.

3.2.2.4 Konzumace čísla

Jako výchozí hodnoty čísla nastaví algoritmus celočíselný typ **integer** a číselnou reprezentace **repr** na prázdný řetězec. Algoritmus postupně **konzumuje aktuální kódové písmeno**, pokud se narazí na znak tečky "." (U+002E) nebo znak "e" (U+0065), převede číslo na typ **float** a zkonzumuje zbytek **kódových písmen** čísla.

Konzumace identifikátor

Algoritmus následující podoperace vrací jeden z tokenů <ident-token>, <function-token> a <url-token> či <bad-url-token>. Nejprve **zkonzumuje jméno**, pokud následuje znak levé závorky "(" (U+0028) a pokud je **zkonzumované jméno** rovno "url" bez rozdílu velikosti písmen, vrátí algoritmus<url-token>; jinak vrátí <function-token> s hodnotou nastavenou na **zkonzumované jméno**; v případech bez následování závorky, vrátí

algoritmus <ident-token> s hodnotou nastavenou na **zkonzumované jméno**.

Konzumace jména

Algoritmus opakovaně konzumuje **name kódová písmena** a ukládá si je do mezivýsledku, jakmile narazí na něco jiného, **rekonzumuje aktuální kódové písmeno** a vrátí mezivýsledek.

Konzumace řetězce

Tato podoperace je volána s **kódovým písmenem pro ukončení** (uvozovka nebo apostrof). Algoritmus opakovaně konzumuje **následující kódové písmeno na vstupu**, narazí-li na **EOF** či na **kódové písmeno pro ukončení** vrátí <string-token> se zkonzumovanou hodnotou; když algoritmus narazí na novou řádku, vznikne parsovací chyba, **rekonzumuje se aktuální kódové písmeno** a vrátí se <bad-string-token> se zkonzumovanou hodnotou; pokud následuje znak obráceného lomítka "\" (U+005C), algoritmus **zkonzumuje escapovaný znak**; v jiných případech se **aktuální kódové písmeno** připojí k hodnotě <string-token>.

Konzumace escapovaného znaku

Algoritmus předpokládá předchozí **zkonzumování** znaku lomítka "\" (U+005C). Pokud jsou **následující kódová písmena** reprezentována jako **hexadecimální číslo** a ve výsledku menší než **maximální kódové písmeno**, vrátí algoritmus toto kódové písmeno ve formě escapovaného znaku; pokud je větší než **maximální kódové písmeno** a nebo escapovanou sekvenci přeruší **EOF** vrátí algoritmus neznámý znak U+FFFD.

3.2.3 Parsování

Parsování neboli syntaktická analýza shlukuje získané tokeny do seznamů významových pravidel[1]. Jednotlivá pravidla řadí do struktury stromu s kořenem reprezentujícím vstupní bod algoritmu - pravidlo Stylesheet. Pravidla mohou být do sebe různě zanořena a mohou tak tvořit podobnou hierarchii jako je třeba souborová struktura - adresáře a soubory. Několik pravidel obsahuje i část "prelude", pro účely této práce přeloženou jako "předehra"[1]. Tato vlastnost slouží k charakterizování samotné hodnoty pravidla.

Ukázka kódu 3.3: Příklad "předehry" (prelude) pravidla Simple Block

```
prelude {
        simple-declaration: 809px;
}
```

3.2.3.1 Definice pojmů fáze parsování

Algoritmus parsování operuje s následujícími pojmy dle [1].

- **Aktuální vstupní token**

  Aktuální vstupní token je token nebo hodnota komponenty, na kterou je zrovna nastaven ukazatel.

- **Následující vstupní token**

  Značí následující token nebo hodnotu komponenty po **aktuálním vstupním tokenu**. Pokud žádný token nenásleduje, algoritmus chápe následující token jako <EOF-token>.

- **<EOF-token>**

  Slouží jako konceptuální token reprezentující konec seznamu tokenů.

- **Konzumace aktuální vstupního tokenu**

  Akce, při které se přehodí **aktuální vstupní token** na **následující vstupní token**.

- **Rekonzumace aktuální vstupního tokenu**

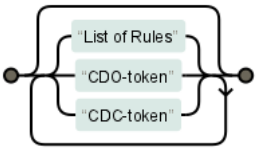
  Pokud je algoritmem vyžadována konzumace **následující vstupního tokenu**, místo přehození ukazatele a vrácení **následující vstupního tokenu**, vrací algoritmus **aktuální vstupní token**.

3.2.3.2 Významy a diagramy gramatických pravidel

Z důvodu větší shody se zdrojovým kódem zůstávají názvy pravidel opět v anglickém jazyce. Diagramy[18] mají pouze informativní charakter, nezohledňují např. prázdné znaky.

Pravidlo Stylesheet

Toto pravidlo je hlavním pravidlem zastřešující ostatní pravidla a reprezentuje celý syntaktický strom.



Obrázek 3.20: Diagram pravidla Stylesheet

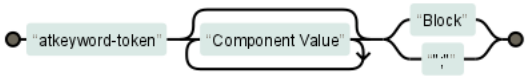
Pravidlo List of Rules

Struktura, která reprezentuje seznam pravidel.

Obrázek 3.21: Diagram pravidla List of Rules

Pravidlo At-Rule

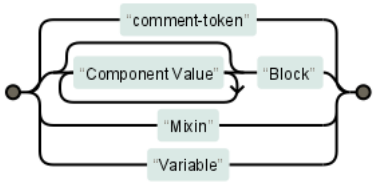
Pravidlo, které začíná zavináčem, zpravidla určuje vlastnosti přesahující rámec statického souboru kaskádových stylů. Například @media určuje, pro která zařízení jsou pravidla platná.



Obrázek 3.22: Diagram pravidla At-Rule

Pravidlo Qualified Rule

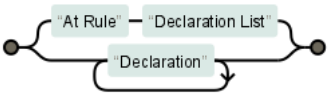
Pravidlo Qualified Rule představuje nejběžnější pravidlo kaskádových stylů. "Předehru" reprezentuje selektor jazyka CSS a následuje jednoduchý blok. Preprocesor toto místo oproti jazyku CSS rozšiřuje o uživatelské proměnné a mixiny.



Obrázek 3.23: Diagram pravidla Qualified Rule

Pravidlo Declaration List

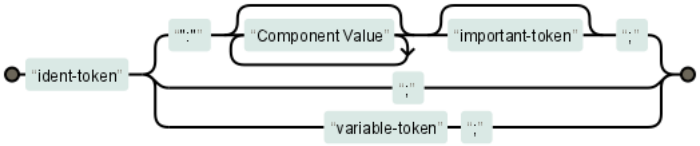
Toto pravidlo reprezentuje seznam deklarací.



Obrázek 3.24: Diagram pravidla Declaration List

Pravidlo Declaration

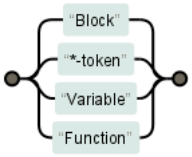
Pravidlo Declaration představuje jednotlivou deklaraci.



Obrázek 3.25: Diagram pravidla Declaration

Pravidlo Component Value

Jedná se o nejjemnější pravidlo, které se se často skládá z jediného tokenu.



Obrázek 3.26: Diagram pravidla Component Value

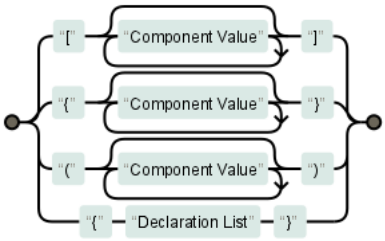
Pravidlo Function

Pravidlo, které značí předdefinované funkce jazyka CSS. Nejedná se však o funkce v matematickém smyslu slova, které by na základě vstupu vracely výstup.

Obrázek 3.27: Diagram pravidla Function

Pravidlo Simple Block

Představuje jednoduchý blok, který je asociován s jedním z tokenů <function-token>, <{-token>, <(-token>, který ho rovněž uzavírá.



Obrázek 3.28: Diagram pravidla Simple Block

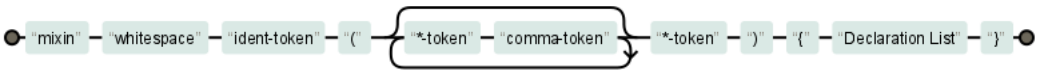
Pravidlo Variable

Toto pravidlo definuje uživatelskou proměnnou a jako jméno využívá <variable-token>.

Obrázek 3.29: Diagram pravidla Variable

Pravidlo Mixin

Pravidlo mixin obsahuje uživatelskou funkci.



Obrázek 3.30: Diagram pravidla Mixin

3.2.3.3 Algoritmus parsování

Algoritmus parsování převádí objekty nižší úrovně (tokeny, kódová písmena) na objekty vyšší úrovně (gramatická pravidla). Algoritmus rozšiřuje aktuální pracovní verzi Syntax Module Level 3[1] a příslušně volá jednotlivé parsovací podoperace.

Konzumace pravidla Stylesheet

Konzumace tohoto pravidla je vstupním bodem celé fáze parsování. Algoritmus provede podoperaci **konzumace se pravidlo List of Rules** a uloží si ho.

Konzumace pravidla List of Rules

Na začátku se vytvoří prázdný seznam pravidel. Algoritmus opakovaně se konzumuje **následující vstupní token** a podle něj postupuje:

- <whitespace-token> - neprovede se žádná akce
- <EOF-token> - vrátí se seznam pravidel
- <CDO-token>
  <CDC-token> - **rekonzumuje se aktuální vstupní token**, provede se **konzumace pravidla Qualified Rule** a pokud se něco vrátí, uloží se do seznamu pravidel
- <at-keyword-token> - **rekonzumuje se aktuální vstupní token**, provede se **konzumace pravidla At-Rule** a pokud se něco vrátí, uloží se do seznamu pravidel
- v jiném případě - **rekonzumuje se aktuální vstupní token**, **zkonzumuje se pravidlo Qualified Rule** a pokud se něco vrátí, uloží se do seznamu pravidel.

Konzumace pravidla At-Rule

Algoritmus zkonzumuje **následující vstupní token**, vytvoří nové **At-Rule pravidlo** se jménem **aktuálního vstupního písmena**, prázdnou "předehrou" a prázdnou vnitřní hodnotou. Algoritmus opakovaně konzumuje **následující vstupní token** a postupuje následovně:

- <semicolon-token>
  <EOF-token> - vrátí se **At-Rule pravidlo**
- <{-token> - **zkonzumuje se pravidlo Simple Block**, přiřadí se jako hodnota a celé pravidlo se vrátí
- v jiném případě - **rekonzumuje se aktuální vstupní token**, provede se **konzumace pravidla Component Value** a uloží se do "předehry".

Konzumace pravidla Qualified Rule

Algoritmus vytvoří nové **Qualified Rule pravidlo** s prázdnou "předehrou" a prázdnou vnitřní hodnotou. Algoritmus opakovaně konzumuje **následující vstupní token** a postupuje následovně:

- <EOF-token> - jedná se o parsovací chybu
- <comment-token> - přiřadí se komentář k vnitřní hodnotě pravidla At-Rule

- <mixin-token> - **zkonzumuje se Mixinu Definition** a připraví se využití
- <variable-token> - **zkonzumuje se Variable Definition** a připraven se k využití
- <mixin-excludeFrom> - vyjme se vlastnost z aktuální CSS třídy
- <mixin-extends> - provede se dědění od aktuální CSS třídy
- <{-token> - **zkonzumuje se pravidlo Simple Block**, přiřadí se jako hodnota a celé **Qualified Rule pravidlo** se vrátí
- v jiném případě - **rekonzumuje se aktuální vstupní token**, provede se **konzumace pravidla Component Value** a uloží se do "předehry".

Konzumace pravidla List of Declarations

Algoritmus vytvoří prázdný seznam deklarací a opakovaně konzumuje **následující vstupní token**:

- <whitespace-token>
  <semicolon-token> - žádná akce se neprovádí
- <EOF-token> - vrátí se seznam deklarací
- <at-keyword-token> - **rekonzumuje se aktuální vstupní token**, **zkonzumuje se pravidlo At-Rule** a připojí se k seznamu deklarací, pokud se jedná o pravidlo volající mixin, @mixin, **zkonzumuje se Mixin Call**
- <ident-token> - vytvoří se dočasný seznam naplněný **aktuálním vstupním token** a dokud není **následující vstupní token** něco jiného než <semicolon-token> či <EOF-token> **konzumuje se pravidlo Component value** a přikládá se do dočasného seznamu; pokud **konzumace pravidla Declaration** nad dočasným seznamem něco vrátí, připojí se k seznamu deklarací
- v jiném případě - jedná se parsovací chybu, **rekonzumuje se aktuální vstupní token**, dokud není **následující vstupní token** něco jiného než <semicolon-token> či <EOF-token> **konzumuje se pravidlo Component value** a získaná hodnota se zahazuje.

Konzumace pravidla Declaration

Algoritmus zkonzumuje **následující vstupní token**. Vytvoří novou deklaraci se jménem **aktuálního vstupního tokenu** a prázdnou vnitřní hodnotou, podle **následujícího vstupního tokenu** pak postupuje následovně:

- zkonzumují se všechny následující tokeny, pokud se jedná o <whitespace-token>
- pokud následující token není <colon-token> jedná se parsovací chybu a algoritmus nic nevrací, v opačném případě **se zkonzumuje aktuální vstupní token**
- dokud je **následující vstupní token** něco jiného než <EOF-token> **konzumuje se pravidlo Component Value** a přiřazuje se do hodnoty deklarace
- vrací se deklarace.

Konzumace pravidla Component Value

Algoritmus zkonzumuje **následující vstupní token**:

- pokud je <variable-token> a je součástí lokální proměnné vrátí se pouze <variable-token>, v opačném případě se vytvoří pravidlo Variable a vrátí se
- pokud je <{-token>, <[-token> nebo <(-token> **zkonzumuje se pravidlo Simple Block** a vrátí se
- pokud je <function-token> **zkonzumuje se pravidlo Function** a vrátí se
- v jiném případě se vrací **aktuální vstupní token**.

Konzumace pravidla Simple Block

Algoritmus zavádí definici **koncového tokenu**, což je zrcadlová varianta jednoho z tokenů <{-token>, <[-token> či <(-token>. Algoritmus vytvoří nový blok asociovaný s **aktuálním vstupním tokenem** a prázdnou vnitřní hodnotou. Opakovaně konzumuje **následující vstupní token** a postupuje:

- <EOF-token>
  koncový token - vrací se blok
- v jiném případě - **rekonzumuje se aktuální vstupní token**, **zkonzumuje se pravidlo Component Value** a získaná hodnota se přiřadí k bloku.

Konzumace pravidla Function

Algoritmus vytvoří se novou funkci se jménem hodnoty **aktuálního vstupního tokenu** a prázdnou vnitřní hodnotou. Algoritmus pak opakovaně konzumuje **následující vstupní token**:

- <EOF-token>
  <)-token> - vrátí se pravidlo Function
- v jiném případě - **rekonzumuje se aktuální vstupní token**, **zkonzumuje se pravidlo Component Value** a získaná hodnota se přiřadí k pravidlu Function.

Konzumace pravidla Mixin Definition

Algoritmus zkonzumuje definice pravidla pro uživatelské funkce, aby byla přístupná pro volání z uživatelského pseudo-kódu, dle **následujícího vstupního tokenu**:

- <EOF-token>
  <}-token> - definice **Mixinu** se uloží pro pozdější použití
- <{-token> - **zkonzumuje pravidlo Declaration List** a přiřadí do **Mixinu**.

Konzumace pravidla Mixin Call

Algoritmus zkonzumuje volání uživatelské funkce a připraví ji k zavolání ve fázi preprocesování.

Konzumace pravidla Variable Definition

Algoritmus zkonzumuje definici pravidla pro uživatelské proměnné, aby byla přístupná pro volání z uživatelského pseudo-kódu, název proměnné tvoří obsah tokenu <variable-token> a dle **následující vstupního tokenu**:

- <whitespace-token> musí následovat bezprostředně po <variable-token>, jinak se jedná o parsovací chybu
- <semicolon-token>
  <EOF-token> -uloží definici **Variable**
- v jiném případě - **rekonzumuje aktuální vstupní token**, **zkonzumuje pravidlo Component value** a získaná hodnota se přiřadí k uživatelské proměnné.

Zavolání uživatelské proměnné

Toto volání se realizuje až ve fázi preprocesní.

3.2.4 Preprocesování

V této fázi dochází k hlavnímu úkolu celého preprocesoru. Program převádí pravidla vytvořená parserem z uživatelského pseudo-CSS kódu na pravidla tisknutelná do standardního CSS kódu. Program bude tento mechanizmus realizovat tak, že projde syntaktický strom a nad jednotlivými uzly a listy provede, v případě požadavku, operace, které jsou blíže popsány níže.

3.2.4.1 Import externích souborů

Lepší orientaci ve velkých projektech zaručuje rozdělení kaskádových stylů do několika souborů. Standardní CSS sice nabízí možnost importovat externí soubor pomocí příkazu `@import`, ten ale za každý tento import provede požadavek na server se zdrojovými soubory a značně tím zpomaluje celkovou odezvu zobrazení webové stránky[17]. Preprocesor bude shlukovat externí soubory a vytvoří jeden celkový pro rychlejší odezvu načtení stránek.
Z důvodu bezpečnosti nebude tato možnost povolena u nastavení v prostředí webové produkce. Mohlo by se stát, že PHP funkce sloužící k načítání externích souborů budou zneužity a vyzradí choulostivé informace architektury a dat projektu[6].

3.2.4.2 Uživatelské proměnné

Jazyku CSS chybí uživatelské proměnné. Zavedením těchto proměnných do preprocesoru se značně usnadní editovatelnost kódu.

3.2.4.3 Uživatelské funkce

Zavedením uživatelských funkcí do jazyka CSS, je umožněno opakovatelně volat menší procedury s různými parametry a získávat tak různé výstupy pro různé případy. Aby nevznikaly kolize mezi názvy vlastností je volání mixinu řešeno pomocí tokenu <at-keyword-token> s vnitřní hodnotou `mixin`.

3.2.4.4 Dědičnost

Preprocesor by měl umožňovat vytvářet potomky na základě rodičovských vlastností.

3.2.4.5 Vnořování

Systém vnořených vlastností bývá v jazyce CSS mnohdy pro člověka nečitelný, navržený preprocesor by měl umožnit pracovat s přehlednou strukturou.

3.2.4.6 Vyjímání vlastností

Preprocesor by měl být schopen vyjímat vlastnosti, tedy extrahovat všechny vlastnosti množiny A, kromě vlastností jež náleží množině B.

3.2.4.7 Funkce zjišťující vlastnosti obrázku

Při komplexním návrhu stylu stránky je mnohdy potřeba znát přesné rozměry obrázků, se kterými se pracuje. Preprocesor by měl umožňovat získání vlastností obrázku.

3.2.4.8 Barevné tónování fotek a funkce s obrázky

Preprocesor by měl umožnit změnu tónů barev obrázku, úpravu jasu a kontrastu. Měl by zachovat originální obrázek a nově vytvořený propojit s finálním výstupem.

3.2.4.9 Barevné palety a barevné funkce

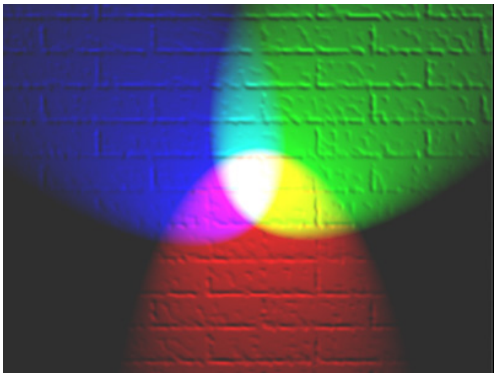
Preprocesor by měl umožnit manipulovat s barvami v různých barevných paletách, vzájemně je mezi sebou převádět, vypisovat dle zvoleného formátu, upravovat světlost a měnit saturaci a odstín. Měl by dokázat pracovat s nejpoužívanějšími barevnými modely - RGB, HSL, HSV, CMYK[36]. Oproti tomu standardní CSS, jak již bylo zmíněno, dovoluje použít pouze modely RGB a od verze CSS3 HSL[17].

Ukázka kódu 3.4: Ukázka použití HSL modelu v prostředí jazyka CSS

```
div {
        color: hsl(123, 45%, 56%);
}
```

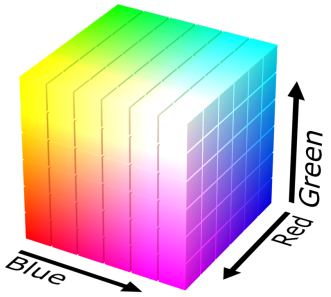
Model RGB

Nejběžněji se na webových stránkách vyskytujícím modelem pro zobrazování je RGB[17]. Jedná se o aditivní míchání barev, kdy výsledná barva odpovídá smíchání barev tří barevných kuželů různých intenzit barev červené (angl. Red), zelené (angl. Green) a modré (angl. Blue), každé v rozmezí 0 až 255. Tyto tři barvy se označují jako primární barvy[36].



Obrázek 3.31: Aditvivní míchání barev z červené, zelené, modré[27]

Barevný prostor může být chápán jako trojrozměrná krychle, kde každá její strana představuje jednu ze základních barev. Tento model se využívá hlavně při vykreslování na monitorech a TV obrazovkách, kdy jednotlivé body tvoří dohromady výslednou barvu[36].



Obrázek 3.32: Krychle reprezentují barevný prostor RGB modelu[27]

Ukázka kódu 3.5: Příklad použití RGB modelu v prostředí jazyka CSS

```
div {
        color: rgb(150 ,16, 50);
}
```

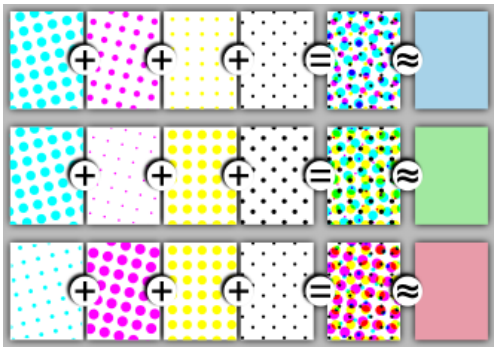
Model CMY(K)

Tento model je oproti předchozímu modelu subtraktivní, což znamená, že s každou přidanou základní barvou se z výsledné barvy ubírá světlo. Na míchání výsledné barvy se podílí azurová (angl. Cyan), purpurová (angl. Magenta), žlutá (angl. Yellow) a v oblasti tisku se přidává navíc černá (angl.

blacK nebo Key). Azurové, purpurové a žluté se nazívají barvami sekundárními. Intenzita barev se vyjadřuje od 0% do 100%[36].

Obrázek 3.33: Subtraktivní míchání barev ze žluté, azurové a purpurové[27]

Černý inkoust se přidává, aby zakryl nechtěné tóny tmavých míst tištěného obrázku a aby vylepšil ostrost a redukoval spotřebu dražších inkoustů[36].

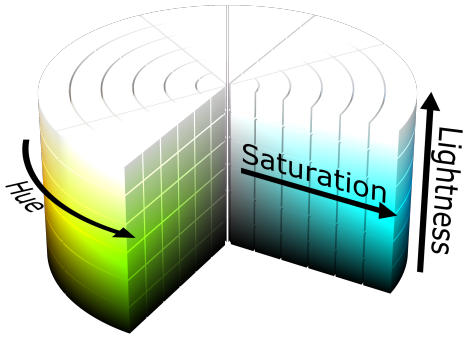


Obrázek 3.34: Využití CMYK modelu při míchání barev pro tisk[27]

Model HSL

O přesnější popisování vztahů mezi barvami z RGB prostoru se snaží modely HSL a HSV, které lépe odpovídají lidskému vnímání barev a umožňují jemnější manipulaci a přechod mezi pocitově bližšími barvami. Jedná se o příbuzné reprezentace barev, obě definují barvu jako bod uvnitř válce. Centrální osa válce jde od černé do bílé. Úhel kolem osy odpovídá odstínu barvy (angl. Hue). Vzdálenost od osy sytosti (angl. Saturation) a kolmice na

základnu představuje u HSL světlost (angl. Lightness) a u HSV hodnotu (angl. Value)[36].



Obrázek 3.35: Válec reprezentující barevné spektrum u HSL modelu[27]

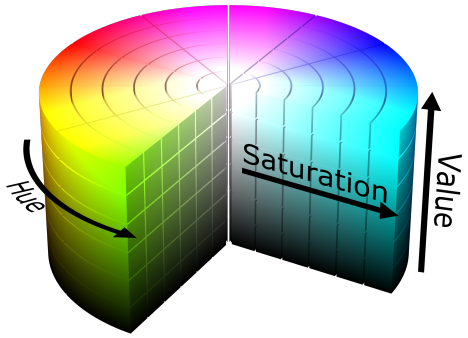
Jak bylo zmíněno, základ HSL (angl. Hue, Saturation, Lightness) modelu tvoří odstín, sytost a světlost. Odstínem se chápe převládající barevný tón, jež se udává ve stupních 0 až 360. Sytost znamená intenzitu barvy a definuje se jako procentuální zastoupení 0% až 100%. Světlost udává relativní světlost nebo tmavost barvy, též v rozmezí 0% až 100%[36]. Verze CSS3 zahrnuje HSL jako součást specifikace[17].

Ukázka kódu 3.6: Příklad použití HSL modelu v prostředí jazyka CSS

```
div {
        color: hsl(187, 16%, 55%);
}
```

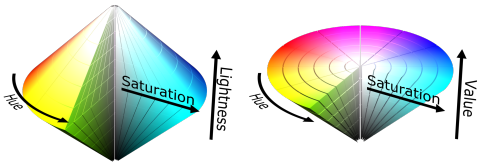
Model HSV

Obdobně jako předchozí model, HSV pracuje s odstínem, sytostí, ale na rozdíl od HSL třetí parametr představuje vlastnost “hodnota”, někdy se též pojmenovává HSB (angl. Hue, Saturation, Brightness). Odstín je udáván ve stupních, sytost a hodnota v procentech[36].



Obrázek 3.36: Válec reprezentující barevné spektrum u HSV modelu[27]

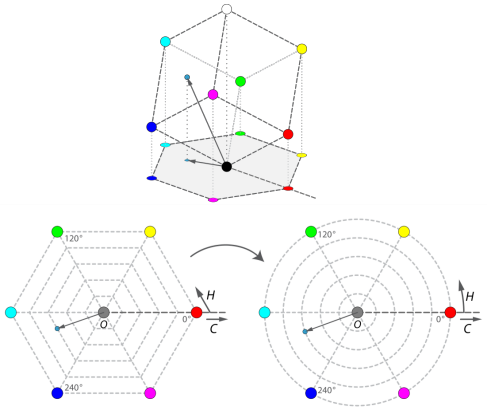
Zatímco vlastnost odstín je u obou modelů identická, pojem sytost se ve své definici dramaticky liší. HSV je možné si představit jako převrácený kužel s bodem reprezentujícím černou barvu dole a plně sytými barvami na svrchním kruhu. HSL by odpovídalo rozložení barev na spojenému dvojkuželu, kde spodní vrchol reprezentuje černou barvu, horní vrchol reprezentuje bílou a plně syté barvy se nacházejí na kruhu spojujícím kužely[36].



Obrázek 3.37: Porovnání vlastností saturace u HSL a HSV modelu[27]

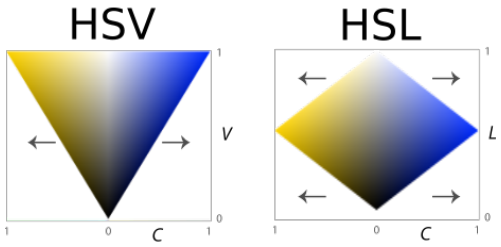
Vzájemné vztahy k RGB modelu

Na webových stránkách se nejvíce používá model RGB. Ten je však z grafického hlediska nevhodný pro návrh, neboť neumožňuje intuitivní práci s barvou. Na člověka podobně působící barvy se mnohdy nacházejí na vzdálených místech krychle reprezentující RGB prostor. Pro lepší manipulaci s barvami převádí preprocesor barvy na HSL model a v něm provádí určité barevné úpravy, jako například změnu odstínu, světlosti či saturace. Jak již je uvedeno výše v textu, HSL a HSV modely reprezentují barvu jako bod uvnitř válce, kde středovou osu tvoří tzv. neutrální osa, složená z barev šedi. Modely HSL a HSV zobrazují RGB krychli na tzv. chromatickou rovinu, která je kolmá na neutrální osu. Projekce tak získá tvar hexagonu, jehož vrcholy tvoří červená, žlutá, zelená, azurová, modrá a fialová. Jako vzdálenost od středu na chromatické rovině tak vznikne pomocná veličina tzv. chroma, označena písmenem C[36].



Obrázek 3.38: Projekce RGB krychle do chromatické roviny[27]

Body ležící na neutrální ose se někdy označují za achromatické[36].



Obrázek 3.39: Chroma rozdělení v modelech HSV a HSL[27]

Převod na HSL

Všechny algoritmy pro převod barev pracují s normalizovanými hodnotami složek RGB, tedy v rozsahu 0 až 1[36].

$$R' = R/255$$

$$G' = G/255$$

$$B' = B/255$$

Chroma neboli vzdálenost bodu reprezentující barvu od středu na chromatické rovině, se vypočítá jako rozdíl maximální a minimální barevné složky.

$$C_{max} = max(R', G', B')$$

$$C_{min} = min(R', G', B')$$

$$C = C_{max} - C_{min}$$

Odstín se definuje jako úhel vektoru, který míří od středu k bodu reprezentujícímu barvu zobrazeného na chromatickou rovinu. Úhel 0° reprezentuje červenou, 60° žlutou, 120° zelenou, 180° azurovou, 240° modrou a 300° fialovou barvu. Původně hexagonální projekce se tak lineárně namapuje na tvar kruhu (viz. obrázek 3.38). Dle dominantní barvy se pak určí segment části kruhu, ve kterém se dopočte rozdíl doplňkových barev ve vztahu k vlastnosti chroma. Pro body, které se zobrazují na počátek chromatické roviny, není odstín definován. Z praktických důvodů práce s barvou se vlastnost odstínu definuje jako 0[36].

$$H = \begin{cases} 0°(\text{nedefinován}), & C = 0 \\ 60° * \frac{G'-B'}{C} \bmod 6, & C_{max} = R' \\ 60° * \frac{B'-R'}{C} + 2, & C_{max} = G' \\ 60° * \frac{R'-G'}{C} + 4, & C_{max} = B' \end{cases}$$

Protože HSL prostor vychází z modelu dvojkuželu (viz. obr. 3.35), světlost se definuje jako průměr nejmenší a největší barevné složky, kdy neutrální osa z jednoho vrcholu přechází postupně z černé a z druhého vrcholu z bílé. Potkávají se na chromatické rovině na které zároveň leží všechny primární a sekundární barvy.

$$L = (C_{max} + C_{min})/2$$

Sytost neboli saturace přestavuje intenzitu barvy. Sytější barvy se jeví jako živější, méně syté pak jako tlumené. Tento pocitový důsledek způsobují příměsi šedé tvořící desaturované barvy. Saturace se u modelů HSL a HSV definuje jako aktuální chroma barvy v poměru k maximální možné hodnotě chroma barevného modelu[36].

$$S = \begin{cases} 0, & L = 1 \\ \frac{C}{1-|2L-1|}, & \text{jinak} \end{cases}$$

Převody mezi modely jsou pak funkcí převáděné barvy na převedenou barvu.

$$RGB2HSL(rgb(R,G,B)) = hsl(H,S,L))$$

$$HSL2RGB(hsl(H,S,L)) = rgb(R,G,B))$$

Převod na HSV

Model opět pracuje s normalizovanými hodnotami složek RGB modelu.

$$R' = R/255$$

$$G' = G/255$$

$$B' = B/255$$

Hodnota odstínu barvy (angl. Hue) a chroma se získává stejně jako u HSL modelu[36].

$$C_{max} = max(R', G', B')$$

$$C_{min} = min(R', G', B')$$

$$C = C_{max} - C_{min}$$

$$H = \begin{cases} 0°(\text{nedefinován}), & C = 0 \\ 60° * \frac{G'-B'}{C} \text{ mod} 6, & C_{max} = R' \\ 60° * \frac{B'-R'}{C} + 2, & C_{max} = G' \\ 60° * \frac{R'-G'}{C} + 4, & C_{max} = B' \end{cases}$$

HSV model reprezentuje barevný prostor jako obrácený kužel, kde hodnota vlastnosti "hodnota" odpovídá největší z barevných složek R', G', B'. Základna kuželu, která je zároveň chromatickou rovinou, obsahuje primární a sekundární barvy společně s bílou (viz. ob. 3.36).

$$V = C_{max}$$

Saturace se opět definuje jako aktuální chroma v poměru k maximální možné hodnotě chroma, která je tentokrát rovná vlastnosti "hodnota"[36].

$$S = \begin{cases} 0, & C_{max} = 0 \\ \frac{C}{V}, & \text{jinak} \end{cases}$$

Pro převodní funkce pak platí:

$$RGB2HSV(rgb(R, G, B)) = hsv(H, S, V))$$

$$HSV2RGB(hsv(H, S, V)) = rgb(R, G, B))$$

Převod na CMYK

Dále má preprocesor schopnost převádět barvy z ostatních barevných modelů do modelu CMYK, který se nejhojněji používá při tisku. Výsledný převod pro tisk má však pouze aproximativní charakter, neboť výsledné zobrazení barvy v tištěné podobě má několik úskalí, které je potřeba si uvědomit. Monitory totiž bývají mnohdy špatně barevně nakalibrovány, což ústí v neodpovídající zabarvování. Při tisku dále hraje velkou roli kvalita tiskárny a v ní vložený inkoust. V neposlední řadě tisk barvy ovlivňuje typ papíru. Jeho barva, hrubost a lesk mohou měnit celkové vyznění barvy. Při převodu model opět pracuje s normalizovanými hodnotami barevných složek z RGB prostoru.

$$R' = R/255$$

$$G' = G/255$$

$$B' = B/255$$

Nejprve se určuje hodnota černé složky jako doplněk největší barevné složky.

$$K = 1 - max(R', G', B')$$

Následně se vypočítají hodnoty azurové, fialové a žluté složky[36].

$$C = (1 - R' - K)/(1 - K)$$

$$M = (1 - G' - K)/(1 - K)$$

$$Y = (1 - B' - K)/(1 - K)$$

Pro převodní funkce tedy platí:

$$RGB2CMYK(rgb(R, G, B)) \approx cmyk(H, S, V))$$

$$CMYK2RGB(cmyk(H, S, V)) \approx rgb(R, G, B))$$

3.2.5 Výpis výstupu

Finální výstup představuje nejcennější informaci celého preprocesoru. Preprocesor musí zaručit, že výstup bude v syntaxi jazyka CSS a bude tedy možné ho využít při tvorbě webových stránek.

3.2.6 Ošetřování chyb

Zpětná vazba v případě chyb ve vstupním řetězci je důležitou součástí návrhu aplikace. Chyby, na které preprocesor při svém běhu narazí, by uživateli měly poskytnout dostatek informací vedoucích k jejich odstranění.

3.2.7 Shrnutí pravidel

Následující sekce shrnuje pravidla navrženého jazyka preprocesoru do formální gramatiky.

Jak již bylo zmíněno výše, formální gramatika je definována jako čtveřice (N, Σ, P, S), kdy N představuje konečnou množinu neterminálů, Σ konečnou množinu terminálů, P definuje konečnou množinu odvozovacích pravidel a S značí počáteční symbol[5].

Navržená gramatika definuje následující terminály Σ:

- symboly končící na "`-token`" představují příslušné tokeny
- `*-token` značí libovolný token
- ε určuje prázdný řetězec.

Neterminální symboly N tvoří pravidla, která jsou výstupem fáze parsování (podkapitola 3.2.3):

- $\langle SS \rangle$ značí pravidlo Stylesheet
- $\langle LoR \rangle$ značí pravidlo List of Rules
- $\langle AR \rangle$ značí pravidlo At-Rule
- $\langle QR \rangle$ značí pravidlo Qualified Rule
- $\langle B \rangle$ značí pravidlo Block
- $\langle DL \rangle$ značí pravidlo Declaration List
- $\langle D \rangle$ značí pravidlo Declaration
- $\langle CV \rangle$ značí pravidlo Component Value
- $\langle F \rangle$ značí pravidlo Function
- $\langle V \rangle$ značí pravidlo Variable
- $\langle M \rangle$ značí pravidlo Mixin.

Pravidla navrženého jazyka preprocesoru P shrnuje následující seznam:

$\langle SS \rangle$ ::= $\langle LoR \rangle$ | `CDO-token` | `CDC-token`
$\langle LoR \rangle$ ::= $\langle QR* \rangle$ | $\langle AR* \rangle$
$\langle AR \rangle$ ::= `atkeyword-token` $\langle CV* \rangle$ ($\langle B \rangle$ | `;`)
$\langle AR* \rangle$ ::= $\langle AR \rangle$ $\langle AR* \rangle$
$\langle QR \rangle$ ::= ($\langle C \rangle$) | ($\langle CV* \rangle$ | $\langle B \rangle$) | ($\langle M \rangle$) | ($\langle V \rangle$)
$\langle QR* \rangle$ ::= $\langle QR \rangle$ $\langle QR* \rangle$
$\langle B \rangle$ ::= (`[` $\langle CV* \rangle$ `]`) | (`{` $\langle DL \rangle$ `}`)
$\langle DL \rangle$ ::= ($\langle AR \rangle$ | $\langle D \rangle$) | ($\langle D* \rangle$)
$\langle D \rangle$ ::= (`ident-token :` $\langle CV* \rangle$ $\langle I \rangle$ `;`) | (`;`) | (`variable-token ;`)
$\langle I \rangle$ ::= `important-token` | ε
$\langle D* \rangle$ ::= $\langle D \rangle$ $\langle D* \rangle$
$\langle CV \rangle$ ::= $\langle B \rangle$ | `*-token` | $\langle V \rangle$ | $\langle F \rangle$
$\langle CV* \rangle$ ::= $\langle CV \rangle$ $\langle CV \rangle$*
$\langle F \rangle$ ::= `function-token` $\langle CV* \rangle$ `)`
$\langle V \rangle$ ::= `variable-token` $\langle CV* \rangle$ `;`
$\langle M \rangle$ ::= `mixin-token ident-token {` $\langle DL \rangle$ `}`

Kapitola 4

Implementace

Následující kapitola popisuje implementaci navrženého jazyka do formy samostatného programu.

4.1 Použité technologie

Při volbě technologií bylo nutno přihlédnout ke skutečnosti, že publikační systém Webgarden je napsán v jazyce PHP[6] a běží na webovém serveru Lighttpd[7]. Použití těchto technologií bylo tedy předpokladem pro návrh preprocesoru a jeho začlenění do systému. Jako verzovací systém byl použit open-sourceový systém GIT[8]. O správu knihoven a externích zdrojů se stará utilita Composer[9]. Testování částí programu umožňuje framework PHPUnit[11]. Profilování a odlazování chyb bylo řešeno za pomoci PHP rozšíření Xdebug[12] a knihovny Tracy[13]. Html šablony dokumentace projektu byly vytvořeny nástrojem ApiGen[32]. Jako vývojové prostředí bylo použito NetBeans[34]. Tyto technologie podrobněji rozebírají následující kapitoly.

4.1.1 Skriptovací jazyk PHP

Jazyk PHP je skriptovací jazyk k obecnému použití, avšak v dnešní době se nejhojněji používá pro programování dynamicky generovaných webových aplikací. V tomto použití se skripty provádějí na straně serveru. PHP se těší velké oblibě, pravděpodobně z důvodu jednoduchosti použití a velké podpory na straně poskytovatelů webových stránek. PHP skripty totiž mohou být nasazeny nezávisle na operačním systému, platformě a zcela zdarma[6].

4.1.2 Serverová část s využitím Lighttpd

Lighttpd se open-source webový server, který se zaměřuje na bezpečnost, rychlost, flexibilitu a standardizovanou kompatibilitu optimalizovanou pro

kriticky zatížené oblasti. V porovnání s ostatními web servery disponuje nízkou paměťovou náročností a často si umí poradit s problémy spojenými s přetěžováním serveru z důvodu mnoha požadavků[7].

4.1.3 Verzovací systémy a využití GITu

Vyvíjet software v týmu nebo jen na různých počítačích s sebou nese rizika spojená s různými verzemi souborů, vzájemným přepisování souborů nebo s nedostupností souborů aktuálních. Tyto komplikace vývoje odstiňují verzovací systémy, které se starají právě o verzování jednotlivých souborů. Nejčastěji se používají pro hlídání změn zdrojového kódu, ale použít se dají pro jakékoli soubory. Systém pro správu verzí monitoruje kdo, kdy, kde a jakým způsobem změnil které soubory. V případě nežádoucích změn je možné se vrátit k některé z předchozích verzí a dohledat osobu za změnu zodpovědnou. Verzovací systémy umožňují spolupráci mnoha programátorů na jednom softwarovém projektu a pomáhají řešit případné úpravy stejných částí kódu více programátory tzv. kolize. GIT byl vytvořen jako svobodný software s decentralizovaným přístupem. Oproti centralizovaným verzovácím systémům, které ukládají data při verzování na jediný server a tento server také potřebují při většině úkonů, GIT využívá decentralizovaný přístup, kdy je systém verzování řešen přístupen rovný-s-rovným (angl. peer-to-peer). Každý v síti si je roven a každý má svou lokální kopii celého úložiště. Programátorům tento přístup umožňuje pracovat na projektu bez sdílení společné sítě např. při ztrátě připojení k internetu[8].

4.1.4 Správa závislostí a utilita Composer

Composer je nástroj na správu závislostí knihoven v PHP. Umožňuje deklarovat knihovny, na kterých projekt závisí a Composer se pak stará o jejich instalaci a aktualizaci[9]. Tento projekt byl vytvořen, aby se mohl stát plnohodnotným balíčkem publikovaným na stránkách PHP Package Repository[10], ke kterému se dá přistupovat skrze utilitu Composer.

4.1.5 Unit testování a framework PHPUnit

Frameworkem se rozumí aplikační rámec, knihovna podpůrných programů orientovaná na určité zjednodušení práce programátora. PHPUnit[11] se stará o zjednodušení unit testování, které bude probráno v sekci 4.2.7 Testem řízený návrh.

4.1.6 Debugger a rozšíření Xdebug

Jedním z nástrojů nejvíce urychlující vývoj aplikací je debugger. Jedná se o nástroj usnadňující hledání chyb a ladění programu. Beze změny zdrojového kódu lze běh aplikace zastavovat či krokovat a sledovat jak se

mění vlastnosti objektů. I přesto, že jazyk PHP je na webovém poli nejrozšířenější, v nativní podobě mu debugger chybí. Jako debugger byl proto použit nástroj Xdebug[12], který ve formě rozšíření serverové konfigurace jazyka PHP, přidává debuggovací rozhraní.

4.1.7 Ladění a knihovna Tracy

Zatímco debugger běží jako rozšíření samotného jazyka PHP, Tracy debugger[13] volí jiný přístup a běží jako součást standardního skriptu. V případě, že jazyk PHP narazí na chybu, vypíše pouze stručné, mnohdy nic neříkající sdělení. Tracy upravuje v případě chyby výstup, doplňuje jej o cenné informace jako napovídání v případě překlepů či historii volaných funkcí, které k chybě vedly.

4.1.8 Dokumentace a nástroj ApiGen

Zdokumentováním zdrojového kódu si programátor ulehčuje budoucí práci a orientaci v kódu, stejně tak jako i jiným programátorům, kteří by na projektu mohli spolupracovat. Dokumentování kódu je podrobněji popsáno v sekci 4.2.9.4, která pojednává o PRS5. Pro vygenerování dokumentace ve formátu HTML, je použit nástroj ApiGen[32], který projde zdrojový kód a za pomoci systému hyperlinků vytvoří proklikávatelnou reprezentaci dokumentace.

4.1.9 Vývojová prostředí a využití NetBeans

Vývojové prostředí, neboli IDE (angl. Integrated Development Environment) znamená prostředí pro zjednodušení a urychlení programátorovi práce. Zpravidla se jedná o textový editor schopný pojmout celou projektovou složku se soubory různých zaměření a spravovat je v jednom programu. Obohacuje editor o našeptávání i možnost proklikávání mezi třídami, které na sebe odkazují. Umožňuje integraci externích nástrojů a knihoven jako GIT, Composer, PHPUnit, Xdebug. Jako vývojové prostředí byl zvolen svobodný open-sourcový software NetBeans[34] s širokou komunitou uživatelů a přispěvatelů. Původně byl navržen pro jazyk Java, ale v současné době podporuje mnoho dalších jazyků. Mimo jiné i jazyky pro tvorbu webových stránek jako HTML, CSS, Javascript, PHP a další.

4.2 Výchozí principy

Před popisem vlastního mechanismu fungování programu budou rozebrány principy a standardy, které se práce snaží dodržovat a využívat. Nikoliv však dogmaticky, neboť v určitých případech dodržení jednoho principu, znamená nedodržení jiného. Přehnaná snaha a honba za diamantově čistým a standardizovaným kódem představuje ve světě skutečných programů pouhý

programátorův sen a mnohdy brzdí požadavky na rychlý vývoj dnešních aplikací. Použití té či oné metodiky by vždy mělo být opodstatněné, programátor by měl vědět proč určitý diskomfort (třeba ve formě implementace složitého návrhového vzoru) začleňuje do svého pracovního postupu a proč stejný návrhový vzor na jiném místě porušuje. Programátorovi by měly tyto principy práci v celkovém vyznění usnadňovat, nikoliv ztěžovat. Názvy následujících kapitol zůstávají ve svém "rodném" anglickém jazyce, neboť se pod těmito termíny objevují i v česky psané literatuře a v jazyce českém vyznívají poněkud kostrbatě.

4.2.1 OOP - Object-Oriented Programming

Objektově orientované programování je programovací paradigma, které představuje styl programování, neboli způsob, jak instruovat počítač, aby provedl požadované operace. Objevuje se na počátku 60. let minulého století, jako alternativa k historicky staršímu procedurálnímu programování. Procedurálnímu programování přistupuje k vyřešení požadovaného úkolu jako k sérii po sobě jdoucích výpočetních kroků, které mohou k řešení používat tzv. procedury - soubory vlastních výpočetních kroků. Jako dominantní styl programování se OOP využívá od 90. let kdy jeho podporu začaly umožňovat tehdejší programovací jazyky[28]. Vychází z několika základních konceptů probraných v následujících podsekcích.

4.2.1.1 Object

Základem OOP je objekt - entita reprezentující prvek reálného světa se všemi náležitostmi, jako je datová podstata a s ní související funkčnost. OOP se tedy snaží namapovat co nejvěrněji skutečný svět do světa počítačového. Datové reprezentaci se říká atributy, funkčnosti operací pak metody. Objekty si uchovávají svůj vnitřní stav a s další částí programu komunikují prostřednictvím tzv. veřejných metod. Vnitřní neveřejné operace se označují jako privátní metody[28].

4.2.1.2 Abstrakce

Abstrakcí se rozumí schopnost zmenšit nutnou znalost, kterou musí programátor mít, aby mohl s objektem pracovat. Vnitřní fungování objektu programátora nemusí zajímat, stačí, že objekt vrací správné výsledky. Na různých úrovních komplexnosti tak může přistupovat v různých abstrakcích[28]. Hráče automatů nezajímá jak automat funguje, stačí mu, když za pár mincí zabliká.

4.2.1.3 Encapsulation

Koncept zaručující, že nelze přistupovat k vnitřním neveřejným částem objektů, které by mohly vést k nekonzistenci se říká zapouzdření (angl. encapsulation)[28]. Pokud program opravuje automat na kuřecí polévku, neměl by mít možnost přistupovat k nastavení tepelného štítu vesmírné lodi[37].

4.2.1.4 Composition

Skládání neboli kompozice zaručuje, že jeden objekt může obsahovat další objekty jako své vlastnosti[28]. Objekt reprezentující sochu Budhy by mohl například obsahovat další tři objekty Oko.

4.2.1.5 Delegation

K provedení určité operace se dá využít jiného objektu - delegování práce mimo původní objekt[28]. Paní domácí deleguje vynášení odpadků na některého ze svých nájemníků.

4.2.1.6 Inheritance

Schopnosti přejímat a rozšiřovat funkcionalitu objektů se říká dědičnost (angl. inheritance). V této souvislosti se hovoří o tzv. třídách a instancích. Třída je teoretická reprezentace objektu[28] či ideje např. kytara. Instance pak konkrétní "zhmotnění"např. kytara GIBSON Les Paul Faded T 2017 WB.

4.2.1.7 Polymorphism

Polymorfismem se rozumí schopnost skupiny objektů poskytovat na navenek stejnou funkcionalitu - neboli rozhraní. Ale konkrétní chování objektu se bude lišit dle své vnitřní implementace[28]. Každý klaun provede vlastní skeč, když bude požádán, aby pobavil děti.

4.2.2 SoC - Separation of Concerns

Česky rozdělení zodpovědností, které doporučuje rozdělit funkcionalitu programu na části, tak aby se minimálně překrývaly a aby tyto části byly zodpovědné pouze za sebe[28]. Např. webové stránky často využívají tento princip, tím že sémantiku tvoří značkovací jazyk HTML, vzhled určuje techologie CSS, ovládání uživatelského rozhraní je realizováno za pomoci jazyku Javascipt a fungování vnitřní logiky serverové části je ovládáno dalším skiptovacím jazykem.

4.2.3 KISS - Keep It Simple Stupid

Jednoduché pravidlo, které říká, že systém pracuje nejlépe pokud se udržuje v jednoduchém a přehledném stavu. Při navrhování by se mělo vyhnout přílišné komplexitě[30].

4.2.4 DRY - Don't Repeat Yourself

Pokyn "Neopakuj se!" klade důraz na minimalizování duplicit jakékoliv informace v programu. Každá informace musí mít v systému jedinou, jednoznačnou a autoritativní reprezentaci[26]. Znamená to, že pokud programátor změní například vnitřní fungování algoritmu pro řazení objektů, tuto změnu provede pouze na jednom místě a ta se pak projeví na všech místech kde se s daným algoritmem pracuje[30].

4.2.5 S.O.L.I.D

Tato mnemotechnická pomůcka v sobě skrývá zkratky názvů strategií, které kladou důraz na udržitelnost a rozšířitelnost systému[30]. Techto pět bodů shrnují následující podkapitoly.

4.2.5.1 SRP - Single Responsibility Principle

Neboli Princip jedné odpovědnosti říká, že třída má mít pouze jeden důvod ke změně[30]. Objekt by tedy měl být zodpovědný pouze za jednu kontrétní funkčnost a tato odpovědnost by měla být v třídě zapouzdřena.

4.2.5.2 OCP - Open/Closed Principle

Tento princip radí, aby entity softwarového návrhu (třídy, funkce, moduly) byly připravené k rozšíření, ale uzavřené k modifikacím[30]. Objekt je tedy možné rozšiřovat bez nutnosti měnit objekt původní.

4.2.5.3 LSP - Liskov Substitution Principle

Princip radící zachování zastupitelnosti tříd, znamená že pokud je třída Dítě podtřídou třídy Člověk, pak nahrazení třídy Člověk třídou Dítě by nemělo fungovaní systému nijak omezit. Třída Dítě může tedy volně vystupovat jako třída Člověk.

4.2.5.4 ISP - Interface Segregation Principle

Uživatel objektu by neměl být závislý na metodách, které nepoužívá[30]. Tento princip doporučuje rozkládat velká "univerzální" rozhraní na menší a pracovat pouze s metodami, které jsou k vyřešení problému potřebné.

4.2.5.5 DIP - Dependency Inversion Principle

Princip obrácení závislostí říká, že objekt by měl záviset na abstrakcích, nikoli na konkrétních vlastnostech. Konkrétnější musí záviset na abstraktnějším a ne naopak. Objekt by neměl záviset na něčem co se může často měnit. Pokud objekty závisejí na abstraktních rozhraních je možné konkrétní implementaci snadno nahradit jinou.

4.2.6 DI - Dependency Injection

Jedná se o často zaměňovaný pojem s podobně znějícím pojmem Dependency Inversion Principle, Dependency Injection je pouze mechanismen k dosažení Dependency Inversion Principle, tedy Principu obrácení závislostí. Pokud nějaký objekt pracuje s instancemi jiných objektů, předávají se mu tyto instance při vytváření původního objektu. Dodržení tohoto principu zaručuje, že objekt má přístup pouze k těm instancím objektům se kterými pracuje. Dále je zaručena jednoduchá testovatelnost částí programu, odstraňují se skryté závislosti a již před zkompilováním kódu se dá hlídat, zda-li do třídy vstupují správné typy objektů[29].

4.2.7 TDD - Test-Driven Development

Vývoj řízený testy je přístup k programování, kdy malé opakující se kroky vedou k zefektivnění celého vývoje. Tento přístup vyžaduje psaní tzv. unit testů. Unit testem se rozumí ověření správnosti fungování určitě malé části programu (např. funkce, třída, metoda)[29].

4.2.7.1 Cyklus vývoje

Cyklus vývoje TDD shrnují následující kroky dle [29].

1. **Napsaní testu**

   Napsáním testu se rozumí připravení ověření správného fungování kódu, vývojář se tím ujistí, že správně pochopil požadavky na testovanou komponentu. Začlenění psaní testů na první místo je nejpodstatnější částí vývojového cyklu.

2. **Spuštění testů a ujištění se že všechny neprojdou**

   V tuto chvíli neexistuje žádný vlastní kód, proto by žádné testy neměly projít. Pokud by snad prošly, znamenalo by to, že jsou špatně napsané a nebo že projdou vždy a jejich využití tedy postrádá smysl.

3. **Napsání vlastního kódu**

   Tento krok si klade za cíl co nejrychleji implementovat funkčnost. Nehledí přitom na čistotu a efektivitu kódu, snaží se pouze o splnění průchodnosti testy.

4. **Kód projde testy**

   Pokud kód prochází testy, tedy splňuje požadavky na něj kladené, je zaručena správná funkcionalita.

5. **Refaktorace**

   Refaktorace znamená přepisování kódu při zachování stejné funkcionality. Až tento krok dovoluje zefektivnění a přepsání kódu do čistější podoby. Odstraňují se při ní duplicity a přetváří se do elegantnější formy.

6. **Opakování**

   Předchozí kroky se s narůstajícími požadavky na program stále dokola opakují a přidávají tak funkcionalitu a umožňují programu kontrolovaně a rychle růst.

4.2.7.2 Metrika Code Coverage

Metrika Code Coverage neboli pokrytí zdrojového kódu při testování určuje, kolik procent kódu bylo testy pokryto. Vyšší pokrytí znamená lepší otestování a snižuje šanci, že zdrojový kód obsahuje nedetekované chyby[29].

4.2.7.3 Kritika a omezení TDD

Mnohdy diskutovaná bývá práce navíc, která s psaním testů souvisí. Objevují se názory, že psaní testů snižuje produktivitu a brzdí vývoj. Testy většinou píše stejná osoba, která psala i testovaný kód, což může vést k přehlížení a zjednodušování. Obecně špatně se testují určité části systému, jako například uživatelské rozhraní či oblasti, kde se pracuje s databázemi.

4.2.8 Design Patterns - návrhové vzory

Podobně jako u stavební architektury poskytují vzory návrhu znovupoužitelné řešení běžně se objevujících problémů. Návrhový vzor není hotovým řešením ve formě například knihovny, ale spíše přístupem k řešení problému. Za zásadní a výchozí knihu softwarových návrhových vzorů se považuje kniha Design Patterns: Elements of Reusable Object-Oriented Software[22], sepsaná čtveřicí autorů, známých jako Gang of Four. Návrhovým vzorům z této knihy se proto často říká GoF Designs Patters. Přestože stále vznikají nové návrhové vzory, je tato více než 20 let stará kniha stále velmi aktuální.

4.2.8.1 Využití návrhových vzorů

Preprocesor využívá několik návrhových vzorů, které budou podrobě probrány v příslušných sekcích později v této kapitole.

4.2.8.2 Kritika

Přehnaná snaha o využívání návrhových vzorů může vést ke zbytečné komplexitě. V domnění, že návrhové vzory zlepšují řešení a snaha za každou cenu použít návrhový vzor, mnohdy odvádí od problému samotného. Vždy platí, že programátor by měl vědět, proč daný mechanismus využívá, znát jeho světlé a stinné stránky. Ne vždy znamená použití nějakého vzoru výhru při řešení problému. Příkladem budiž návrhový vzor Singletor. Kromě problémů při řešení vícevláknové aplikace, také zavádí globální stav, který ztěžuje testování a může vést k nekontrolovanému chování aplikace.

4.2.9 PSR - PHP Standard Recommendation

Tato specifikace se snaží o větší interoperabilitu mezi jednotlivými komponentami, jež jsou napsané v jazyce PHP. Poskytuje společné technického rozhraní, doporučuje prověřené koncepty a optimalizuje programovací a testovací zvyky. V současné době existuje 17 doporučení[31] pro různé oblasti a zaměření aplikace, většina z nich se nyní nachází ve stavu pracovní verze a prochází vývojem standardizace. Tato práce se drží následujících čtyř doporučení.

4.2.9.1 PSR 1

Určuje základní kódovací standart. Definuje jak např. pojmenovávat programové entity. Zaručuje vysokou úroveň technické interoperability mezi sdíleným PHP kódem.

4.2.9.2 PSR 2

Předkládá návod na styl kódování, ve kterém sjednocuje styly kódování pro práci mezi různými autory. Definuje např. jak má vypadat tabulatura, kde se mají umisťovat mezery a kde nové řádky.

4.2.9.3 PSR 4

Definuje jak přistupovat k automatickému nahrávání tříd a jak mapovat tzv. jmenné prostory (balík funkcionalit) na souborovou strukturu.

4.2.9.4 PSR 5

Popisuje, jak pomocí komentování zdrojového kódu tvořit PHP dokumentaci. Pomocí tzv. anotací, což jsou klíčová slova začínající zavináčem, umožňuje standardizovanou formou zjednodušeně popsat fungování tříd, funkcí, výjimek a dalších.

Ukázka kódu 4.1: Příklad tvorby dokumentace v jazyku PHP

```
/**
 * This file is part of Preprasor CSS Preprocessor
 * Created by Filip Vorel 2016
 */

namespace Preprasor\Preprocess\Helper;

use Preprasor\PreprasorErrorException;

/**
 * Class for checking directory's availability
 *
 * @package Preprasor\Preprocess\Helper
 */
class DirectoryDiver {
 /**
  * @var string
  */
 public $pathToDirectory;

 /**
  * DirectoryDiver constructor.
  *
  * @param $pathToDirectory
  * @throws DirectoryNotFound
  */
 public function __construct($pathToDirectory) {
  if($pathToDirectory=="") {
   $pathToDirectory = "./";
  }
  if(!file_exists($pathToDirectory)) {
   throw new DirectoryNotFound("Could not find $pathToDirectory.");
  }
  $this->pathToDirectory = $pathToDirectory;
  }
  /**
   * returns its path to directory
   *
   * @return string
   */
  public function getPath() {
   return $this->pathToDirectory;
  }
}
```

Jak je vidět na příkladu, popisem k třídě a jednotlivým funkcím se kód zpřehledňuje. Anotace pomáhají vývojovému prostředí napovídat jakého typu vyžadují funkce proměnné a jaké výjimky se od nich dají očekávat.

4.3 Implementace fáze nastavení

V konfiguraci je možné nastavit, například zda bude možné využívat potenciálně nebezpečné funkce pro webovou produkci či vypisování barev ve formátu kompatibilním i ve starších prohlížečích. Starší prohlížeče totiž umí pracovat pouze s hexadecimálním tvarem RGB barevného modelu.

Ukázka kódu 4.2: Příklad jediného možného zápisu barvy podporováného ve starších prohlížečích

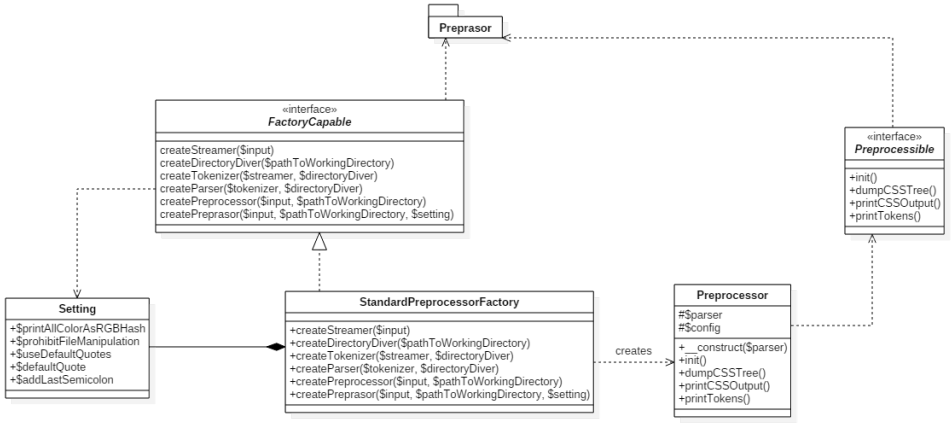
```
div {
        color: #8A2BE2;
}
```

Z uvedeného příkladu není pro nezkušené oko patrné rozdělení jednotlivých složek barev - červené (138), zelené (43) a modré (226).

Ukázka kódu 4.3: Příklad možných zápisů barev od verze CSS 3

```
div {
        color: rgb(138, 43, 226);
        background-color: hsl(9, 68, 26);
}
```

Preprocesor umožňuje nastavit, který formát zobrazování barev bude použit. Pokud není vybráno žádné nastavení, defaultně se vybere zpětně kompatibilní mód webové produkce. Tvorba preprocesoru je realizována návrhovým vzorem Abstract Factory. Abstract Factory poskytuje rozhraní pro tvorbu skupin závislých objektů bez specifikace jejich konkrétních tříd.



Obrázek 4.1: UML diagram realizace nastavení

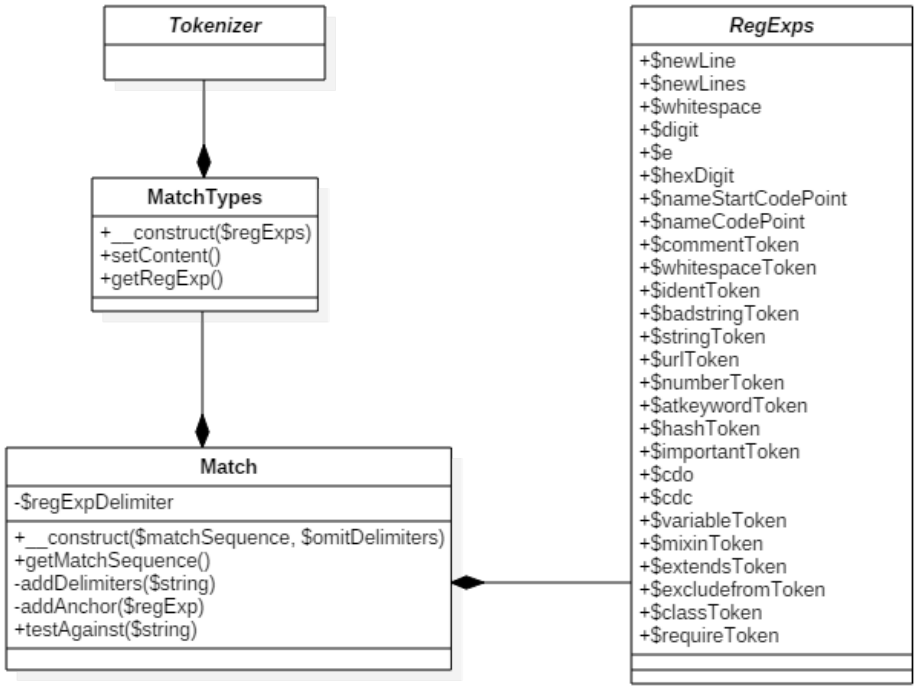
Abstract Factory vytvoří preprocesor se zvolenou konfigurací. Díky rozhraní Preprocessible může program s objektem pracovat nezávisle na jeho implementaci.

4.4 Implementace tokenizace

Jak již bylo zmíněno výše, fáze tokenizace mění znaky ze vstupního řetězce na proud tokenů. K realizaci využívá rozšíření jazyka PHP MultiByte. Zatímco některé jazyky reprezentují jeden znak jako jeden byte (např. angličtina), některé jich potřebují více, neboť byte dokáže reprezentovat pouze 256 znaků. Využití extenze MultiByte zaručuje, že prováděné operace nad řetězci budou správně přistupovat k rozsahu znaků.

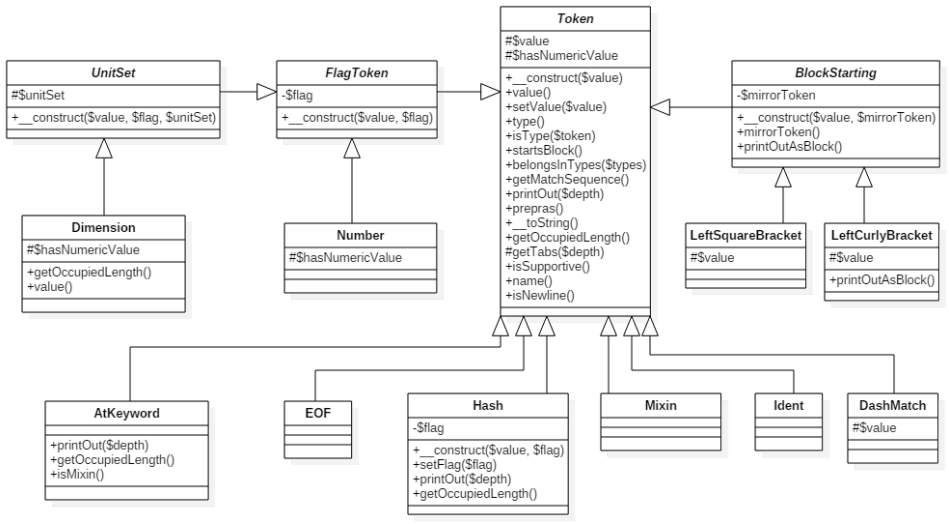
4.4.1 Cyklus získání tokenu

Na nejnižší úrovni zpracování vstupu operuje třída Streamer. Ta nejprve přijme vstupní textový řetězec, ten si uloží a čeká na povely třídy rozhraní Tokenizerable, se kterou úzce spolupracuje. Třída Streamer má za úkol především posouvat ukazatel **aktuálního kódového písmena**, dále porovnává vstup s regulárními výrazy jednotlivých tokenů. Každý jednotlivý regulární výraz je pak pro lepší manipulaci obalen do objektu Match. Tyto objekty zastřešuje datová struktura MatchTypes, která umožňuje snadnější porovnávání regulárních výrazů se vstupním řetězcem (4.2).



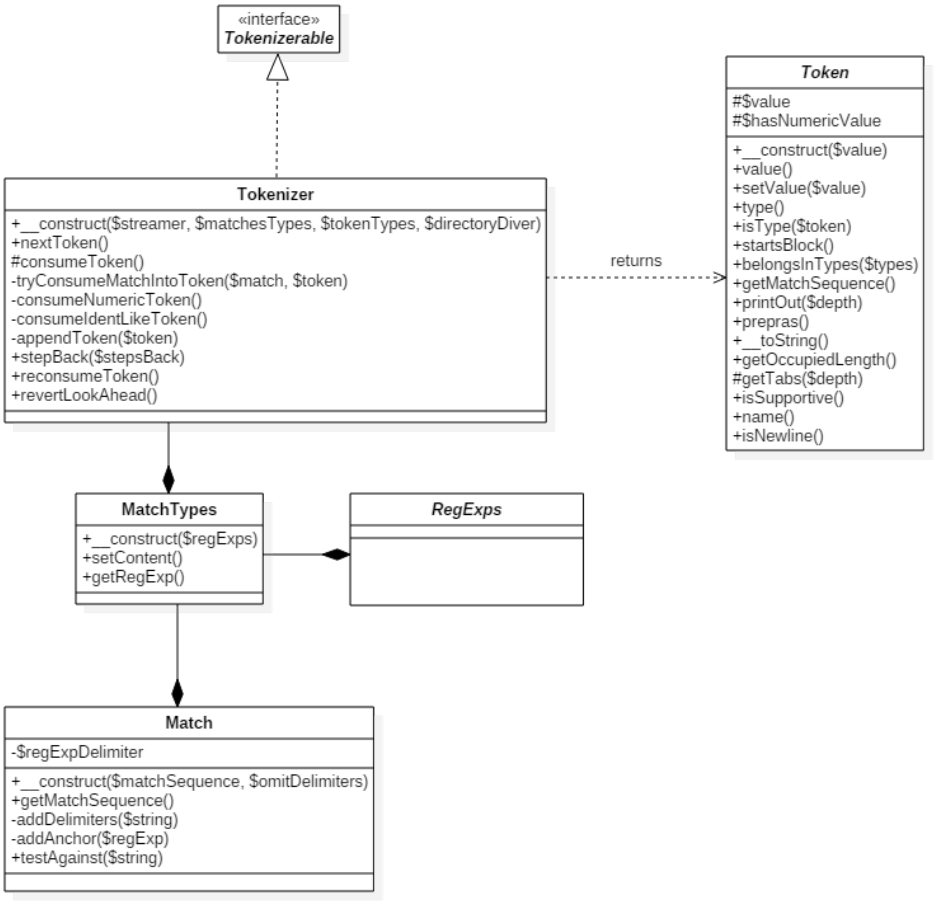
Obrázek 4.2: UML diagram umístění regulárních výrazů v tokenizéru

Hlavní funkcí třídy rozhraní Tokenizerable je pak nextToken(), která se stará právě o přiřazování tokenů na základě získaných **kódových písmen**. Funkce nextToken() zavolá proceduru **konzumace tokenu** z předchozí sekce a podle shod s regulárními výrazy na **aktuálních kódových písmenech** vytváří příslušné tokeny. Tokenů je několik druhů. Všechny dědí vlastnosti od základní třídy Token, některé jsou rozšířeny příslušnými podtřídami (4.3).



Obrázek 4.3: UML diagram několika tříd odvozených od základních tříd tokenů

Schéma závislostí tokenizéru ukazuje obrázek (4.4).

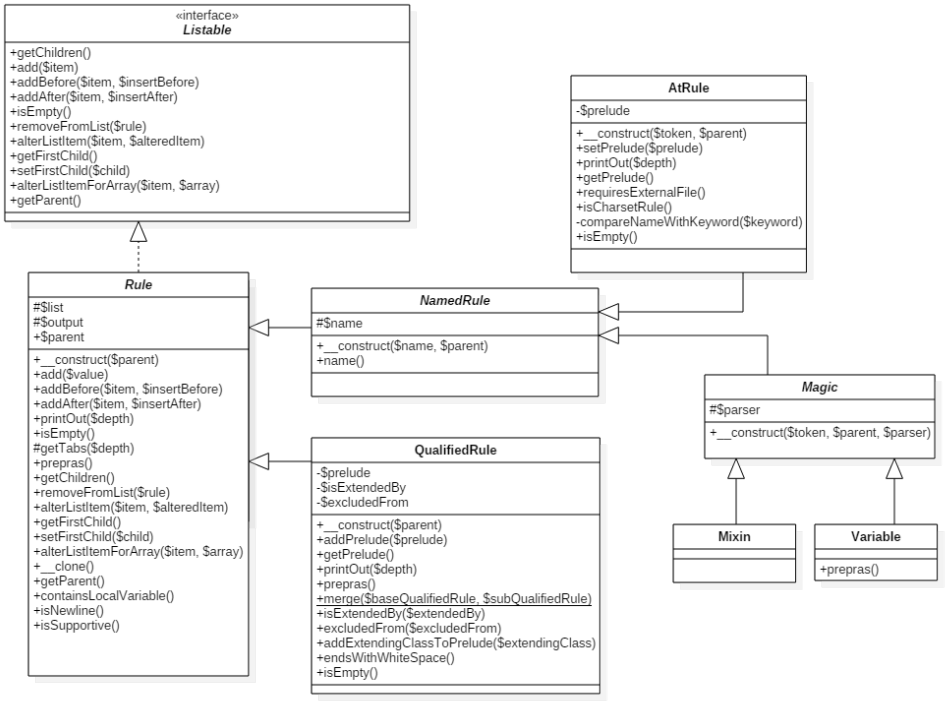


Obrázek 4.4: UML diagram schéma tokenizéru

Celý mechanismus se opakuje a příslušně posunuje **aktuální kódové písmeno**. Výstupem tokenizace je proud tokenů, určený k parsování do významových pravidel, které popisuje následující sekce.

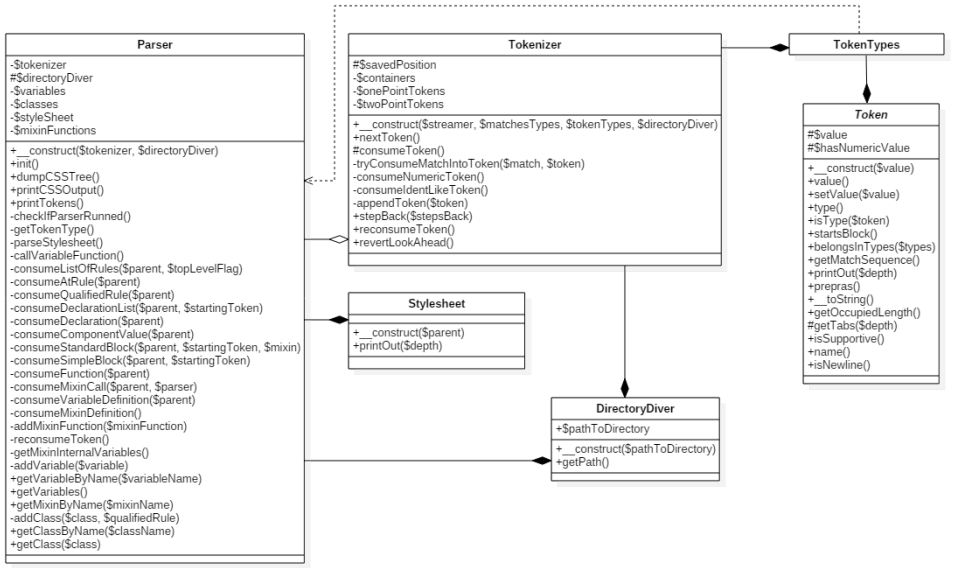
4.5 Implementace parsování

Objektů reprezentující pravidla, které při algoritmu parsování mohou vzniknout, je několik. Všechny však dědí vlastnosti ze základní abstraktní třídy Rule, a příslušně ji v případě potřeby rozšiřují (obr 4.5).



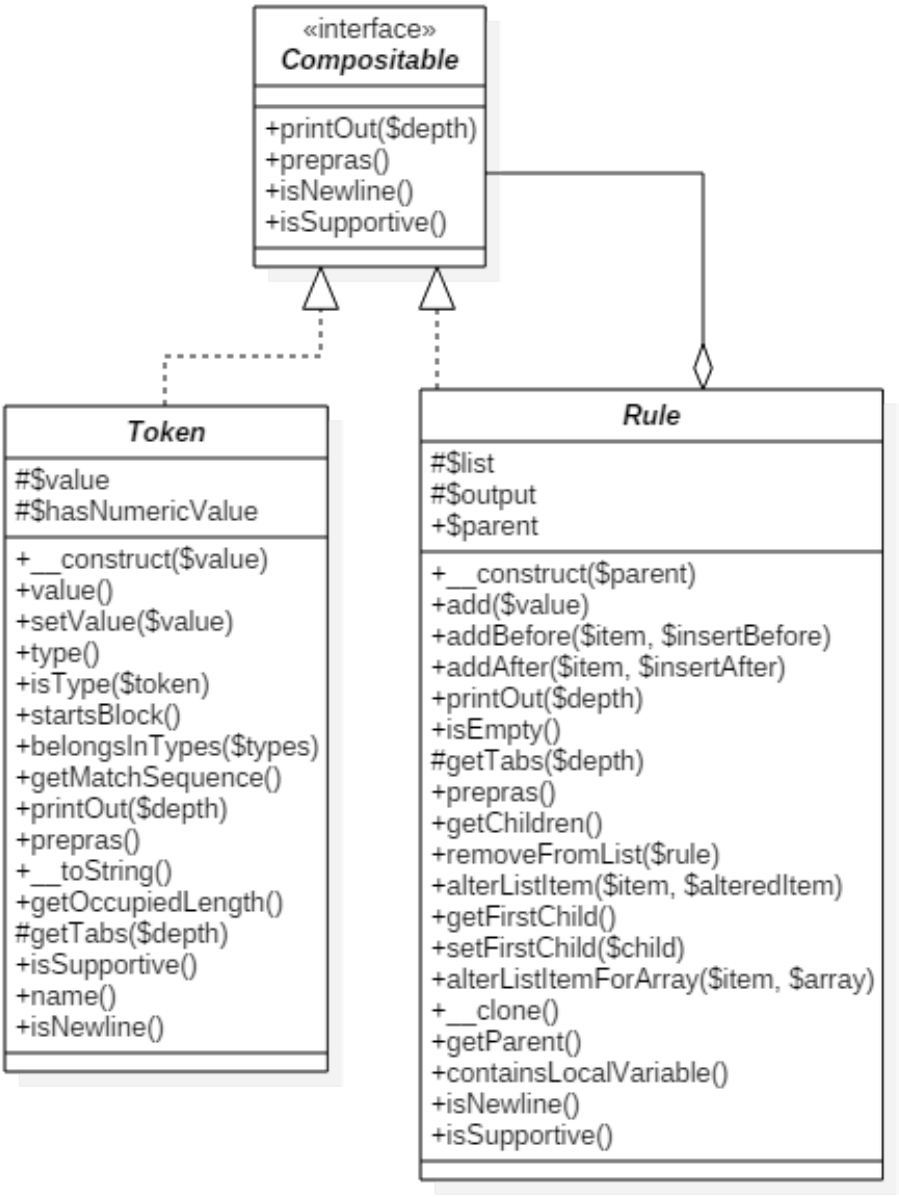
Obrázek 4.5: UML diagram několika tříd odvozených od základní třídy pravidel

Třída Parser přijímá od třídy Tokenizer postupně následující tokeny. Ty pak pomocí datové struktury TokenTypes porovnává a určuje, která pravidla dohromady tvoří. Výsledným produktem této fáze je objekt reprezentující celou strukturu vstupního souboru - objekt pravidla **Stylesheet** (obr 4.6). Jak třída Parser tak i třída Tokenizer využívají k operacím v pracovním adresáři pomocný objekt DirectoryDiver. Tento objekt hlídá, aby pracovní adresář existoval a byl dostupný.



Obrázek 4.6: UML diagram schéma parseru

Pro manipulaci do sebe zanořených **pravidel** a přidružených **tokenů** využívá preprocesor návrhový vzor **Composite**[22]. Tento návrhový vzor zabezpečuje, že se skupinou objektů může být nakládáno stejně jako s jednotlivým objektem. Účelem je vytvořit strom, který stejným způsobem pracuje jak s jednoduchými tak i se složenými objekty stromu. Prvky CSS stromu musí implementovat rozhraní Compositable. To zaručuje, že objekty stromu (**pravidla** a **tokeny**) jsou tisknutelné a je možné nad nimi provádět preprocesování (obr. 4.7). **Pravidlo** (třída Rule) může obsahovat další **pravidla** nebo **tokeny** (třída Token). Použijeme-li paralelu se souborovým systémem, **pravidla** tak představují složky a **tokeny** samotné soubory.



Obrázek 4.7: UML diagram návrhového vzoru Composite[22] v prostředí proprocesoru

Fáze parsování tedy vytvoří ze seznamu po sobě jdoucích tokenů pravidla a nad některými z nich se následně provede fáze preprocesování.

4.6 Implementace preprocesování

Jak již bylo zmíněno, v této fázi dochází k hlavnímu úkolu preprocesoru. Aplikace převádí pravidla jazyka preprocesoru na pravidla tisknutelná do standardního CSS kódu. Preprocesor byl napsán v jazyce PHP. Z toho důvodu se i syntaxe navrženého jazyka preprocesoru snaží jazyku PHP připodobnit. Program prochází syntaktický strom a volá metodu preprocesování nad jednotlivými pravidly. Následující sekce popisují implementace rozšíření oproti jazyku CSS.

4.6.1 Import externích souborů

Za vkládání externího souboru je zodpovědná třída Streamer. Ta pracuje na nejnižší úrovni přímo se vstupním řetězcem a do něj injektuje obsah externího souboru. Nad vstupním řetězcem se pak pokračuje v tokenizaci.

4.6.1.1 Příklad použití

Ukázka kódu 4.4: Vyžádání externího souboru v syntaxi jazyka preprocesoru

```
body {
        color: blue;
}
@require '../css/simpleDiv.css';
```

Výsledný výstup bude zahrnovat i obsahem importovaného souboru.

Ukázka kódu 4.5: Výstup v synaxi jazyka CSS

```
body {
        color: blue;
}
div {
        color: red;
}
```

4.6.2 Uživatelské proměnné

Proměnná se definuje znakem $ po němž následuje jméno a následně deklarace vnitřní hodnoty proměnné. Preprocesor si při procházení vstupního řetězce zapamatuje definované uživatelské proměnné a ty v případě potřeby poskytne pro vložení do syntaktického stromu.

4.6.2.1 Příklad použití

Ukázka kódu 4.6: Vstup v syntaxi jazyka preprocesoru

```
$fontFace Arial, sans-serif;
p1 {
```

```
        font: 15px $fontFace;
}
p2 {
        font: 20px $fontFace;
}
```

Na výstup se vytisknou obsahy proměnných.

Ukázka kódu 4.7: Výstup v synaxi jazyka CSS

```
p1 {
        font: 15px Arial, sans-serif;
}
p2 {
        font: 20px Arial, sans-serif;
}
```

4.6.3 Uživatelské funkce

Obdobně jako uživatelské proměnné si preprocesor při procházení vstupního řetězce zapamatuje uživatelské funkce. Jejich zavoláním se v CSS stromu vytvoří příslušné struktury v závislosti na parametrech, se kterými byly zavolány.

4.6.3.1 Příklad použití

Ukázka kódu 4.8: Vstup v syntaxi jazyka preprocesoru

```
mixin paragraph($fontface,$fontSize) {
        font: $fontface $fontSize;
        text-align: right;
}
p1 {
        @mixin paragraph(15, Arial);
}
p2 {
        @mixin paragraph(20, Helvetica);
}
```

Výstup bude obsahovat příslušné struktury dle jejich parametrů.

Ukázka kódu 4.9: Výstup v syntaxi jazyka CSS

```
p1 {
        font: 15 Arial;
        text-align: right;
}
p2 {
        font: 20 Helvetica;
        text-align: right;
}
```

4.6.4 Dědičnost

Preprocesor si třídy jazyka CSS udržuje v paměti spolu s jejich vlastnostmi. Tyto vlastnosti pak při dědění třída přejímá.

4.6.4.1 Příklad použití

Ukázka kódu 4.10: Vstup v syntaxi jazyka preprocesoru

```
.navigation {
        font-family: "Times New Roman";
        font-size: 30px;
}
.detail extends .navigation {
        color: red;
}
```

Společné vlastnosti pak spojí do jednoho bloku.

Ukázka kódu 4.11: Výstup v syntaxi jazyka CSS

```
.navigation, .detail {
        font-family: "Times New Roman";
        font-size: 30px;
}
.detail {
        color: red;
}
```

4.6.5 Vnořování

Vnořené bloky postupně probublávají až k nejvyššímu bloku, který připraví strukturu rozpoznatelnou pro jazyk CSS.

4.6.5.1 Příklad použití

Ukázka kódu 4.12: Vstup v syntaxi jazyka preprocesoru

```
div {
        span {
                margin: 0;
                padding: 0;
                list-style: none;
        }
        a {
                color: red;
                text-decoration: none;
        }
}
```

Tato nová struktura vypadá při výpisu následovně:

Ukázka kódu 4.13: Výstup v syntaxi jazyka CSS

```
div span {
        margin: 0;
        padding: 0;
        list-style: none;
}
div a {
        color: red;
        text-decoration: none;
}
```

4.6.6 Vyjímání vlastností

Preprocesor si třídy jazyka CSS udržuje v paměti spolu s jejich vlastnostmi. Tyto třidy pak slouží jako základ pro vyjímání vlastností.

4.6.6.1 Příklad použití

Ukázka kódu 4.14: Vstup v syntaxi jazyka preprocesoru

```
.paragraph {
        font-family: "Times New Roman";
        font-size: 30px;
        color: red;
        child: 4n+10;
}
.colorlessParagraph excludeFrom .paragraph {
        color;
}
```

Vyjmutí vlstností třídy A se provádí jako podědění všech vlastností třídy A, kromě uvedených vlastností v třídě B.

Ukázka kódu 4.15: Výstup v syntaxi jazyka CSS

```
.paragraph {
        font-family: "Times New Roman";
        font-size: 30px;
        color: red;
        child: 4n+10;
}
.colorlessParagraph  {
        font-family: "Times New Roman";
        font-size: 30px;
        child: 4n+10;
}
```

4.6.7 Funkce zjišťující vlastnosti obrázku

Preprocesor zjistí rozměry obrázku a ty pak reprezentuje jako uživatelskou proměnnou.

4.6.7.1 Příklad použití

Ukázka kódu 4.16: Vstup v syntaxi jazyka preprocesoru

```
$height image-height('../images/bolog.jpg');
$width image-width('../images/bolog.jpg');
.imageContainer {
        height: $height;
        width: $width;
}
```

Proměnné obsahují rozměry obrázku ve správných jednotkách a ty následně preprocesor vypíše.

Ukázka kódu 4.17: Výstup v syntaxi jazyka CSS

```
.imageContainer {
        height: 447px;
        width: 600px;
}
```

4.6.8 Barevné tónování fotek a funkce s obrázky

Preprocesor neposkytuje možnost manipulace s obrázky na komplexní úrovni, neboť požadavkem na preprocesor bylo využití standardní knihovny jazyka PHP. Osvobozuje však od nutnosti využít externí nástroj pro základní manipulace s obrázky jako jsou změna kontrastu, světlosti, rozostření, změna barevného tónu, negace a převod na stupně šedi. Pracuje tak, že vezme originální obrázek, vytvoří kopii, na kterou aplikuje požadovanou změnu tónů barev a ve struktuře syntaktického stromu vytvoří na nový obrázek odkaz. Z důvodu zápisu do souborů při operacích s obrázky je tato možnost při nastavení webové produkce zakázána.

4.6.8.1 Příklad použití

Ukázka kódu 4.18: Vstup v syntaxi jazyka preprocesoru

```
.div4 {
    image: contrast_image('../images/bolog.jpg', 10);
}
.div5 {
    image: colorize_image('../images/bolog.jpg', 0, 255, 2, 0);
}
```

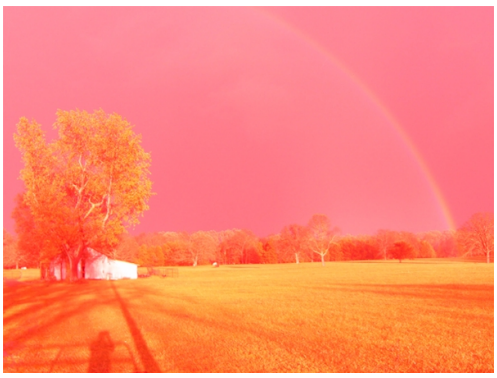
Po zavolání je nový obrázek (obr .4.9) vytvořen a správně nalinkován.

Ukázka kódu 4.19: Výstup v syntaxi jazyka CSS

```
.div4 {
        image: url('../images/bolog-contrast-10.jpg');
}
.div5 {
```

```
        image: url('../images/bolog-colorize-0-255-2-0.jpg');
}
```

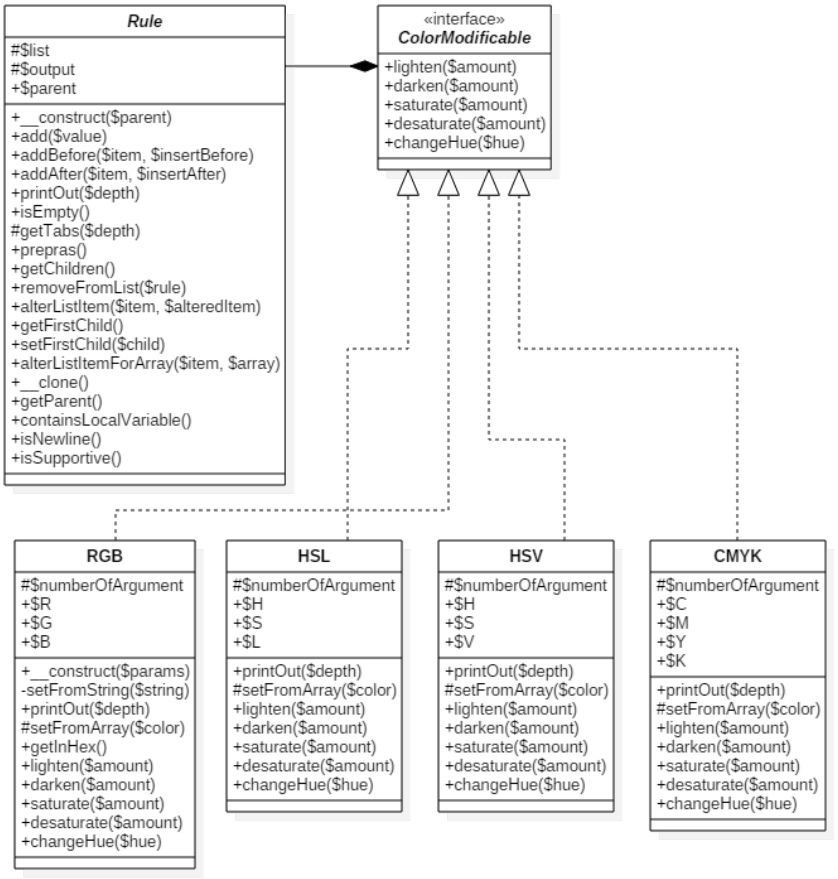
Obrázek 4.8: Originální obrázek[15]



Obrázek 4.9: Obrázek po aplikaci barevného tónování

4.6.9 Barevné palety a barevné funkce

Pro manipulaci s barvami je využit návrhový vzor **Strategy**[22]. Tento návrhový vzor zapouzdřuje určitý druh algoritmů nebo objektů, které se mají měnit, tak, aby byly pro uživatele kódu zaměnitelné. Jednotlivé barevné modely reprezentují různé strategie práce s barvou (obr. 4.10).



Obrázek 4.10: UML diagram návrhového vzoru Strategy[22] v prostředí preprocesoru

4.6.9.1 Příklad použití 1

Tímto způsobem je velice snadné s barvami pracovat bez použití dalšího nástroje. Preprocesor umožňuje měnit charakteristiky barvy nezávisle na tom, ve kterém barevném modelu byla původně specifikována. Je tedy možné měnit i ty charakteristiky, které pro původní model nejsou přístupné.

Ukázka kódu 4.20: Vstup v syntaxi jazyka preprocesoru

```
.div1 {
    color: rgb(123, 45, 56);
}
.div2 {
    color: lighten(rgb(123, 45, 56), 20);
}
```

Výše zmíněný příklad barvu zesvětlí (obr. 4.11) a vypíše.

Obrázek 4.11: Zesvětlení barvy

Ukázka kódu 4.21: Výstup v syntaxi jazyka CSS

```
.div1 {
        color: rgb(123, 45, 56);
}
.div2 {
        color: rgb(188, 76, 93);
}
```

Jak je vidět, pro nezkušeného uživatele není změna světlosti barvy v RGB modelu intuitivní. Preprocesor od nutnosti této znalosti odprošťuje.

4.6.9.2 Příklad použití 2

Operace s barvami je možné kombinovat. Bez použití jiného nástroje tak lze docílit velmi jemných barevných rozdílů (obr. 4.12).

Ukázka kódu 4.22: Vstup v syntaxi jazyka preprocesoru

```
.div1 {
    color: rgb(40, 117, 14);
}
.div2 {
    color: darken(saturate(rgb(40, 117, 14), 20), 5);
}
```

Obrázek 4.12: Jemné saturování barvy

Ukázka kódu 4.23: Výstup v syntaxi jazyka CSS

```
.div1 {
        color: rgb(40, 117, 14);
}
.div2 {
        color: rgb(26, 95, 0);
}
```

4.7 Implementace výpisu výstupu

Díky využití návrhového vzoru **Composite**[22] stačí nad výsledným zpreprocesovaným stromem zavolat funkci pro vytištění. Jednotlivé uzly a listy stromu následně volají stejnou funkci. Každý objekt stromu je tak zodpovědný za vytištění svého malého celku.

4.7.1 Analýza

Volitelně preprocesor umožňuje vytisknout přehlednou analýzu zatížení paměti a celkový čas běhu programu. O mechanismus se stará třída Analyzer, která v případě vyžádání zanalyzuje běh programu. Příklad textového výstupu analýzy může vypadat následovně:

```
Použitá paměť: 11212152 bytů ~10949.37Ki bytů
Čas běhu programu: 1.5273900032043 ms
```

4.8 Implementace ošetřování chyb

Zpracování chyb při běhu programu je realizováno rozšířením výjimek, které tvoří součást jazyka PHP. Výjimka je konstrukt mnoha programovacích jazyků, starající se o zpracování chyb na které program narazí. Jedná se o objekt nesoucí chybové hlášení, který má schopnost procházet od nejmenších součástí aplikace až do těch největších a čeká až bude zachycen. Zachycení chyby, a tím i její potlačení, může být žádoucím efektem. U některých parsovacích chyb se nadbytečné či nepotřebné informace zahazují. Tato nepotlačená chyba by způsobila ukončení celého běhu programu i v případech, kdy to není nezbytné. Odchytávání výjimek se v mnoha programovacích jazycích realizuje try-catch blokem.

Ukázka kódu 4.24: Příklad odchytávání výjimek

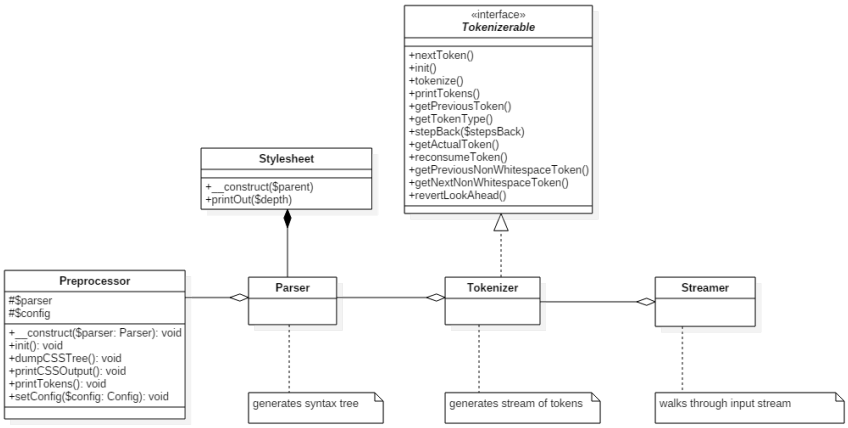
```
try {
  tryToDoSomething();
} catch (Exception $e) {
  somethingWentWrong();
}
```

Výjimky preprocesoru rozšiřují standardní výjimky jazyka PHP především o možnost zjistit, kde ve vstupních řetězci k chybě došlo.

4.9 Celkový přehled implementace

Preprocesor pracuje tak, že vstupní řetězec postupně prochází třída Streamer. Ta je kontrolována třídou Tokenizer a ze získaných znaků tvoří tokeny. Získané tokeny pak rozloží do významových pravidel třída Parser.

Preprocesor pak má tato pravidla k dispozici ve formě syntaktického stromu. A nad uzly a listy tohoto stromu se pak provede fáze preprocesování.



Obrázek 4.13: UML diagram struktury preprocesoru

Modifikovaná struktura stromu se pak vytiskne jako výstup v syntaxi jazyka kaskádových stylů pro použití při tvorbě webových stránek.

Kapitola 5

Výhody použití preprocesoru oproti jazyku CSS

Následující sekce popisuje výhody použití preprocesoru oproti klasickým kaskádovým stylům na komplexních příkladech.

5.1 Znovupoužitelnost kódu

Díky zavedení uživatelských proměnných a funkcí se kód oproti možnostem jazyka CSS stává znovu použitelný.

5.1.1 Příklad použití

Ukázka kódu 5.1: Vstup v syntaxi jazyka preprocesoru

```
mixin paragraph($fontface,$fontSize, $color) {
        font: $fontface $fontSize;
        color: $color;
        padding: 10px;
        background-color: #2e3436;
        border-radius: 10px;
        border: 3px solid $color;
}
.text1 {
        @mixin paragraph(15px, "Courier New", #761c19);
}
.text2 {
        @mixin paragraph(20px, "Times New Roman", #1b6d85);
        text-decoration: underline;
}
.text2 {
        @mixin paragraph(18px, "Verdana", #133d85);
        text-align: center;
```

```
}
```

Preprocesor opakovaně zavolá definovanou proceduru paragraph s různými parametry a na základě parametrů proceduru vypíše v syntaxi jazyka CSS.

Ukázka kódu 5.2: Výstup v syntaxi jazyka CSS

```
.text1 {
        font: 15px "Courier New";
        color: rgb(118, 28, 25);
        padding: 10px;
        background-color: rgb(46, 52, 54);
        border-radius: 10px;
        border: 3px solid rgb(118, 28, 25);
}
.text2 {
        font: 20px "Times New Roman";
        color: rgb(27, 109, 133);
        padding: 10px;
        background-color: rgb(46, 52, 54);
        border-radius: 10px;
        border: 3px solid rgb(27, 109, 133);
        text-decoration: underline;
}
.text2 {
        font: 18px "Verdana";
        color: rgb(19, 61, 133);
        padding: 10px;
        background-color: rgb(46, 52, 54);
        border-radius: 10px;
        border: 3px solid rgb(19, 61, 133);
        text-align: center;
}
```

5.2 Udržitelnost

Především zavedením dědičnosti a vnořování se oproti jazyku CSS zaručí udržitelnost kódu.

5.2.1 Příklad použití

Ukázka kódu 5.3: Vstup v syntaxi jazyka preprocesoru

```
.paragraph {
        color: black;
        padding: 10px;
}
.articles extends .paragraph {
        span {
                a {
```

```
                        text-decoration: none;
                }
        }
        div {
                p {
                        text-align: left;
                }
        }
}
```

V jazyku navrženém pro preprocesor se mezi zanořenými bloky snadno orientuje a kód se tak lépe udržuje.

Ukázka kódu 5.4: Výstup v syntaxi jazyka CSS

```
.paragraph, .articles {
        color: black;
        padding: 10px;
}
.articles  span a {
        text-decoration: none;
}

.articles  div p {
        text-align: left;
}
```

5.3 Zkrácení délky kódu

Definování vlastností v syntaxi navrženého jazyka je oproti jazyku kaskádových stylů v mnoha případech kratší.

5.3.1 Příklad použití

Ukázka kódu 5.5: Vstup v syntaxi jazyka preprocesoru

```
mixin barItem($size) {
    font-family: Raleway Dots, sans-serif;
    color: rgb(255, 255, 255);
    font-size: $size;
    background-color: rgb(140, 74, 79);
    border-left: 1px solid rgb(246, 150, 139);
    border-right: 1px solid rgb(246, 150, 139);
    color: rgb(255, 255, 255);
    margin: 10 10px;
    padding: 5px;
    font-style: italic;
}
.text1 {
    @mixin barItem(5em);
```

```
}
.text2 {
    @mixin barItem(10px);
    font-weight: normal;
}
.text3 excludeFrom .text2 {
    font-style;
}
```

Znovu použitelnost procedur a mechanismus vyjímání vlastností umožňují zkrátit zápis pro definování požadovaných vlastností.

Ukázka kódu 5.6: Výstup v syntaxi jazyka CSS

```
.text1 {
        font-family: Raleway Dots, sans-serif;
        color: rgb(255, 255, 255);
        font-size: 5em;
        background-color: rgb(140, 74, 79);
        border-left: 1px solid rgb(246, 150, 139);
        border-right: 1px solid rgb(246, 150, 139);
        color: rgb(255, 255, 255);
        margin: 10 10px;
        padding: 5px;
        font-style: italic;
}
.text2 {
        font-family: Raleway Dots, sans-serif;
        color: rgb(255, 255, 255);
        font-size: 10px;
        background-color: rgb(140, 74, 79);
        border-left: 1px solid rgb(246, 150, 139);
        border-right: 1px solid rgb(246, 150, 139);
        color: rgb(255, 255, 255);
        margin: 10 10px;
        padding: 5px;
        font-style: italic;
        font-weight: normal;
}
.text3 {
        font-family: Raleway Dots, sans-serif;
        color: rgb(255, 255, 255);
        font-size: 10px;
        background-color: rgb(140, 74, 79);
        border-left: 1px solid rgb(246, 150, 139);
        border-right: 1px solid rgb(246, 150, 139);
        margin: 10 10px;
        padding: 5px;
        font-weight: normal;
}
```

5.4 Odstranění duplicit

Preprocesor zavádí uživatelské proměnné a uživatelské funkce, které svým aplikováním odstraňují duplicity, mnohdy přítomné v jazyku CSS.

5.4.1 Příklad použití

Ukázka kódu 5.7: Vstup v syntaxi jazyka preprocesoru

```
mixin paragraph($fontface, $fontSize, $color) {
        font: $fontface $fontSize;
        color: $color;
        padding: 10px;
        border-radius: 4px;
        -moz-border-radius: 4px;
        -webkit-border-radius: 4px;
        -moz-box-shadow: 5px 5px 8px #CCC;
        -webkit-box-shadow: 5px 5px 8px #CCC;
        box-shadow: 5px 5px 8px #CCC
        border: 1px solid #999999;
        background-color: #EEEEEE;
        padding: 5px;
}
.class1 {
        @mixin paragraph(15px, "Courier New", #761c19);
}
.class2 {
        @mixin paragraph(20px, "Arial", #87D29E);
}
```

Vlastnosti jsou tedy definovány na jednom místě a neduplikují se.

Ukázka kódu 5.8: Výstup v syntaxi jazyka CSS

```
.class1 {
        font: 15px "Courier New";
        color: rgb(118, 28, 25);
        padding: 10px;
        border-radius: 4px;
        -moz-border-radius: 4px;
        -webkit-border-radius: 4px;
        -moz-box-shadow: 5px 5px 8px #CCC;
        -webkit-box-shadow: 5px 5px 8px #CCC;
        box-shadow: 5px 5px 8px #CCC
        border: 1px solid rgb(153, 153, 153);
        background-color: rgb(238, 238, 238);
        padding: 5px;
}
.class2 {
        font: 20px "Arial";
```

```
        color: rgb(135, 210, 158);
        padding: 10px;
        border-radius: 4px;
        -moz-border-radius: 4px;
        -webkit-border-radius: 4px;
        -moz-box-shadow: 5px 5px 8px #CCC;
        -webkit-box-shadow: 5px 5px 8px #CCC;
        box-shadow: 5px 5px 8px #CCC
        border: 1px solid rgb(153, 153, 153);
        background-color: rgb(238, 238, 238);
        padding: 5px;
}
```

5.5 Snadná editovatelnost kódu

Jazyk CSS neumožňuje definovat určitou vlastnost na jednom místě a pak ji opětovně používat. V případě nutnosti změn je pak potřeba tuto vlastnost změnit na všech místech jejího výskytu. Pokud by například bylo nutné změnit barevné rozvržení stránky, v každém místě, kde se s barvou pracuje, by bylo ji nutné upravit. Definováním barvy na začátku kódu a jejím použitím jako proměnné se této nutnosti v jazyku preprocesoru předejde.

5.5.1 Příklad použití

Ukázka kódu 5.9: Vstup v syntaxi jazyka preprocesoru

```
$color #1d2e3a;
body {
        font-family: "Arial CE", arial;
        color: $color;
}
.new-style > li h3:after {
        color: $color;
        font-weight: bold;
        font-size: 40px;
        line-height: 39px;
}
.functionTable table td .go {
        color: $color;
        font-weight: bold;
        font-size: 1.2em;
}
```

V případě nutnosti editace stačí barvu změnit pouze na jednom místě.

Ukázka kódu 5.10: Výstup v syntaxi jazyka CSS

```
body {
        font-family: "Arial CE", arial;
        color: #1d2e3a;
```

```
}
.new-style > li h3:after {
        color: #1d2e3a;
        font-weight: bold;
        font-size: 40px;
        line-height: 39px;
}
.functionTable table td .go {
        color: #1d2e3a;
        font-weight: bold;
        font-size: 1.2em;
}
```

Kapitola 6

Integrace do publikačního systému Webgarden

Publikační systém Webgarden[16] je internetová platforma pro správu obsahu na webových stránkách. Jako jeden z prvních podobných systémů se na českém internetu objevil v roce 2006[16]. Uživatelsky příjemnou formou umožňuje návštěvníkům portálu vytvářet a spravovat webové stránky i bez hlubší znalosti webové problematiky.

6.1 Architektura MVC

Jako mnoho podobných projektů stojí i Webgarden na softwarové architektuře MVC[39], neboli model-view-controller, někdy též chápán jako návrhový vzor, MVC rozděluje aplikaci na tři na sobě minimálně závislé logické celky.

6.1.1 Část Model (čes. model)

Model v sobě drží data a logiku aplikace, např. data a práce s databází.

6.1.2 Část View (čes. pohled)

Pohled může být jakýkoliv výstup informací z modelu, zpravidla v podobě vhodné k další uživatelské interakci. Několik pohledů může reprezentovat stejnou informaci. Příkladem může být uživateli zobrazená webová stránka.

6.1.3 Část Controller (čes. řadič)

Tato část má na starosti reagovat zpravidla na uživatelský vstup a provádí změny v modelu a pohledu.

6.1.4 Mechanismus MVC

Celek pak funguje následujícími po sobě jdoucími kroky:

1. Uživatel provede nějakou uživatelskou akci, například ve formě stisknutí tlačítka.
2. Objekt uživatelského rozhraní informuje o akci řadit.
3. Řadič v případě potřeby aktualizuje model, například upraví informace o uživateli.
4. Model na základě změněných dat zpracuje vnitřní stav, například upraví věk uživatele.
5. Pohled pak použije upravený model a zobrazí uživateli aktuální data.
6. Následně se čeká na uživatelský vstup a celý cyklus se opakuje.

6.2 Použití preprocesoru s přihlédnutím ke komerčnímu přístupu

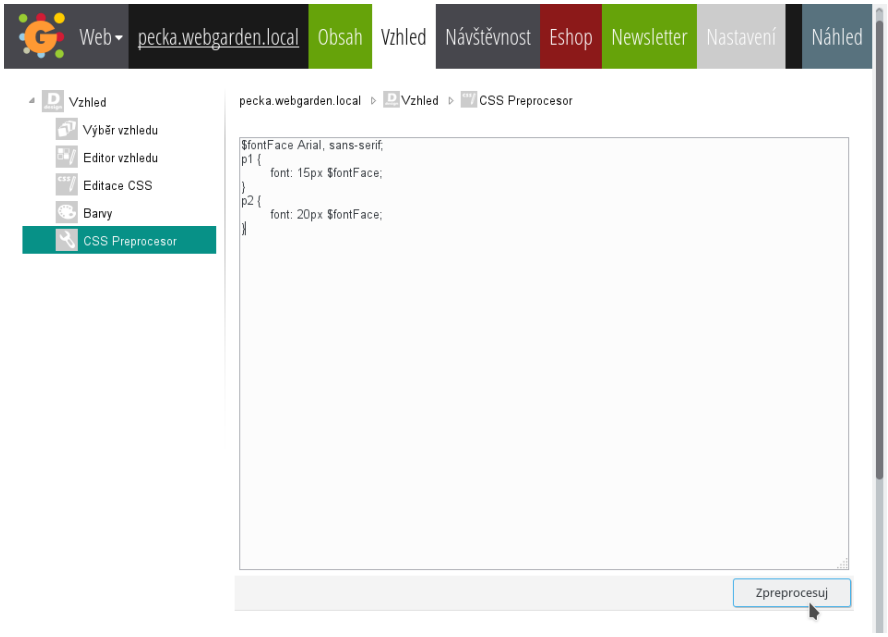
Publikačního systému Webgarden má dvě základní varianty, variantu placenou a neplacenou. Neplacenou variantu lze používat bez omezení funkčnosti systému, omezen je pouze datový tok, adresa stránek je reprezentována jako doména druhého řádu (např. kosmickevedomi.webgarden.cz) a na stránky je umisťována reklama. Placená varianta uživatele nijak neomezuje. Používat preprocesor lze v nezměněné funkčnosti i v neplacené verzi, s tím omezením, že systém drží v paměti pouze jeden preprocesní skript na webový projekt.

6.3 Propojení publikačního systému a preprocesoru

Díky využití pokynů z kapitoly o doporučených PHP standardech (viz. sekce 4.2.9) a připravení projektu jako balík pro Composer (viz. sekce 4.1.4) se preprocesor velmi snadno integruje. Na začátku skriptu stačí pouze vložit knihovnu pro autoloadování a díky využití PSR4 (kapitola 4.2.9.3) se automaticky pracuje pouze se skripty které jsou potřeba. Dále řekneme programu, aby pracoval s hlavní třídou preprocesoru.

Ukázka kódu 6.1: Načtení preprocesoru

```
require_once "vendor/autoload.php";
use Preprasor\Preprasor;
```



Obrázek 6.1: Integrace preprocesoru do prostření Webgarden

Při odeslání formuláře převezme řadič vstup a předá ho preprocesoru, ten vstup zpreprocesuje a předá modelu.

Ukázka kódu 6.2: Spuštění preprocesoru

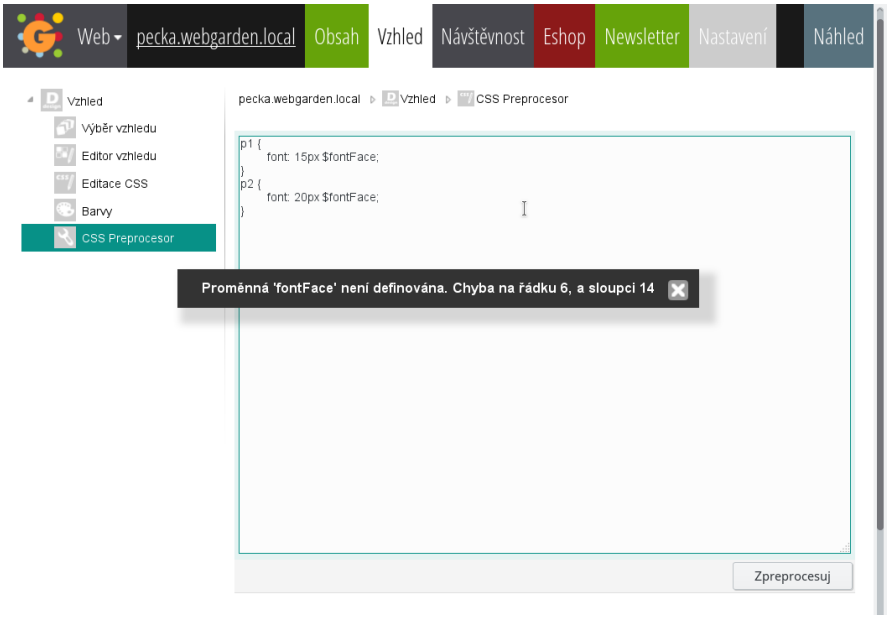
```
$factory = new StandardPreprocessorFactory;
$preprasor = $factory->createPreprasor($input, $workingDirectory);
$preprasor->init();
$output = $preprasor->printCSSOutput();
```

Výstup preprocesoru se pak zobrazí uživateli a uloží do databáze k další manipulaci. Zvýraznění syntaxe jazyka CSS zajišťuje knihovna Highlighter[33].

Obrázek 6.2: Výstup preprocesoru v prostředí Webgarden

6.4 Ošetřování chyb

Pokud se uživatel dopustí chyby ve vstupním souboru, preprocesor ho srozumitelně informuje.



Obrázek 6.3: Zobrazení uživatelské chyby s jejím výskytem

Zobrazování chyb v prostředí Webgarden je realizováno zachytáváním výjimek. Výjimky preprocesoru rozšiřují standardní výjimky jazyka PHP o přidruženou informaci ve formě řádku a sloupce, kde k chybě došlo.

Závěr

Předložená práce popisující vývoj preprocesoru pro webový projekt Webgarden si v úvodu vytyčila následující cíle:

- Vytvořit teoretický návrh preprocesoru, který bude sestaven na míru požadavkům projektu Webgarden.
- Implementovat aplikaci s použitím stávajících technologií projektu Webgarden.
- Integrovat preprocesor do publikačního systému Webgarden.

V rámci přípravy teoretického návrhu preprocesoru vykrystalizovala potřeba projektu Webgarden v konkrétní požadavky na rozšíření jazyka kaskádových stylů a to o uživatelské proměnné, uživatelské funkce, dědění, vnořování, vyjímání vlastností, práci s barvami v různých modelech a barevné tónování fotek. Na základě těchto požadavků byl navržen jazyk preprocesoru, který jazyk kaskádových stylů (CSS) o tyto vlastnosti rozšířil. Preprocesor se tak stal nástrojem pro snazší tvorbu udržitelnějšího a rozšiřitelnějšího kódu.

Pro konkrétní implementaci preprocesoru byly zvoleny stávající technologie používané publikačním systémem Webgarden, především skriptovací jazyk PHP. Kromě toho byla k tvorbě preprocesoru zvolena široká paleta nástrojů zjednodušující různá odvětví vývoje a rozšiřitelnosti softwaru.

Dodržením programovacích standardů se preprocesor podařilo bez problémů integrovat do prostředí publikačního systému Webgarden a v budoucnu bude umožňovat jak uživatelům systému, tak i jeho grafikům, jeho výhod využívat. Tímto došlo ke splnění i posledního z výše uvedených cílů.

Jako teoretický základ k výše uvedeným cílům bylo součástí práce i zevrubné představení problematiky jazyka kaskádových stylů (CSS) spolu s jeho výhodami a nedostatky. V práci byly představeny základní i pokročilé

principy objektově orientovaného návrhu spolu s vývojem řízeným testy a několika programovacími standardy jazyka PHP.

Pro úplnost je třeba zmínit, že předložený preprocesor opomíjí matematické funkce, převody mezi různými soustavami a jednotkami či operacemi nad řetězci, neboť zmíněné vlastnosti uživatelé při běžném návrhu v prostředí Webgarden nevyužijí. Umožňuje však jednoduše vyjímat vlastnosti z tříd jazyka kaskádových stylů a také provádět základní barevné operace s obrázky a to bez využití dalších nástrojů, z čehož budou uživatelé prostředí Webgarden významně benefitovat.

Preprocesor je nyní spuštěn online v testovací verzi. Kompletní podoba aplikace je také nahrána na CD přiloženém k této práci.

Literatura

[1] *CSS Syntax Module Level 3. [online].* [cit. 2016-12-20]. Dostupné z: `https://drafts.csswg.org/css-syntax-3/`

[2] *Požívanost CSS preprocesorů. [online].* [cit. 2017-08-01]. Dostupné z: `https://css-tricks.com/poll-results-popularity-of-css-preprocessors/`

[3] *SASS dokumentace. [online].* [cit. 2017-08-01]. Dostupné z: `http://www.sass-lang.com/documentation/`

[4] *LESS dokumentace. [online].* [cit. 2017-08-01]. Dostupné z: `http://lesscss.org/features/`

[5] Černá Ivana, Křetínský Mojmír, Kučera Antonín *Formální jazyky a automaty I* Elportál, první vydání, 2006, ISSN 1802-128X.

[6] *PHP Dokumentace. [online].* [cit. 2016-05-01]. Dostupné z: `http://php.net/`

[7] *Dokumentace k serveru Lighttpd. [online].* [cit. 2016-03-19]. Dostupné z: `http://redmine.lighttpd.net/projects/lighttpd/wiki`

[8] *Dokumentace verzovacího systému GIT. [online].* [cit. 2016-03-19]. Dostupné z: `https://git-scm.com/doc`

[9] *Dokumentace k utilitě Composer. [online].* [cit. 2016-03-19]. Dostupné z: `https://getcomposer.org/doc/`

[10] *The PHP Package Repository. [online].* [cit. 2016-12-19]. Dostupné z: `https://packagist.org/`

[11] *Dokumentace ke knihovně PHPunit. [online].* [cit. 2016-03-19]. Dostupné z: `https://phpunit.de/documentation.html`

[12] *Dokumentace k rozšíření Xdebug. [online].* [cit. 2016-03-19]. Dostupné z: `https://xdebug.org/docs/`

[13] *Dokumentace ke knihovně Tracy. [online].* [cit. 2016-03-19] Dostupné z: `https://phpunit.de/documentation.html`

[14] M.G. Helander, T.K. Landauer, P.V. Prabhu *Handbook of Human-Computer Interaction* Elsevier, 1997, str. 594 - 610

[15] *Obrázky pod licencí Creative Commons CC0. [online].* [cit. 2016-12-19]. Dostupné z: `https://pixabay.com/`

[16] *Publikační systém Webgarden. [online].* [cit. 2016-12-19]. Dostupný na: `http://www.webgarden.cz/`

[17] *Standard jazyka CSS. [online].* [cit. 2016-12-19]. Dostupný na: `https://www.w3.org/Style/CSS/`

[18] *Generátor RegExp obrázků. [online].* [cit. 2016-12-19]. Dostupný na: `https://regexper.com/`

[19] Robert C. Martin *Clean Code: A Handbook of Agile Software Craftsmanship* Prentice Hall PTR Upper Saddle River, NJ, USA 2008. ISBN 0132350882.

[20] Martin Fowler, Kent Beck, John Brant, William Opdyke, Don Roberts *Refactoring: Improving the Design of Existing Code* Springer-Verlag London, UK 2002. ISBN 0201485672.

[21] Joe Kutner *The Healthy Programmer: Get Fit, Feel Better, and Keep Coding* Pragmatic Bookshelf, 2013. ISBN 1937785319.

[22] Steven Fraser, Erich Gamma, Richard Helm, Ralph Johnson *Design Patterns: Elements of Reusable Object-Oriented Software* Addison-Wesley Professional, 1994. ISBN 0201633612.

[23] Jakub Vrána *1001 tipů a triků pro PHP* COMPUTER PRESS, 2012

[24] Marian Böhmer *Návrhové vzory v PHP* COMPUTER PRESS, 2012

[25] Steven Levithan, Jan Goyvaerts *Regular Expressions Cookbook: Detailed Solutions in Eight Programming Languages* O'Reilly Media, druhé vydání, 2012. ISBN 1449319432.

[26] Andrew Hunt, David Thomas *Pragmatic Programmer, The: From Journeyman to Master* Addison Wesley, 1999. ISBN 020161622X.

[27] Obrázky pod licencí Creative Commons *Wikimedia Commons* [cit. 2017-08-01]. Dostupné z: https://commons.wikimedia.org/

[28] Rudolf Pecinovsky *OOP - Learn Object Oriented Thinking and Programming* Tomas Bruckner, 2013. ISBN 8090466184.

[29] Kent Beck *Extreme programming explained: embrace change* Addison-Wesley Longman Publishing Co., Inc. Boston, USA, 2000

[30] Robert C. Martin *Agile Software Development: Principles, Patterns, and Practices* Pearson Education, 2003. ISBN 0135974445.

[31] *PHP Standards Recommendations. [online].* [cit. 2016-12-19]. Dostupné z: `http://www.php-fig.org/psr/`

[32] *Generátor dokumentace ve formě HTML stránek. [online].* [cit. 2016-12-19]. Dostupné z: `http://www.apigen.org/`

[33] *Zvýražnění syntaxe jazyka CSS pro PHP. [online].* [cit. 2016-12-19]. Dostupné z: `https://github.com/scrivo/highlight.php`

[34] *Vývojové prostředí NetBeans. [online].* [cit. 2016-12-19]. Dostupné z: `https://netbeans.org`

[35] Håkon Wium Lie, Bert Bos *Cascading Style Sheets, designing for the Web. [online].* [cit. 2017-08-01]. Dostupné z: `https://www.w3.org/Style/LieBos2e/history/Overview.html` Addison Wesley, druhé vydání, 1999, ISBN 0-201-59625-3. Kapitola 20.

[36] John Child, Mark Galer *Photographic lighting: essential skills* čtvrté vydání, 2008, Focal Press, Oxford, UK. ISBN 0240519647.

[37] *Červený trpaslík, 1. díl, 1. série, díl Konec* [Red dwarf]. [seriál]. Režie Ed Bye. UK, 1988.

[38] Kochurkin, Ivan *Tree structures processing and unified AST. [online].* [cit. 2017-08-01]. Dostupné z: `http://blog.ptsecurity.com/2016/07/tree-structures-processing-and-unified.html` Positive Research Center.

[39] Krasner, Glenn E.; Pope, Stephen T. *A cookbook for using the model–view controller user interface paradigm in Smalltalk-80.* [cit. 2017-08-01]. Dostupné z: `http://www.global-webnet.com/Adventures/Files/DescriptionOfMvcUi-KrasnerPope.pdf` ParcPlace Systems

Příloha **A**

Seznam použitých zkratek

CSS Cascading Style Sheets

PHP PHP: Hypertext Preprocessor

SASS Syntactically Awesome Style Sheets

XHTML Extensible HyperText Markup Language

HTML HyperText Markup Language

XML Extensible Markup Language

W3C World Wide Web Consortium

DRY Don't repeat yourself

MVC Model View Controller

IDE Integrated Development Environment

Příloha **B**

Obsah přiloženého CD

Charakteristika nejpodstatnějších souborů na přiloženém CD
Následující graf usnadňuje orientaci mezi nejvýznamnějšími adresáři a soubory práce.

- `.GIT` ..... soubory verzovacího systému GIT
- `doc` ..... adresář s dokumentací
- `preprasor` ..... zdrojové kódy implementace
- `text` ..... soubory, které jsou součástí textu práce
  - `thesis.tex` ..... zdrojová forma práce ve formátu LaTeX
  - `thesis.pdf` ..... text práce ve formátu PDF
- `tests` ..... testy pro Unit testovaní
- `vendor` ..... externí knihovny
- `www` ..... pomocné soubory běžné součásti webových stránek
  - `index.html` ..... vstup do jednoduchého uživatelského rozhraní pro testování preprocesoru online
  - `examples.html` ..... příklady použití prerocesoru
- `README.MD` ..... charakteristika preprocesoru jako GIT repozitáře
- `README.txt` ..... stručná charakteristika preprocesoru