МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ  
УЧРЕЖДЕНИЕ ОБРАЗОВАНИЯ «ВИТЕБСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМЕНИ П.М. МАШЕРОВА»  
  
Факультет математики и информационных технологий  
???

КУРСОВАЯ РАБОТА

по дисциплине специализации

Оптимизация HTML-верстки

Бакштай Виталий Викторович,

студент 3 курса, группы 31

Руководитель:

???,

???

Витебск, 2017

Содержание

[Введение 3](#_Toc501661365)

[1 Теоретическая часть 4](#_Toc501661366)

[1.1 Постановка задачи 4](#_Toc501661367)

[1.2 Терминология 4](#_Toc501661368)

[1.3 Antlr – краткие сведения 5](#_Toc501661369)

[1.4 Обход AST 6](#_Toc501661370)

[1.5 Visitor и Listener 6](#_Toc501661371)

[1.6 Antlr в Visual Studio 6](#_Toc501661372)

[1.7 Построение сокращенных имен 7](#_Toc501661373)

[1.8 Поиска имен идентификаторов и классов 8](#_Toc501661374)

[1.8.1 HTML 8](#_Toc501661375)

[1.8.2 CSS 8](#_Toc501661376)

[1.8.3 JavaScript 8](#_Toc501661377)

[1.9 Алгоритм работы приложения 8](#_Toc501661378)

[2 Практическая часть 9](#_Toc501661379)

[2.1 9](#_Toc501661380)

[2.2 Разбор CSS-файла. 9](#_Toc501661381)

[2.3 Хранение селекторов и их свойств. 10](#_Toc501661382)

[2.4 Построение дерева CSS-файла 12](#_Toc501661383)

[2.5 Удаление наследуемых правил 15](#_Toc501661384)

[Заключение 16](#_Toc501661385)

[Список использованной литературы 17](#_Toc501661386)

[Приложение А 18](#_Toc501661387)

[Приложение Б 19](#_Toc501661388)

# Введение

В наши дни на просторах интернета практический каждый сайт написан с использованием таких языков, как html, css, javacript. Но не каждый разработчик задумывается над оптимальным написанием идентификаторов и имен классов в этих файлах. Да это и не логично на этапе разработке сайта, так как давать короткие и не говорящие имена не удобно. Но на момент размещения сайта на серверах данная информация не несет никакого смысла. Отсюда на лицо проблема с неоправданно большим размером загружаемых файлов для отображения страниц в браузере. Решение данной проблемы имеет важное теоретическое и практическое значение, так как чем меньше будет размер загружаемых файлов, тем быстрее они будут отображаться для пользователей. Поисковые системы также присваивают сайтам с меньшей скоростью загрузки более высокий ранг в списке выдачи для поискового запроса при прочих равных параметрах. На некоторых серверах (хостингах) ограничен размер хранимых данных, что также является важной проблемой.

Объект: html-верстка представляющая набор файлов css, js, html.

Целью данной курсовой работы является задача оптимизации html-верстки представлящаяя набор файлов css, js, html. Основное внимание будет уделено именованию идентификаторов и классов.

Для достижения данной цели возникают следующие задачи:

* лексический анализ файлов html, css, js;
* построение AST (abstract syntax tree) для каждого типа файлов;
* обработка полученных AST;
* построение алгоритма выявления и сокращение имен идентификаторов и классов.

# Теоретическая часть

## Постановка задачи

Имеется html-верстка. Целью данной курсовой работы является задача оптимизации данной верстки представляющая набор файлов css, js, html. А конкретнее, это сокращение именования имен идентификаторов и классов, используемых в этих трех типах файлах.

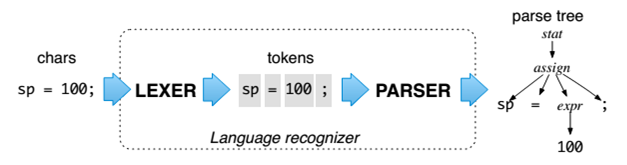
Чтобы произвести нужную оптимизацию, требуется считывать последовательно из набора файлы и разобрать их. То есть найти все упоминания имен идентификаторов и классов. После разбора необходимо сократить все найденные именования.

## Терминология

Парсинг — процесс преобразования исходного кода в структурированный вид. Типичный парсер представляет собой комбинацию лексера и парсера. Лексер группирует символы исходного кода в значащие последовательности, которые называются лексемами. После этого определяется тип лексемы (идентификатор, число, строка и т.п.). Токеном называется совокупность значения лексемы и ее типа. В примере на рисунке 1.2.1 токенами являются “*sp”, “=”, “100”*. Парсер же из потока токенов строит связную древовидную структуру, которая называется деревом разбора. В данном случае “*assign”* является одним из узлов дерева. Абстрактное синтаксическое дерево или AST — дерево разбора на более высоком уровне, из которого удалены не значимые токены, такие как скобки, запятые. Однако существуют парсеры, в которых шаг лексирования и разбора совмещены.

Для описания различных узлов AST используются правила. Объединение всех правил называют грамматикой языка. Существуют инструменты, генерирующие код под определенную платформу для разбора языков на основе грамматик. Они называются генераторами парсеров. Например, ANTLR, Bison, Coco/R.

Рисунок 1.2.1 – Пример парсинга



## Antlr – краткие сведения

Для разбора файлов было решено воспользоваться фрэймворком Antlr (ANother Tool for Language Recognition – “ещё одно средство распознавания языков“). Одна из основных причин выбора, это наличие на сайте разработчика готовых файлов описания граматики для большинства используемых языков. ANTLR позволяет создавать лексические и синтаксические анализаторы на различных языках (Java, C, C++, Python, C#, ActionScript, JavaScript, PHP) на основе грамматик.

Процесс создания анализатора можно описать следующим образом:

* создание грамматики для лексического анализатора
* создание грамматики для синтаксического анализатора
* отладка грамматики
* генерация классов лексического и синтаксического анализатора
* компиляция классов лексера(лексического анализатора) и парсера(синтаксического анализатора)
* использование классов для обработки текста

## Обход AST

Как известно, парсер преобразует исходный код в дерево разбора (в дерево, в котором убраны все незначимые токены), называемом AST. Существуют различные способы обработать такое дерево. Простейший — заключается в обработке дерева с помощью рекурсивного обхода потомков в глубину. Однако данный способ применим только для совсем простых случаев, в котором существует немного типов узлов и логика обработки простая. В остальных случаях, необходимо выносить логику обработки каждого типа узла в отдельные методы. Это осуществляется с помощью двух типовых подходов (паттернов проектирования): Visitor и Listener.

## Visitor и Listener

В Visitor для посещения потомков узла необходимо вручную вызывать методы обхода дочерних узлов. При этом если родитель имеет три потомка, и вызвать методы только для двух узлов, то часть поддерева вообще не будет обработана. В Listener же методы посещения всех потомков вызываются автоматически. В Listener существует метод, вызываемый в начале посещения узла (enterNode) и метод, вызываемый после посещения узла (exitNode). Методы Visitor, в отличие от Listener, могут возвращать объекты и даже могут быть типизированными, т.е. при объявлении \*Visitor каждый метод Visit, будет возвращать объект AstNode, который в нашем случае является общим предком для всех остальных узлов унифицированного AST.

Таким образом, код преобразования дерева с помощью Visitor получается более функциональным и лаконичным за счет того, что в нем нет необходимости хранить информацию о посещенных узлах.

## Построение сокращенных имен

Имена идентификаторов и классов могут начинаться только с латинских букв строчных либо прописных, а последующие символы могу содержать цифр. Так же имена с прописными буквами и строчными считаются, как два разных имени. Следовательно, генерация имен будет заключаться в последовательном переборе символов. При переборе всех символов в k-ой позиции будет произведен переход к следующему символу в k+1-ой позиции, а в текущей позиции переход к первому символу. На первую позицию налаживается ограничение, оговоренное выше.

Пусть имеется словарь, сопоставляющий некоторому числу символ.

Пусть имеется массив, содержащий набор индексов символов, представляющий текущее имя.

Алгоритм:

1. увеличить первый индекс символа на 1;
2. k=0, где k – номер текущего индекса в массиве;
3. проверить на выход k-ого индекса за границу доступных символов, если выход есть, то перейти к Шагу 4, иначе к Шагу 7;
4. обнулить k-ый индекс;
5. увеличить на единицу значение k;
6. выполнить Шаг 3;
7. преобразовать текущий массив в строчное представление используя ранее описанный словарь символов.

Выполняя данный алгоритм на одном и том же массиве можно получать не повторяющиеся имена с минимальной длинной.

## Поиска имен идентификаторов и классов

### HTML

В HTML идентификаторы хранятся в значении атрибута “id”, а имена классов в значении атрибута “class”. Имена классов записаны через пробел, поэтому их легко получить, разделив строку по пробельному символу.

### CSS

В CSS идентификаторы могут записывать в селекторах. А записываются в виде строки, где первый символ это “#”, а остальная часть, это название идентификатор. С классами аналогичная ситуация только первым символом является “.”.

### JavaScript

В JavaScript есть множество способов обращения к DOM объектам страницы. Примеры функций:

* getElementById(name:string) – получение узла с идентификатором name.
* getElementsByClassName(names:string) – получение списка узлов с именами классов names, где в names имена записаны через пробел.

Так как очень сложно отследить какая именно была передана в данные функции, было принято решение сокращать литеральные строки, полностью совпадающие с именами ранее сокращенных имен. В большинстве случаев разработчик передает данные в данные функции именно литералом.

## 

## Алгоритм работы приложения

Полученное в результате приложение должно делать копию существующей директории. Данный алгоритм будет рекурсивно просматривать директории, при этом оптимизируя нужные файлы. На вход подается начальная директория.

Алгоритм работы:

1. просмотреть все вложенные директории и выполнить этот же алгоритм для каждой найденной директории;
2. найти следующий файл, если нет, то перейти к Шагу 8;
3. определить тип файла;
4. если это файл типа “.js” то поместить его в очередь;
5. изменить все имена идентификаторов и классов;
6. записать измененный файл в новую директорию;
7. перейти к Шагу 2;
8. взять файл из очереди;
9. изменить имена на основании найденных имен;
10. выполнить Шаг 6;
11. перейти к шагу 8.

# Практическая часть

Для решения поставленных задач был выбран язык программирования C#, а средой разработки Visual Studio.

## Antlr в Visual Studio

Так как средой разработки была выбрана Visual Studio необходимо автоматизировать процесс генерации классов лексера и парсера при изменении граматики. Данную возможность предлагает Visual Studio. Для работы с Antlr достаточно установить через менеджер управлением пакетов NuGet пакет “Antlr4”. Это возможно сделать через графический интерфейс NuGet или выполнить команду из листинга 2.1.1.

Листинг 2.1.1 – Команда для установки пакетов Antlr

Install-Package Antlr4

Установив данный пакет в окне добавления элемента в проект появится дополнительные пункты, связанные с Antlr, такие как “Antlr 4 Lexer”, “Antlr 4 Parser”, “Antlr 4 Combine”. Последний из представленных создает файл, описывающий одновременно и лексер и парсер грамматики.

После создания одного из представленных файлов Visual Studio будет используя установленный пакет компилировать файл в классы реализующих лексический анализ, это файл \*Lexer, и синтаксический анализ, это файл \*Parser, где \* это название файлов, описывающих граматику. Данная компиляция будет происходить автоматически при сохранениях файла граматики.

## Реализация алгоритма генерации имен

На основнии алгоритма, описанного в пункте 1.6 был реализован следующий класс NameGenerator, представленный в листингах 2.2.1 – 2.2.2.

Листинг 2.2.1 – Реализация класса NameGenerator

class NameGenerator : INameGenerator {

//возможные значения для первого символа

private char[] posibleValueFirst;

//возможные значения для остальных символов

private char[] posibleValue;

private IList<int> curentCode;

public NameGenerator() {

//реализация конструктора

}

//метод получения следующего имени

public string GetNext() {

StringBuilder sb = new StringBuilder();

int i = 0;

sb.Append(posibleValueFirst[curentCode[i]]);

i++;

while (i < curentCode.Count) {

sb.Append(posibleValue[curentCode[i]]);

i++;

}

int cur = 0;

if (curentCode[cur] + 1 == posibleValueFirst.Length) {

curentCode[cur] = 0;

cur++;

}

while (cur < curentCode.Count

&& curentCode[cur] + 1 == posibleValue.Length) {

curentCode[cur] = 0;

cur++;

}

if (cur == curentCode.Count) {

curentCode.Add(-1);

}

curentCode[cur]++;

return sb.ToString();

}

}

Листинг 2.2.2 – Реализация конструктора класса NameGenerator

public NameGenerator() {

//заполнение массивов возможных значений

curentCode = new List<int>();

var tempVal = new List<char>();

for (char i = 'a'; i <= 'z'; i++) {

tempVal.Add(i);

}

for (char i = 'A'; i <= 'Z'; i++) {

tempVal.Add(i);

}

posibleValueFirst = tempVal.ToArray();

for (char i = '0'; i <= '9'; i++) {

tempVal.Add(i);

}

posibleValue = tempVal.ToArray();

//начальное значение

curentCode.Add(0);

}

Класс NameGenerator реализует интерфес INameGenerator описанный в листинге 2.2.3. Используя такой подход классы использующие данный интерфейс, не зависят от конкретной реализации данного интерфейса.

Листинг 2.2.3 – Описание интерфейса INameGenerator

public INameGenerator () {

string GetNext();

}

Ниже приведен пример имен, которые генерирует класс.

Листинг 2.2.4-Пример имен

[a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z,A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z,aa,ba,ca,da,ea,fa,ga,ha,ia,ja,ka,la,ma,na,oa,pa,qa,ra,sa,ta,ua,va,wa,xa,ya,za,Aa,Ba,Ca,Da,Ea,Fa,Ga,Ha,Ia,Ja,Ka,La,Ma,Na,Oa,Pa,Qa,Ra,Sa,Ta,Ua,Va,Wa,Xa,Ya,Za,ab,bb,cb]

## Реализация фабрики сокращенных имен

После того как будут найдены нужные имена необходимо где-то хранить новые имена идентификаторов и классов. Для этого был написан класс FactoryNames, который описан в листинге 2.3.1, реализующий интерфейс IFactoryNames описанный в листинге 2.3.2.

## Разбор CSS-файла.

Разбор CSS-файла будет производиться фреймворком «*Alba.CsCss*». Данный фреймворк производит полный разбор CSS файла. Основан на парсерах Firefox.

Пример использования данного фреймворка описан в листинге 2.1.1.

Листинг 2.1.1 Пример разбора содержимого CSS-файла.

CssLoader cssLoader = new CssLoader();

CssStyleSheet css = cssLoader.ParseSheet("h1{color: red;}",

new Uri("http://my.ru/test.css"),

new Uri("http://my.ru"));

Console.WriteLine(css.StyleRules.Single()

.Declaration.Color.Color.R);

Результат выполнения данного будет вывод в консоль красной составляющей цвета свойства «*color*» селектора «*h1*».

Так же данный фреймворк поддерживает валидацию CSS-стилей. Для этого в классе CssLoader содержится событыие «ParseError». Оно будет вызываться данным классом тогда, когда будет найдена ошибка.

Листинг 2.1.2 Пример кода, реагирующий на ошибки в CSS-стилях.

CssLoader cssLoader = new CssLoader();

StringBuilder sb = new StringBuilder();

cssLoader.ParseError += (sender,e)=> {//подписывание на событие

sb.Append(e.Message).Append("\n");

sb.Append("Number line: ")

.Append(e.LineNumber)

.Append("\n");

};

CssStyleSheet css = cssLoader.ParseSheet("h1{colir: red;}",

new Uri("http://my.ru/test.css"),

new Uri("http://my.ru"));

if(sb.Length != 0) {

Console.WriteLine(sb);

}

В данном примере указано правило «*colir*», которого нет в спецификации CSS.

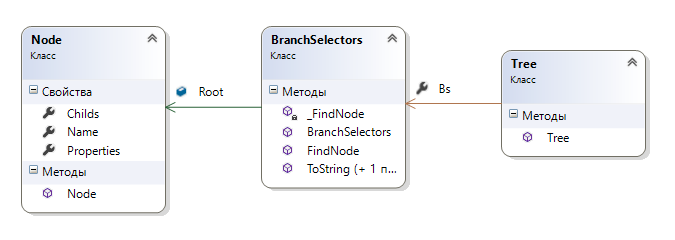
Рисунок 2.1.1 Результат работы программы с не существующим правилом.



## Хранение селекторов и их свойств.

Хранение дерево селекторов будет организованно следующим образом.

Рисунок 2.2.1 Диаграмма классов



Класс «*Node*» является информационным экспертом и хранит в себе: имя селектора, правила, ссылки на узлы, которые наследуют правила данного селектора.

Листинг 2.2.1 Описание класса «*Node*»

class Node {

public string Name {get; set;}

public Dictionary<CssProperty,PropertyValue>

Properties{get; }

public List<Node> Childs { get; }

public Node() {

Name = string.Empty;

Properties = new Dictionary<CssProperty,PropertyValue>();

Childs = new List<Node>();

}

}

struct PropertyValue {

public CssValue Value { get; set; }

public bool Important { get; set; }

}

Класс «*BranchSelectors*» является деревом содержащий в себе селектора и их правила. И имеет метод поиска селектора по названию.

Листинг 2.2.2 Описание класса «*BranchSelectors*»

class BranchSelectors {

public Node Root;

public BranchSelectors() {

Root = new Node { Name = "\*"};

}

public Node FindNode(String cs) {

return \_FindNode(cs,Root);

}

private Node \_FindNode(String cs,Node node) {

if(node.Name.CompareTo(cs) == 0) {

return node;

}

foreach(Node child in node.Childs) {

Node result = \_FindNode(cs,child);

if(result != null) {

return result;

}

}

return null;

}

}

Класс «*Tree*» содержит в себе дерево селекторов. Это сделано для того, чтобы в дальнейшем можно было добавить поддержку других частей CSS-файла, таких как @media, @keyframe, @font и так далее.

Листинг 2.2.3 Описание класса «*Tree*»

class Tree {

public BranchSelectors Bs{ get; }

public Tree() {

Bs = new BranchSelectors();

}

}

## Построение дерева CSS-файла

После разбора CSS-файла на основе разобранных данных необходимо построить дерево селекторов. За построение дерева отвечают два класса: «*ParseRules*» и «*ParsingSelector*».

Класс «*ParseRules*» используя класс «*ParsingSelector*» заполняет дерево селекторов и заполняет каждый селектор правилами с учетом «*!important*». После работы данного класса создается заполненное дерево селекторов.

Листинг 2.3.1 Описание класса «*ParseRules*»

class ParseRules {

public void Parse(Tree tree,CssStyleSheet css) {…}

private void parseStuleRule(Tree tree,CssStyleRule sr) {…}

private void addNodeProperty(Node node,CssDeclaration cd) {…}

}

Метод «*Parse*» для каждого селектора из файла вызывает функцию «parseStuleRule».

Листинг 2.3.2 Реализация метода «Parse»

public void Parse(Tree tree,CssStyleSheet css) {

foreach(CssStyleRule sr in css.StyleRules) {

parseStuleRule(tree,sr);

}

}

Метод «*parseStuleRule*» при помощи класса «*ParsingSelector*» получает селектор из дерева и записывает в его правила.

Листинг 2.3.3 Реализация метода «parseStuleRule»

private void parseStuleRule(Tree tree,CssStyleRule sr) {

ParsingSelector ps = new ParsingSelector(tree.Bs);

foreach(CssSelectorGroup sg in sr.SelectorGroups) {

Node node = ps.Parse(sg);

addNodeProperty(node,sr.Declaration);

}

}

Листинг 2.3.4 Реализация метода «addNodeProperty»

private void addNodeProperty(Node node,CssDeclaration cd) {

var prop = node.Properties;

CssPropertyManager cpm = new CssPropertyManager();

foreach(CssPropertyValue pv in cd.ImportantData) {

if(cpm.isHave(pv.Property)) {

if(prop.ContainsKey(pv.Property)) {

prop[pv.Property] = new PropertyValue {

Value = pv.Value,

Important = true };

} else {

prop.Add(pv.Property,new PropertyValue {

Value = pv.Value,

Important = true }

);

}

}

}

foreach(CssPropertyValue pv in cd.AllData) {

if(prop.ContainsKey(pv.Property)) {

var propValue = prop[pv.Property];

if(propValue.Important == false) {

propValue.Value = pv.Value;

}

} else {

if(cpm.isHave(pv.Property)) {

if(prop.ContainsKey(pv.Property)) {

prop[pv.Property] = new PropertyValue {

Value = pv.Value,

Important = false };

} else {

prop.Add(pv.Property,new PropertyValue {

Value = pv.Value,

Important = false }

);

}

}

}

}

}

Класс «*ParsingSelector*» отвечает за разбор селекторов и помещении в дерево селекторов.

Листинг 2.3.5 Описание класса «*ParsingSelector*»

class ParsingSelector {

public BranchSelectors Bs { get; }

public ParsingSelector(BranchSelectors bs){…}

public Node Parse(CssSelectorGroup csg) {…}

private List<Node> getPartNodes(CssSelector cs,

StringBuilder sb){…}

private bool isExtend(char oper){…}

}

Метод «*Parse*» производит анализ частей селекторов и операторов между ними. В зависимости от этого добавляет в дерево новые селектора и устанавливает связи между ними. Возвращает ссылку на узел дерева в который следует поместить правила.

Метод «*getPartNodes*» возвращает список селекторов, которые участвуют в селекторе.

Метод «*isExtend*» определяет будет ли наследовать селектор после оператора правила селектора до оператора.

## Удаление наследуемых правил

Удаление лишних правил реализует класс «*OptimiseBranchSelectors*».

Листинг 2.4.1 Описание класса «*OptimiseBranchSelectors*»

class OptimiseBranchSelectors {

public void ToOptimize(BranchSelectors bs) {…}

private void \_ToOprimize(Node node,

Dictionary<CssProperty,

CssValue> properties){…}

}

Метод «*ToOptimize*» запускает рекурсивную функцию «\_*ToOptimize*» передавая в качестве параметров корень дерева и пустой список свойств.

Листинг 2.4.2 Реализация метода «*ToOptimize*»

public void ToOptimize(BranchSelectors bs) {

\_ToOprimize(bs.Root,new Dictionary<CssProperty,CssValue>());

}

Листинг 2.4.3 Реализация метода «\_*ToOptimize*»

private void \_ToOprimize(Node node,

Dictionary<CssProperty,

CssValue> properties) {

var cpm = new CSSManager.CssPropertyManager();

foreach(CssProperty cp in properties.Keys) {

if(node.Properties.ContainsKey(cp)

&& cpm.isInherited(cp)) {

var val = node.Properties[cp].Value;

if(val.Equals(properties[cp])) {

Console.WriteLine("Оптимизация");

Console.WriteLine(node.Name);

Console.WriteLine(cpm.GetStringValue(cp));

node.Properties.Remove(cp);

}

}

}

properties = new Dictionary<CssProperty,CssValue>

(properties);

foreach(CssProperty cp in node.Properties.Keys) {

var val = node.Properties[cp];

if(properties.ContainsKey(cp)) {

properties[cp] = val.Value;

} else {

properties.Add(cp,val.Value);

}

}

foreach(var child in node.Childs) {

\_ToOprimize(child,properties);

}

}

# Заключение

В ходе выполнения курсовой работы проводился анализ CSS-файла. Целью курсовой работы являлась его минификация путем удаления лишних правил, которые наследуются.

В курсовой работе представлены ряд алгоритмов, позволяющих решить данные задачи.

Результатом курсовой работы является приложение, которое производит валидацию CSS-файла, анализирует и удаляет лишние правила в CSS-файле.

В дальнейшем планируется ввести следующие изменения и усовершенствования приложения:

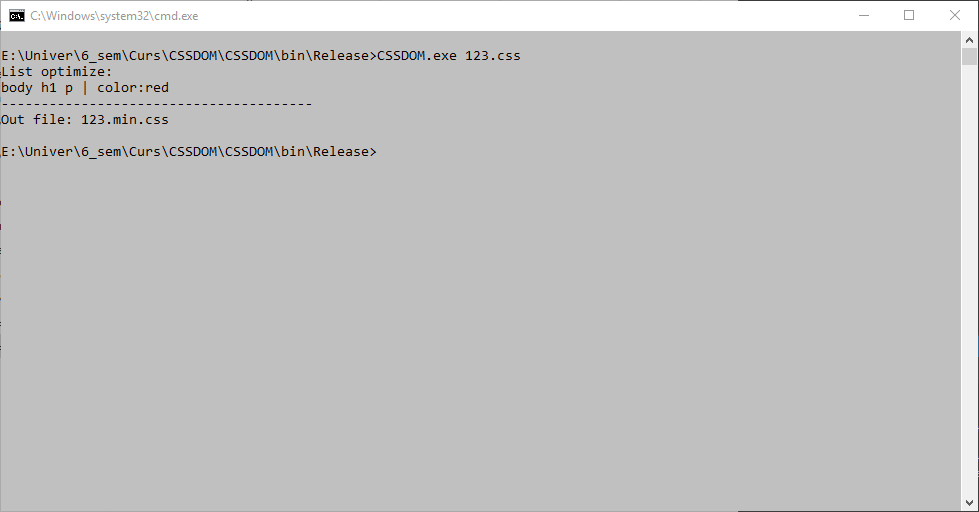
* улучшить алгоритм удаления лишних правил в селекторах;
* добавление поддержки @media, @keyframe, @font.
* реализовать оптимизацию в совокупности с HTML-страницами.

# Список использованной литературы

1. Cascading Style Sheets [Электронный ресурс]. - Режим доступа: https://www.w3.org . Дата доступа: 10.05.2017.
2. Дэвид Макфарланд, CSS3: The Missing Manual 3-е издание. — 3-е. — Санкт-Петербург, 2014. — С. 608 — ISBN 978-5-496-00428-2
3. Свойства в CSS, наследуемые по умолчанию [Электронный ресурс] – Режим доступа: <http://cssfollow.ru/inherited-by-default/> . Дата доступа : 07.05.2017.
4. Все спецификации CSS [Электронный ресурс] – Режим доступа: https://www.w3.org/Style/CSS/specs.ru.html . Дата доступа : 08.05.2017.

# Приложение А

**Пример запуска приложения**



# Приложение Б

**Содержимое дискового носителя**

1. \CSSDOM - файлы исходного кода проекта.
2. \Release - разработанное приложение
3. Записка.docx - пояснительная записка к курсовой работе.