

Hochschule Osnabrück
University of Applied Sciences

Implementierung eines Source-To-Source-Compilers zur Optimierung von CSS-Dateien im Webstack

Oliver Erxleben (oliver.erxleben@hs-osnabrueck.de)

Sergej Hert (sergej.hert@hs-osnabrueck.de)

Jörn Voßgröne (joern.vossgroene@hs-osnabrueck.de)

Hochschule Osnabrück
Ingenieurwissenschaften und Informatik
Informatik - Mobile und Verteilte Anwendungen
Compilerbau - Sommersemester 2013

20. September 2013

Inhaltsverzeichnis

1	Einleitung	1
1.1	Motivation	1
1.2	Anforderungen	1
2	Einführung in CSS	3
2.1	Cascading Stylesheets	3
2.2	Revisionen und Probleme	3
2.3	Allgemeiner Aufbau	3
2.4	Grammatik	4
3	Kommandozeilenwerkzeug	6
3.1	Makefile und Start des Programms	6
3.2	Datenstrukturen	7
3.3	Grundaufbau des Kommandozeilenwerkzeugs	7
4	Baumgenerierung	8
4.1	Abbildung der CSS Grammatik in Flex	8
4.2	Benötigte Datenstrukturen zur Erstellung des Baums	9
4.3	Baumgenerierung mit Bison	11
5	Optimierung	14
5.1	Zusammenfassen von Regeln mit gleichem Selektor	14
5.2	Zusammenfassen von Regeln mit gleicher Deklarationsliste	14
5.3	CSS Shorthands	15
6	Messungen und Ergebnis	16
6.1	Grundlage	16
6.2	Bewertung	17
6.3	Fazit	19
	Literaturverzeichnis	20
	Anhänge	21

Abbildungsverzeichnis

1	Workflow der Anwendung	2
2	Screenshot der Messdaten	16
3	Ladezeiten ohne Optimierung	18
4	Ladezeiten mit Optimierung	18

Listings

1	Aufbau einer CSS-Regel	4
2	Grammatik des W3C	4
3	Makefile	6
4	Programmstart	7
5	Datenstrukturen	7
6	Beschreibung der CSS Grammatik in Flex	8
7	C Strukturen für die Baumgenerierung	10
8	Baumgenerierung mit Bison	11
9	Optimize-Funktion	14
10	Ruby Webservice für Messdaten	17

Zusammenfassung:

Die vorliegende Ausarbeitung wurde in LaTeX verfasst und ist eine gemeinsame Arbeit von Oliver Erxleben, Sergej Hert und Jörn Voßgröne an der Hochschule Osnabrück / University of Applied Sciences im Fachbereich Ingenieurwissenschaften und Informatik für das Fach Compilerbau im Sommersemester 2013. Die Arbeit beschäftigt sich mit der Optimierung von Cascading Stylesheets für Webseiten.

Die Arbeit gliedert sich in mehrere Abschnitte. Im ersten Abschnitt wird der Hintergrund des Themas beschrieben und es werden Anforderungen an die Anwendung aufgestellt.

Im zweiten Teil, Cascading Style Sheets, werden die Sprache, dessen Verwendung, sowie die zugrundeliegende Grammatik erläutert.

Der dritte Teil beschreibt den Aufbau des CLI-Tools und dessen Datenstrukturen.

Das Thema Baumgenerierung, vierter Teil, beschreibt die verwendete Grammatik, den Aufbau des Baums und dessen Generierung.

Die Optimierungen auf dem Baum werden im Abschnitt fünf aufgezeigt.

Der Abschnitt Messungen und Ergebnis erklärt das Vorgehen für Tests und es wird die Arbeit resümiert und das Ergebnis zusammengefasst.

1 Einleitung

1.1 Motivation

Im Alltag eines Frontend-Entwicklers, eines Mitarbeiters an einem Web-Projekt oder des Entwicklers für das grafische User Interface kommt es nicht selten vor, dass die Beschreibung der grafischen Elemente durch Cascading Stylesheets geschieht. Noch seltener sind diese Style-Angaben fehlerfrei. Sei es aufgrund von Zeitdruck, unterschiedlichen Entwicklern oder durch nachträgliches Bugfixing, oft sind CSS-Regeln inkonsistent aufgestellt, zum Beispiel existieren noch Regeln, die gar nicht mehr im DOM (Document Object Model) der Seite zu finden sind. Es werden Regeln mehrmals überschrieben und es wird nicht auf Optimierung von Selektoren geachtet.

Seit 2010 berücksichtigt der Page Ranking Algorithmus von Google auch die Ladezeiten für Websites¹. Seiten, die neben SEO², ein gutes Page Ranking erhalten, werden demnach auch durch ihre Ladezeiten bestimmt. Die Ladezeiten spielen zwar im Vergleich mit SEO nur eine kleine Rolle, können aber zu einem besseren Ergebnis beitragen (Weite Informationen: <http://googlewebmastercentral.blogspot.de/2010/04/using-site-speed-in-web-search-ranking.html>).

Ladezeiten von mobilen Websites und Web-Anwendungen bzw. Seiten, die über mobile Internetverbindungen geladen werden, sollten schnell und nur wenig Daten übertragen, um ein konsistentes Benutzererlebnis zu gewährleisten. Statistiken zeigen, dass Benutzer auf Websites eher verbleiben wenn diese schnell geladen werden und der Benutzer schnell Informationen abrufen oder mit der Anwendung interagieren kann.

1.2 Anforderungen

Im Rahmen der Hausarbeit für das Fach Compilerbau im Sommersemester 2013 im Master-Studiengang Informatik - Verteilte und Mobile Anwendungen an der Hochschule Osnabrück / University of Applied Sciences soll ein Werkzeug entwickelt werden mit dem sich Stylesheets optimieren lassen. Das Parsen der CSS-Quellen soll mittels Flex und Yacc, bzw. Bison geschehen. Vgl. siehe ANDREW W. APPEL (1998): Modern Compiler Construction in C.

Weiterhin soll ein Kommandozeilentool entwickelt werden, womit CSS-Optimierungen durchgeführt und die Ergebnisse ausgegeben werden können.

Die Anforderungen an den CSS-Optimierer werden in der Abbildung 1 als Flow-Diagramm beschrieben.

Ausgabeformat Das Ausgabeformat soll ebenfalls als CSS-Datei geschrieben werden.

¹Offizieller Blogpost von Google: <http://googlewebmastercentral.blogspot.de/2010/04/using-site-speed-in-web-search-ranking.html>

²SEO: Search Engine Optimization / Suchmaschinenoptimierung

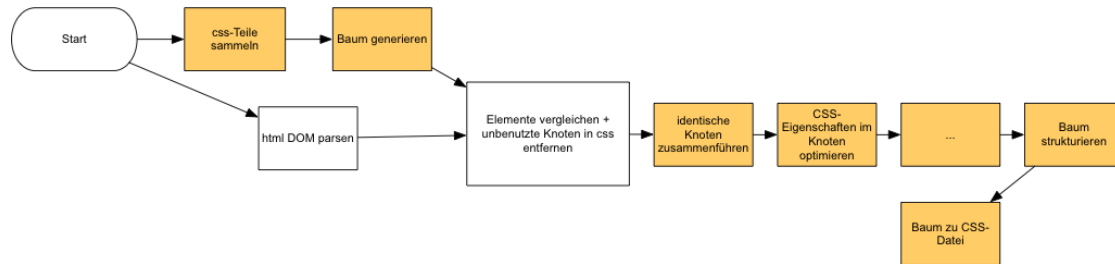


Abbildung 1: Workflow der Anwendung

Ungenutztes CSS entfernen Deklarierte Regeln die in HTML-Dateien nicht eingesetzt werden, sollen entfernt werden, da sie überflüssig sind.

Gleiche CSS-Regeln zusammenführen Selektoren die mehrmals in den CSS-Dateien vorkommen, werden zu einer Regel zusammengeführt, wobei bei gleichen Eigenschaften der neuere den älteren Wert überschreibt (Kaskade).

Shorthands für Hex-Farbwerte Manche Farbwerte lassen sich verkürzt darstellen. So erzeugt der Hex-Farbwert `#fff` ein gleiches Ergebnis wie der (längere) Wert `#ffffff`.

2 Einführung in CSS

2.1 Cascading Stylesheets

Cascading Stylesheets gilt aktuell als Standard-Stylesheetsprache für Websites. Es findet aber auch Verwendung in anderen Umgebungen, wie zum Beispiel JavaFX. Mit CSS ist es möglich Ausgabemedien unterschiedliche Darstellung vorzugeben. Beispielsweise soll einen Hyperlink³ beim Druck einer Seite in einer anderen Farbe dargestellt werden oder die Elemente einer Bildergalerie sollen auf Geräten mit kleinerer Auflösung (wie etwa Tablets oder Smartphones) gelistet werden und bei einem Display mit FullHD-Auflösung können Bilder in einem Gitternetz dargestellt werden.

Verschiedene spezielle CSS-Eigenschaften können für die unterschiedlichen Einsatzgebiete von CSS existieren. Die vorliegende Ausarbeitung bezieht sich auf CSS im Web-Stack, bzw. untersucht nur CSS für Webseiten.

2.2 Revisionen und Probleme

Die Entwicklung von CSS wurde 1993 begonnen und hat nach mehreren Versionen 1995 den Stand der ersten Version erreicht. 1998 wurde die Version zwei vorgestellt. Diese ist aktuell und wurde bis heute allerdings von keinem der Mainstream-Browser vollständig implementiert. Einige Browser setzen große Teile des CSS2, bzw. CSS2.1-Standards korrekt um. Es kann aber zu Problemen bei unterschiedlichen Browsern führen. 2011 wurde die Recommendation für CSS 2.1 veröffentlicht.

Die aktuelle Entwicklung für den dritten Standard von CSS begann bereits 2000. Zu den Neuerungen in dieser Revision gehören vor allem der modulare Aufbau, womit CSS-Eigenschaften nun schrittweise entwickelt werden können. Fast alle Browser implementieren Kernfunktionen des dritten CSS-Standards, die allerdings variieren können.

2.3 Allgemeiner Aufbau

Damit eine Optimierung von bestehendem CSS-Quelltext durchgeführt werden kann, muss zuvor analysiert werden wie ein Browser CSS verarbeitet und wie Kaskadierung im Browser genutzt wird um Style-Informationen zu verknüpfen.

Eine CSS-Datei besteht aus einer Menge von Anweisungen, oder Regeln, die das Layout einzelner Elemente (mit bestimmter ID), Klassen von Elementen oder allgemein gültigen Elementen bestimmen. Regeln, auf die identische Eigenschaften angewendet werden sollen, können durch Kommata getrennt werden.

Der allgemeine Aufbau einer CSS-Regel wird im Listing 1 veranschaulicht.

³Hyperlink: http://www.w3schools.com/html/html_links.asp

Listing 1: Aufbau einer CSS-Regel

```

1 a {
2     background:#000;
3     color: #fff;
4 }
5 #special {
6     border: 1px solid red;
7 }
8
9 .left, top.myClass {
10     border: 1px dotted green;
11 }

```

Im Listing 1 sind drei Regeln gegeben. Die erste Regel wird für ein HTML-Element angewendet. Jedes a-Element (Hyperlinks im Browser) werden mit schwarzen Hintergrund angezeigt und der textliche Inhalt des Elements wird weiß dargestellt. Bei dem zweiten Element wird eine Regel auf ein Element mit der ID **special** angewendet. Dies erhält einen durchgehenden roten Rahmen um den Inhalt des Elements. Die dritte Regel legt eine gepunktete grüne Linie um den Inhalt der Elemente mit der Klasse **left**, sowie der HTML-Elemente top mit der zugewiesenen Klasse **myClass**.

2.4 Grammatik

Die zugrundeliegende Grammatik stammt von der Seite des W3C (Details, siehe: (2003)CSS Grammatik des W3C). Dies entspricht im Wesentlichen der Grammatik für die CSS Version 2.1. Das Listing 2 zeigt einen Auszug der Grammatik des W3C.

Listing 2: Grammatik des W3C

```

1
2 /* ... */
3 ruleset
4 : selector [ ',' S* selector ]*
5   '{' S* declaration? [ ';' S* declaration? ]* '}' S*
6 ;
7 selector
8 : simple_selector [ combinator selector | S+ [ combinator? selector
9   ]? ]?
10 ;
11 simple_selector
12 : element_name [ HASH | class | attrib | pseudo ]*
13   | [ HASH | class | attrib | pseudo ]+
14 ;
15 class
16 : '.' IDENT
17 ;
18 element_name
19 : IDENT | '*'

```



```
20 attrib
21   : '[' S* IDENT S* [ [ '=' | INCLUDES | DASHMATCH ] S*
22     [ IDENT | STRING ] S* ]? ']'
23   ;
24 pseudo
25   : ':' [ IDENT | FUNCTION S* [IDENT S*]? ')' ]
26   ;
27 declaration
28   : property ':' S* expr prio?
29   ;
30 prio
31   : IMPORTANT_SYM S*
32   ;
33 expr
34   : term [ operator? term ]*
35   ;
36
37 /* ... */
```

Für die Optimierung einer CSS-Datei ist diese Grammatik allerdings ungeeignet. Zum einen unterstützt die Grammatik nicht offiziell CSS3-Elemente, wenn auch Teile davon, und zum anderen ist die Aufgabe des Optimierers nicht die Darstellung oder die Interpretation der CSS-Quelldateien, sondern die Verbesserung der definierten Elemente nach definierten Regeln. Dafür wurde eine eigene Grammatik entwickelt. Diese wird im Abschnitt 4.1 vorgestellt.

3 Kommandozeilenwerkzeug

Um Optimierungen durchzuführen wurde im Rahmen der Hausarbeit ein Kommandozeilenwerkzeug erstellt, welches in den folgenden Teilabschnitten vorgestellt wird.

3.1 Makefile und Start des Programms

Zur Kompilierung und Bereitstellung wurde ein Makefile erstellt (Details siehe Listing 3). Mit dem Makefile besteht ebenfalls die Möglichkeit Einzelteile des Projekts zu erstellen. Zum Beispiel kann mittels **make parser** nur der Parser erstellt werden.

Listing 3: Makefile

```
1 #include grammar/makefile.mk
2
3 all: app
4
5 app: lex.yy.c css.tab.c css.tab.h grammar/css_types.h
6     cc grammar/css.tab.c grammar/lex.yy.c grammar/css_types.c main.c
7         cli_parse.c css_merge.c optimizer.c output.c -o optimCSS
8
9 parser: lex.yy.c css.tab.c css.tab.h grammar/css_types.h
10        cc grammar/css.tab.c grammar/css_types.c grammar/lex.yy.c -g -o
11            parser
12
13 css.tab.c css.tab.h: grammar/css.y
14     bison -d grammar/css.y -o grammar/css.tab.c
15
16 lex.yy.c: grammar/css.l css.tab.h
17     flex -o grammar/lex.yy.c grammar/css.l
18
19 clean:
20     rm -f lex.yy.c grammar/css.tab.h grammar/css.tab.c grammar/parser
21         optimCSS
```

Der Start des Programms geschieht über die Konsole unter Angabe von Parametern. Zur Verfügung stehen die folgenden Parameter:

- f **dateiname** Angabe der Quelldatei. Dies muss eine HTML-Datei sein. Aus der angegebenen HTML-Datei werden die genutzten CSS-Dateien extrahiert und für die Optimierung zusammengeführt.
- m Minifizierte Ausgabe.
- s Strukturierte Ausgabe.

Zum Start des Programms ist der Parameter -f zwingend nötig um dem Optimierer CSS-Dateien zur Verfügung zu stellen. Ein Beispiel ist in Listing 4 dargestellt.

Listing 4: Programmstart

```
1
2 ./optimCSS -f index.html
```

3.2 Datenstrukturen

Zum Einlesen der Kommandozeilenparameter und der HTML- bzw. CSS-Dateien wurden Datenstrukturen definiert. Diese sind in Listing 5 dargestellt. Mit Hilfe der Strukturen ist es möglich den Ausgabebetyp, Eingabetyp, Eingabedatei, sowie eine temporär zusammengeführte CSS-Datei zu halten.

Listing 5: Datenstrukturen

```
1
2 enum output_type {STRUCTURED, MINIFIED}; // -s, -m
3 enum input_type {FILE_INPUT, PATH_INPUT}; // -f, -p
4
5 struct input_data {
6     int output_type;
7     int input_type;
8     char* src;
9 };
10
11 struct css_data {
12     char* src_html;
13     char* merged_css;
14 };
```

3.3 Grundaufbau des Kommandozeilenwerkzeugs

Das Kommandozeilenwerkzeug lässt sich in fünf Komponenten unterteilen. In der ersten Komponente werden die Kommandozeilenparameter eingelesen und für die spätere Nutzung in einer Struktur abgelegt. Die folgende Komponente ermittelt aus der übergebenen HTML-Datei alle genutzten CSS-Dateien und führt diese zu einer temporären CSS-Datei zusammen, welche später zur Optimierung verwendet wird. Nachdem nun eine einzelne CSS-Datei zum Optimieren bereitsteht wird diese zunächst, mittels eines durch Flex und Bison erzeugten Parsers, in eine Baumstruktur überführt und so für die eigentliche Optimierung aufbereitet. Die hierzu genutzten Flex- und Bison-Quellen sind in den Listings 6 und 8 dargestellt. Auf dem nun vorliegenden CSS-Baum werden nachfolgend einige Optimierungen durchgeführt, welche in Kapitel 5 detaillierter beschrieben werden. Abschließend wird der optimierte CSS-Baum in Textform als CSS-Datei ausgegeben. Die Ausgabe kann sowohl strukturiert (gut lesbar) als auch minimiert erfolgen. Dies wird durch Kommandozeilenparameter eingestellt.

4 Baumgenerierung

Im folgenden wird die Erstellung des Baumes beschrieben. Dabei wird zum Einen auf die Implementierung der Bison und Lex Dateien eingegangen und zum Anderen wird auf die Abbildung des Baumes und es werden die erstellten CSS Strukturen erläutert.

4.1 Abbildung der CSS Grammatik in Flex

Da sich die Nutzung der Beschreibung der CSS Grammatik durch die W3C für uns als nicht nutzbar erwiesen hat, haben wir uns dafür entschieden diese selber zu entwickeln. Es wurde entschieden, die Beschreibung einfach zu halten. Dies bedeutet, dass bei der Beschreibung nicht alles akribisch interpretiert wird. Interpretation ist auch nicht notwendig. Im folgenden werden die Punkte aufgelistet die unterschieden werden.

- String: kann ein Selektor, Deklarationschlüssel, etc. sein
- Komma
- Doppelpunkt
- Semikolon
- Geschweifteklammer auf
- Geschweifteklammer zu
- Punkt

Diese einfache Unterteilung hat den Vorteil, dass die Beschreibung der CSS Grammatik in Flex simpel erstellt werden kann. Es wird zur Unterscheidung von Selektoren / Deklaration die oben beschriebenen Sonderzeichen benötigt.

Listing 6: Beschreibung der CSS Grammatik in Flex

```

1  %{
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  // include generated bison header
6  #include "css_types.h"
7  #include "css.tab.h"
8
9  %}
10
11 %option noyywrap
12
13 /*
14    patterns can be described here
15 */
16
17 SEL_STRING      ["#" | "." ]*[a-zA-Z0-9#"'( )" "" ". "/" \ - "%" "\ [ \ ]"*"" _ ""
                  = "' ! \" @ \" \" ? \" ] +
```

```
18 RCHEVRON      ">"
19 PLUS          "+"
20
21 %x COMMENT
22
23 %%
24 %{
25 /*
26     rules for pattern recognition
27 */
28 %}
29
30 " ," { return COMMA; }
31 " :" { return COLON; }
32 " ; " { return SEMICOLON; }
33 "{ " { return LBRACE ; }
34 "} " { return RBRACE ; }
35 ". " { return DOT ; }
36
37 [ \t\n\r]      ;    // ignore whitespace
38
39 {SEL_STRING}[ ]{RCHEVRON}[ ]{SEL_STRING} { yylval.sval = strdup(
    yytext); return STRING; }
40 {SEL_STRING}[ ]{PLUS}[ ]{SEL_STRING} { yylval.sval = strdup(yytext);
    return STRING; }
41 {SEL_STRING} {yylval.sval = strdup(yytext); return STRING; }
42
43 .          ;    // ignore everything else
44
45 %%
46
47 /*
48     user code, lexen etc.
49     moved to test.bison
50 */
```

Im Listing 6 ist die Implementierung zu sehen. Dabei werden in den Zeilen 39 bis 41 die verschiedenen CSS Konstellationen beachtet und als String interpretiert. Im Vergleich zur CSS-Grammatik der W3C ist keine detaillierte Unterscheidung möglich, aber auch nicht gewünscht.

4.2 Benötigte Datenstrukturen zur Erstellung des Baums

Bevor die interpretierte CSS-Grammatik auf einen Baum abgebildet werden kann, werden Datenstrukturen benötigt. Diese ermöglichen es eine automatische Baumgenerierung und Verknüpfung zu gewährleisten. Dabei werden für folgende Punkte Strukturen abgebildet:

css_RuleList Beinhaltet die gesamten Regeln und ist der Wurzelknoten des Baums

css_Rule Eine CSS Regel, die Selektoren und ihre Deklarationen beinhalten

css_SelectorList Ist eine Selektorliste, die einzelne Selektoren beinhaltet

css_Selector Beinhaltet nur den Namen. Richtige Zuordnung geschieht in der CSS Rule

css_DeclarationList Ist eine Deklarationsliste, die die einzelnen Deklarationen beinhaltet

css_Declaration Beinhaltet der Deklarationsschlüssel und Deklarationswert. Richtige Zuordnung geschieht in der CSS Rule

Diese Punkte sind dem Listing 7 zu sehen. Dabei sind auch die Funktionen zur Erstellung der einzelnen Elemente zu sehen. Diese werden bei der Baumgenerierung benötigt und werden im folgenden Abschnitt genutzt.

Listing 7: C Strukturen für die Baumgenerierung

```

1  #ifndef CSS_TYPES_H_
2  #define CSS_TYPES_H_
3
4  /**
5   *
6   * Structures for CSS tree
7   *
8   */
9
10 typedef struct css_Selector_ * css_Selector;
11 typedef struct css_Declaration_ * css_Declaration;
12 typedef struct css_SelectorList_ * css_SelectorList;
13 typedef struct css_DeclarationList_ * css_DeclarationList;
14 typedef struct css_Rule_ * css_Rule;
15 typedef struct css_RuleList_ * css_RuleList;
16
17 struct css_Selector_ {
18     char* name;
19 };
20
21 struct css_Declaration_ {
22     char* dec_key;
23     char* dec_val;
24 };
25
26 struct css_SelectorList_ {
27     css_Selector selector;
28     css_SelectorList next;
29 };
30
31 struct css_DeclarationList_ {
32     css_Declaration declaration;
33     css_DeclarationList next;
34 };
35
36 struct css_Rule_ {
37     css_SelectorList selectorList;
38     css_DeclarationList declarationList;

```

```

39 };
40
41 struct css_RuleList_ {
42     css_Rule rule;
43     css_RuleList next;
44 };
45
46 css_Selector create_CSSSelector(char* _name);
47 css_Declaration create_CSSDeclaration(char* _dec_key, char* _dec_val);
48 css_SelectorList create_CSSSelectorList(css_Selector _selector,
49     css_SelectorList _next);
49 css_DeclarationList create_CSSDeclarationList(css_Declaration
50     _declaration, css_DeclarationList _next);
50 css_Rule create_CSSRule(css_SelectorList _selectorList,
51     css_DeclarationList _declarationList);
51 css_RuleList create_CSSRuleList(css_Rule _rule, css_RuleList _next);
52
53 #endif

```

4.3 Baumgenerierung mit Bison

Nachdem die Beschreibung der CSS-Grammatik in Lex und die benötigten Datenstrukturen beschrieben wurden. Wird jetzt die Baumgenerierung mit Bison erläutert. Dabei liefert die Grammatik verschiedene Tokens (Merkmale), welche im Abschnitt 4.1 beschrieben sind. Diese Tokens werden von Bison genutzt, um den Baum zu generieren.

Listing 8: Baumgenerierung mit Bison

```

1 %{
2     #include <stdio.h>
3     #include <stdlib.h>
4     #include <ncurses.h>
5     #include <string.h>
6     #include "css_types.h"
7     #include "css.tab.h"
8     #include "parsecss.h"
9
10    css_RuleList root;
11
12    extern int yyparse();
13    extern FILE *yyin;
14
15    void yyerror(const char *s);
16    #define BUFFER_SIZE 4096
17
18 %}
19
20 // types which are found/returned by flex
21 %union{
22     char* sval;
23     css_Selector aSelector;

```

```

24     css_Declaration aDeclaration;
25     css_SelectorList aSelectorList;
26     css_DeclarationList aDeclarationList;
27     css_Rule aRule;
28     css_RuleList aRuleList;
29 }
30
31
32 // Terminal symbols of our language
33 %token <sval> STRING
34
35 %token LBRACE RBRACE COMMA DOT SEMICOLON COLON
36
37 %type <aRuleList> rulelist css
38 %type <aRule> rule
39 %type <aDeclarationList> declarationlist
40 %type <aDeclaration> declaration
41 %type <aSelectorList> selectorlist
42 %type <aSelector> selector
43
44 %%
45 // grammar section, parsing rules
46
47 css:      rulelist { $$ = root = $1; }
48 ;
49
50 rulelist: rule rulelist { $$ = create_CSSRuleList($1,$2); }
51         | rule { $$ = create_CSSRuleList($1,NULL); }
52 ;
53 rule:     selectorlist LBRACE declarationlist RBRACE { $$ =
54         create_CSSRule($1, $3); }
55 ;
56 selectorlist: selector COMMA selectorlist { $$ =
57         create_CSSSelectorList($1,$3); }
58         | selector { $$ = create_CSSSelectorList($1, NULL); }
59 ;
60 selector: STRING { $$ = create_CSSSelector($1); }
61         | STRING COLON STRING { char buffer[256] = {0}; strcat(buffer,
62         $1); strcat(buffer, ":"); strcat(buffer, $3); $$ =
63         create_CSSSelector(strdup(buffer)); }
64 ;
65 declarationlist: declaration SEMICOLON declarationlist { $$ =
66         create_CSSDeclarationList($1,$3); }
67         | declaration SEMICOLON { $$ = create_CSSDeclarationList($1,
68         NULL); }
69         | declaration { $$ = create_CSSDeclarationList($1, NULL); }
70 ;
71 declaration: STRING COLON STRING { $$ = create_CSSDeclaration($1,
72         $3); }
73         | STRING COLON STRING COMMA STRING COMMA STRING {
74         char buffer[BUFFER_SIZE] = {0}; strcat(buffer, $3); strcat(
75         buffer, ","); strcat(buffer, $5); strcat(buffer, ",");
76         strcat(buffer, $7);
77         $$ = create_CSSDeclaration($1, strdup(buffer)); }

```

```

69     | STRING COLON STRING COMMA STRING COMMA STRING COMMA STRING {
70         char buffer[BUFFER_SIZE] = {0}; strcat(buffer, $3); strcat(
            buffer, ","); strcat(buffer, $5); strcat(buffer, ",");
            strcat(buffer, $7);
71         strcat(buffer, ","); strcat(buffer, $9);
72         $$ = create_CSSDeclaration($1, strdup(buffer)); }
73     ;
74
75 %%
76 // user code section
77
78 css_RuleList parseCSS(char* fileName) {
79     // set inputfile
80     FILE *inFile = fopen(fileName, "r");
81     if(!inFile) {
82         printf("Could not open input file!\n");
83         exit(-1);
84     }
85     yyin = inFile;
86
87     // bison: parse until there is no input anymore
88     do {
89         yyparse();
90     } while(!feof(yyin));
91
92     return root;
93 }
94
95
96 void yyerror(const char *s) {
97     printf("EEK, parse error! Message: %s\n", s);
98     exit(-1);
99 }

```

Im Listing 8 ist die Erstellung des Baums aufgeführt. Im folgenden wird beispielhaft erläutert, wie der Baum generiert wird. In den Zeilen 46 bis 73 zeigt Aufbau des Baums. Als erstes wird eine CSS Regelliste erkannt und erstellt. Dieser nimmt CSS regeln auf. Die CSS Regeln müssen ein Selektor und eine Deklaration beinhalten. Der erstellte Baum ist eine verkettete Liste mit den einzelnen CSS Regeln.

5 Optimierung

Die Optimierung findet auf dem vom Parser erzeugten CSS-Baum statt. Listing 9 zeigt die Grundidee der Optimierung. So werden nacheinander verschiedene Optimierungsmöglichkeiten auf den geparsten Baum angewendet. Zunächst werden dabei Regeln mit selbem Selektor zusammengefügt. Im Anschluss werden doppelte Deklarationen innerhalb einer Regel aufgelöst. Abschließend werden Regeln, welche die selbe Deklarationsliste besitzen zusammengefügt um die Redundanz zu verringern.

Es wäre hier leicht möglich weitere Optimierungsmöglichkeiten einzufügen, sodass die CSS-Datei weiter verbessert werden kann. Innerhalb der einzelnen Optimierungsschritte werden zusätzlich einige Shorthand-Optimierung der CSS-Deklarationen vorgenommen.

Die einzelnen Optimierungen werden nachfolgend detailliert besprochen. Der zugehörige Quellcode kann auf der Dokumentations-CD in der Datei *optimizer.c* gefunden werden.

Listing 9: Optimize-Funktion

```
1
2 css_RuleList optimize(css_RuleList list, char* filename) {
3     // merge nodes with same selector
4     list = mergeNodes(list);
5     list = removeDoubleDeclarations(list);
6     list = mergeDoubleDeclarations(list);
7
8     return list;
9 }
```

5.1 Zusammenfassen von Regeln mit gleichem Selektor

Im ersten Schritt der Optimierung werden zunächst Regeln mit gleichem Selektor zusammengefasst. Hierzu wird jeweils ein Selektor aus dem Baum entnommen und der Rest des CSS-Baums nach diesem Selektor durchsucht. Dies wird für alle Selektoren des Baums durchgeführt. Wird bei der Suche nach einem Selektor eine Regel mit dem selben Selektor gefunden, so werden die beiden zugehörigen CSS-Regeln zu einer neuen Regel zusammengefügt und die bisherigen Regeln aus dem Baum entfernt.

Nachdem die Regeln zusammengefügt wurden, werden in einem weiteren Schritt doppelte Deklarationen innerhalb einer CSS-Regel aufgelöst. Hierbei wird die Kaskadierung von CSS-Dateien beachtet.

5.2 Zusammenfassen von Regeln mit gleicher Deklarationsliste

Zur Zusammenfassung von Regeln mit gleicher Deklarationsliste wird zunächst eine Regel aus der Liste entnommen. Daraufhin wird die Liste mit denen aller nachfolgenden Regeln

verglichen. Werden übereinstimmende Deklarationsliste gefunden, so werden die zugehörigen Selektorlisten zusammengefügt und eine der beiden Regeln aus dem CSS-Baum entfernt. Dies wird für alle Regeln des CSS-Baums durchgeführt um so eine bestmögliche Zusammenfassung der Regeln zu erreichen.

5.3 CSS Shorthands

Zusätzlich zu den zuvor genannten Optimierungen wurden einige CSS-Shorthands implementiert. Zum Einen die Abkürzung von hexadezimalen Farbcodes und zum Anderen das Auslassen von Mengeneinheiten bei 0-Werten. Diese Shorthands werden auf alle im CSS-Baum vorhandenen Deklarationen angewendet um so einen möglichst minimalen CSS-Code zu erhalten.

6 Messungen und Ergebnis

6.1 Grundlage

Als Grundlage zur Bewertung von Optimierungen, soll eine **Single Page Applikation** verwendet werden. Single Page Anwendungen folgen dem Paradigma niemals die gesamte Seite neu zu laden, sondern nur gewünschte Teile der Anwendung zu aktualisieren. Oft werden dabei asynchrone Requests oder auch Web-Sockets verwendet. Dies verbessert Ladezeiten oder auch SEO. Weiterhin wird versucht ohne das Neuladen von Seiten die Web-Anwendung wie eine Desktop-Anwendung darzustellen.

Die Messdaten basieren auf einem frei erhältlichen HTML5-Template, welches von der Seite <http://egrappler.com> heruntergeladen werden kann. Die Abbildung 2 zeigt einen Ausschnitt der Seite.

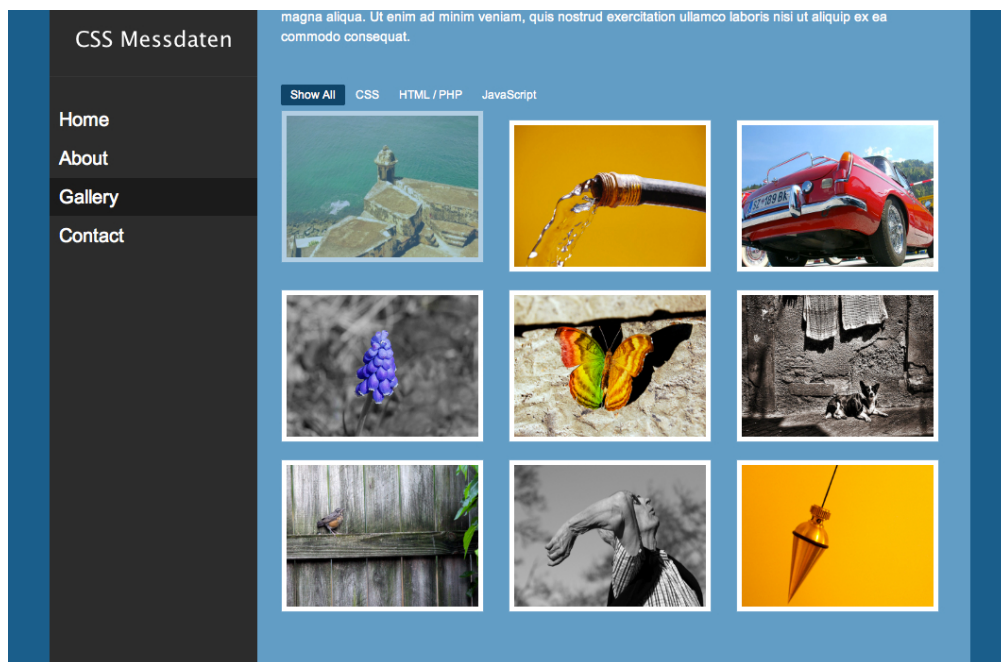


Abbildung 2: Screenshot der Messdaten

Weiterhin wurde ein eigener minimalistischer Webservice, basierend auf *Ruby Rack*⁴ und *WEBrick*⁵, entwickelt. Der Webdienst dient zur Bereitstellung der Testdaten vor und nach der Optimierung. Das Listing 10 zeigt die Implementierung des Dienstes, welcher den aus der Standard-Ruby-Implementierung mitgelieferten Web-Server nutzt und auf Port 80 lauscht.

⁴Ruby Rack: <http://rack.github.io>

⁵WEBrick: <http://www.ruby-doc.org/stdlib-2.0.0/libdoc/webrick/rdoc/index.html>

Listing 10: Ruby Webservice für Messdaten

```

1  #!/usr/bin/env ruby
2
3  # USAGE: ./server.rb path/to/the/root/dir
4
5  require 'rubygems'
6  require 'rack' # rack it up
7
8  serve = Rack::Builder.new do
9    use Rack::Static,
10     :urls => ["/images", "/js", "/css"],
11     :root => ARGV[0],
12     :index => 'index.html'
13    run Rack::File.new(ARGV[0])
14  end
15
16  # by default WEBrick doesn't listen to sigkill, cause the script runs
in a new session.
17  Signal.trap('INT') {
18    Rack::Handler::WEBrick.shutdown
19  }
20
21  # takeoff
22  Rack::Handler::WEBrick.run(serve, :port => 80, 'Content-Type'=>'text/
    html')
```

6.2 Bewertung

Das Ergebnis soll durch folgende Arten gemessen werden:

- Anzahl Dateien
- Dateigröße
- Ladezeit im Browser
- Anzahl Knoten im generierten Baum

Die Tabelle 1 grenzt die Messergebnisse von einander ab. Nachfolgend werden die Optimierungen erläutert.

Art	Ursprung	Optimierung
Anzahl Dateien	1..n Dateien	1 Datei
Größe	ca. 37 KB	ca. 33 KB
Ladezeit	26 ms	6 ms
Anzahl Knoten im Baum	317	302

Tabelle 1: Ergebnis der Messungen











 index.html /Users/olivererxlel	GET	Suc...	text...	Other	0 B 11.0 K	1 ms 1 ms	
 style.css /Users/olivererxlel	GET	Suc...	text...	index.ht... Parser	0 B 11.3 K	1 ms 1 ms	
 flexslider.css /Users/olivererxlel	GET	Suc...	text...	index.ht... Parser	0 B 2.7 KB	5 ms 5 ms	
 prettyPhoto.css /Users/olivererxlel	GET	Suc...	text...	index.ht... Parser	0 B 19.6 K	8 ms 7 ms	
 tipsy.css	GET	Suc...	text...	index.ht...	0 B	8 ms	

Abbildung 3: Ladezeiten ohne Optimierung

 index.html /Users/olivererxlel	GET	Suc...	text...	Other	0 B 10.7 K	1 ms 1 ms	
 output.css /Users/olivererxlel	GET	Suc...	text...	index.ht... Parser	0 B 36.8 K	6 ms 3 ms	

Abbildung 4: Ladezeiten mit Optimierung

Die Anzahl der Dateien wird verringert. Im Beispiel der Messdaten wird aus vier CSS-Dateien eine CSS-Datei zusammengeführt. Dies verringert den Overhead für den Browser und verbessert die Ladezeiten sehr.

Die Abbildungen 3 und 4 zeigen Ausschnitte aus den Entwicklerwerkzeugen von Google Chrome. Mit diesen ist es möglich alle zu ladenden Dateien zu überwachen und mit welcher Geschwindigkeit der Browser diese lädt. Wie die Abbildungen zeigen benötigen die vier nicht optimierten Dateien aus Abbildung 3 länger als die optimierte CSS-Datei.

Im Beispiel werden mehrere Knoten im Baum entfernt. Dies geschieht durch das Zusammenführen identischer Knoten. Außerdem wird die optimierte CSS-Datei beim Schreiben minimiert, sodass Whitespaces und Linebreaks entfernt werden. Dies führt zur Reduktion der Ausgabedatei.

6.3 Fazit

Mittels (F)lex, Yacc/Bison und der verwendeten Grammatik kann ein Baum generiert und dessen Knoten optimiert werden. Die CSS-Dateien können mit dem entwickelten Tool Webseiten beschleunigen. Wie die Testdaten zeigen, ergibt sich eine Beschleunigung von ca. 75 % beim Laden der CSS-Daten.

Die Anwendung benötigt neben Flex und Yacc/Bison keine weiteren Abhängigkeiten, sodass eine gute Plattformunabhängigkeit gewährleistet ist. Der Sourcecode wurde im Rahmen der Hausarbeit unter Ubuntu 10.10, Ubuntu 12.04 mittels gcc und unter Mac OS X mittels clang und gcc erfolgreich kompiliert.

Der vorgestellte CSS-Optimierer lässt sich für statische Websites und für Single Page Web-Apps gut einsetzen. Allerdings kann es zu Problemen kommen, wenn die Web-Anwendung eine Template-Engine einsetzt oder dynamisch CSS-Ressourcen in die Seite lädt. Neue Ressourcen, HTML-Elemente oder auch CSS-Stile können somit nicht optimiert werden.

Auf CSS selbst können noch weitere Optimierungen folgen, die allerdings im Rahmen dieser Hausarbeit nicht durchgeführt wurden.

Neben CSS sollte vor allem auch JavaScript optimiert werden, da so alle Ressourcen einer Web-Anwendung verbessert werden und eine noch größere Reduktion der Ladezeiten entstehen kann.

Literatur

CSS Grammatik des W3C. 2003 (URL: <http://www.w3.org/TR/CSS21/grammar.html>).

ANDREW W. APPEL, MAIA GINSBURG: Modern Compiler Construction in C. Cambridge University Press, 1998.

Virtuelle Maschine

Zur Messung der Testdaten wird eine virtuelle Maschine erstellt. Diese liegt dem Datenträger als VirtualBox-Maschine dieser Hausarbeit bei.

Betriebssystem

Als Betriebssystem dient ein aktuelles Debian Linux mit einem virtuellen Kern, 256 MB Arbeitsspeicher und 8 GB Massenspeicher. Diese Hardwarespezifikation entspricht im wesentlichen eingebetteten Systemen oder Shared Webhosting.

Benutzer- und Logindaten

Für die VM wurde folgende Benutzer mit Passwort angelegt:

- cbh:compiler (Standardbenutzer)
- root:r00t (Superuser)

Dienste und Tools

Auf der virtuellen Maschine sind folgende Dienste und Werkzeuge installiert:

- FTP
- Ruby
- SSH
- Git

Dem FTP- und SSH-Server ist der Standardbenutzer zugewiesen, sodass Dateien auf, bzw. von dem Server geladen werden können.

Ordnerstruktur

Im Homeverzeichnis des Benutzers *cbh* befindet sich das Git-Projekt *compilerconstruction*.