# PROJECT REPORT

## Data Structures and Algorithms

## (UCS406)

# CSS Minifier & SCSS Preprocessor

## By


| | |
|---|---|
| **Nikhil Jain** | **101603206** |
| **Prateek Chhikara** | **101603247** |
| **Prakhar Gupta** | **101610066** |

**Computer Science & Engineering Department**

**Thapar Institute of Engineering and Technology, Patiala**


**May 2018**

# TABLE OF CONTENT

# Section 1

## 1. Problem Formulation

### 1.1 Minifying CSS

It is the process of minimizing code and markup in your web pages and script files. It’s one of the main methods used to reduce load times and bandwidth usage on websites. Minification dramatically improves site speed and accessibility, directly translating into a better user experience. It's also beneficial to users accessing your website through a limited data plan and who would like to save on their bandwidth usage while surfing the web.

### 1.2 SCSS Preprocessor

Sassy Cascading Style Sheet (SCSS) Preprocessor used to convert a SCSS file to CSS file using preprocessing. CSS on its own can be fun, but style sheets are getting larger, more complex, and harder to maintain. This is where a preprocessor can help. Syntactically Awesome Style Sheet (SASS) lets you use features that don't exist in CSS yet like variables, nesting, mixings, inheritance and other nifty goodies that make writing CSS fun again. Once you start tinkering with Sass, it will take your preprocessed Sass file and save it as a normal CSS file that you can use in your website.

# Section 2

## 2. Analysis of Problem

### 2.1 Problem 1: CSS Minifier

Minified source code is especially useful for interpreted languages deployed and transmitted on the Internet, because it reduces the amount of data that needs to be transferred. In programmer culture, aiming at extremely minified source code is the purpose of recreational code golf competitions.

When creating HTML, CSS and JavaScript (JS) files, developers tend to use spacing, comments and well-named variables to make code and markup readable for themselves. It also helps others who might later work on the assets. While this is a plus in the development phase, it becomes a negative when it comes to serving your pages. Web servers and browsers can parse file content without comments and well-structured code, both of which create additional network traffic without providing any functional benefit.

So, the following points must be covered by CSS Minifier.Removes useless white spaces, indentation characters and line breaks.

1. Strips all comments
2. Removes empty CSS declarations
3. Keeps a single charset per CSS file removing all extra declarations
4. Do not differentiate between uppercase and lowercase.
5. Do not replace CSS rules which has more importance.

#### 2.1.1 Overview

When it comes to generating a page or running a script, web browsers aren't concerned about the readability of code. Minification strips a code file of all data that isn't required for the file to be executed. Unlike traditional compression techniques, minified files don't need to be decompressed before they can be read, modified or executed.

Minification is performed after the code for a web application is written, but before the application is deployed. When a user requests a webpage, the minified version is sent instead of the full version, resulting in faster response times and lower bandwidth costs. Minification is used in websites ranging from small personal blogs to multi-million user services.

### 2.1.2 How Minification Works

Minification works by analyzing and rewriting the text-based parts of a website to reduce its overall file size. Minification extends to scripts, style sheets, and other components that the web browser uses to render the site.

Minification is performed on the web server before a response is sent. After minification, the web server uses the minified assets in place of the original assets for faster distribution to users. **Here's a step-by-step description of how minification works:**

1. A web developer creates a JavaScript or CSS file to be used in a web application. These files are formatted for the developer's convenience, which means they make use of whitespace, comments, long variable names, and other practices for readability.
2. The developer applies a minification technique (see below) to convert the file into one that's more optimized, but harder to read. Common minification techniques include removing whitespace, shortening variable names, and replacing verbose functions with shorter, more concise functions.
3. The web server uses the minified file when responding to web requests, resulting in lower bandwidth usage without sacrificing functionality.

The benefit of minification is that it only needs to be performed when the source file changes. When combined with other compression techniques, minification can greatly reduce bandwidth usage for both the enterprise and the user.

## 2.2 Problem 2: SCSS Preprocessor

### 2.2.1 Overview

Sass consists of two syntaxes. The original syntax, called "the indented syntax", uses a syntax like HTML. It uses indentation to separate code blocks and newline characters to separate rules. The newer syntax, "SCSS", uses block formatting like that of CSS. It uses braces to denote code blocks

and semicolons to separate lines within a block. The SCSS files are traditionally given the extensions. SCSS, respectively.

SCSS is a nested meta language, as valid CSS is valid SCSS with the same semantics. Sass supports integration with the Firefox extension Firebug.

SCSS Script provides the following mechanisms: variables, nesting etc.

### 2.2.2 Variables

Variables begin with a dollar sign ($). Variable assignment is done with a colon (:).

Sass Script supports four data types:

- Numbers (including units)
- Strings (with quotes or without)
- Colors (name, or names)
- Booleans

Variables can be arguments to or results from one of several available functions. During translation, the values of the variables are inserted into the output CSS document.

**In SCSS style**

```
$primary-color: #3bbfce;

$margin: 16px;

.content-navigation {

  Margin-left: $margin;

  color: $primary-color;

}
```

**Would compile to:**

```
.content-navigation {

Margin-left: 16px;

  color: #3bbfce;

}
```

### 2.2.3 Nesting

CSS does support logical nesting, but the code blocks themselves are not nested. Sass allows the nested code to be inserted within each other.

**In SCSS style-**

```
table.hl {
  margin: 2em 0;
  td.ln {
    text-align: right;
  }
}
```

**Would compile to:**

```
table.hl {
  margin: 2em 0;
}
table.hl td.ln {
  text-align: right;
}
```

# Section 3

## 3. Data Structure and Algorithmic Technique

### 3.1 Data Structures Used

Following Data Structures are used-

1. Binary Search Tree(BST)
    - BST is used for selection of class, id, simple tags etc.
2. Linked List
    - Linked List is used to store declarations which consists of properties and values of the class.
3. Stack as a Linked List
    - Statements such as *@import:url("abc.txt")* are encountered then we need to store them separately, that's why we are using stack as a Linked List. Preprocessor also uses 2 stacks, one for storing selector (Doubly LL) and other for decleation(Singly LL)
4. HashMap<String,String> map=new HashMap<>()
    - It's used to store the variable as key and their corresponding value as value. If we are in state of storing variable, say $var1:12px; then key is $var1 and value is 12px. If key is encountered in declaration, then it is replaced by its corresponding value. HashMap is used because it has O (1) complexity for both set() and get() function.

### 3.2 Algorithmic Techniques

BST consists of following fields-

Selector Name (i.e. class/id) and address of the head of the Linked List. The head of the linked list stores the declaration of this selector (i.e. either left or right node of the root).

When a node is encountered then it is compared lexicographically and stored according to properties of BST. However, we also need to find that if selected already exists, then new selector with same name should not be created, if that selected already exists then append new declarations to linked list whose head is stored in this BST node.

# Section 4

# 4. Methodology

## 4.1 Insertion Function

```
1. BSTNode insert(BSTNode root1, String name, StyleNode d)
2.     if(root1 == null)      //If root is null
3.         root1 ←new BSTNode(name)
4.         root1.data←d
5.      else if(((root1.name).compareTo(name))=0)   // if class found
6.         if(root1.data == null)
7.             root1.data ← d
8.         else
9.             StyleNode ptr ← root1.data       //else create new BSTnode
10.            StyleNode prev ← null
11.            while(ptr!=null)
12.                if(((ptr.property).compareTo(d.property))=0)   //if style property found
13.                    Prev← ptr
14.                    Ptr← ptr.next
15.    if(ptr == null)
16.        prev.next ← d
17.    else
18.        if(d.isimp || (!d.isimp && !ptr.isimp))   //Check importance and then override
19.            ptr.value← d.value;
20.            if(d.isimp)
21.                ptr.isimp = true
22.        else if(((root1.name).compareTo(name))>0)
23.            root1.left←insert(root1.left,name,d)
24.        else
25.            root1.right←insert(root1.right,name,d)
26.   return root1
```

Time Complexity= O(n+logN) = O(n)

Space Complexity = O(n + N)

## 4.2 Push Function in Stack of Minifier

1. **int push(item){**
2. if stack is full
3. return null
4. else
5. top←top+1
6. stack[top]← item}

Time Complexity= O (1)

Space Complexity = O(n)

## 4.3 Pop Function in Stack of Minifier

1. void **pop()**{
2. if stack is empty
3. return null
4. else
5. item← stack[top]
6. top←top-1
7. }

Time Complexity= O (1)

Space Complexity = O(n)

## 4.4 Display Function

1. **void displayLL1( head)**
2. while(head != null)
3. if(head.isimp)
4. print(head.property+":"+head.value+"!important"+";")
5. else
6. print(head.property+":"+head.value+";")
7. Head← head.next

Time Complexity= O (n) & Space Complexity = O (1)

## 4.5 Inorder Traversal Function

```
1. void Inorder(root)
2.     if(root=NULL)
3.         return
4.     else
5.         Inorder(root→left)
6.      Print classname &  DisplayLL(root→data)
7.         Inorder(root→right)
```

Time Complexity= O (n)

Space Complexity = O(n)

## 4.6 Hash Function for Variables

**String variable(String str)**

```
1       int i<-0
2       int state<-1
3       String part1<-part2<-""
4       HashMap<String, String> map <- new HashMap<>()
5       while(i<str.length())
6               char ch<-str.charAt(i)
7               if(ch='{')    // state =2 is state in which we replace key by value
8                       state<-2
9               if(ch='}')  //state =1 is state in which we add key value pair to hashmap
10                      state<-1
11              if(ch='$')
12                      if(state=1)
13                              part1<-""
14                              while(i<str.length()&&ch!<-';')
15                                      part1+<-ch
```

```
16                                  ch<-str.charAt(++i)
17                          String [] arrOfStr <- part1.split(":", 2)
18                          map.put(arrOfStr[0].trim(),arrOfStr[1].trim())
19                  if(state=2)
20                          part2<-""
21                          int end3 <- 0
22                          int start3 <- i
23                          while(i<str.length()&&ch!<-")
24                                  part2+<-ch
25                                  ch<-str.charAt(++i)
26                                  end3 <- I
27                           part2<-part2.trim()
28                          if (map.containsKey(part2))
29                                  String a <- map.get(part2)
31                                  str <- str.substring(0,start3)+ a+ str.substring(end3)
34              else
35                          i++
36      str <- str.replaceAll("\\$.*?","")
37      return str
```

Time Complexity= O (1) & Space Complexity = O(n)

## 4.7 Push and Pop Functions

1. **void popclass()**
2. if(classtop!=null)
3. printf()
4. className ptr←classbottom

```
5.          while(ptr!=null)
6.              printf(ptr.name+" ")
7.              ptr ←ptr.next
8.               classtop = classtop.prev
9.      if(classtop!=null)
10.     classtop.next←null
```

Time Complexity= O (1) & Space Complexity = O(n)

```
1. void popclass()
2.              if(classtop!<-null)
3.                      System.out.println()
4.                      className ptr <- classbottom
5.                      while(ptr!<-null)
6.                              System.out.print(ptr.name+" ")
7.                              ptr <- ptr.next
8.                      classtop <- classtop.prev
9.                      if(classtop!<-null)
10.                     classtop.next<-null
```

Time Complexity= O (n) & Space Complexity = O(n)

**void pushprop(String s,char c)**

```
1.              Node temp <- new Node(s,c)
2.              if(Nodetop = null)
3.                      Nodetop <- temp
4.              else
5.                      temp.next <- Nodetop
6.                      Nodetop <- temp
```

```
1. void popprop()
2.      if(Nodetop = null)
3.                      return
4.      else
5.              Node ptr<- Nodetop
```

```
6.                  System.out.print("\n"+ptr.prop+"")
7.                  Nodetop <- Nodetop.next
```

## 4.8 Nesting Function

```
1. void nesting(String path)
2.     String s<- Content of file(path)
3.             s <- variable(s)  // call variable function and preprocess all variables
4.             s <- s.replaceAll("\\/\\*.*?\\*\\/", "")    //replace comments
5.             s <- s.replaceAll("\r\n  +\t", ""). trim().toLowerCase()\\ replace newline and
   extra space and convert to lowercase
6.             int i<-0
7.             String Parentname <- null
8.             int start<-0,end <-0,end1<-0,child1<-0,count <- 0,start1<-0,flag <- 0
9.             while(i<s.length())
10.                if(s.charAt(i)='{' && flag = 0)
11.                        pushprop(null,")
12.                        end<- i-1
13.                        Parentname <- s.substring(start,end+1)
14.                        Parentname <- Parentname.trim()
15.                        pushclass(Parentname)
16.                        flag <- 1
17.                        start1 <- i+1
18.                else if(s.charAt(i) = ';')
19.                        end1<-i-1
20.                        String prop <- s.substring(start1,end1+1)
21.                        prop <- prop.trim()
22.                        pushprop(prop,'\0')
23.                        start1 <- i+1
24.                else if(s.charAt(i)='{' && flag = 1)
25.                        pushprop(null,")
26.                        end1<- i-1
27.                        String childname <- s.substring(start1,end1+1)
28.                        childname <- childname.trim().replaceAll("\r\n  +\t", "")
```

```
                                    start1 <- i+1
                            count++
                        else if(s.charAt(i)='}' &&count !<-0)
                          start1 <- i+1
                            count--
                            popclass()
                            System.out.print("")
                            while(Nodetop.ch !<- '')
                                        popprop()
                            System.out.print("\n")
                                    Nodetop <- Nodetop.next
                                    System.out.println()
                        else if(s.charAt(i)='}' &&count =0)
                            start <- i+1
                            int check <-0
                            if(Nodetop.next!<-null)
                                    check <-1
                                    System.out.print("\n")
                                    popclass()
                                    System.out.print("")
                                    while(Nodetop.ch !<- '')
                                                popprop()
                                    System.out.println("\n")
                                    Nodetop <- Nodetop.next
                                    if(check = 0)
                                    classtop <- classtop.prev
                                            if(classtop!<-null)
                                            classtop.next<-null
                            flag <-0
                            i++
```

Time Complexity= O (N+n) & Space Complexity = O(N + n)

# Section 5

## 5. Test Cases

### 5.1 CSS Minifier

Input-

```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-

        CSS Minimizer & SCSS Preprocessor

1-Preprocessor
2-Minimizer
3-Exit

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-

Enter your choice  2


-----------------------------------------------------
Input File is :

@import "e:/qwdd.ceda.dd/wdsxcx.r";
a,h1,h2,h3{
/*font: times new roman; */
background : blue;
}

.cl1    /*      .cl3 :hover{
    font-size:12px  ;

    COLOR:red
} */

.cl7+.cl9>div #div{
    font-family:google fonts;
    color: #123421;
        COLOR: #333;
}

@import "f:/qwdd.ceda.dd/wdsxcx.r";
.cl2{
    display: none
}

a:hover li{
    text-decoration:none;
            background:none ;
    baclground:url("data:12%+122.txt");
}

a:hover li{
    background:black !important;
    background:123 ;
}
```

*Figure 5.1 (a) Input to CSS Minifier*

Output-

```
Minimized  file is


@import "f:/qwdd.ceda.dd/wdsxcx.r"@import "e:/qwdd.ceda.dd/wdsxcx.r".cl1 .cl7+.cl9>div #div{font-family:
googlefonts;color:#333;}.cl2{display:none;}a,h1,h2,h3{background:blue;}a:hover li{text-decoration:none;b
ackground:black!important;baclground:url("data:12%+122.txt");}
```

*Figure 5.1 (b). Output of CSS Minifier*

## 5.2 SCSS Preprocessor

Input-

```
        CSS Minimizer & SCSS Preprocessor

1-Preprocessor
2-Minimizer
3-Exit

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-

Enter your choice  1


-----------------------------------------------------
Input File is :
$prakhar: 13px;
$p : red;
nav {
  ul {
    margin: 0;
    padding: 0;
    list-style: none;
  }

  li { display: inline-block; }

  a {
    display: block;
    font-size:$prakhar   ;
        color:$p;
    /* text-decoration: none; */
        b{
                text:hello;
        }
  }
}
nik
{
color: black;
c{font:blue;
        d
        {
         background:blue;
        }
}
```

*Figure 5.2 (a). Input to SCSS Preprocessor*

Output-

```
------------------------------------------------------------
Preprocessed  file is

nav ul {
list-style: none;
padding: 0;
margin: 0;
}

nav li {
display: inline-block;
}

nav a b {
text:hello;
}

nav a {
color:red;
font-size:13px;
display: block;
}


nik c d {
background:blue;
}

nik c {
font:blue;
}


nik {
color: black;
}
```

*Figure 5.2 (b). Output of SCSS Preprocessor*

# Section 6

# 6. Results and Conclusion

## 6.1 CSS Minifier

### 6.1.1 Benefits of Minification

1. Users load content faster as less unnecessary data needs to be downloaded. Users experience identical service without the additional overhead.
2. Businesses see lower bandwidth costs as less data is transmitted over the network. The extra content that only developers care about is no longer being sent to users.
3. Businesses also see lower resource usage since less data needs to be processed for each request. The minified content – which only needs to be generated once – can be used for an unlimited number of requests.

### 6.1.2 Conclusion

Minification is a fast and easy way to reduce a web application's resource usage. Even with standard compression techniques, minification can improve the time needed to render a page by over 60%. Minifying your website can lead to large performance gains without compromising your users' experience. Time complexity of our minifier is O(log N +n) where N is number of seectors and n is number of style rules/decorations. Space complexity is O(N+n)

## 6.2 SCSS Complier

### 6.2.1 Conclusion

Once this software is installed, you can compile your SCSS to CSS using the SCSS command. You'll need to tell SCSS which file to built from, and where to output CSS to. For example, running SCSS input.scss output.css from your terminal would take a single SCSS file, input.scss and compile that file to output.css. Time complexity of SCSS compiler of our project is O(N + n) and space complexity is O(N+n) , ) where N is number of seectors and n is number of style rules/decorations.

# Section 7

# 7. References


1. https://www.incapsula.com/cdn-guide/glossary/minification.html
2. https://cssminifier.com/
3. https://blog.stackpath.com/glossary/minification/
4. https://sass-lang.com/guide
5. https://docs.oracle.com/javase/6/docs/api/java/util/regex/Pattern.html
6. https://www.geeksforgeeks.org/java-util-hashmap-in-java/
7. https://www.javatpoint.com/java-string-split
8. https://www.tutorialspoint.com/java/io/java_io_filereader.htm