

Web-Informationssysteme, WS 2009/10

Seminar-Übung 2: Information-Retrieval, CSS, Extraktion von Web-Tabellen

Besprechung am Mi 04.11.2009

— SÜ-2.1 —

Was Ihr wissen müßt über “Information Retrieval (Vektorraum-Modell)”

Wiederholung

- Was ist die **Aufgabe** einer (klassischen) Suchmaschine?

Gegeben eine Anfrage sollen diejenigen Dokumente gefunden werden, die für diese Anfrage relevant sind. Da bei vagen, keyword-basierten Anfragen eine genaue Antwort meist nicht möglich ist, wird eine nach absteigender Ähnlichkeit zur Anfrage geordnete (*ranked*) Liste von möglicherweise relevanten Dokumenten geliefert.

- Wie wird die **Qualität** von Suchmaschinen gemessen?

Seien D die Dokumente in der Dokumentensammlung, $Q_S(D)$ die von der Suchmaschine als Antwort auf Q zurückgegebenen Dokumente. Desweiteren nehmen wir an, dass $r_Q(D)$ die Dokumente in der Dokumentensammlung, die relevant für Q sind (z.B. von Hand ermittelt). Mit anderen Worten $r_Q(D)$ sind die Dokumente, die eine “perfekte” Suchmaschine auf Q zurückgibt.

Dann ist

$$\text{Precision}_S(Q) = \frac{|Q_S(D) \cap r_Q(D)|}{|Q_S(D)|} \quad \text{Recall}_S(Q) = \frac{|Q_S(D) \cap r_Q(D)|}{|r_Q(D)|}$$

Precision ist als der Anteil der “richtigen” unter allen Antworten, *Recall* der Anteil der “richtigen” Antworten unter allen relevanten Dokumenten. Im Idealfall gibt eine Suchmaschine genau alle relevanten Dokumente zurück, also $Q_S(D) = r_Q(D)$ und damit Precision und Recall 1.

- Warum ist dieses Qualitätsmaß für das **Web** zumindest problematisch.

Precision ist auch für Web-Suchmaschinen ein sinnvolles Maß. Aber die Messung von Recall ist allenfalls stichprobenartig möglich und hängt oft sehr stark von der Domäne der Suche ab.

- Was ist die Grundidee des **Vektorraum-Modells**?

Jedes Dokument und jede Anfrage werden durch einen Vektor über einem Feature-Raum mit einer Dimension pro Term (Feature). Ähnlichkeit zwischen Dokumenten/Anfragen wird dann zurückgeführt auf Ähnlichkeit zwischen Vektoren, z.B. Winkel (Cosinus-Ähnlichkeit).

- Ihr solltet **binäre, einfache, normalisierte und TF/IDF** Varianten des Vektorraummodells anwenden können.

Im folgenden sei N die Anzahl an Dokumenten in der Dokumentensammlung, T ein Term, D ein Dokument, $DF(T)$ die Anzahl der Dokumente, in denen T vorkommt, $RTF(T, D)$ die Anzahl der Vorkommen von T in D , $DL(D)$ die Länge des Dokuments D (Summe der $RTF(T, D)$ über alle T).

1. **Binäre Term-Frequenz:** Eine Zelle m_{ij} der binären Term-Dokument Matrix ist 1 genau dann, wenn Term i in Dokument j vorkommt.
2. **Einfache Term-Frequenz:** Eine Zelle m_{ij} der einfachen Term-Dokument Matrix ist $RTF(i, j)$, also die einfache Term-Frequenz von i in j (die Anzahl der Vorkommen von i in j).
3. **Normalisierte Term-Frequenz:** Eine Zelle m_{ij} der normalisierten Term-Dokument Matrix ist $TF(i, j) = \frac{RTF(i, j)}{DL(j)}$.
4. **TF/IDF:** Eine Zelle m_{ij} der TF/IDF Term-Dokument Matrix ist $TF(i, j) \cdot IDF(j)$ wobei $IDF(i)$ die logarithmisch-gedämpfte *inverse document frequency* sei, also $IDF(i) = \log \frac{N}{DF(i)}$.

- Ihr solltet zumindest das **Cosinus-Ähnlichkeitsmaß** kennen und abschätzen können.

$$\text{sim}(Q, D_j) = \frac{Q \cdot D}{\|Q\| \cdot \|D\|} = \frac{\sum_i (q_i m_{ij})}{\sqrt{\sum_i q_i^2} \sqrt{\sum_i m_{ij}^2}}$$

— SÜ-2.2 —

Auswertung von CSS Selektoren

Programmierung

Implementieren Sie einen einfachen Auswerter für CSS Selektoren. Gegeben ein CSS Selektor und ein HTML (oder XML) Dokument soll der Auswerter alle Elemente des Dokuments ausgeben, die mit dem Selektor matchen.

Jedes Element soll identifiziert werden durch seine *hierarchische* Position, z.B. 1.2.14.7 identifiziert das 7. Kind des 14. Kinds des 2. Kinds des 1. Kinds der (virtuellen) Dokumentwurzel, also des Knotens vom Typ `org.w3c.dom.Document` in der gegebenen DOM Repräsentation. Dazu soll das jeweilige Label des Elements ausgegeben werden.

Beispielsweise auf der Webseite der Vorlesung sollte der CSS Selektor `table.blaetter td > a` beispielsweise die folgende Ausgabe liefern:

```

CSS Engine with input wis-index.html for selector table.blaetter td > a!
2  table.blaetter td > a

4  =====
6  There are 29 total answers for table.blaetter td > a on wis-index.html:
8  D.1.2.30.2.3.1--a: Lageplan, Erdgeschoss
   D.1.2.35.9.2.1--a: Übungsblatt 08 (.pdf)
10 D.1.2.35.12.2.1--a: Übungsblatt 10 (.pdf)
   D.1.2.42.6.2.1--a: Themenblatt 04 (.pdf)
12 D.1.2.42.3.2.1--a: Themenblatt 01 (.pdf)
   D.1.2.30.3.3.1--a: Lageplan, Erdgeschoss
   D.1.2.42.9.2.1--a: Themenblatt 07 (.pdf)
   D.1.2.35.10.2.1--a: Übungsblatt 09 (.pdf)
14 D.1.2.35.8.2.1--a: Übungsblatt 07 (.pdf)
   D.1.2.35.15.2.1--a: Wiederholungsblatt 02 (.pdf)
16 D.1.2.42.10.2.1--a: Themenblatt 08 (.pdf)
   ...

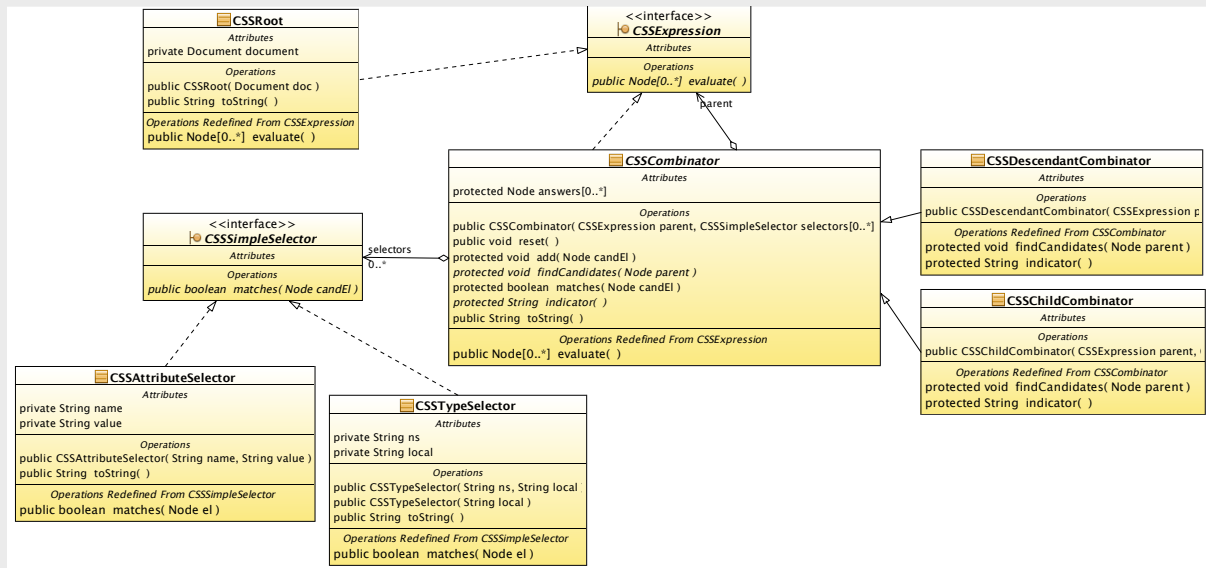
```

Der Auswerter muß nicht alle CSS3 Selektoren unterstützen. Er sollte aber zumindest die folgenden einfachen Selektoren sowie die Kombinatoren *descendant* (" ") und *child* (">") unterstützen.

- Universeller Selektor `*`.
- Typ-Selektoren wie `h1` (ohne Namensraum-Behandlung).
- ID-Selektor (wie `#nav`).
- Class-Selektor (wie `.menu`).

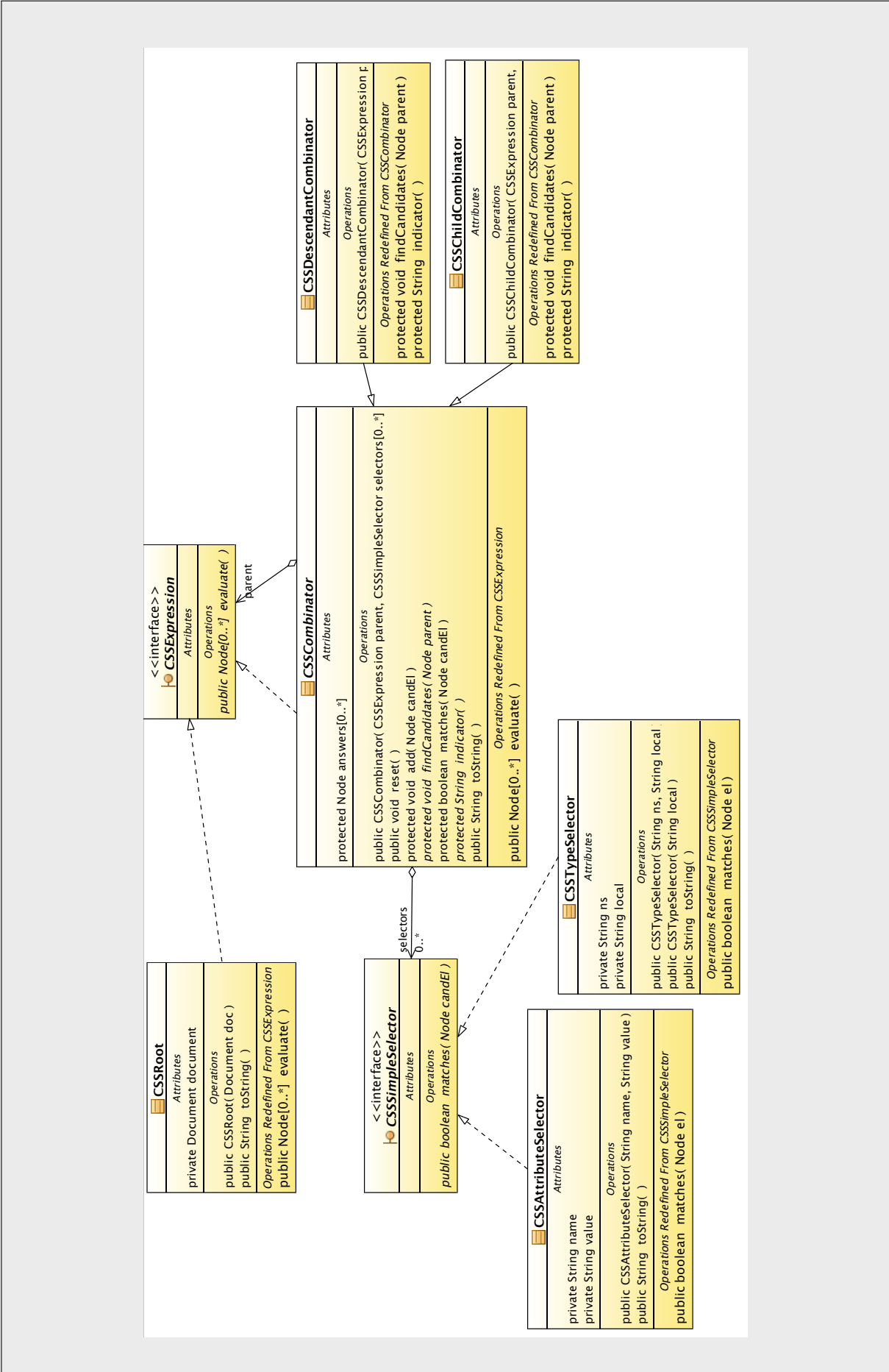
Wir verzichten also auf Pseudo-Klassen, Pseudo-Elemente, und Namensräume. Es ist auch nicht nötig Abkürzungen wie `#nav` für `*#nav` zu realisieren.

Hinweis: Wenn Ihr nicht alleine zu recht kommt (und nur dann): Als Hilfestellung findet Ihr in den Anlagen ([t02CLIDriver.java](#), [t02/css](#) Ordner) das Skelett eines Lösungsvorschlages. Das folgende UML-Diagramm zeigt den groben Aufbau des Lösungsvorschlages:



- Die Implementierung ist so aufgebaut, dass ein CSSCombinator ausgehend von den Antworten für einen übergeordneten Teil-Ausdruck (parent im Konstruktor) mögliche Antworten für seinen Kombinator-Typ (z.B. alle Kinder von solchen Antworten) berechnet und für jeden prüft, ob die zugeordneten *simple selectors* (CSSSimpleSelector) alle erfüllt sind.
- Der Parser ist etwas fragil. In der Praxis ist CSS bereits komplex genug, um den Einsatz eines Parser-generators wie ANTLR oder JavaCC zu rechtfertigen. Für den Lösungsvorschlag haben wir Euch davor bewahren wollen.

In den Anlagen-Dateien sind noch zu implementierende Stellen deutlich mit @TODO markiert.



CSSDescendantCombinator

```
2 package t02.css;

4 import java.util.LinkedList;
  import java.util.List;
6 import org.w3c.dom.Node;
  import org.w3c.dom.NodeList;

8
10 /**
   * The descendant combinator selects all descendants of an answer of the parent
   * expression.
12  * @author Tim Furche
   */
14 public class CSSDescendantCombinator extends CSSCombinator {

16
17     public CSSDescendantCombinator(CSSExpression parent, List<CSSSimpleSelector> selectors) {
18         super(parent, selectors);
19     }

20
21     @Override
22     protected void findCandidates(Node parent) {
23         NodeList nl = parent.getChildNodes();
24         for (int i = 0; i < nl.getLength(); i++) {
25             if(nl.item(i).getNodeType() == Node.ELEMENT_NODE) {
26                 this.add(nl.item(i));
27                 this.findCandidates(nl.item(i));
28             }
29         }
30     }

31
32     protected String indicator() { return " "; }
33
34 }
```

CSSCombinator

```

2 package t02.css;

4 import java.util.HashSet;
  import java.util.List;
6 import java.util.Set;
  import org.w3c.dom.Node;

8

10 /**
   * In CSS a combinator, such as ">" or "+", connects simple selectors. In effect
   * combinators are like XPath axes and advance the context from one node set to
12 * another. This class is abstract and must be extended to provide the functionality
   * of each specific combinator.
14 */
  public abstract class CSSCombinator implements CSSExpression {

16

18     /**
       * The list of simple selectors used to filter candidate answers produced
20     * by specific combinator.
       */
22     private List<CSSSimpleSelector> selectors;

24     /**
       * The parent expression in the CSS syntax tree. Usually again a combinator
26     * except for the top-level case where it is a {@link CSSRoot} expression.
       */
28     private CSSExpression parent;

30     /**
       * The set of answer nodes.
32     */
  protected Set<Node> answers;

34

36     /**
       * The constructor for combinators takes the parent expression and the list of
       * filters.
38     */
  public CSSCombinator(CSSExpression parent, List<CSSSimpleSelector> selectors) {
40     this.parent = parent; this.selectors = selectors;
  }

42

44     /**
       * Resets the answer buffer. Currently not used.
       */
46     public void reset() {
        this.answers = null;
48     }

50     /**
       * Returns the set of nodes that (a) are reachable by the specific type of
52     * combinator from any answer of the parent expression and (b) pass all
       * simple selectors.
54     */
  public Set<Node> evaluate() {

```

```

56     if (answers != null) return answers;
    answers = new HashSet<Node>();
58     for(Node parEl : parent.evaluate()) {
        this.findCandidates(parEl);
60     }

62     return answers;
    }

64
    /**
66     * Adds an answer to the answer buffer. First checks if the answer has already
    * been computed, then checks all filters.
68     */
    protected void add(Node candEl) {
70         if(this.answers.contains(candEl)) return;
        if(this.matches(candEl))
72             this.answers.add(candEl);
    }

74
    /**
76     * Abstract function for finding the candidate answers for a given answer.
    * Depends on the kind of combinator and must be overridden by the specific
78     * combinators extending this class.
    */
80    protected abstract void findCandidates(Node parent);

82
    /**
    * Checks if a node matches all simple selectors associated with the current combinator.
84     */
    protected boolean matches(Node candEl) {
86         for(CSSSimpleSelector sel : selectors) {
            if(!sel.matches(candEl)) return false;
88         }
        return true;
90     }

92    protected abstract String indicator();

94    @Override
    public String toString() {
96        StringBuilder sb = new StringBuilder();
        sb.append(parent.toString());
98        sb.append(this.indicator());
        for(CSSSimpleSelector sel : selectors) {
100            sb.append(sel.toString());
        }
102        return sb.toString();
    }

104
106 }

```


CSSChildCombinator

```
2 package t02.css;

4 import java.util.List;
  import org.w3c.dom.Node;
6 import org.w3c.dom.NodeList;

8 /**
   * The child combinator selects all children of each answer of the parent expression.
10  * @author Tim Furché
   */
12 public class CSSChildCombinator extends CSSCombinator {

14     public CSSChildCombinator(CSSExpression parent, List<CSSSimpleSelector> selectors) {
16         super(parent, selectors);
17     }

18     @Override
20     protected void findCandidates(Node parent) {
21         NodeList nl = parent.getChildNodes();
22         for (int i = 0; i < nl.getLength(); i++) {
23             if(nl.item(i).getNodeType() == Node.ELEMENT_NODE) {
24                 this.add(nl.item(i));
25             }
26         }
27     }

28     protected String indicator() { return "> "; }

30 }

32 }
```

CSSTypeSelector

```
1
package t02.css;

3
import org.w3c.dom.Node;

5
/**
7  * A type selector tests that the local name and/or namespace of the given
8  * element matches with its specification. Also implements the universal selector
9  * (for fun).
10  *
11  * @author Tim Furche
12  */
13 public class CSSTypeSelector implements CSSSimpleSelector {

15     private String ns;
16     private String local;
17
18
19     public CSSTypeSelector(String ns, String local) {
20         this.ns = ns; this.local = local;
21     }

22
23     public CSSTypeSelector(String local){
24         this("", local);
25     }

26
27     public boolean matches(Node el) {
28         if (el.getNodeType() != Node.ELEMENT_NODE) return false;
29         if (!ns.equals("")) && !el.getPrefix().equals(ns) return false;
30         if (!local.equals("")) && !el.getNodeName().equals(local) return false;
31         return true;
32     }

33
34     @Override
35     public String toString() {
36         if (this.ns.equals("")) return this.local;
37         else return this.ns + "|" + this.local;
38     }
39
40
41 }
```

CSSParser

```

2 package t02.css;

4 import java.util.LinkedList;
  import java.util.List;

6
8 /**
9  * Parses a string into a CSS expression.
10  * @author Tim Furche
11  */
12 public class CSSParser {
13
14     /**
15      * Enumeration class for selectors.
16      */
17     enum Selector {
18         TYPE (-1) {
19             public CSSSimpleSelector getSelector(String value) {
20                 return new CSSTypeSelector(value);
21             }
22             @Override
23             public int findKey(String where, int start, int stop){
24                 return -1;
25             }
26         },
27         CLASS ('.') {
28             public CSSSimpleSelector getSelector(String value) {
29                 return new CSSAttributeSelector("class", value);
30             }
31         },
32         ID ('#') {
33             public CSSSimpleSelector getSelector(String value) {
34                 return new CSSAttributeSelector("id", value);
35             }
36         };
37
38         private final int key;
39
40         Selector(int key) {
41             this.key = key;
42         }
43
44         public int key() { return key; }
45
46         public int findKey(String where, int start, int stop){
47             int pos = where.indexOf(this.key, start);
48             return (pos < stop) ? pos : -1;
49         }
50
51         abstract public CSSSimpleSelector getSelector(String value);
52     }
53
54     /**
55      * Enumeration class for combinators.

```

```

56  */
enum Combinator {
58  CHILD  ('>') {
    public CSSExpression getExpression(CSSExpression parent, List<CSSSimpleSelector>
        selectors) {
60      return new CSSChildCombinator(parent, selectors);
    }
62  },
  DESCENDANT  (' ') {
64      public CSSExpression getExpression(CSSExpression parent, List<CSSSimpleSelector>
        selectors) {
        return new CSSDescendantCombinator(parent, selectors);
66      }
    @Override
68      public int findKey(String where, int start){
        int pos = where.indexOf(this.key, start);
70      if (pos == -1 || pos == start) return -1;
        if (where.substring(start, pos).trim().equals("")) return -1;
72      while (where.charAt(pos + 1) == ' ') pos++;
        if (where.charAt(pos + 1) == CHILD.key || where.charAt(pos + 1) == ADJACENT.key
74          || where.charAt(pos + 1) == SIBLING.key) return -1;
        return pos;
76      }
    },
78  ADJACENT  ('+') {
    public CSSExpression getExpression(CSSExpression parent, List<CSSSimpleSelector>
        selectors) {
80      return new CSSChildCombinator(parent, selectors); //TODO
    }
82  },
  SIBLING  ('~') {
84      public CSSExpression getExpression(CSSExpression parent, List<CSSSimpleSelector>
        selectors) {
        return new CSSChildCombinator(parent, selectors); //TODO
86      }
    };

    protected final int key;

    Combinator(int key) {
92      this.key = key;
    }

    public int key()  { return key; }

96      public int findKey(String where, int start){ return where.indexOf(this.key, start);}

98      abstract public CSSExpression getExpression(CSSExpression parent, List<CSSSimpleSelector>
        selectors);
100  }

102  /**
104   * Parses combinator sections, i.e. sequences of simple selectors only.
    */
106  private static List<CSSSimpleSelector> parseSelectors(String input, int start, int stop) {

```

```

List<CSSSimpleSelector> results = new LinkedList<CSSSimpleSelector>();
Selector last = Selector.TYPE;
while( last != null && start < stop) {
    // Find the next selector
    Selector next = null; int nextPos = stop;
    for (Selector s: Selector.values()) {
        int newPos = s.findKey(input, start, stop);
        if(newPos != -1 && newPos < nextPos) {
            next = s; nextPos = newPos;
        }
    }
    // Everything from start to nextPos is now a character sequence
    results.add(last.getSelector(input.substring(start, nextPos).trim()));

    // For the next round we "advance" the combinator and the start position
    last = next; start = nextPos + 1;
}

return results;
}

/**
 * The actual (static) parser that slices up a string into combinator sections
 * and then parses those with {@link #parseSelectors(java.lang.String, int, int)}.
 */
public static CSSExpression parse(String input, CSSExpression expr) {
    int start = 0;
    // The initial combinator is always a descendant
    Combinator last = Combinator.DESCDANT;
    while( last != null && start < input.length()) {
        // Find the next combinator
        Combinator next = null; int nextPos = input.length();
        for (Combinator c: Combinator.values()) {
            int newPos = c.findKey(input, start);
            if(newPos != -1 && newPos < nextPos) {
                next = c; nextPos = newPos;
            }
        }
        // Everything from start to nextPos is now a sequence of simple selectors
        List<CSSSimpleSelector> selectors = parseSelectors(input, start, nextPos);

        // Create the last combinator
        expr = last.getExpression(expr, selectors);

        // For the next round we "advance" the combinator and the start position
        last = next; start = nextPos + 1;
    }

    return expr;
}
}

```