

Westfälische Wilhelms-Universität Münster
Englisches Seminar
Seminar: Computational Text Analysis
Lecturer: Hendrik Cyrus
WS 2007/08
22.01.2008

COIFFEUR — COMPUTATIONAL TEXT ANALYSIS IN PERL

Eugen Ruppert
Oberschlesierstr. 5
48151 Münster
0251/7038791
studium@euge.de
Matrikel 343206
Language, Text and Information

Contents

1	Introduction	1
2	Theory: The distinction between Type and Token	1
3	The program ‘Coiffeur’	2
3.1	Requirements	4
3.2	Running Coiffeur	4
3.3	Customization	5
4	Code Explanation	7
4.1	Read Text File	8
4.2	Lexical Exceptions	8
4.3	Character Exceptions and Tagging	8
4.4	Tokenization	9
4.5	Types and Tokens	10
4.6	HTML Output	11
5	Outlook	12
	References	13
	Appendix	14
	The Perl program: coiffeur.pl	14
	The exceptions file: exceptions.txt	27
	The characters file: characters.txt	28
	The Cascading Style Sheet: coiffeur.css	30
	The example text file: mytext.txt	32
	The ‘tokenized’ text: mytext.txt	33
	The result HTML: mytext.html	36
	The printed HTML file	40

1 Introduction

In this term paper on computational text analysis I am going to describe my program 'Coiffeur'. The groundwork has been done in a group work in the seminar where we created a program which calculates the Type-Token Ratio of a text. It was executed in Perl, the seminar was based on the book Hammond (2003). I will first explain lexical density and its use. There are also some problems with lexical density and the distinction between 'Types' and 'Tokens' that need to be considered. I will go on with my program, describe its usage and the difference from the program we created during the seminar.

The biggest part will be the explanation of the source code. I will explain the different processing steps with an example text. Because my program is complex, highly customizable and works well there are other possible usages for this program. I will mention these in the outlook at the end of the paper.

2 Theory: The distinction between Type and Token

The Type-Token Ratio (TTR) determines the lexical density of the text, the diversion in the vocabulary, the richer the vocabulary the higher the TTR. The idea of Types and Tokens dates back to Saussure's definition of the linguistic sign, and its distinction between the "signifier" (shape of a word) and the "signified" (the underlying concept) (de Saussure 1916). Applied to text linguistics a text would contain many signifiers (words) that represent signifieds (meanings).

Peirce (1931–1958) actually proposes the terms 'Type' and 'Token'. Types are "things that exist", therefore he excludes *the* from the Type definition because it has no meaning itself. A Token is a single thing or event with a 'limited identity', 'limited' in the sense that it is bound to the text (and can represent other things or events in other texts). Types only occur as Tokens, thus the Type-Token Ratio can never be larger than '1'.

Hutton (1990) criticizes the distinction of Types and Tokens, at least the lexical one. It cannot account for the diversity of meanings a word has. The opposite direction is not possible as well, synonyms cannot be processed in most cases. But in my opinion a computational analysis that would solve all the semantical problems is not possible, either. And I think that lexical density can provide information worth studying, especially comparing text from different languages. My program even makes it possible to exclude words like *the* or *a* (or any other word; see Customization, p. 5) from the analysis.

A further challenge for Type-Token Ratios is the fact that different texts are not easily comparable. The texts have to be of approximately the same size, as the lexical density decreases over time (cf. Figure 1, which is taken from Tweedie and Baayen (1998)). The text genres have to be at least similar, it would not make sense to compare a poem with a rich vocabulary to a financial report, which has a narrow vocabulary. When comparing text extracts, their relative position in the text (beginning, ending)

has to be considered. Brown and Yule (1983) and Meyer (2004) explain that there are both lexical and topical differences.

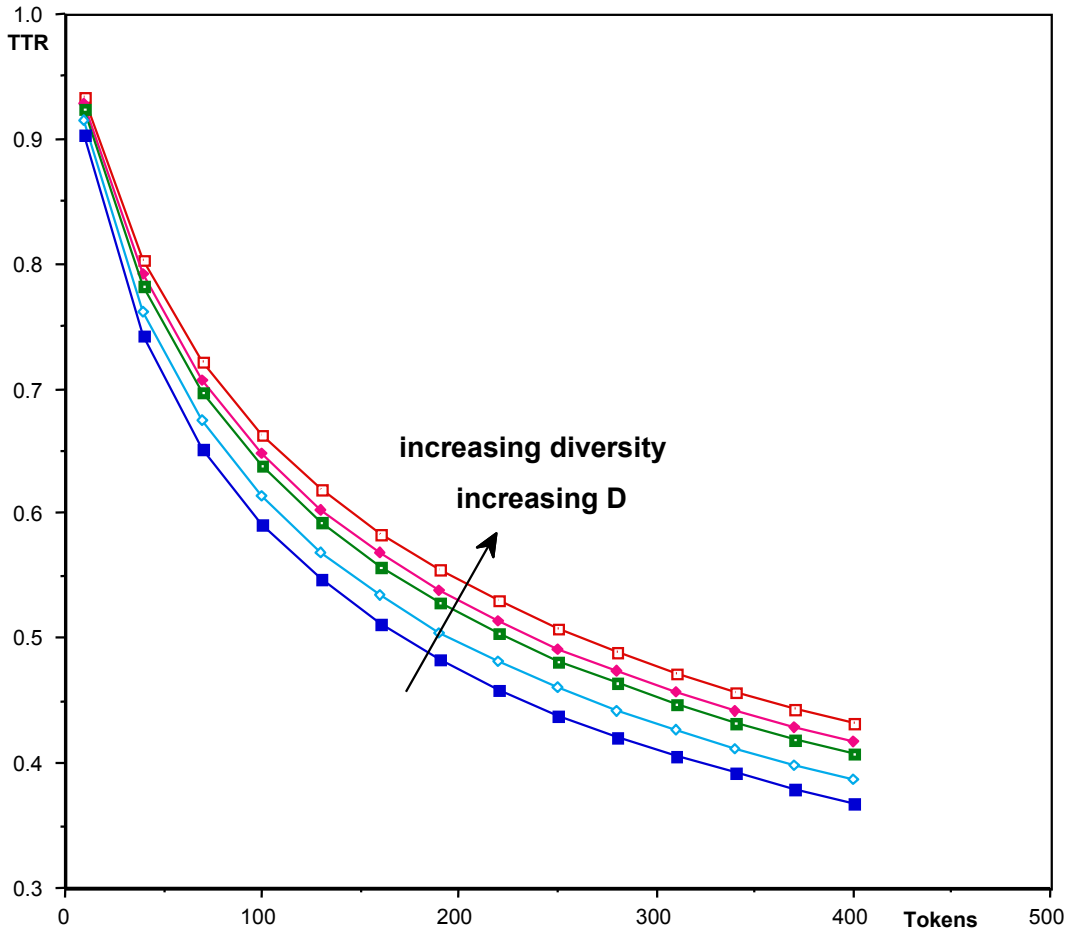


Figure 1: The decreasing Type-Token Ratio in longer texts

3 The program 'Coiffeur'

I called my program 'Coiffeur', the name comes from the French word *coiffeur* and means *hairdresser*. I think of Coiffeur as a (humorous) metaphor for a hairdresser who beautifies a customer and gets acquainted with him or her in a chat. When the hairdresser is done he can draw some conclusions about his customer, he 'analyzed' the customer, hence the slogan: *Coiffeur — Making you pretty while analyzing your content*.

As mentioned in the Introduction this program is based on the group work for the seminar Computational Text Analysis where we built a simple Perl program which calculates the Type-Token Ratio for a specified text. The source code is provided in Listing 1.

Listing 1: Source code of our group work

```

1 use diagnostics;

# Read file
open (FL, $ARGV[0]) or die("File '$ARGV[0]' could not be opened!\n");
while (<FL>) {
6   chomp;
   # Split a line into an array of words
   @line = split(/[.,;:!\?"'()*\s+[\.,;:!\?"'()*\s*--\s*|^\\"'\'(\)|
       [\"'\(\)]?[.!?]+[\"'\(\)]?\\Z/, $_);
   # Add each word to the hash and the array
11  foreach $word(@line) {
       # If a word is not empty
       if ($word ne "") {
           # Add it to the words-array
           push (@words, $word);
16         # Convert to lowercase
           $type = lc $word;
           # If the word is not in the hash, add it
           if (!exists $hash{$type}) {
21             $hash{$type} = 1;
           }
       }
   }
}
close(FL);
26
foreach $word(@words) {
    print "$word\n";
}

31 # Calculate token-count, type-count and the type-token-ratio
$tokenc = $#words + 1;
$typec = keys(%hash);
$ttr = $typec / $tokenc;

36 # Return results
print "The text consists of $tokenc words.\n";
print "The text has $typec types.\n";
print "The Type-Token-Ratio is: $ttr.\n";

```

We can see that this is a very basic and straightforward approach:

- Read the text file.
- Strip the text of (almost) all special, non-word characters.
- Add each word into a 'words' array.
- Check whether the (lowercase) word has been encountered before.
- If not, add it to a hash (associative array).
- Calculate the Type-Token Ratio using the Type and the Token count.

- Print the result.

But this program is small, very easy to understand and does not need many resources. What then are the features and improvements of Coiffeur which has 15 times more lines of code?

Coiffeur has a different focus than the group work. For the group work we only needed the result 'Type-Token Ratio', everything else was removed by the 'split' command. Coiffeur aims at an accurate representation of the text in HTML. All non-word characters are kept and in many cases turned to HTML entities to create valid HTML.

3.1 Requirements

To use the program you will need the following:

- Perl, you need a computer that runs Perl. If your computer does not run Perl, you can get it at the following URI:
<http://www.perl.com/download.csp>
- The Coiffeur package¹, which consists of the files:
 - *coiffeur.pl*, the main program.
 - *exceptions.txt*, lexical exceptions, words you want to include in your results and possibly assign to (other) word meanings. With this file you can define that *Dr.* will be included in the results for *Doctor* and *doctor*.
 - *characters.txt*, special characters or sequences of characters you want to exclude from your results. This can be punctuation marks, parentheses and quotes but can also be used to exclude certain words from your results.
 - (optional) *coiffeur.css*, a stylesheet for the created HTML files that produces a nice presentation. You probably want to create your own or edit it to suit your needs.
 - (optional) *coiffeur.png*, an image used by the provided stylesheet.
 - (optional) *Coiffeur.pdf*, the documentation you are reading now.
- A text file that you want to process. There is an example text file *mytext.txt* provided but it is more interesting to analyze your own text files. Any file extension will be processed.

3.2 Running Coiffeur

You execute the program with

```
perl /path/to/coiffeur.pl /path/to/yourtextfile
```

¹You can get the latest version at <http://shorty.euge.de/program/coiffeur/>

If you have the files in your Coiffeur directory, 'perl coiffeur.pl yourtextfile' will start the program. If your text file has the extension '.txt', like filename.txt, the program will produce a HTML-file with '.html' as extension, filename.html. If your text has a different or no file extension '.html' will be appended to the file name.

The program terminates with the message *Analysis complete*. Then you will find your processed HTML file in the directory of your text file. You can view it from there but it will look best if you first copy the stylesheet coiffeur.css and the image coiffeur.png into this folder. The stylesheet features a nice colour scale, each word will have a colour corresponding to its frequency of occurrence in the text. You can see a screenshot in Figure 2.

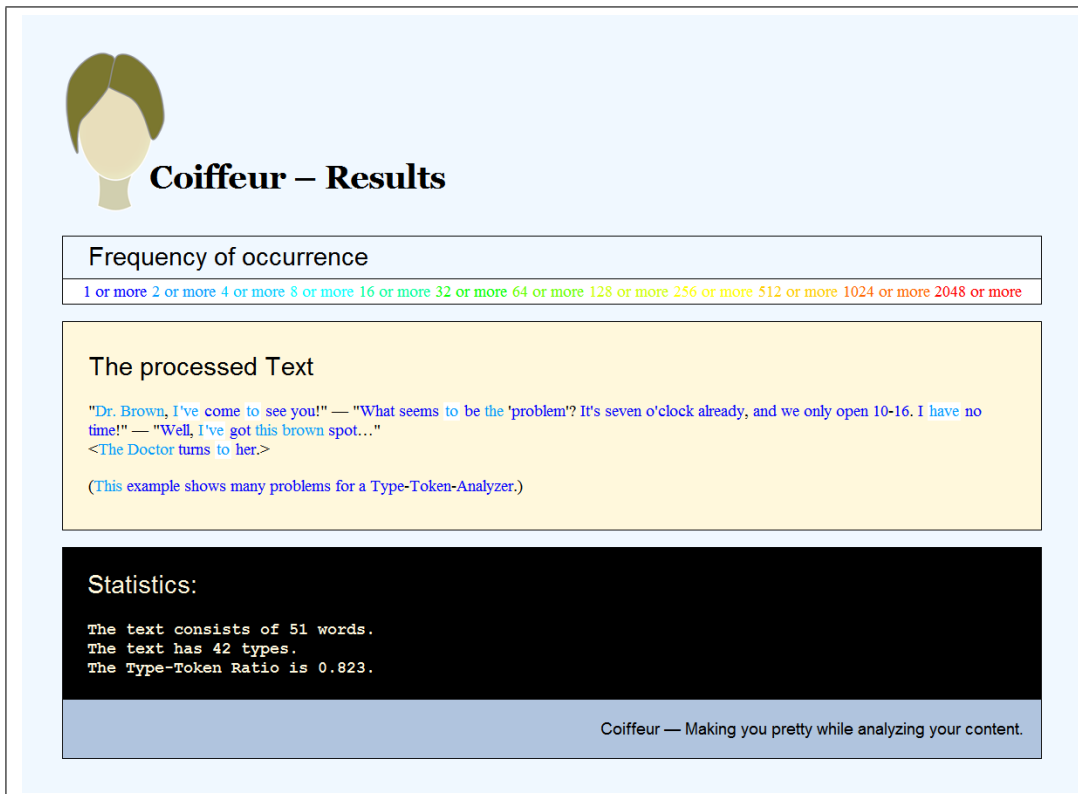


Figure 2: The appearance of the HTML file.

3.3 Customization

There are some parameters you can set directly in the program.

- You usually specify \$textfile as an argument for the program. But you can set it to a fixed value.
- The variables \$exfile and \$charfile can be changed to other files, especially if these files are in a folder other than the .pl-file.

- `$lbase` is mainly used for the computation of the 'word class' that will be assigned to a word. Words with a single occurrence will get word class '0', with `$lbase` you specify the steps for the other higher word classes. With the predefined value '2', the minimal number of occurrences for a word class doubles, so word class '1' would require the word to occur at least 2 times ($2^1 = 2$), the word class '2' would require 4 occurrences ($2^2 = 4$) and so on. Do not set this value to '1' or lower. If you do, it will be ignored and set to '2'.
- You can set `$debug` to '1' or '0' to enable or disable the debug mode. In this mode you will get much more output to your console. It can be useful to trace down bugs or to see how the program works.

You can edit the files `characters.txt`² and `exceptions.txt`² to adjust Coiffeur to your needs. These files contain exceptions of two kinds:

1. (Non-word) character exceptions are specified in `characters.txt` (Appendix, p. 28); you can exclude characters from the computation. This is actually necessary for a proper calculation. Without it, (1) would consist of two words, "'Hello,' and 'World!'" because all characters except the 'space' character are treated as word characters. The provided 'characters' file already covers most of such characters.

(1) "Hello, World!"

Another usage of this file is the exclusion of certain words from the computation. In my tests the most frequent words were in most cases the determiners *the* and *a*, *of* and *and*. If you do not want to analyze these, exclude them by using this file. (There are, in fact, some examples in the file but they are commented out.)

2. Lexical exceptions are defined in `exceptions.txt` (Appendix, p. 27); you can define words or parts of words to be treated on their own. Consider (2) as an example. You can define *'ve* from *I've* to be a 'word' on its own and link it to the word meaning *have*, if you want to. Then the results for *have* and *'ve* are combined.

(2) I've become a monster!

A meaning is not needed every time. The word *o'clock* needs to be in the exceptions file but does not need to be linked with a meaning from another word, as it is a full word on its own.

If you decide to include a meaning, type it in lowercase because in the analysis every word is transformed to lowercase. This inevitably leads to some errors in the computation. Coiffeur makes no distinction between *Brown* in *Dr. Brown* and the word *brown* in *this brown spot* (in the example file `mytext.txt`). But I

²The files described in the following have descriptions on their usage with many examples included, you can read them and create your exceptions accordingly.

consider this example much less frequent than *The* at the beginning of a sentence. The program could look whether the preceding character is a punctuation mark, but there would be many exceptions from this simple scheme, like parenthesis, dashes, quotation marks and even (multiple) spaces. Also, users can edit the exception files, the tag names can change and the program would then produce very inaccurate results. If the program would not do the lowercase transformation, some users would be wondering why the determiner *the* is not the most frequent word in (3)³.

(3) The boy sees the girl. The girl sees the boy.

Another general problem is the ambiguity of some words or word parts. *Let's*, *it's* and *John's* all contain 's but the meaning is different each time: *us*, *is* and 'the genitive s'. If you process a text with a wide variety of meanings for certain exception, it is probably best to leave the meanings out.

The last file you can modify is the Cascading Stylesheet for the HTML presentation, `coiffeur.css` (Appendix, p. 30). The most important classes will be the word classes `word0–word11`⁴. Words with the largest number of occurrences are also assigned the class `max`. In the default setting they have a different background in the screen presentation and are underlined when printed out.

Listing 2: Examples from the CSS file

```
.word0 {
    color: #0000FF;
}
.max {
    background: white;
    padding-left: 0.1em;
    padding-right: 0.1em;
}
```

4 Code Explanation

This will be a step-by-step code explanation where I will show on an example how my program processes a text. I type the small text from (4) and save it as `mytext.txt`.⁵ I run the program with `perl coiffeur.pl mytext.txt`.

(4) "Dr. Brown, I've come to see you!" --- "What seems to be the
'problem'? It's seven o'clock already, and we only open 10-16.
I have no time!" --- "Well, I've got this brown spot..."

³In this example all words would have the same Type count, if your program is case sensitive.

⁴You can define any word class, a word class `word100` is not wrong, but `2100` is a very large number, words with this word class are highly unlikely.

⁵This text file is included in the Coiffeur package so you can test it yourself.

Now that all characters that can mark a tag are replaced by tags (no intersections with the following code), the use of tags is enabled (lines 150–222). The characters or character sequences are read from the file specified in `$charfile` at the beginning. Each exception is in a new line (comments have to be ignored again), separated from its tag name by a tabulator character. These are stored in a hash `%tags`. For every exception a HTML-entity can be defined. If it exists, it is added to the `%entities` hash.

Then the substitution of character exceptions with tags occurs. The exception characters that will be substituted are first put into the array `@chararr`. There they are sorted from longest to shortest and applied to the text: each occurrence is substituted with the proper tag.⁷

When all other characters have been tagged, the space character can be turned into the tag `<'space'>` (lines 202ff). This step of tagging has to be the last, because other characters from the characters file might rely on spaces as part of a regular expression.

In the last step the non-text characters that are part of lexical exceptions are resubstituted with their decoding characters (lines 208ff). In *Dr.* the period is not tagged (see (6)) This was our intention, that is the reason we put *Dr.* in the exceptions file.

```
(6) <'qdb'>Dr.<'space'>Brown<'pccm'><'space'>I've<'space'>come
<'space'>to<'space'>see<'space'>you<'pcex'><'qdb'><'space'>
<'dem'><'space'><'qdb'>What<'space'>seems<'space'>to<'space'>
be<'space'>the<'space'><'qsg'>problem<'qsg'><'pcqu'><'space'>
It's<'space'>seven<'space'>o'clock<'space'>already<'pccm'>
<'space'>and<'space'>we<'space'>only<'space'>open<'space'>10
<'dhp'>16<'pcfs'><'space'>I<'space'>have<'space'>no<'space'>
time<'pcex'><'qdb'><'space'><'dem'><'space'><'qdb'>Well<'pccm'>
<'space'>I've<'space'>got<'space'>this<'space'>brown<'space'>
spot<'pell'><'qdb'><'lbr'><'lt'>The<'space'>Doctor<'space'>
turns<'space'>to<'space'>her<'pcfs'><'gt'><'pbr'><'pno'>This
<'space'>example<'space'>shows<'space'>many<'space'>problems
<'space'>for<'space'>a<'space'>Type<'dhp'>Token<'dhp'>
Analyzer<'pcfs'><'pnc'>
```

4.4 Tokenization

The ‘tokenization’—in the sense that every entity, be it a word or a tag, is a single item in the array `@words`—is easy to accomplish since every word is separated from other words by tags (see lines 226–235). The `split` command splits the word at a tag start `<`, a tag end `>` or between two tags `><`. Empty words, that result when the text starts or ends with a tag, are removed in a separate step.

Example (7) shows the first sentence “*Dr. Brown, I’ve come to see you!*”. The full text

⁷This solution is not perfect, especially with regular expressions there can be shorter exceptions that would substitute larger parts of the text. But if you know this you can adjust your expressions accordingly.

can be seen in the Appendix, p. 33. Tags are only enclosed in single quotes now, since the split command removed '`<`' and '`>`'. But they are still distinguishable from the words.

```
(7) 'qdb'
    Dr.
    'space'
    Brown
    'pccm'
    'space'
    I've
    'space'
    come
    'space'
    to
    'space'
    see
    'space'
    you
    'pcex'
    'qdb'
```

4.5 Types and Tokens

This part of the program (lines 251–338) calculates the Type count, the Token count and in the end the Type-Token Ratio. The array @words is traversed and each item is first checked whether it is a word or a tag. Tags are not processed, since they are not needed for the count.

If \$word is a regular word with no occurrences of non-alphanumerical characters, the Token count is incremented. The word is transformed to lowercase and the count for this word in the hash %types is also incremented.

Words that contain a quote or a period are a little more complicated to count. When the word contains a period it is likely to be an abbreviation, thus the program looks the word up in the %exceptions hash. If a result is found (this would be the 'meaning' mentioned earlier) the Type count for the meaning is incremented. If there is no meaning, the Type count for the word itself is incremented. In both cases the program increases the Token count.

A word that contains a quote like *I've* is even more complicated, because it probably consists of two words, *I* and *'ve* or *have*, depending on how you view it. So first the proper exception must be found in @exarray. If nothing is found (which should not happen), the whole word is counted.

If there is one exception found, \$word is splitted into \$word1 and \$word2, where \$word2 is the exception (like *'ve*) and \$word1 is the complement of \$word2 in \$word (like *I*, which is the complement of *'ve* in *I've*). \$word1 can be empty with words like

o'clock where the whole word is the exception. *Coiffeur* accounts for this fact.

Both of these words are processed in the following way: if there is one meaning defined in the exceptions, the Type count for the meaning is raised, else the Type count for the word itself. For both words the Token count is incremented as well.

The Type-Token Ratio is calculated in lines 326–338. If there are more than zero tokens the Type-Token Ratio is the division $TTR = \text{Type count} / \text{Token count}$, otherwise $TTR = 0$. The resulting number can have many decimal places, they are reduced to three.

4.6 HTML Output

Before writing the HTML file the program needs to set or check some variables (lines 343ff). The word number `$wnr`, which is a counter variable for the sequence position of the word in the text is set to '1'. The highest number of occurrences is retrieved from the `%types` hash and stored in `$maxocc`. To prevent division by zero the `$lbase` variable is checked. If it is set to 1 or below, the program corrects the number to 2 ($\log(1) = 0$; $\log(<1) < 0$). To be able to quickly look up the character-representation of a tag in the text, the `%tags` hash is reversed and stored as `%characters`, the keys are exchanged with the values. The file name of the output is also established.

After this the file is opened for writing. The HTML file starts with the 'doctype' definition. Since my program produces clean and valid code, it is set to *XHTML 1.0 Strict*. Next comes the HTML head with the title and stylesheet information. The title is generated from the `@words` array, the first five non-tags are included.

The interesting thing from the body—aside from the word output—is the generated table in which the minimum amount of words for a group `word0–word11` is calculated and printed into the file. Note that every section of the HTML code has an own id, you can use it to access and style style the output.

The functionality of the text output (lines 446–638) is similar to the procedure from Type and Token counting, only this time the number values are not stored but retrieved, so I will not discuss this in detail. Each word is put into a `` tag, which is an inline element, so it does not disturb the presentation.

The following attributes are applied to a `` tag:

- The 'word`n`' class, where `n` is an integer number equal to or greater than 0. It is calculated from the number of occurrences `$occnr` and the variable `$lbase` set by the user. `$lbase` is standardly set to '2', this means that to reach the next higher group, the word frequency has to be twice as high. (The frequencies for the word classes 0–11: 1 2 4 8 16 32 64 128 256 512 1024 2048)
- If the current word is the most frequent word, the class 'max' is also included.
- The 'word id', a unique number representing the position of the word in the text, for example the first word would have `id="word1"` as an attribute of its `` tag.

- A title attribute with the number of occurrences, so that when you move your mouse over a word a tool tip will show up and inform you about the number of occurrences.

The HTML output also attends tags, they are needed for the representation. First, the quotes enclosing the tag name are removed. If there is an entity defined it is printed out. If no entity is defined the hard coded tags 'gt', 'lt' and 'qsg' are checked. Should they not match either, the %characters hash is used to retrieve the character representation for a tag. These characters often contain parts of a regular expression, like escape characters. These 'active' characters are removed. Should the character turn out to be a space character, a line break with tabulator keys is printed after the space to increase the readability of the HTML output. A linebreak in the HML file can only occur after a space character because otherwise you will get space characters where there should be none, e.g. between a quote character and the word.

The last part (lines 642ff) is the statistics output, which is already calculated. When it has been included in the document the program has finished processing the text. You can see the resulted HTML code in the Appendix, page 36. A printout of the HTML document is on page 40.

5 Outlook

As mentioned before, Coiffeur can be used for other tasks than just Type-Token relation. Because of its high extensibility and preservation of all text content this program can provide a basis for a range of applications.

Lists with word frequencies can be generated with ease, since all the frequencies are already computed. The words are sorted from highest to lowest frequency of occurrence in the array @frtypes. Another possible application is the computation of the mean sentence length. Punctuation is tagged in the @words array, exceptions like *Dr.* are excluded with exceptions lists. So you can count the number of words between (the tags of) two punctuation marks to get your results.

An advanced application would be the implementation of a 'text processor' that automatically converts the user input (plain text) into HTML. Coiffeur does this already but needs to be extended in the syntax to be able to use text formatting like 'bold' or 'italics' and HTML elements other than <p> and
. Such a processor could then be included into a Content Management System. Such systems are common with web services, like Wikipedia⁸ and online weblog editors⁹.

⁸<http://www.wikipedia.org/>

⁹A well working example is Textile, you can test it at this URI:
<http://textism.com/tools/textile/index.php>

References

- [Brown and Yule1983] Brown, G. and G. Yule. 1983. *Discourse Analysis*. Cambridge: Cambridge University Press.
- [de Saussure1916] de Saussure, F. 1916. *Cours de linguistique générale*. Paris: Payot.
- [Hammond2003] Hammond, M. 2003. *Programming for Linguistics: Perl for Language Researchers*. Malden, Massachusetts: Blackwell Publishers.
- [Hutton1990] Hutton, C. M. 1990. *Abstraction & Instance*. Language and Communication Library, vol. 11. Oxford: Pegasus Press.
- [Meyer2004] Meyer, C. F. 2004. Can You Really Study Language Variation in Linguistic Corpora? *American Speech* 79: 339–355.
- [Peirce1931–1958] Peirce, C. S. 1931–1958. *Collected Papers of Charles Sanders Peirce*. Cambridge: Harvard University Press.
- [Tweedie and Baayen1998] Tweedie, F. J. and R. H. Baayen. 1998. How Variable May a Constand be? Measures of Lexical Richness in Perspective. *Computers and the Humanities* 32: 323–352.

All of the mentioned websites were available at the point of writing this term paper.

Appendix

The Perl program: coiffeur.pl

```

#use diagnostics;
2 use POSIX;

#####

# Coiffeur -- Making you pretty while analyzing your content
7
#####

# Text file
$textfile = $ARGV[0];
12 # Exceptions-file
$exfile = 'exceptions.txt';
# Character file
$charfile = 'characters.txt';

17 # The logarithm base
# Use 2 for short texts, 4 for very long texts
# Probably '2' will suffice for texts below 50.000 words
$base = 2;

22 # Variable for debugging (1=on, 0=off)
# Prints information on what the program is doing
$debug = 1;

# Special characters to mark the beginning and end of a tag
27 # before parenthesis are processed (especially '<' and '>')
$starttag = chr(17);
$endtag = chr(18);

# These characters are needed for exception-handling
32 # (such as: 'Dr.' -> 'Dr\chr(21)')
$colon = chr(19);
$period = chr(20);
$comma = chr(21);
$quote = chr(22);
37
#####

# TEXTFILE
# File Reading
42 open (FL, $textfile) or die("File '$textfile' could not be opened!\n");
print "\nAnalyzing file '$textfile'\n";
print "-----\n\n";

# Put the whole text into one string
47 $text = join("", <>);

# Debugging

```

```

if ($debug) {
    print "The original text:\n\n";
52  print "$text\n\n";
}
close (FL);

57 #####

# EXCEPTIONS
# Read Exceptions File

62 if ($debug) {
    print "The exception list:\n";
    print "Exception --\t'meaning'\n";
}

67 open (EX, $exfile) or die("File '$exfile' could not be opened!\n");
while (<EX>) {
    chomp;
    # if not empty and not a comment, started with a single quote '''
    if ($_ ne "" && $_ !~ m/\A\'/) {
72     # each line has one exception, form and meaning separated by '\t'
        @exception = split(/\t/, $_);
        # build an exception-hash
        if ($#exception > 0) {
            $exceptions{$exception[0]} = $exception[1];
77         if ($debug) {
            print "'$exception[0]'\t--\t'$exception[1]'\n";
        }
    }
}
82 }
close (EX);

# Process Exceptions
87 @exarray = keys(%exceptions);

foreach $ex(@exarray) {
    $exreplace = $ex;
    # substitute special characters with unused characters
92 $exreplace =~ s/\\. /$period/g; #.
    $exreplace =~ s/\\,/,$comma/g; #,
    $exreplace =~ s/\\\'/$quote/g; #'
    $exreplace =~ s/\\:/$colon/g; #:

97 # substitute occurrences of the exceptions with 'tagged exceptions'
    $text =~ s/$ex/$exreplace/g;
}

if ($debug) {
102 print "\nThe text with marked exceptions:\n";

```

```

    print "$text";
}

107 # Now that the escape characters '\ ' are removed from the text,
    # we need to remove them from the hash, too,
    # to be able to access exceptions later
    foreach $ex(@exarray) {
        # create new key
112 $newex = $ex;
        $newex =~ s/\\//g;
        # delete old key, retrieve value
        $val = delete $exceptions{$ex};
        # insert new key-value pair
117 $exceptions{$newex} = $val;
    }

    # Build a new @exarray
    @exarray = keys(%exceptions);
122

#####

127 # TAGGING
    # Mark all special, non-word characters globally with tags

    # We first need the single quote (') because
    # a tag has the form <'NAME'>
132 $text =~ s/\'/${starttag}\'qsg\'${endtag}/g; #single quote

    # Quotes can be used now
    # To enable use of tags, we translate:
    # '<' --> '<'lt'>' and '>' --> '<'gt'>'
137 $text =~ s/\</${starttag}\'lt\'${endtag}/g; #<
    $text =~ s/\>/${starttag}\'gt\'${endtag}/g; #>

    # Now we turn '$starttag' to '<' and '$endtag' to '>'
    $text =~ s/${starttag}/</g;
142 $text =~ s/${endtag}/>/g;

    # Every tag can be used now, so we
    # proceed with the other special characters from the characters file
147
    ###-----###

    # SPECIAL CHARACTERS
    # These are used to distinguish words from non-words,
152 # non-words specified in the $charfile are put into tags

    if ($debug) {
        print "\nSpecial characters:\n";
    }

```

```

    print "Character\tTagname\t'HTML-Entity'(optional)\n";
157 }

# File Reading
open (CF, $charfile) or die("File '$charfile' could not be opened!\n");
162 while (<CF>) {
    chomp;
    # if not empty and not comment line (started with a single quote)
    if ($_ !~ m/\'.*/ && $_ ne "") {
        # split line at tabs
167     @charcontent = split(/\t/, $_);
        # add the tags to a tags-hash
        $tags{$charcontent[0]} = $charcontent[1];
        # add the entities to an entities-hash, if existent
        if ($#charcontent > 1) {
172     $entities{$charcontent[1]} = $charcontent[2];
        }
        if ($debug) {
            print "$charcontent[0]\t\t\t$charcontent[1]";
            if ($#charcontent > 1) {
177     print "\t\t\t$entities{$charcontent[1]}\n";
            }
            else {
                print "\n";
            }
        }
182     }
    }
}
close(CF);

187
# Substitution
# The array needs to be sorted, otherwise smaller entities,
# like endash, might be processed before emdash
@chararr = sort{length($b)<=>length($a)}(keys(%tags));
192 if ($debug) {
    print "\nNon-word characters will be processed in this order:\n";
    print "@chararr\n\n";
}

197 # substitute characters with tags
foreach $char(@chararr) {
    $text =~ s/$char/\<\'$tags{$char}\'\>/g;
}

202 # Spaces
# now spaces can be tagged, as the other characters,
# possibly depending on '\s', are already tagged
$tags{" "} = "space";
$text =~ s/\ /<\'space\'\>/g;
207
# Resubstitution

```

```

# We can now resubstitute the exceptions
# so that they are processed in words (e.g Ive)
$text =~ s/$colon/\:/g; #:
212 $text =~ s/$period/\./g; #,
$text =~ s/$comma/\./g; #.
$text =~ s/$quote/\'/g; #'

217 # Test of the tagging result
if ($debug) {
    print "Text after tagging:\n";
    print "$text\n\n\n";
}
222

#####

# TOKENIZATION
227 # Split the text at tag openings or closings: '<', '><' or '>'
@tempwords = split (/<|\>|\<|\>/, $text);

# There can be some 'empty' words, so we sort them out.
foreach (@tempwords) {
232   if ($_ ne "") {
        push(@words, $_);
    }
}

237 # Test whether Tokenization runs correctly
# Words should be separated,
# Special characters marked with single quotes: e.g 'space'
if ($debug) {
    print "The tokens (with 'tagged' symbols):\n";
242   foreach $word(@words) {
        print "$word\n";
    }
    print "\n";
}
247

#####

# TYPES
252 # Determining the types
foreach $word(@words) {
    # if the word is not a tag
    if ($word !~ m/\'.+\'/) {
        # look for a non-word character, as in 'Mr.' or 'I've'
257     if ($word =~ m/\W/) {
            # separation between abbreviations with a period, 'Mr.',
            # and those containing quotes: I've
            if ($word =~ m/\./) {
                # substitute with meaning, else add it to type hash

```

```

262         if (exists $exceptions{$word}) {
                $types{$exceptions{$word}}++;
                $tokenc++;
            }
            else {
267         $types{$word}++;
                $tokenc++;
            }
        }
        # words with quotes
272     else {
            # search in @exarray
            foreach $ex(@exarray) {
                if ($word =~ m/$ex/) {
277                 $word2 = $ex;
                    last;
                }
            }
            # if nothing found increase typecount for the whole $word
            if ($word2 eq "") {
282         $types{$word}++;
                $tokenc++;
            }
            # if found
            else {
287         # increase count of the meaning of $word2
                if (exists $exceptions{$word2}) {
                    $types{$exceptions{$word2}}++;
                    $tokenc++;
                }
                else {
292         $types{$word2}++;
                    $tokenc++;
                }
                # $word1 is the $word without $word2
297         $word1 = $word;
                $word1 =~ s/$word2//;
                # add $word1 if not empty
                if ($word1 ne "") {
302         $types{$word1}++;
                    $tokenc++;
                }
            }
        }
    }
}
307 # if regular word (none of the above)
else {
    # increase tokencount
    $tokenc++;
    # increase typecount for the lowercase 'word' in the hash
312 $types{lc$word}++;
}
}

```

```

}

317

# Calculation Type-Token Ratio
$typec = keys(%types);
# Prevent division by Zero
322 $ttr = 0;
if ($tokenc > 0) {
    $ttr = $typec / $tokenc;
    # truncate digits after period, if there are more than 3
    if ($ttr =~ m/\.\d{4,}/) {
327         while ($ttr !~ m/\d\.\d{3}$/) {
            $ttr =~ s/\d$//;
        }
    }
}
}

332

#####

# WRITING HTML
337
###-----###

# Set some variables before writing file
# wordnumber
342 $wnr = 1;
# the reversed tags hash, now you can substitute tags with characters
%characters = reverse %tags;
# highest number of occurrences
%revtypes = reverse %types;
347 @frtypes = sort{ $b > $a }(keys (%revtypes));
$maxocc = $frtypes[0];

if ($debug) {
    print "\n\nMaximum number of occurrences: $maxocc\n\n";
352 }

# Correct $lbase, if '1' or lower
# to prevent wrong results and division by Zero
if ($lbase <= 1) {
357     print "The base for logarithmic computation is too low: $lbase.\n";
    print "Setting it to '2'.\n";
    $lbase = 2;
}

362

# Determine HTML file name
$htmlfile = $textfile;
367 # if text file has the ending .txt

```

```

if ($htmlfile =~ m/\.txt$/) {
    #substitute '.txt' with '.html'
    $htmlfile =~ s/\.txt$/\.html/;
}
372 # else attach .html
else {
    $htmlfile = $htmlfile.".html";
}

377
print "Using '$htmlfile' for output.\n\n";

###-----###
# Writing File
382 open (FW, '>', $htmlfile);

# HTML Header
print FW "<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"\n";
print FW "\"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">\n\n";
387 print FW "<html>\n";
print FW "\t<head>\n";
# Title
print FW "\t\t<title>Coiffeur:\n";
# Prints the first five words from the words array
392 $titnr = 5;
$i=0;
for ($c=0;$c < $titnr; $c++) {
    # if index in array boundaries,
    if ($i <= $#words +1) {
397     # search for a non-tag word and print it
        while ($words[$i] =~ m/\'.\+\'/ && $i <= $#words) {
            $i++;
        }
        print FW " $words[$i]";
402     $i++;
    }
}
print FW "\n</title>\n";

407 print FW "\t\t<link rel=\"stylesheet\" href=\"coiffeur.css\" ";
print FW "type=\"text/css\" media=\"screen, projection, print\" />\n";
print FW "\t</head>\n\n";

# HTML Body
412 print FW "\t<body>\n";
print FW "\t\t<div id=\"header\">\n";
# Heading
print FW "\t\t\t<h1>Coiffeur &ndash; Results</h1>\n";
print FW "\t\t</div>\n";
417
# Table with colours for the relative frequency
print FW "\t\t<div id=\"reftable\">\n";
print FW "\t\t\t<h2>Frequency of occurrence</h2>\n";

```

```

print FW "\t\t\t<table>\n";
422 print FW "\t\t\t\t<tr>\n";
# twelve columns,
# show the minimum amount of occurrences for each wordclass
for ($col = 0; $col<12;$col++) {
    $occnr = $lbase ** $col;
427 print FW "\t\t\t\t<td class=\"word$col\">\n";
    print FW "\t\t\t\t$occnr or more \n";
    print FW "\t\t\t\t</td>\n";
}

432 print FW "\t\t\t\t</tr>\n";
print FW "\t\t\t</table>\n";
print FW "\t\t</div>\n";

437 ###-----###

# Write the text
print FW "\t\t<div id=\"text\">\n";
print FW "\t\t\t<h2>The processed Text</h2>\n";
442 print FW "\t\t\t<p>";

# Insert the text from the array
foreach $word(@words) {
    # check whether word or tag
447 # if word
    if ($word !~ m/\'.+\'/) {
        $lword = lc $word;
        # check, whether a tokencount exists
        #(exceptions don't have a tokencount)
452 if (exists $types{$lword}) {
            $occnr = $types{$lword};
        }
        # else find exception and process it
        # (similar to typecount before)
457 else {
            # separation between abbreviations with a period, 'Mr.',
            # and those containing quotes I've
            if ($word =~ m/\./) {
                # get tokencount for the meaning
462 if (exists $exceptions{$word}) {
                    $occnr = $types{$exceptions{$word}};
                }
                # else set occurrence to 1
                else {
467 $occnr = 1;
                }
            }
            # words with quotes
            else {
472 # search in @exarray
                foreach $ex(@exarray) {

```

```

    if ($word =~ m/$ex/) {
        $word2 = $ex;
        last;
477     }
    }
    # if nothing found set occurrence to 1
    if ($word2 eq "") {
482     $occnr = 1;
    }
    # if found
    else {
        # split into two words
        $word1 = $word;
487     $word1 =~ s/$word2//;

        # process word1 if not empty
        if ($word1 ne "") {
            print FW "<span class=\"word\"";
492     # if there is a typecount
            if (exists $types{$word1}){
                $occnr = $types{$word1};
                $freqlog = floor(log($occnr)/log($lbase));
            }
497     else {
                $occnr = 1;
                $freqlog = 0;
            }
            # Print attributes
502     print FW "$freqlog";
            if ($maxocc == $occnr) {
                print FW " max\" ";
            }
            else {
507     print FW "\" ";
            }

            print FW "id=\"word$wnr\" ";
            print FW "title=\" $occnr occurrence\"";
512     # determines sg or pl ending
            if ($occnr > 1) {
                print FW "s\">";
            }
            else {
517     print FW "\">";
            }

            print FW "$word1";
            $wnr++;
522     print FW "</span>";
        }
        # proceed with the second word, if not empty
        if ($word2 ne "") {
            print FW "<span class=\"word\"";

```

```

527         # if there is a typecount
         if (exists $types{$exceptions{$word2}}) {
             $occnr = $types{$exceptions{$word2}};
             $freqlog = floor(log($occnr)/log($lbase));
         }
532     else {
         $occnr = 1;
         $freqlog = 0;
     }
     # Print attributes
537     print FW "$freqlog";
     if ($maxocc == $occnr) {
         print FW " max\" ";
     }
     else {
542         print FW "\" ";
     }

     print FW "id=\"word$wnr\" ";
     print FW "title=\"$occnr occurrence";
547     # determines sg or pl ending
     if ($occnr > 1) {
         print FW "s\">";
     }
     else {
552         print FW "\">";
     }

     print FW "$word2";
     $wnr++;
557     print FW "</span>";
    }
    }
    next;
}

print FW "<span class=\"word";

567     # Determine the frequency logarithmically to the defined base:
     # e.g. log(frequency)/log(2) rounded down.
     $freqlog = floor(log($occnr)/log($lbase));

     # following information is included for every word:
572     # frequency group (0-11), wordnumber
     # and the frequency of occurrences as the title attribute
     print FW "$freqlog";
     if ($maxocc == $occnr) {
         print FW " max\" ";
     }
577 }
     else {
         print FW "\" ";
     }

```

```

    }
    print FW "id=\"word$wnr\" title=\"${occnr} occurrence";
582 # determines sg or pl ending
    if ($occnr > 1) {
        print FW "s\">";
    }
    else {
587     print FW "\">";
    }
    print FW "$word";
    print FW "</span>";
    $wnr++;
592 }

# if tag
else {
    # remove quotes
597 $word =~ s/\'//g;
    # if there is an entity defined use it
    if (exists $entities{$word}) {
        print FW "$entities{$word}";
    }
602 # else if is 'lt' (<) print '&lt;';
    elsif ($word eq "lt") {
        print FW "&lt;";
    }
    # else if 'gt' (>) print '&gt;';
607 elsif ($word eq "gt") {
        print FW "&gt;";
    }
    # else if 'qsg' ('), print ''
    elsif ($word eq "qsg") {
612     print FW "'";
    }
    # else use the character
    else {
        $char = $characters{$word};
617     $char =~ s/\\/\\/g;
        $char =~ s/\\/+/g;
        $char =~ s/\\*/g;
        print FW "$char";
        # insert linebreak if space
622     if ($char eq " ") {
            print FW "\n\t\t\t";
        }
    }
}
}
627 }

print FW "\n\t\t\t</p>\n";
print FW "\t\t</div>\n";
632

```

```
###-----###  
  
# Write Statistics  
print FW "\t\t<div id=\"statistics\">\n";  
637 print FW "\t\t\t<h2>Statistics:</h2>\n";  
  
print FW "\t\t\t<p>\n";  
print FW "\t\t\tThe text consists of $tokenc words.<br />\n";  
print FW "\t\t\tThe text has $typec types.<br />\n";  
642 print FW "\t\t\tThe Type-Token-Ratio is $ttr.<br />\n";  
  
print FW "\t\t\t</p>\n";  
print FW "\t\t</div>\n";  
  
647 # Footer  
print FW "\t\t<div id=\"footer\">\n";  
print FW "\t\t\t<p>\n";  
print FW "\t\t\tCoiffeur &mdash; ";  
print FW "Making you pretty while analyzing your content.\n";  
652 print FW "\t\t\t</p>\n";  
print FW "\t\t</div>\n";  
  
print FW "\t</body>\n";  
print FW "</html>\n";  
657  
# File Writing Complete  
close (FW);  
  
# All done  
662 print "\nAnalysis complete.\n\n";
```

The exceptions file: exceptions.txt

```
' The exceptions list
' -----
' Each exception is on a separate line
' A 'meaning' is possible too, separate with 'TAB'
' This meaning is the word your exception will be counted to.
'
' For example, if you want 'Dr.' to be included
' in the count of 'doctor' and 'Doctor' you type:
' Dr\.    doctor
'
' If you don't want this, include 'Dr\.' without a 'meaning':
' Dr\.
'
' Keep your meanings in lowercase for a proper count
' because all occurrences of them in the text
' will be transformed to lowercase.

' You can insert comment lines by placing a single quote
' at the beginning of the line.
```

```
Mr\.    mister
Ms\.    ms
Dr\.    doctor
Prof\.  professor
etc\.   et cetera
vs\.    versus
n\'t    not
\'m     am
\'re    are
\'ve    have
\'d     had
\'s     is
\'ll    will
o\'clock o'clock
```

The characters file: characters.txt

```

' You can comment using a single quote at the beginning of a line
,
' This file lists all non-word characters you want to process
' They are else treated as word-characters
,
' The Format is: Character TAB Tag cTAB HTML-Entity(optional)
' Example:          \!                pex
' Example:          \<                lt                &lt;;
,
' Regular expressions are possible, too:
' \n\n+    pbr          </p></p>
' \n      lbr          <br />
,
' Please not that you will need to escape all 'active' characters,
' character, that can occur in a regular exrepsson.
,
' The following characters are hard coded in the program,
' do not reassign them or you will get false results
' \ '      qsg
' \ ;      psc
' \ <      lt          &lt;;
' \ >      gt          &gt;;
' \       space
,
' If you use characters, that can be part of a regular expression,
' such as '*', '+' or '\', you need to use a HTML entity
' which will be inserted instead of the character.
' Example:
' \\      slb          \
,
' This limitation allows the user to leave out HTML entities
' for all the other characters, so it is rather good than evil. :-)
,
' You can exclude certain words, like functionwords
' or determiners from your results.
' it is best to provide uppercase and lowercase examples
' uppercase:
'\sTo\s      To          To
'\sThe\s     The          The
'\sA\s       A            A
'\sIs\s      Is           Is
'\sOf\s      Of           Of

' lowercase
'\sto\s     to
'\sthe\s    the          the
'\sa\s      a
'\sis\s     is
'\sof\s     of
,
' #####

```

```
\&      amp      &amp;
\"       qdb      &quot;

' Punctuation
\.       pcfs     .
\\.\\.\\.+ pell    &hellip;
\,       pccm
\!       pcex     !
\?       pcqu     ?
\:       pcco

' Line- and Paragraphbreaks
\n\n+   pbr      </p><p>
\n      lbr      <br />

' Parenthesis
\(       pno
\)       pnc
\[       pso
\]       psc
\{       pco
\}       pcc

' Dashes
\---+   dem      &mdash;
\--     den      &ndash;
\ -     dhp

' Slashes
\\      slb      \
\/      slf
```

The Cascading Style Sheet: coiffeur.css

```
1 @media all{
  body {
    background: aliceblue;
    size: 1.3em;
  }
6  h2 {
    font-family: sans-serif;
    font-weight: lighter;
  }
  h1 {
11  font-family: Georgia, serif;
    letter-spacing: -0.02em;
  }
  #text {
16  background: cornsilk;
    border: 1px solid black;
    padding: 0.5em 1.5em 1em 1.5em;
    margin: 1em 2em 0em 2em;
  }
  #statistics {
21  background: black;
    color: cornsilk;
    padding: 0.1em 1.5em 0.3em 1.5em;
    margin: 1em 2em 0em 2em;
  }
26  #statistics p {
    font-family: monospace;
    font-size: 1.2em;
    font-weight: bold;
  }
31  #header {
    margin: 2em 2em 0 2em;
    padding: 100px 0 20px 100px;
    background-image: url('coiffeur.png');
    background-position: top left;
36  background-repeat: no-repeat;
  }
  #footer {
    background: lightsteelblue;
    border: 1px solid black;
41  font-family: sans-serif;
    text-align: right;
    padding: 0.1em 1em 0.2em 1.5em;
    margin: 0em 2em 2em 2em;
  }
46  #reftable {
    border: 1px solid black;
    margin: 0em 2em 0em 2em;
  }
  #reftable h2 {
51  margin: 0.2em 0 0.3em 1em;
```

```
    }
    #reftable table {
        background: white;
        width: 100%;
56     border-top: 1px solid black;
        padding-left: 1em;
        padding-right: 1em;
    }
    .max {
61     background: white;
        padding-left: 0.1em;
        padding-right: 0.1em;
    }
    .word0 {
66     color: #0000FF;
    }
    .word1 {
        color: #0099FF;
    }
71     .word2 {
        color: #00CCFF;
    }
    .word3 {
        color: #00FFFF;
76     }
    .word4 {
        color: #00FF99;
    }
    .word5 {
81     color: #00FF00;
    }
    .word6 {
        color: #66FF00;
    }
86     .word7 {
        color: #CCFF00;
    }
    .word8 {
        color: #FFFF00;
91     }
    .word9 {
        color: #FFCC00;
    }
    .word10 {
96     color: #FF6600;
    }
    .word11 {
        color: #FF0000;
    }
101 }
@media print{
    #reftable {
```

```
border: 0;
padding:0;
margin: 0;
}
#reftable table {
width: 100%;
border: 0;
border-top: 1px solid black;
border-bottom: 1px solid black;
padding: 0;
}
#text {
border: 0;
margin: 0.1em;
}
.max {
text-decoration: underline;
}
#statistics {
color: black;
border: 1px solid black;
padding:1em;
margin: 0;
}
h1 {
text-align: left;
margin-top: 0;
padding: 0;
}
#footer {
border: 0;
margin: 0;
padding: 0;
}
}
```

The example text file: mytext.txt

```
"Dr. Brown, I've come to see you!" --- "What seems to be the 'problem'?"
It's seven o'clock already, and we only open 10-16. I have no time!"
--- "Well, I've got this brown spot..."
<The Doctor turns to her.>
```

(This example shows many problems for a Type-Token-Analyzer.)

The 'tokenized' text: mytext.txt

The tokens (with 'tagged' symbols):

'qdb'

Dr.

'space'

Brown

'pccm'

'space'

I've

'space'

come

'space'

to

'space'

see

'space'

you

'pcex'

'qdb'

'space'

'dem'

'space'

'qdb'

What

'space'

seems

'space'

to

'space'

be

'space'

the

'space'

'qsg'

problem

'qsg'

'pcqu'

'space'

It's

'space'

seven

'space'

o'clock
'space'
already
'pccm'
'space'
and
'space'
we
'space'
only
'space'
open
'space'
10
'dhp'
16
'pcfs'
'space'
I
'space'
have
'space'
no
'space'
time
'pcex'
'qdb'
'space'
'dem'
'space'
'qdb'
Well
'pccm'
'space'
I've
'space'
got
'space'
this
'space'
brown
'space'
spot

'pell'
'qdb'
'lbr'
'lt'
The
'space'
Doctor
'space'
turns
'space'
to
'space'
her
'pcfs'
'gt'
'pbr'
'pno'
This
'space'
example
'space'
shows
'space'
many
'space'
problems
'space'
for
'space'
a
'space'
Type
'dhp'
Token
'dhp'
Analyzer
'pcfs'
'pnc'

The result HTML: mytext.html

```
1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html>
  <head>
6     <title>Coiffeur:" Dr. Brown I've come to"</title>
     <link rel="stylesheet" href="coiffeur.css" type="text/css"
       media="screen, projection, print" />
  </head>

  <body>
11     <div id="header">
        <h1>Coiffeur &ndash; Results</h1>
     </div>
     <div id="reftable">
        <h2>Frequency of occurrence</h2>
16     <table>
        <tr>
            <td class="word0">
                1 or more
21            </td>
            <td class="word1">
                2 or more
            </td>
            <td class="word2">
26            4 or more
            </td>
            <td class="word3">
                8 or more
            </td>
            <td class="word4">
31            16 or more
            </td>
            <td class="word5">
                32 or more
            </td>
            <td class="word6">
36            64 or more
            </td>
            <td class="word7">
                128 or more
            </td>
            <td class="word8">
41            256 or more
            </td>
            <td class="word9">
            <td class="word10">
46            512 or more
            </td>
            <td class="word10">
                1024 or more
            </td>
```

```

51         <td class="word11">
           2048 or more
           </td>
         </tr>
       </table>
56 </div>
<div id="text">
  <h2>The processed Text</h2>
  <p>&quot;;<span class="word1" id="word1" title="2
    occurrences">Dr.</span>
  <span class="word1" id="word2" title="2
    occurrences">Brown</span>,
61 <span class="word1" id="word3" title="2
    occurrences">I</span><span class="word1 max" id="word4"
    title="3 occurrences">'ve</span>
  <span class="word0" id="word5" title="1
    occurrence">come</span>
  <span class="word1 max" id="word6" title="3
    occurrences">to</span>
  <span class="word0" id="word7" title="1
    occurrence">see</span>
  <span class="word0" id="word8" title="1
    occurrence">you</span>!&quot;;
66 &mdash;
  &quot;;<span class="word0" id="word9" title="1
    occurrence">What</span>
  <span class="word0" id="word10" title="1
    occurrence">seems</span>
  <span class="word1 max" id="word11" title="3
    occurrences">to</span>
  <span class="word0" id="word12" title="1
    occurrence">be</span>
71 <span class="word1" id="word13" title="2
    occurrences">the</span>
  '<span class="word0" id="word14" title="1
    occurrence">problem</span>'?
  <span class="word0" id="word15" title="1
    occurrence">It</span><span class="word0" id="word16"
    title="1 occurrence">'s</span>
  <span class="word0" id="word17" title="1
    occurrence">seven</span>
  <span class="word0" id="word18" title="1
    occurrence">o' clock</span>
76 <span class="word0" id="word19" title="1
    occurrence">already</span>,
  <span class="word0" id="word20" title="1
    occurrence">and</span>
  <span class="word0" id="word21" title="1
    occurrence">we</span>
  <span class="word0" id="word22" title="1
    occurrence">only</span>
  <span class="word0" id="word23" title="1
    occurrence">open</span>

```

```

81     <span class="word0" id="word24" title="1
        occurrence">10</span>-<span class="word0" id="word25"
            title="1 occurrence">16</span>.
    <span class="word0" id="word26" title="1 occurrence">I</span>
    <span class="word1 max" id="word27" title="3
        occurrences">have</span>
    <span class="word0" id="word28" title="1
        occurrence">no</span>
    <span class="word0" id="word29" title="1
        occurrence">time</span>!&quot;
86    &mdash;
    &quot;<span class="word0" id="word30" title="1
        occurrence">Well</span>,
    <span class="word1" id="word31" title="2
        occurrences">I</span><span class="word1 max" id="word32"
            title="3 occurrences">'ve</span>
    <span class="word0" id="word33" title="1
        occurrence">got</span>
    <span class="word1" id="word34" title="2
        occurrences">this</span>
91    <span class="word1" id="word35" title="2
        occurrences">brown</span>
    <span class="word0" id="word36" title="1
        occurrence">spot</span>&hellip;&quot;<br />&lt;<span
            class="word1" id="word37" title="2 occurrences">The</span>
    <span class="word1" id="word38" title="2
        occurrences">Doctor</span>
    <span class="word0" id="word39" title="1
        occurrence">turns</span>
    <span class="word1 max" id="word40" title="3
        occurrences">to</span>
96    <span class="word0" id="word41" title="1
        occurrence">her</span>.&gt;</p><p>( <span class="word1"
            id="word42" title="2 occurrences">This</span>
    <span class="word0" id="word43" title="1
        occurrence">example</span>
    <span class="word0" id="word44" title="1
        occurrence">shows</span>
    <span class="word0" id="word45" title="1
        occurrence">many</span>
    <span class="word0" id="word46" title="1
        occurrence">problems</span>
101    <span class="word0" id="word47" title="1
        occurrence">for</span>
    <span class="word0" id="word48" title="1 occurrence">a</span>
    <span class="word0" id="word49" title="1
        occurrence">Type</span>-<span class="word0" id="word50"
            title="1 occurrence">Token</span>-<span class="word0"
                id="word51" title="1 occurrence">Analyzer</span>.)
    </p>
    </div>
106    <div id="statistics">
        <h2>Statistics:</h2>

```

```
    <p>
    The text consists of 51 words.<br />
    The text has 42 types.<br />
111   The Type-Token Ratio is 0.823.<br />
    </p>
  </div>
  <div id="footer">
    <p>
116   Coiffeur &mdash; Making you pretty while analyzing your
        content.
    </p>
  </div>
</body>
</html>
```

Die vorliegende Arbeit “COIFFEUR — COMPUTATIONAL TEXT ANALYSIS IN PERL ” umfasst etwa 4000 Wörter¹⁰. Der Quelltext und die Beispieldateien sind dabei nicht berücksichtigt.

Ich versichere hiermit, dass ich sie selbstständig und ohne fremde Hilfe verfasst habe, keine anderen Quellen und Hilfsmittel als die angegeben benutzt habe und dass die Stellen der Arbeit, die anderen Werken — auch elektronischen Medien — dem Wortlaut oder Sinn nach entnommen wurden, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht worden sind.

Eugen Ruppert

¹⁰Wörterzählung mit ‘LaTeX word count’:
<http://folk.uio.no/einarro/Services/texcount.html>