Requirements for and design of a flexible tokenization system

Meike Aulbach
Cognitive Science, University of Osnabrück
maulbach@uos.de

Bachelor's thesis

Supervisors:
Stefan Evert, Bettina Schrader

December 22, 2006

Contents

I hereby confirm that I wrote this thesis independently and that I have not made use of any other resources or means than those indicated.

Hiermit bestätige ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Meike Aulbach, Frankfurt am Main, December 2006

Acknowledgements

The first and primary 'Thank you' goes to my advisors Stefan Evert and Bettina Schrader for numerous suggestions, discussions, explanations and last but not least: encouragement and cake/coffee/tea sessions. I'd also like to thank the IKW administrators for providing software, hardware and support during the evaluation phase. A big thanks goes to Edinburgh to Claire Grover for providing me with a beta version of LT-TTT2 and answering my questions.
Furthermore I wish to thank the people who had to stand my whining during the writing process of this thesis: My parents, Patrick Ullrich, Miriam Kreyenborg, Benjamin Balewski, Andreas Karsten and many more ...

1 Introduction

These days, the availability of an almost infinite amount of digital texts and the desire to make those to the subject of linguistic analysis has increased the demand for Natural Language Processing (NLP) tools. From an automatic annotation of texts to build lingustic corpora to more real-life oriented operations like text-to-speech synthesis, spell-checking or dictionary look-up, there is a wide range of applications.

One of the initial obligatory steps in NLP is *word segmentation*. Here, a sequence of characters is segmented into a sequence of words. Identified by Webster and Kit (1992), this type of preprocessing which digital texts undergo is called *tokenization*. During this process, the input text is split into useful linguistic units like words, numbers or punctuation marks which can be analyzed further in subsequent steps. Such an unit is called *token*. Figure 1 illustrates the task of a tokenizer. The input text consisting of a

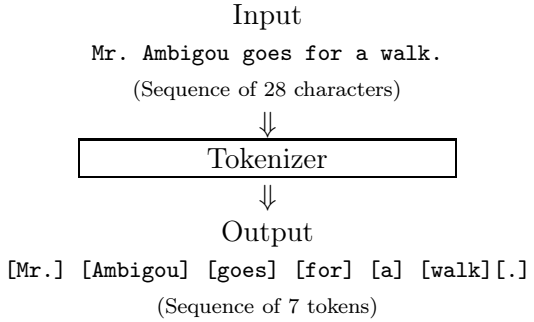

Figure 1: Input and output of a tokenizer

sequence of 28 characters including whitespaces is split by the tokenizer into a sequence of 7 tokens. The period becomes a separate token since it marks the end of a sentence. For an abbreviation like *Mr.*, the same character is part of the preceding token. Throughout this document tokenization is indicated by enclosing tokens in square brackets as in the example above. Of course this is not the only way to represent tokens, but it is useful for demonstration purposes.

For segmented Indo-European languages[1] the task of a tokenizer seems to be rather trivial: splitting on whitespace and punctuation should be sufficient, and many studies and applications in computational linguistics have used such a simple implementation. This naive approach already produces a reasonable performance, but there are many reported complications (Grefenstette and Tapanainen, 1994; Forst and Kaplan, 2006) leading to un-

[1]E.g. English or German. Those are opposed to non-segmented languages like Chinese or Japanese. For a more thorough introduction to this subject see Mikheev's chapter about text segmentation in the Oxford handbook of Linguistics (Mikheev, 2003)

expected and errorneous results[2]. Even more sophisticated and enhanced implementations of tokenizers fail to eliminate those errors completely. The error rate is pretty low overall, but it should not be underestimated since the tokenized text is the basis for all subsequent steps in NLP. According to Mikheev (2003), errors made during tokenization are likely to induce more errors at later stages of text processing. Also the recent work of Forst and Kaplan (2006) has pointed out the importance of precise preprocessing especially during tokenization.

So what are the biggest problems for tokenization? Is it possible to design a flexible and robust tokenizer which is suitable to be used as a preprocessor for different NLP tools with different requirements[3]? Yet the number of contributions to this area of computational linguistics is small, and until today these and more questions are still unresolved.

My thesis begins with providing a theoretical overview of tokenization and common approaches in section 2. I will discuss the requirements of a flexible tokenizer including a comprehensive list of concrete problem cases in section 3. Section 4 presents a test corpus which covers those problem cases and several existing tokenizers will be introduced and evaluated. The architecture of a flexible tokenization system—FTS is specified in section 5.

2 Tokenization basics

This section deals with the theoretical problems of tokenization. First, I discuss the difficulties involved in a token definition and present a few tokenization properties. Then I introduce a number of computational approaches to the word segmentation task.

2.1 Token definition

According to previous work in this area (Guo, 1997; Mikheev, 2003) it is generally accepted that tokens represent words—the basic linguistic units and 'primary building blocks in almost all linguistic theories'. But what exactly is a word? How should punctuation marks in a text be handled? Which special expressions should be retained?

There is no exact and universally valid definition of what a token is because this is entirely dependent on the subsequent utilization of the tokenized data. The following tokenization possibilites for the sentence '`Mr. Ambigou and Teddy aren't hungry.`' illustrate the different focus of varying applications. The matching of linguistic criteria and classification of special cases might be the main focus if the tokenization is the basis of a corpus for linguistic work:

[2]Problem cases for tokenization will be presented in section 3.2

[3]Some examples for different requirements will be given in section 2.1

```
[Mr.][][Ambigou][][and][][Teddy][][are][n't][][hungry][.]
```

Here, the tokenization represents a sequence of linguistic units along with word and sentence separators (spaces and period). The word `aren't` has been split into two tokens which reflect the contraction of two words `are` and `not`. However, if the tokenizer is used as a preprocessor for a part of speech tagger, such information might not play an important role:

```
[Mr.][Ambigou][and][Teddy][aren]['t][hungry][.]
```

The word separators are simply left out and the word `aren't` has been split at the apostrophe. For a tagger which has been trained by unsupverised learning, the most important factor is not exact linguistic matching, but the tokenization has to be consistent with the training data (which might use tokens in the style `[aren]['t]`). Hence, tokens should always be chosen and formatted in a supportive manner for a specific application.

There are many influential decisions which have to be reached in order to obtain a complete token definition which can serve as the basis for a consistent tokenization. After having observed a variety of special cases which call for a definition, I want to distinguish between two different levels. The first level, which I refer to as *linguistic level*, specifies the handling of cases like the one illustrated in Table 1. The table shows three reasonable possibilities

Original string	`aren't`
Type A	`[aren't]`
Type B	`[are][n't]`
Type C	`[are][not]`
Type E*	`[aren]['t]`
Type F*	`[aren'][t]`

Table 1: *Linguistic level* tokenization examples and counterexamples*

on the linguistic level to tokenize the string `aren't`. A tokenization of type A simply ignores the internal apostrophe and defines the whole composition as a single token. In contrast, a tokenization of type B and C reflects the fact that `aren't` is a contraction of the two individual words `are` and `not` with the apostrophe marking the omission of the letter `o`. Even though there is no explicit delimiter, the string is split into two separate tokens. Moreover, type C performs a transformation: `[n't]` $\Rightarrow$ `[not]`. However if one intuitively regards the apostrophe as a delimiter, tokenizations of type E* and F* have to be taken into account. Those might be reasonable from an intuitive point of view but are not linguistically justified. For this reason I don't count them as tokenization possibilities on the linguistic level, they should rather be regarded as counterexamples [4]

[4]From a combinatory viewpoint there is at least one more tokenization possibility:

The second level is what I call the *structural level*. Definitions required on the structural level are mostly caused by punctuation symbols (e.g.-"/). They carry important information about structure, i.e. signal the end of a sentence, dialogue turn, quotation, etc. Table 2 demonstrates possibilities for a structured tokenization of a complex German compound word: `"Deutsche Post"-Marken`[5]. Type A represents a simple tokenization where the complete

Original string	"Deutsche Post"-Marken
Type A	["Deutsche Post"-Marken]
Type B	["Deutsche Post"][-][Marken]
Type B′	[["Deutsche Post"][-][Marken]]
Type C	["][Deutsche] [Post]["][-][Marken]
Type C′	[["][Deutsche] [Post]["][-][Marken]]
Type D	["[Deutsche] [Post]"][-][Marken]
Type D′	[["[Deutsche] [Post]"][-][Marken]]

Table 2: *Structural level* tokenization examples

word is marked as an individual token. Type B is similarly simple, the compound word is split into three constitutent parts: proper name, hyphen and object. Another simple tokenization like type C separates words and punctuation marks into a sequence of individual tokens. Type D is the first tokenization accounting for the complex structure of the word. It still consists of a sequence of three constitutent parts like type B, but the internal structure of the first part has been hierarchically tokenized on a deeper level. This hierarchical tokenization is also represented by the tokenizations type B′-D′: based on the token sequences of type B-D respectively the additional outer square brackets reflect the important fact that the complex construct is basically a single token.

Before starting with implementing a computational identification of tokens, a lot of decisions on the linguistic and structural level have to be made to achieve a token definition which can be used in the implementation. As there is no universally applicable token definition, a specification should in the first place make sense in respect of the further utilization of the tokenized text.

2.2 Tokenization properties

To create terminology which allows for a more accurate description of a specific tokenization style, this section introduces a list of general tokenization

`[aren]['][t]`. If a linguistically motivated reason for such a tokenization can be given, it can be counted as a tokenization possibility on the linguistic level too.

[5]This expression can be translated roughly as "German Post"-stamps. *German Post* here is enclosed in quotes because it is the proper name of the German post company.

properties. This is intended as a neutral overview of various options for a tokenization, and does not argue in favour or against particular choices.

I. Degree of segmentation

(a) *Strong segmentation* means that the tokenizer tries to segment as much as possible. In this mode, very little or even no grouping of character sequences is performed.

(b) *Weak segmentation* in contrast produces fewer, but longer tokens than strong segmentation.

Example: `I'm buying heart-shaped cookies.`
(a) ⇒ `[I]['][m] [buying] [heart][-][shaped] [cookies][.]`
(b) ⇒ `[I'm] [buying] [heart-shaped] [cookies][.]`

II. Annotation[6]

(a) *With annotation* each token is labeled with information already available during the tokenization process, e.g. whether the characters marked are punctuation symbols, words or abbreviations. This can be done using a mark-up language, e.g. XML.

(b) *Without annotation*, no classification is performed.

Example: `Mr. Ambigou waves.`
(a) ⇒ `<a>Mr.</a> <w>Ambigou</w> <w>waves</w><se>.</se>` [7]
(b) ⇒ `[Mr.] [Ambigou] [waves][.]`

III. Transformation

(a) *With transformation*, the source text undergoes changes during tokenization. This might be the case when the tokenizer performs normalization processes on the text to achieve better matching of rules, or when the original token delimiter is omitted in the output text. But this also means that the tokenization process is irreversible[8].

(b) *Without transformation* on the other side, no information from the original text is lost since everything is kept (even the delimiting whitespaces (␣)) and only classified, e.g. marked up.

Example: `She's␣so␣glad.`
(a) ⇒ `[She][is][so][glad][.]`
(b) ⇒ `[She]['s][␣][so][␣][glad][.]`

[6]Throughout this document, the terms *mark-up* and *label* have been used as appropriate equivalents of *annotate*.

[7]XML style mark-up: `a` = abbreviation, `w`=word, `se`=sentence-ending. `<>` marks the opening and `</>` the closing of a token.

[8]For a corpus linguist, transformation of source text is a capital sin. Results of normalization should be annotated instead. However, for a NLP application, transformation in favour of a consistent tokenization might be useful.

IV. Hierarchy

(a) *A hierarchical tokenization* detects nested structures and complex tokens inside a text and marks tokens hierarchically by examining the opening and closing of special grouping characters (e.g. `"'`).

(b) *Without hierarchical tokenization* the text is just a stream of simple tokens, occuring grouping characters are either ignored or themselves marked as tokens.

Example: `"16-year-old"`
(a) ⇒ `[["][[16][-][year][-][old]]["]]`
(b) ⇒ `["][16-year-old]["]`

There is nothing like an universal tokenization style which is generally suitable as each application which relies on tokenized input requires other tokenization properties. Thus, the applied tokenization style has to be custom-tailored to the specific requirements of the subsequent utilization.

2.3 Approaches to tokenization

The aim of this section is to show miscellaneous concepts and fundamental types of computational approaches to the tokenization task. References are given to a few existing tokenizers which employ a specific approach.

2.3.1 Regular expressions

Regular expressions are a powerful function of many programming languages. They allow the specification of complex rules for character sequences, so-called *patterns*, which can be applied on a given input-text to generate *matches* which correspond to the specified patterns. For tokenization, regular expressions offer a tool to implement several operations. They can be used to define delimiting character sequences, for example the pattern `\s` matches whitespace characters like spaces, tabs and linebreaks. The pattern `\S+` matches at least one or more *non*-whitespace character[9]. Figure 2 gives an example how a tokenizer using this pattern would split the input.

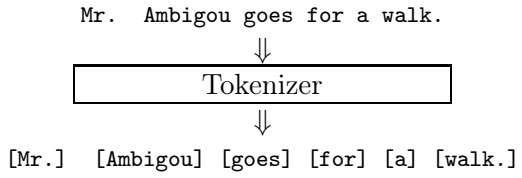

Figure 2: Tokenizer using the regular expression `\S+` to identify words

[9]The regular expressions used in the examples are compliant with the PCRE (Perl Compatible Regular Expressions) library. Url: `http://www.pcre.org/`

But regular expressions are not only useful for defining delimiters, they also support the identification of fixed expressions like dates or numbers. The extended pattern presented in Figure 3 does not only split at whitespace but additionally matches numbers as a single token even if they contain whitespace characters.

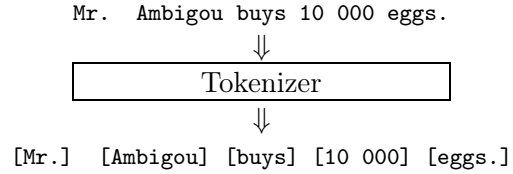

Figure 3: Tokenizer using the regular expression pattern `\d+(\s*\d+)*|\S+`

Almost all existing tokenizers are based on or at least partly employ the regular expression technique. Many NLP toolkits, e.g. the Stanford parser (Klein and Manning, 2003) or the N-Gram Statistics Package (NSP) (Pedersen, 2006), include a simple tokenizer using regular expressions. A tokenizer which works solely by specifiying regular expressions is jTokeniser with the `RegexTokeniser` Java class by Roberts (2006). Version 2.0 offers a graphical user interface (GUI) for a real-time supervision of the effects of a given regular expression by providing a window with the source text and another one with the result of the tokenization.

Although regular expressions offer a considerable amount of features, they have a big disadvantage: it requires a large manual effort to find rules, moreover those rules are mostly close-tailored to a specific language or text genre and thus not very flexible.

2.3.2 Heuristics

Heuristics are general rules or principles that are not derived from a well-developed theoretical understanding. An heuristic for tokenization would typically reflect linguistic intuitions. Trying to model those intuitions in a tokenizer can be done using a programming language—and of course with regular expressions.

Palmer and Hearst (1997) make use of heuristics for their sentence boundary disambiguation tool, SATZ. One of their heuristics says:

> If a word contains an internal period it is assumed to be an abbreviation.

Using this rule, words like *U.S., i.e., p.m., e.t.c.* can be recognized as abbreviations. Another heuristic discussed by Grefenstette and Tapanainen (1994) is based on the following idea:

> If a hyphen appears on the end of a line, the word before the hyphen can be rejoined with the first word on the following line.

But this cannot be safely assumed: If the respective word happens to be a compound word with an internal hyphen, the hyphen itself is a substantial part and would have been falsely removed.

Hence, developing heuristics requires taking care of possible side-effects. Moreover–just like all rule based approaches–they are very likely to be highly language and genre-dependent.

2.3.3 Modular Approaches

Modularization, the separation of tasks, is a concept rather than a way to realize a tokenizer. In a modular approach, the tokenization process is split into several modules, each of it responsible for a small sub-task.

In his Master's thesis about a database for storing and processing of text corpora, Zierl (1997) develops such a modular approach for tokenization. He defines the following sub-tasks: Remove whitespace characters, separate words tied together, cut punctuation marks, undo separations at line end, merge numbers and disambiguation of periods. His modules are realized with a script working with regular expressions, but of course in theory each operation could have been implemented with a different and suitable application or programming language.

Dale (1997) has proposed a framework for a tokenizer where tokenization is separated into three distinct modules. The *bundler* creates a sequence of simple tokens which is later transformed into complex tokens by the *compounder*. The tokenizer has an *interface* module which converts tokens into the form of input expected by the application who makes use of the tokenized data.

Also the LT-TTT (Grover et al., 2000) tokenizing software and its successor LT-TTT2 developed by the Language Technology Group at the University of Edinburgh are based on modules performing different operations which are tied together over a command pipeline.

The modular approach allows the independent optimization of different operations and the plug-in of appropriate tools during the tokenization process. The definition and ordering of the required modules however is still a challenging task.

2.3.4 Machine learning and statistics

In all of the previously presented approaches, it is the programmers task to discover rules for tokenization. But if already processed data is available (i.e. tokenized data), it could serve as training data for a machine learning algorithm trying to extract rules and patterns.

So far, such an approach has not been used for tokenization in segmented languages, only for the related task of sentence boundary disambiguation. The SATZ system (Palmer and Hearst, 1997) trains neural networks or decision trees with training data gathered from existing corpora and is quite successful in disambiguating sentence boundaries.

The advantage lies not only in the reduced effort, it also means more flexibility and portability since the usage is not restricted to a specific language or text-genre. However, a large amount of manually processed or validated data is required.

2.3.5 External knowledge base: Lexicon

Lexicon lookup is a common strategy especially for disambiguating the period following a word from a possible sentence ending or an abbreviation. When a period appears at the end of an abbreviation, it belongs to the abbreviation itself and does not mark the end of a sentence. Thus, detecting abbreviations is an important role for a tokenizer[10]. Their lookup from an external source is a straightforward solution approach.

Some tokenizers include internal abbreviation lists (Schmid, 1994). LT-TTT (Grover et al., 2000) employs the usage of a more comprehensive lexicon file.

For words contained in the lexicon, the lookup is a reliable strategy and it improves the success of the period disambiguation. But if a word is not present, consulting the lexicon is of little help. And even the most enhanced lexicon is probably incomplete. New abbreviations are invented and they might vary in their form between different domains. So even if the technique is generally useful, the maintenance of an exhaustive and up-to-date abbreviation lexicon requires large effort.

3 Requirements for a flexible tokenization system

After the previous initial insight into the basics of tokenization, this section is the first step towards an implementation of a flexible tokenization system. First, I discuss the criteria a flexible tokenizer should fulfil. Then I give a listing of occuring problem cases a tokenizer has to deal with during the tokenization process.

3.1 Flexibility criteria

Words are the basic units of any syntactic analysis, and virtually all NLP processing techniques operate on the level of words. In common practice, a more or less simple tokenizer for their identification is included in nearly

[10] Why abbreviations pose a concrete problem for a tokenizers is described in more thorough detail in section 3.2.1.

every NLP tool. But a simple tokenizer is prone to produce errors since tokenization is not as simple as often assumed. This will be shown in section 3.2. To avoid follow-up errors based on mistakes during tokenization, it would be advisable to delegate the tokenization task to an external but more sophisticated tokenizer.

Such a tokenizer which can be plugged in another NLP tool requires a high degree of flexibility and the behavior should be highly adaptable. The following list describes four key criteria a flexible tokenizer should fulfil:

I. Flexible token definition
   As presented in section 2.1, the exact token definition depends on the subsequent utilization of the tokenized data. Thus it should be possible to customize the definition of tokens and the style of tokenization.

II. Flexible language support
   Each language has speficic rules and special cases which play a role in the development of a tokenizer and its definition of tokens. A flexible tokenizer should be able to use the appropriate ruleset whenever available. The tokenizer might even be able to detect the language of the input text automatically.

III. Flexible input
   The ability to deal with different input begins with the *input source*, which might be e.g. standard input (STDIN), a file, a database or an url. Additionally there are different *input encodings* (ASCII, UTF-8, Latin1, ...) which have to be taken into account. Modern digital texts often occur in different *input formats* (Word-DOC, PDF, HTML, ...) of which the input is extracted by specialized converters to feed the tokenizer with plain text input.

IV. Flexible output
   The output format is a very critical part since it directly interconnects with the subsequent tool. The tokenizer should support numerous generic formats (e.g. XML, TEI, XCES, SGML ...) and offer the possibility to customize a specific output format. Also it is trivial to derive a specific output format if the output appears in a well-defined and standardized format with a complete specification contained in the documentation.

3.2 Concrete problem cases

During the tokenization process itself some concrete problems occur which need to be addressed. In contrast to the definition issues on the linguistic and structural level presented in section 2, the problem cases presented here

are on a *computational level*[11]. This covers common and actually occuring problems which need to be handled by the program code of a tokenizer. Ambiguities arise due to alternating utilization of a few character symbols. A flexible tokenizer should thus employ clever disambiguation strategies to minimize errorneous tokenization results.

3.2.1 Abbreviations

Many experts have identified abbreviations as the number one source of ambiguity in computational processing (Grefenstette and Tapanainen, 1994; Palmer and Hearst, 1997; Mikheev, 2003). The question whether a word with a trailing period is an abbreviation or the last word at the end of a sentence cannot be reliably answered without further exploration. However, a reasonable and common token definition requires an abbrevation to make up an individual token together with the trailing period, but a period marking the end of a sentence should not be tokenized together with the word before it. So, how can a computer program tell the difference between an abbrevation-period and an sentence-boundary-period? Take a look at the following character sequence:

```
Mr. Ambigou goes to visit Dr. Palpably.
He leaves early.
```

The reasonable common tokenization should look like this:

```
[Mr.] [Ambigou] [goes] [to] [visit] [Dr.] [Palpably][.]
[He] [leaves] [early][.]
```

A simple tokenizer using a regular expression approach (see section 2.3.1) for splitting at whitespace characters would result in two incorrect tokens: `[Palpably.]` and `[early.]`:

```
[Mr.] [Ambigou] [goes] [to] [visit] [Dr.] [Palpably.]
[He] [leaves] [early.]
```

While `[Mr.]` can be considered a reasonable token, `[early.]` clearly can't. But on the character level there is no difference between the two. Due to this reason a simple regular expression `\.\s[A-Z]` (also known as period–space–capital-letter) for detecting sentence boundaries would fail in our example and split the sequence into four sentences which are enclosed with brackets:

```
{Mr.} {Ambigou goes to visit Dr.} {Palpably.}
{He leaves early.}
```

[11] The usage of the term *computational* here does `not` imply any relation to the runtime of an algorithm

To solve this ambiguity, a detection of abbreviations is required. The usage of heuristics and lexical lookup (see section 2.3.2 and 2.3.5) is often suggested or applied as solution approach (Grefenstette and Tapanainen, 1994; Schmid, 1994; Mikheev, 2003). Palmer and Hearst (1997) make use of a machine learning approach (see section 2.3.4) for their sentence boundary disambiguation tool. But the issue is further complicated. Even if an abbrevation could be reliably detected, what if a sentence ends with an abbreviation?

```
Mr. Ambigou buys wheat, eggs, milk etc.
He wants to bake muffins.
```

In such cases, an explicit sentence ending period is omitted. The sentence boundary shares the trailing period with the abbreviation. Only syntactic and semantic analysis of the context could succeed in a correct disambiguation. And even if it was possible to reliably disambiguate sentence boundaries from abbreviation periods, how would a reasonable tokenization for the above example look like?

```
[Mr.] [Ambigou] [buys] [wheat][,] [eggs][,] [milk] [etc.]
[He] [wants] [to] [bake] [muffins][.]
```

In this tokenization example, the first sentence lacks an explicit sentence boundary token because the period has been tokenized together with the abbreviation `[etc.]`. This may cause problems if the tokenization is used as basis for NLP applications which expect sentence boundary tokens. Additional delimiter tokens could be added to the tokenization to indicate sentence boundaries explicitly.

```
[Mr.] [Ambigou] [buys] [wheat][,] [eggs][,] [milk] [etc.][%SB%]
[He] [wants] [to] [bake] [muffins][.][%SB%]
```

In this example, `[%SB%]` (sentence boundary) acts as such a delimiter token. However, sentence boundary disambiguation actually is considered the task of *sentence segmentation*. Admittedly, the responsibilities clearly overlap here. A tokenizer may also be concerned with the period disambiguation when it is required to tokenize abbreviations together with their period but sentence endings as separate tokens.

3.2.2 Punctuation marks I: Apostrophe and quote symbols

The apostrophe is a frequently appearing punctuation mark. It is mainly used to form the English possessive (`Mr. Ambigou's muffins`) and mark omissions (`don't`), but there are many exceptions where an apostrophe may occur in another context, e.g. in proper names such as persons or geographic locations.

Quote symbols are used as quotation marks to point out citations or spoken dialogues, moreover they are sometimes used to highlight an individual

word or a phrase. They have an enclosing character and appear in a pair of an opening and a closing symbol. This allows for a nested structure, i.e. a citation might appear within a spoken dialogue or vice versa.

In modern and extended character sets like Unicode or locales which also provide language specific characters many symbols explicitly designed to be used as quotation marks can be found. Table 3 demonstrates Unicode character symbols[12] for the apostrophe and a few quotation marks. However

Unicode Range	Punctuation mark	Example
U+0022	Quotation mark	"
U+0027	Apostrophe	'
U+2018	Left single quotation mark	‘
U+2019	Right single quotation mark	’
U+201A	Single low-9 quotation mark	‚
U+201B	Single high-reversed-9 quotation mark	‛
U+201C	Left double quotation mark	“
U+201D	Right double quotation mark	”
U+201E	Double low-9 quotation mark	„
U+201F	Double high-reversed-9 quotation mark	‟

Table 3: Unicode character symbols for quotes and apostrophe

the appropriate Unicode symbols are rarely used throughout the majority of digital documents. This might be due to several reasons: First, Unicode encoding might not be available or simply not set up correctly. Second, unicode symbols are not present on a standard computer keyboard layout and third it is hard to get somebody to change his or her habit. So people retain the usage of symbols of the simple and rather ancient ASCII [13] character set. Table 4 shows four ASCII symbols which are used to represent the apostrophe and quotation marks[14]. The tokenization of complex hierarchical words

Ascii hex value	Common symbol name	Example
0x22	Double quote	"
0x27	Single quote	'
0x60	Backquote, backtick or accent grave	`
0x2C	Comma	,

Table 4: ASCII symbols used as quotes and apostrophe

requires the detection of recursive structures and grouping. This means, the tokenizer has to carefully examine the appearance of quotation marks. Com-

[12] As defined in the Unicode Standard 4.1. Url: `http://www.unicode.org/charts/`

[13] American Standard Code for Information Interchange.

[14] Sometimes the acute accent (´) is also used, but it is not part of the ASCII character set.

plexity arises since the used character symbols and their combinations vary a lot over digital texts and printed media. The four symbols from the ASCII character set alone are used in a variety of combinations, some of which are attempts to imitate the 'real' quotation marks:

```
'Good night Teddy,' Mr. Ambigou said.
`Good night Teddy,' Mr. Ambigou said.
``Good night Teddy," Mr. Ambigou said.
,,Good night Teddy,`` Mr. Ambigou said.
```

The issue is further complicated because the ASCII symbols are not only used to denote quotation marks. They also frequently appear in other contexts in which they take up a different function:

- *To create the possessive form*
  `Mr. Ambigou's teddy, Chris' bike`
- *To mark the omission of letters*
  `Y'know what I'm thinkin' 'bout talkin' too much`
- *In proper or geographic names*
  `O'Reilly, Martha's Vineyard`
- *To highlight individual words*
  `The eleven 'in' countries informed their 'out' partners.`
- *As abbreviations for feet and inch*
  `7'4"` — 7 feet and 4 inches
- *As abbreviations for the angular measurement*
  `12°10'59"` — 12 degree, 10 arcminutes and 59 arcseconds

All of those examples may actually occur in a text together. The following text demonstrates the ambiguous use of the ASCII apostrophe symbol.

```
Mr.  Ambigou was flying to Martha's Vineyard to visit his
old friend Dr.  Chris O'Reilly.  Mr.  Ambigou's teddy was
sitting next to him.  `Y' know what I'm thinkin' 'bout
flyin' too high,' Mr.  Ambigou muttered.  `I'm getting
sick in here.' The plane reached a height of 32.105'76".
He took his cell phone and tried to dial Chris' number.
```

It should be obvious that the apostrophe symbol introduces a major source of ambiguity. This finding has been supported by the work of Fontenelle (2005). To correctly tokenize the above text and create a hierarchical tokenization, a flexible tokenizer is required to disambiguate the ASCII apostrophe symbol and recognize the applied quotation style to be able to observe the opening and closing of quotes, phrases etc. Furthermore not only the handling of ASCII symbols but also of quotation symbols of extended character sets, e.g. Unicode (see table 3) is required.

3.2.3 Punctuation marks II: Hyphen and dashes

Another class of ambiguous punctuation marks is represented through hyphen and dash symbols. Table 5 demonstrates the various symbols which exist in Unicode. The hyphen minus (U+002D) symbol is the only symbol which is also offered in the ASCII character set (hex value 0x2D). Unicode presents two individual symbols for the hyphen minus, the hyphen (U+2010) and the mathematical minus sign (U+2212). There is the figure dash (U+2012) to be used in a sequence of numbers, e.g. a phone number. The en dash (U+2013) indicates a closed range and the em dash (U+2014) is often used in pairs to offset parenthetical text or to mark a pause, like a break in thought.

Unicode Range	Punctuation mark	Example
U+002D	Hyphen minus	-
U+2010	Hyphen	-
U+2012	Figure dash	–
U+2013	En dash	–
U+2014	Em dash	—
U+2212	Minus sign	−

Table 5: Unicode character symbols for hyphen and dashes

However the ASCII character set offers only the hyphen minus symbol and due to this reason it has become one of the most ambiguous symbols for a tokenizer. Its usage is not restricted to hyphenated compounds, it also appears as replacement symbol for the different types of dashes. Because the dash symbol is slightly longer than the hyphen symbol, sometimes two or three successive hyphens (double- and triple-hyphen dash) are used. If those hyphen dashes would be used with surrounding whitespaces, this poses no problem for a tokenizer. But unfortunately there are cases, i.e. in most *Project Gutenberg*[15] texts, when surrounding whitespaces are omitted like in the following example[16]:

```
"Remarkable--most remarkable!" said Holmes, rising
and taking his hat.
```

Due to the lack of an explicit whitespace boundary, a whitespace tokenizer might wrongly create a token like `[Remarkable--most]`. This can easily be avoided by adding a rule which reflects the fact that the sequence of two or more successive hyphens acts as a delimiter. But there is no straightforward way to distinguish between a hyphen inside a hyphenated compound or a single-hyphen dash:

[15] The first producer of free electronic books, Url: `http://www.gutenberg.org`

[16] Excerpt taken from the Gutenberg e-book: *The Adventure of the Devil's Foot*—Sir Arthur Conan Doyle

```
Will you still need me, will you still feed me-when I'm
sixty-four?
```
[17]

On the character level there is no difference between the sequence `me-when` and `sixty-four`. So unless a tokenizer is capable of distinguishing between a 'real' hyphen and a single-hyphen dash, errors in tokenization are to be expected.

Line break hyphens

As already introduced in section 2.3.2, Grefenstette and Tapanainen (1994) have raised the subject of end-line hyphenation as problem for tokenization. If the hyphen symbol is used to separate a word at line breaks for layout purposes by a typesetting program, every hyphen appearing at the end of a line is an ambiguous character: Is it a substantial part of a hyphenated compound, or is it a line break hyphen breaking a word which is actually written without an internal hyphen? Many so-called *legacy* texts which have been processed by a typesetting program, i.e. PDF documents or the resulting texts of OCR[18] on scanned book pages, contain such line break hyphens. The following excerpt of *Moby Dick* by Herman Melville[19] has been manually formatted in a typesetting style:

```
Hoisting sail, it glided down the Acushnet river.  On one
side, New Bedford rose in terraces of streets, their ice-
covered trees all glittering in the clear, cold air. Huge
hills and mountains of casks on casks were piled upon her
wharves, and side by side the world-wandering whale ships
lay silent  and safely moored at last;  while from others
came a sound of carpenters and coopers, with blended noi-
ses of fires and forges to melt the pitch, all betokening
```

A simple whitespace tokenizer would create four incorrect tokens `[ice-]`, `[covered]`, `[noi-]` and `[ses]`. To avoid this, words with a hyphen just before a line-break could be rejoined with the next word on the following line. Following this rule, the tokens `[icecovered]` and `[noises]` are the result after the tokenization. While the last token is correct, the first one is not. Thus, a tokenizer is required to implement a strategy to avoid incorrect rejoining if it has to deal with legacy input text.

3.2.4 Special expressions

Special expressions are usually of a technical nature. Dates, sequences of numbers, formulae or e-mail addresses need to be handled correctly. They

[17] Lyrics ©1967 by 'The Beatles' on the album 'Sgt. Peppers Lonely Heart Club Band'
[18] Optical Character Recognition
[19] Also taken from the *Project Gutenberg* e-book archive.

are likely to contain special character symbols (i.e. slashes, mathematical symbols, brackets, ...) and there are many variations in their linguistic notation. The task to deal with special expressions is a challenging one, especially the varying standards across and within Indo-European languages demand a lot of attention.

It might be important to mark such an expression as an individual entity even though there are whitespace characters between the constitutent parts. For example a database might contain a lot of historic date references which vary naturally in their notation:

```
Friday, the 13th of November 1854 (...)
13.11.1854 (...)
13. November 1845 (...)
11/13/1845 (...)
```

Someone submitting a query to this database would have to take all possible notations into account when searching for entries with a specific date. But if date expressions are correctly recognized they could internally be marked up with a normalized date which can then be used in the search query. Another difficulty is presented through numerical expressions. For example digits with decimal commas or telephone numbers are often split with a strong degree of segmentation by a tokenizer:

```
[Phone] [(][0541][)][34421][-][0]
[9][,][000][,][000] [people]
```

However the recognition of special expressions might be considered to be of no concern for a tokenizer. But at least it should be ensured that such entities are not strongly segmented into units which are hard to put back together in later stages of processing.

Thus, it might be useful if a tokenizer is able to detect such entities and tokenizes them appropriately.

4 Evaluation of existing tokenizers

The evaluation of existing tokenizers provides an overview about the state of art. It serves the purpose to discover strengths and weaknesses of different approaches. Along with Habert et al. (1998) I agree that is not trivial to do a comparative evaluation since diverging tokenization choices lead to sharply different tokenized texts. Hence, there is no one and only gold standard each tokenization could be compared against. This evaluation rather aims at examining to which extent the evaluated tokenizers fulfil the flexibility criteria presented in section 3.1 and how they handle the concrete problem cases presented in section 3.2.

First, the experimental setup is described in detail. Subsequently, the evaluated tokenizers are presented together with the evaluation results. Then a final summary of the results is given.

4.1 Experimental setup

4.1.1 Test environment

All evaluation processes took place on an 64-bit x86 computer platform with a Linux operating system (Kernel version 2.6) of the University of Osnabrück (`quickie.cogsci.uos.de`). An individual directory structure with all relevant files has been created on the file system. The latest versions of the participating tokenizers were installed and used for the evaluation.

4.1.2 Test corpus

The concrete problem cases on the computational level described in section 3.2 were manually compiled into a test corpus. The test corpus consists of a collection of excerpts considered problematic for a tokenizer. It contains only ASCII character symbols and has a maximum of 65 characters per line. Because the ASCII encoding is the only encoding supported by all tokenizers in the evaluation, the text is stored as an ASCII text file. The full text of the test corpus file is attached to this thesis in Appendix A.1.

To obtain a text which might actually occur in real-life application, the contents have not been made up but were collected from different sources of existing digital texts. Included were short excerpts of classics available as digital ASCII text documents from the *Project Gutenberg* collection. The excerpts have been chosen from the following list of books: *The Hound of the Baskervilles* (Sir Arthur Conan Doyle), *Ullyses* (James Joyce), *Dracula* (Bram Stoker), *Alice's Adventures in Wonderland* (Lewis Carroll) and *Thus spake Zarathustra* (Friedrich Wilhelm Nietzsche). A part of *The full Project Gutenberg Licence* has also been included. Additionally, a few paragraphs with input text suggested by others who have contributed to tokenization have been added. Short extracts of the Wallstreet journal and the MEDLINE corpus were provided in a tutorial for the tokenizer of the Natural Lanugage Toolkit (NLTK). Grover et al. (2000) discuss the tokenization of bibliographic material, a few references discussed in their article have been included. They are originally from a reference list provided for the BibEdit project (Matheson and Dale, 1993). A few lines of a test text which is distributed with LT-TTT (Grover et al., 2000) and a last additional line with an own example for the ambiguous abbreviated feet and inch notation have been added.

Random samples of the concrete problem cases to be checked during the evaluation were chosen. In the following, a categorized overview about the random samples is given.

Abbreviations

1. *Common abbreviations with trailing period*
   `Mr. | rev. | S. | Corp. | etc.`

2. *Abbreviations with internal periods*
   `M.R.C.S. | C.C.H.`

Quotation mark symbols

1. *Opening symbol ", closing symbol "*
   `"Penang lawyer." | "Superman"`

2. *Opening symbol `, closing symbol '*
   `` `and what is the use of a book,' ``

3. *Posessives and contractions*
   `Foundation's | It's | 1992's | I'm`

4. *Abbreviated feet and inch notation*
   `10"6'`

Hyphen symbol

1. *Hyphenated compounds in running text*
   `old-fashioned | by-the-bye | Friday-the-13th`

2. *Hyphenated compounds broken at line end*
   `hearth-%LINEBREAK%rug`[20]

3. *Line break hyphens*
   `inte-%LINEBREAK%gration`

4. *Double-hyphen dash*
   `carry--dignified | Bistritz.--Left`

Special expressions

1. *Time expressions*
   `1884 | 3 May | 8:35 P.M. | 1st May | July 1995`

2. *Contact data*
   `809 North 1500 West | UT 84116 | (801) 596-1887`
   `business@pglaf.org | http://pglaf.org`

3. *Medical expressions and formulae*
   `alpha-galactosyl-1,4-beta-galactosyl-specific`
   `4.53 +/- 0.15% | 8.16 +/- 0.23%`

[20] `%LINEBREAK%` is used here to indicate the position of the line break

4. *References*
   ```
   Cabelli, H.F., 1990. Promoter occlusion: transcription
   through a promoter may inhibit its activity.
   Cell 29 939-944.
   ```

5. *Monetary expressions*
   ```
   $23.625 | $102 million | 34 cents
   ```

4.1.3 Test procedure

Each tokenizer was evaluated individually. The following chart provides a step-by-step listing of the test procedure:

1. *Consult the documentation:* See if any help files are present and read it to be able to answer at least the following questions:
   (a) How is the tokenizer called on the commandline?
   (b) Which configuration options are present?

2. *Checklist flexibility criteria*: Determine (from the documentation and help files) to what extent the tokenizer fulfils the four criteria requirements for a flexible tokenizer described in section 3.1.

3. *Input format check:* Feed the tokenizer with Latin1 and UTF-8 encoded input files containing extended characters[21] and examine the results.

4. *Tokenization:* Feed the tokenizer with the test corpus input text, write the output to a result file and store it in the test environment.

5. *Checklist random samples:* Analyze the output in the result file and note how the tokenizer has dealt with the random samples of concrete problem cases which were encoded in the test corpus.

6. *Write a test protocol* which lists the tokenization of the chosen random samples.

4.2 Results

The following paragraphs subsequently introduce five different tokenizers with a description of their functionality, configuration options and a summary of the evaluation results. The test protocols of the tokenization of the random samples are attached in Appendix A.2 - A.6.

[21] The German word 'Grüßli' was used

4.2.1 String::Tokenizer

To show the dangerous consequences of reducing tokenization to a simple 'split-at-whitespace' task, this approach should be included in the evaluation. To represent this strategy, a Perl script using the CPAN[22] module String::Tokenizer (Little, 2004) has been chosen. The module performs tokenization by using a regular expression matching whitespace characters and is thus perfectly suited for the demonstration. To call the module and read the input from a file given with a commandline argument, a simple wrapper Perl script (see Figure 4) has been written.

```
#!/usr/bin/perl
#
# Usage: ./S-Tok.pl <input-file>

# Load module
use String::Tokenizer;

# Open file from first argument and read it in an array.
open(INPUT, $ARGV[0]) || die("Could not open $ARGV[0]\n");
my @input = <INPUT>;
close(INPUT);

# Remove line-breaks and put everything into $input string
my $input = join ' ', map /(.*)/, @input;

# Create the tokenizer and tokenize the $input string
my $tokenizer = String::Tokenizer->new();
$tokenizer->tokenize($input);

# print tokens enclosed in square brackets
print "[", join "][" => $tokenizer->getTokens(), "]";
```

Figure 4: S-Tok.pl – Perl script using the String::Tokenizer module

To adapt the module's behavior, String::Tokenizer offers two methods. By default, the module collects only tokens and omits the delimiting whitespace characters in the result. Thus, if whitespace characters should be retained, this can be done setting the constant `RETAIN_WHITESPACE` with the method `handleWhitespace()`. The other method, `setDelimiter()`, allows to specify a list of characters to be used as delimiters instead of whitespaces. Besides the possibility to define delimiters, the token definition cannot be adapted further. The input format is flexible to the degree Perl is able to deal with different character encodings. Latin1 and UTF-8 were both tested

[22] Comprehensive Perl Archive Network - Url: `http://www.cpan.org`

successfully. As the module only provides a list of tokens, the formatting of the output is completely left to the programmer.

The analysis of the tokenization results of String::Tokenizer has shown exactly the behavior which was expected of a whitespace tokenizer. Due to the lack of any intelligent rules, correct tokenization in some cases can be regarded as pure matter of luck. Abbreviations were acceptably classified (`[Mr.]`, `[rev.]`) only when they were surrounded by whitespaces to the left and right. If that was not the case, additional punctuation symbols were added to the token (`[etc.,]`, `[C.C.H.,"]`). Quotation marks were also attached to the previous or next word (`["Penang]`, `[`and]`). For possessives, contractions and the abbreviated feet and inch notation, String::Tokenizer produced acceptable tokens (`[Foundation's]`, `[It's]`, `[10'6"]`) again only by chance, an error was made (`["I'm]`). Hyphenated compounds in running text were tokenized as individual token if surrounded by whitespaces (`[old-fashioned]`). But when it was broken at linebreak, a hyphenated compound was tokenized the same way as a word with an internal linebreak hyphen: the hyphen was attached to the previous word (`[hearth-][rug]`, `[inte-][gration]`). The double-hyphen dash was not separated from the surrounding words either (`[carry--dignified,]`).

4.2.2 ET - Efficient Tokenizer

This 'Efficient Tokenizer' (Covington, 2003) has been written with a declarative logical progamming language: Prolog. It is a finite-state transition network with a two-character lookahead during recursion through the input string. It splits the text into three different kinds of tokens, which are tagged appropriately: words, numbers and special characters. Those are represented as prolog facts where each fact is considered an individual token in form of a list of single characters:

`Hello world!` ⇒ `[w([h,e,l,l,o]), w([w,o,r,l,d]), s(['!'])]`

To load the contents of a text file into Prolog, the wrapper shell script from Figure 5 has been used.

Without directly changing the source code it is not possible to adapt the token definition ET uses. It provides no support for other languages, moreover it is intended to be used on English text only as the implementation contains special rules for handling apostrophes based on English morphology. The formatting of the tokenization consists of a list of Prolog facts as described above. ET can handle UTF-8 as input format. For Latin1 encoded input text, extended characters appear as broken in the output.

The tokenization results of ET exhibit a very strong segmentation and the resulting tokens are written completely in lowercase letters. Every punctuation symbol is tokenized as individual token, the other tokens consist of pure letter or number sequences. Abbreviations consist of a sequence of letter

```
#!/bin/sh

PL=/usr/bin/pl
ET='et.pl'

if [ "$#" -ne "1" ]; then
   echo "Usage: $0 <file>"
   exit 1
fi

$PL -f $ET <<EOF
tokenize_file('$1',X),print(X).
EOF
```

Figure 5: et.sh – reads the contents of a text file into ET

and punctuation tokens (`[mr][.]`, `[m][.][r][.][c][.][s][.]`). Quotation marks are separated accordingly (`['"'][superman]['"']`). The apostrophe in possessives and contractions is removed (`[foundation][s]`, `[it][s]`). Hyphenated compounds in running text, at line break and linebreak hyphens are tokenized consistently (`[old][-][fashioned]`, `[inte][-][gration]`). The special time expression 1st May has been split further (`[1][st][may]`).

4.2.3 Cass tokenizer

The next tokenizer to be evaluated is part of a larger NLP Tool: Cass by Abney (2006) is a partial parser written in C including a separate program for tokenization. According to the manpage of the documentation, `token` breaks running text into tokens based on the following token definition:

> A token is approximately a stretch of alphanumeric characters, possibly with an embedded apostrophe. Each non-alphanumeric character becomes a token to itself. All control characters are replaced with the special token \control, and numbers are replaced with the special token \num.

The `token` program takes no commandline options, it reads input from `STDIN` and prints it to `STDOUT`, so the program call is pretty straightforward and simple:

```
token < input-file > output-file
```

Cass provides no possibility to adapt the token definition used by the `token` program as the output is intended to be processed by the partial parser only. Hence, the output format is prepared as a 3-column list, each row representing a token. Here is an example for the tokenization of the string `Hello world!`:

```
hello   Hello   \s
world   world   -
!       !       \n
```

The first column represents the token, the second column contains the token in the original and capitalized form and the third column[23] denotes the whitespace character appearing after the token. Only ASCII input text is correctly handled by Cass, the octal and decimal representation is printed for extended characters from the Latin1 or UTF-8 encoding.

The tokenization results of the Cass tokenizer are not as straightforward as for the previous two tokenizers. The period is always tokenized as an invididual token, except when it appears in a numerical context. Thus, abbreviations are tokenized without their period (`[mr][.]`, `[rev][.]`) but numbers with (`[0.15]`, `[1884.]`). Apostrophes occuring in a letter context are kept (`[Foundation's]`, `[It's]`, `[I'm]`) but separated in numerical context (`[1992]['][s]`). All other punctuation symbols are tokenized individually (`[+][/][-]`, `[10]["][6][']`), also the hyphenated words (`[old][-][fashioned]`, `[inte][-][gration]`). However in a numerical context, the hyphen has been split of the previous word and tokenized together with the next word (`[596][-1887]`, `[939][-944]`). The double-hyphen dash has been bundled (`[--]`).

4.2.4 TreeTagger tokenizer

Along with the IMS TreeTagger by Schmid (1994), a program doing basic tokenization is provided. It is run automatically by the shell wrapper scripts supplied with the tagger, but can also be used to run on its own. The tokenizing process is done by the program `separate-punctuation` written in C. It uses an external file containing a list of word forms ending with a period. To call the tokenizer together with the correct path to the abbreviation file, a modified wrapper script (see Figure 6) has been written [24].

In contrast to the previously demonstrated tokenizers, the TreeTagger's `separate-punctuation` offers a few configuration options on the commandline. The token definition can be influenced by adding the `+s` option for separate punctuation. As this is a very short description, a few tests have shown that enabling this option leads to a stronger segmentation of a sequence of punctuation symbols[25]. The output format can be adapted using the flag `+m` to print end of sentence and end of paragraph markers, the used marker can be specified with `-d` and `-D` [26]. The flag `+1` defines that the

[23] `\s` = space, `\n` = newline, `-` = no whitespace

[24] The original version of the wrapper script used to call the UNIX tool `sed` after tokenization to separate clitics (as in *I've*, *don't* etc.). This has been left out since it is not important for the evaluation.

[25] The sequence `!?!` was tokenized `[!?!]` without and `[!][?][!]` with the option `+s`.

[26] In the modified wrapper script the option `+m` has been added for the evaluation.

```
#!/bin/sh

BIN=/home/student/m/maulbach/evaluation/tokenizers/TreeTagger/bin
LIB=/home/student/m/maulbach/evaluation/tokenizers/TreeTagger/lib

TOKENIZER=${BIN}/separate-punctuation
ABBR_LIST=${LIB}/english-abbreviations

# put all on one line
cat $* |
# do tokenization
$TOKENIZER +1 +s +m +l $ABBR_LIST
```

Figure 6: tree-tokenizer.sh – shell script to call `separate-punctuation`

output consists of one word per line. The TreeTagger characterizes itself as language independent. For the tokenizer alone, this is only true insofar as there is the possibility to use a language specific abbreviation file. The standard TreeTagger distribution includes abbreviation files for English, German and Spanish. The TreeTagger could be confirmed to correctly handle extended characters from both Latin1 and UTF-8 encoding.

The evaluation of the tokenization results of `separate-punctuation` which makes use of an abbreviation list has shown what was predicted: this strategy is reliable only for words contained in the abbreviation list (`[Mr.]`, `[Corp.]`, `[etc.]`). The only tokenization error which occurred was `[rev][.]` – this abbreviation was included in the case-sensitive abbreviation list with the first letter in uppercase only (*Rev.*). Even though single letter abbreviations (`[S.]`) and abbreviations with internal periods (`[M.R.C.S.]`) were not included in the abbreviation list, they were also handled appropriately, maybe due to additional usage of heuristics. All quotation marks were separated from the following and preceeding tokens (`["][Superman]["]`). The apostrophe is kept in possessives and contractions (`[Foundation's]`, `[I'm]`). However, for the abbreviated feet and inch notation this leads to a weird result (`[10'6]["]`). The results for handling the hyphen symbol pretty much resemble the results of String::Tokenizer. In running text, hyphenated compounds are tokenized appropriately (`[old-fashioned]`) but broken at linebreak, the hyphen is tokenized together with the preceeding word (`[hearth-][rug]`, `[inte-][gration]`). The affinity to a whitespace tokenizer is demonstrated further by the handling of the double-hyphen dash (`[carry-dignified]`).

4.2.5 LT-TTT2

This section presents a rule-based and modular software for tokenization: LT-TTT2, a text tokenization toolkit (Grover and Tobin, 2006) is the successor of LT-TTT (Grover et al., 2000). It is realized partly with tools provided by the LT XML2 toolset (Thompson et al., 1997) which has also been developed by the Language Technology Group of the University of Edinburgh. For tokenization, LT-TTT2 uses a pipeline of sequential calls to two components:

- `lxtransduce` is the core component of LT-TTT2 and consists of a general purpose transducer which reads rules from a grammar stored in a file and rewrites the input according to those rules.
- `lxreplace` is part of the LT XML2 toolset and is able to perform replacements in an XML document

The LT-TTT2 distribution includes the wrapper script `runplain-alt` (see Figure 7) which ties the different calls together.

```
# runplain-alt
# [...]
bin/lxplain2xml |
bin/lxtransduce -q TEXT gram/char/paras.gr |
bin/lxtransduce -q p gram/char/pretok.gr |
bin/lxtransduce -q p gram/xml/tok.gr |
bin/lxtransduce -q p -l lex=lex/numbers.lex gram/xml/hyph.gr |
bin/lxreplace -q w/w |
bin/lxtransduce -q p gram/xml/sents.gr |
bin/lxreplace -q cg |
bin/lxtransduce -q s -l lex=lex/numbers.lex gram/xml/numbers.gr |
bin/lxreplace -q "phr/phr" |
bin/lxreplace -q "phr[w][count(node())=1]" -t "&children;" |
bin/lxtransduce -q s -l lex=lex/currency.lex gram/xml/numex.gr |
bin/lxreplace -q "phr[not(@c='cd') and not(@c='yrrange')]" |
bin/lxtransduce -q s -l lex=lex/timex.lex -l \
     numblex=lex/numbers.lex gram/xml/timex.gr |
bin/lxreplace -q phr
```

Figure 7: `runplain-alt` – wrapper script distributed with LT-TTT2

The 'XML processing paradigm' of LT-TTT2 requires input text to be formatted in XML. However using the `lxplain2xml` tool, plain text can be converted to XML easily [27]. The token definition is highly adaptable but complex: the rules for tokenization are defined in external and customizable

[27] In fact, the first call of the wrapper script goes to `lxplain2xml`.

grammar files. Additionally, extendable lexicon files for special expressions (numbers, currencies and time expressions) are included. LT-TTT2 does not transform the input text, it just adds XML mark-up. Thus the resulting output format is XML, but again, conversion to other formats should pose no problem. The following example demonstrates the XML annotation style of LT-TTT2 for the input string `Hello world!`:

```
<TEXT>
<p><s><w c="w">Hello</w> <w c="w">world</w><w c=".">!</w></s></p>
</TEXT>
```

The tags `<p>` and `<s>` enclose paragraphs and sentences. The tag `<w>` is used for tagging tokens, the attribute `c="w"` marks a token as a word and `c="."` marks a token as a punctuation symbol. The support of UTF-8 encoded input has been tested but the extended characters could not be processed and were marked-up with the label *what*. Latin1 encoded input text resulted in an error message[28].

Being the most sophisticated tokenizer in the evaluation, LT-TTT2 has raised and met high expectations. The chosen abbreviations were tokenized in a rather uncommon style: the trailing period was stripped from the prevailing word (`[Mr][.]`, `[etc][.]`, `[M.R.C.S][.]`, `[C.C.H][.]`). This allows for a consistent tokenization; if the period was supposed to be part of the previous word because if it is an abbreviation it can still be rejoined in a later stage of processing along with annotating the token with the label *abbr*. Quotation marks are tokenized separately, opening and closing quotation marks are respectively annotated with the labels *lquote* and *rquote*. Possessives were tokenized with separated clitics (`[Foundation]['s]`, `[1992]['s]`) and tagged with the label *aposs*. The ambiguous `[It]['s]` (here standing for `It is`) was errorneously tagged as aposs too, even though it is a contraction. Those were tokenized the same way (`[I]['m]`) but tagged as word. In the context of the abbreviated feet and inch notation, the quotation marks could not be disambiguated: For `10'6"` the tokenization was `[10]['][6]["]`, but the symbol `'` was tagged with the label *quote* and the symbol `"` with the label *rquote*. Hyphenated compounds in running text were tagged with the label *hyw* for hyphen word and tokenized as an individual token (`[old-fashioned]`). The hyphens appearing at linebreak were tokenized separately (`[hearth][-][rug]`, `[inte][-][gration]`). The double-hyphen dash has been neatly separated from the surrounding words and punctuation (`[carry][--][dignified]`, `[Bistritz][.][--][Left]`). LT-TTT2 did a very good job in the recognition of special expressions. Time and monetary expressions were reliably detected and appropriately classified. Because the

[28] After the evaluation, the authors of LT-TTT2 have confirmed to me in a personal e-mail note that Latin1 support is present in more recent versions of the toolkit. The input encoding has been parameterised into the command line call.

rich annotation is hard to describe in a simplified way, here is an example of the XML output for a few time- and monetary expressions:

- `3 May` ⇒ `<timex type="date"><w c="cd">3</w> <w c="w">May</w></timex>`
- `8:35 P.M.` ⇒ `<timex type="time"><w c="cd">8:35</w> <w c="abbr">P.M.</w></timex>`
- `$102 million` ⇒ `<numex type="money"><w c="sym">$</w> <w c="cd">102</w><w c="w">million</w></numex>`

The *timex* tag distinguishes between the types *date* and *time* and encloses all tokens belonging to a time expression. For the other categories of special expressions, no annotation has been performed but the tokenization was pretty much acceptable for references, medical expressions and formulae.

4.3 Summary

4.3.1 Evaluation of flexibility criteria

Along with an detailed description about the results of the evaluation of flexibility criteria in this section, an overview is given in table 6.

Flexibility crit.	Str::Tok	ET	Cass	TreeTagger	LT-TTT2
Token definition	—	—	—	Option for punctuation	Customizable grammar files
Language	—	—	— —	Language spec. abbr. list	In theory extendable
Input	ASCII, UTF-8, Latin1	ASCII, UTF-8	ASCII	ASCII, UTF-8, Latin1	ASCII, UTF-8
Output	—	—	—	Options for slight changes	—

Table 6: Overview about results for flexibility criteria

The assessment of the flexibility with regard to token definition has yielded poor results. Of the tested tokenizers, only LT-TTT2 allows for a flexible token definition through the utilization of customizable grammar files. The drawback here is that it requires expertise and a large manual effort to build appropriate grammar rules.

Only TreeTagger explicitly claims to be language independent. Because its tokenizer this does not include language specific tokenization rules, only the usage of a language specific abbreviation file can be counted towards the fulfilment of the language flexibility criterion. Even though LT-TTT2

provides grammars and lexicon files for English only, it would in principle be possible to extend the approach to support other languages.

All of the presented tokenizers are able to handle ASCII encoded input, and except of Cass, they are moreover able to deal with Unicode (UTF-8 encoding) input texts. However only TreeTagger and String::Tokenizer (or to be exact, Perl) were able to process input which was encoded with Latin1.

The output format of the evaluated tokenizers varies a lot. ET and the String::Tokenizer module do not print any output at all, the tokens can be either utilized further by the program or the output needs to be formatted and printed by an appropriate output function. Cass, being part of a larger NLP tool, prints specialized columns of tokens; The TreeTagger tokenizer simply prints a sequence of tokens; LT-TTT2 with its rich annotation style prepares the tokenized text in XML. While any output format can in fact be adapted as desired by programming a simple converter, none of the tokenizers offers actually flexible output.

4.3.2 Evaluation of concrete problem cases

An overview about the evaluation results concerning the handling of concrete problem cases is presented in table 7. I have used specific symbols for a simplified result rating. The symbol ✗ means the tokenizer has produced an unacceptable result. When the result is neither really bad nor really good, the symbol — indicates a neutral rating. With the symbol ✓, a good result is specified. In the following I give a more detailed description about the

Problem case	Str::Tok	ET	Cass	TreeTagger	LT-TTT2
Abbreviations	✗	—	—	✓	—
Quotation marks	✗	—	—	—	✓
Hyphenated compounds	✗	—	—	✓	✓
Line break hyphen	✗	✗	✗	✗	✗
Special expressions	—	—	—	—	✓

Table 7: Overview about results of concrete problem cases

results of the evaluation for each concrete problem case.

As predicted, abbreviation periods pose a problem when they should be tokenized together with the abbreviation but sentence boundary periods should be tokenized individually. The only tokenizer in the evaluation attempting such a tokenization, the TreeTagger tokenizer, is successful only when it is able to detect an abbrevation from the abbrevation list or simple heuristics. The other tokenizers avoided this problem by employing a tokenization style of strong segmentation, where punctuation symbols are tokenized separately from letters (ET, Cass tokenizer). For words not yet

recognized as abbreviations, LT-TTT2 tokenizes the trailing period individually (`[Mr][.]`) to leave the decision and a possible combining of the period over to another stage of processing.

Quotation marks are simply tokenized separately without regard to their structural properties by most tokenizers. Only LT-TTT2 attempts to characterize opening and closing quotation marks as lquote and rquote.

None of the tested tokenizers cared about rejoining of words separated with a linebreak hyphen as it may occur in legacy text documents. At least some tokenizers were able to separate the double-hyphen dash from the surrounding words. However, considering the strong segmentation style of ET and Cass which pretty much separate any punctuation this result cannot be credited. The only remarkable detection of the double-hyphen dash is performed by LT-TTT2.

The recognition of special expressions was incorporated into the tokenization process only by LT-TTT2. As observed, the other tokenizers simply tokenized special expressions according to their individual rule implementation.

Especially the poor overall quality of the results of String::Tokenizer are cause for concern because a lot of NLP toolkits make use of a simple tokenizer. This accentuates the need for a flexible tokenization system which can be used to replace simple implementations of tokenizers.

5 Design of a flexible tokenization system

This section describes the architecture for a Flexible Tokenization System (FTS) based on the presented flexibility criteria (see section 3.1). It is considered to be a sketch, thus concrete implementation issues are not discussed.

This section presents the architecture, external sources and the modular structure of FTS. Finally a summary is given.

5.1 Architecture

The architecture consists of four modules. A flowchart of the architecture is demonstrated in figure 8. First, the input is converted to Unicode by the *input converter*. A first and strong segmentation of the characters is performed by the *low level tokenizer*. This data is then passed on to the complex *high level tokenizer* which adapts the sequence of tokens according to a given token definition and based on many sub-grammars. During this stage, the high level tokenizer also integrates external sources. In the last step, the internal representation of tokens is passed on to the *output formatter* which cares for the desired output format. Each module will be discussed in more thorough detail in section 5.3.

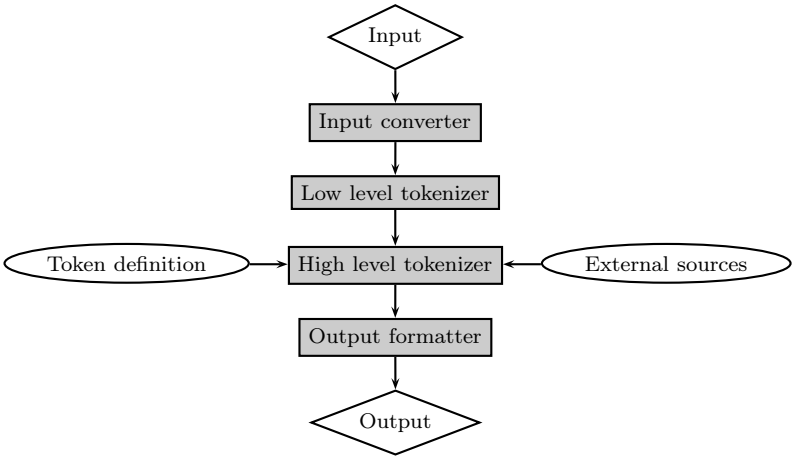

Figure 8: Flowchart demonstrating a sketch of the basic FTS architecture

5.2 External sources

Like the TreeTagger tokenizer which makes use of an abbreviation list or LT-TTT2 which includes several lexicon files, FTS is able to integrate gathered information from external sources. Such information is stored in an appropriate lexicon structure in separate Unicode files. A word lexicon and an abbreviation lexicon in each language FTS supports should be provided. Both the word and the abbreviation lexicon can improve the disambiguation of periods and line break hyphens.

5.3 Modular structure

5.3.1 Input converter

FTS is able to deal with different encodings of the input text. Because the Unicode encoding (UTF-8) offers interesting features, it has been chosen to be the character set used during processing. Unicode does not only provide characters for all languages, the characters contain meta-information representing their function. The so-called Unicode properties come in useful when defining regular expressions. Table 8 demonstrates a few standard Unicode properties [29]. The short description consisting of a single uppercase letter is a basic property whereas an attached lowercase letter signals a certain subproperty. For example `L` denotes the basic property Letter, containing both uppercase (`Lu`) and lowercase (`Ll`) letters. If a digital text is set up correctly with the appropriate Unicode characters, their properties are a benefit for

[29] Taken from the Perl documentation for perlunicode - Unicode support in Perl
Url: `http://perldoc.perl.org/perlunicode.html`

Short description	Long description
L	Letter
Lu	UppercaseLetter
Ll	LowercaseLetter
N	Number
Nd	DecimalNumber
Nl	LetterNumber
P	Punctuation
Pd	DashPunctuation
Ps	OpenPunctuation
Pe	ClosePunctuation
Po	OtherPunctuation
S	Symbol
Sm	MathSymbol
Sc	CurrencySymbol
Sk	ModifierSymbol
Z	Separator
Zs	SpaceSeparator
Zl	LineSeparator
Zp	ParagraphSeparator
C	Other
Cc	Control
Cf	Format

Table 8: A few Standard Unicode character properties

the hierarchical tokenizer: `Ps` and `Pe` help to distinguish between opening and closing punctuation symbols.

In case the input text is not encoded with Unicode, it should be converted by the input converter module[30]. However, this conversion can't replace incorrectly used punctuation symbols of other character sets (e.g. ASCII) with their correct Unicode counterpart. And even if the input text is encoded with Unicode, the characters might have not been used as intended. Thus, FTS never relies solely on the Unicode properties of characters, but if meta-information is available it is able to use it to the full capacity.

5.3.2 Low level tokenizer

During low level tokenization, the input text is split into *low level tokens* according to the definition in a grammar file which can be adapted as desired. Typically, the low level tokenizer performs a strong segmentation by splitting sequences of characters of the same Unicode sub-property. This includes also whitespace and control characters which themselves make up tokens. Once

[30]The vast amount of characters contained in Unicode implies a lossless conversion.

again this is illustrated with the `"Deutsche Post"-Marken` example:

```
["][D][eutsche][ ][P][ost]["][-][M][arken]
```

The resulting low level tokens are not only sequenced but annotated with their level and the respective property. Like LT-TTT2 (Grover and Tobin, 2006), FTS uses XML as mark-up language. For the annotation, a generic XML format for the internal representation has been chosen:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes">
<tokenization>
   <token level="low" sprop="Po">"</token>
   <token level="low" sprop="Lu">D</token>
   <token level="low" sprop="Ll">eutsche</token>
   <token level="low" sprop="Zs"> </token>
   <token level="low" sprop="Lu">P</token>
   <token level="low" sprop="Ll">ost</token>
   <token level="low" sprop="Po">"</token>
   <token level="low" sprop="Pd">-</token>
   <token level="low" sprop="Lu">M</token>
   <token level="low" sprop="Ll">arken</token>
</tokenization>
```

Low level tokens are simple and precise. No other linguistic knowledge than about the different character classes is used for their definition.

It is to be noted that no transformation is performed during the low level tokenization process. The XML annotation allows to carry available information through the several stages of processing and is completely reversible, i.e. no information present in the original text is lost.

5.3.3 High level tokenizer

This module is the first to exhibit intelligent, that is linguistically sensitive behavior. According to rules defined in grammar files, the low level tokens created by the low level tokenizer are now combined to more complex *high level tokens*.

FTS distinguishes between two different types of grammars. First, there are *language independent grammars* which specify basic and universal tokenization tasks, e.g. the handling of abbreviations, hyphens, quotation marks or special expressions. Second, *language specific grammars* can be added to allow the specialised handling of particular phenomena restricted to a specific language. All grammar files can be activated and/or customized as desired and thus allow for a completely flexible token definition. The external lexicon files can be used in the grammars.

An examplary set of rules required for the hierarchical tokenization of the `"Deutsche Post"-Marken` example is given.

1. `[Lu][Ll]` → `[LuLl]`
2. `[Any][-][Any]` → `[[Any][-][[Any]]`
3. `["][Any]["]` → `[["][Any]["]]`

The first rule shows how the separation of upper- and lowercase letters can be undone. An uppercase low level token which is followed by a lowercase low level token is combined to a single token. The structure of a complex hyphen and a complex phrasal token is defined by rules 2 and 3.

During the high level tokenization process, the annotation of tokens is changed. The final hierarchical tokenization of the complete complex token is marked up with the internal XML format like presented here:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes">
<tokenization>
    <token type="complex-hyphen">
        <token type="complex-phrasal">
            <token type="open-quotation">"</token>
            <token type="word">Deutsche</token>
            <token type="space"> </token>
            <token type="word">Post</token>
            <token type="close-quotation">"</token>
        </token>
        <token type="hyphen">-</token>
        <token type="word">Marken</token>
    </token>
</tokenization>
```

This hierarchical tokenization fully accounts for the complex structure: the entire character sequence is a token of the type *complex-hyphen* with three components: a *complex-phrasal* token, a hyphen and a word token. The complex-phrasal token itself consists of 5 tokens, only two of it of the type *word*. Counting all word tokens in the entire character sequence, it is evident that there are only three word tokens in total: `[Deutsche]`, `[Post]` and `[Marken]`.

5.3.4 Output formatting module

By default, FTS offers a range of general output formats. As for XML formats, TEI from the Text Encoding Initiative[31], XCES, an Corpus Encoding Standard for XML[32] and the internal XML format (which is completely specified in the documentation) are available.

[31] Url: http://www.tei-c.org/
[32] Url: http://www.cs.vassar.edu/XCES/

Morever it is possible to define a single delimiter which is printed between each token or to define an opening and closing delimiter which is printed before and after each token. And last but not least, a completely customized output format can be defined via a configuration grammar file.

5.4 Summary

The presented architecture of FTS provides a design for a flexible tokenization system. All of the listed flexibility criteria (see section 3.1) have been incorporated into the sketch:

Flexible token definition
FTS provides a flexible token definition by allowing the customization of the tokenization grammars which are used by the high level tokenizer.

Flexible language support
FTS provides flexible language support, because its design distinguishes between language specific and language independent grammars. It is possible to put extended and specialized language specific grammars on top of the basic language independent tokenization grammars.

Flexible input
FTS provides flexible input because the input converter can handle the majority of character sets.

Flexible output
FTS provides flexible output because its internal token representation is prepared in the desired way by the output formatter module.

The evaluation has shown that tokenization is not as trivial as often assumed. Thus, an enhanced tokenization system should be used instead of quick and dirty tokenizer implementations as they are part of many NLP tools. FTS has been designed to provide the possibility to be neatly plugged in a bigger NLP tool by exhibiting the necessary flexibility.

6 Outlook

For real-life applications, besides flexibility another major criterion is robustness. But *errare humanum est*—to err is human. And especially in those situations the input text might not comply with assumed standards. Even if all of the concrete problem cases addressed in this thesis could sucessfully be processed by a tokenizer, rule-based approaches to tokenization are most likely to fail if the input text contains incorrect spelling, typos, wrong or inconsequent usage of punctuation symbols at critical decision points of

the process. For example a misspelt abbreviation cannot be looked up from an abbreviation database; a pattern that should match a special expression might be dependent on the occurrence of specific punctuation symbols.

For instance, the *WebCorp* project[33] is designed to allow access to the 'Web as corpus'. Linguists can make use of a collection of texts gathered from websites to extract linguistic knowledge. However, the requirements the processing tools have to meet are high: arbitrary texts from the web are far from being correctly and consistently written and encoded. Here, a flexible and robust tokenization system can enhance the overall quality of the corpus contents.

Future work should include an extensive evaluation of the properties of input text occuring in real-life application, e.g. on arbitrary websites. Effort should be made to minimize the consequences of possibly malformed input. Next to the straightforward rejection on detecting errors, normalization operations could be a solution but should be planned with caution.

[33] Url: http://www.webcorp.org.uk/

A Appendix

A.1 Test corpus

Mr. Sherlock Holmes, who was usually very late in the mornings, save upon those not infrequent occasions when he was up all night, was seated at the breakfast table. I stood upon the hearth-rug and picked up the stick which our visitor had left behind him the night before. It was a fine, thick piece of wood, bulbous-headed, of the sort which is known as a "Penang lawyer." Just under the head was a broad silver band nearly an inch across. "To James Mortimer, M.R.C.S., from his friends of the C.C.H.," was engraved upon it, with the date "1884." It was just such a stick as the old-fashioned family practitioner used to carry--dignified, solid, and reassuring.

Amongst the clergy present were the very rev. William Delany, S. J., L. L. D.; the rt rev. Gerald Molloy, D. D.; the rev. P. J. Kavanagh, C. S. Sp.; the rev. T. Waters, C. C.; the rev. John M. Ivers, P. P.; the rev. P. J. Cleary, O. S. F.; the rev. L. J. Hickey, O. P.; the very rev. Fr. Nicholas, O. S. F. C.; the very rev. B. Gorman, O. D. C.; the rev. T. Maher, S. J.; the very rev. James Murphy, S. J.; the rev. John Lavery, V. F.; the very rev. William Doherty, D. D.; the rev. Peter Fagan, O. M.; the rev. T. Brangan, O. S. A.; the rev. J. Flavin, C. C.; the rev. M. A. Hackett, C. C.; the rev. W. Hurley, C. C.; the rt rev. Mgr M'Manus, V. G.; the rev. B. R. Slattery, O. M. I.; the very rev. M. D. Scally, P. P.; the rev. F. T. Purcell, O. P.; the very rev. Timothy canon Gorman, P. P.; the rev. J. Flanagan, C. C. The laity included P. Fay, T. Quirke, etc., etc.

3 May. Bistritz.--Left Munich at 8:35 P.M., on 1st May, arriving at Vienna early next morning; should have arrived at 6:46, but train was an hour late.

Alice was beginning to get very tired of sitting by her sister on the bank, and of having nothing to do: once or twice she had peeped into the book her sister was reading, but it had no pictures or conversations in it, ‘and what is the use of a book,’ thought Alice ‘without pictures or conversation?’

In his private notes on the subject the author uses the expression "Superman" (always in the singular, by-the-bye), as signifying "the most thoroughly well-constituted type," as

opposed to "modern man"; above all, however, he designates Zarathustra himself as an example of the Superman.

The Foundation's principal office is located at 4557 Melan Dr. S. Fairbanks, AK, 99712., but its volunteers and employees are scattered throughout numerous locations. Its business office is located at 809 North 1500 West, Salt Lake City, UT 84116, (801) 596-1887, email business@pglaf.org. Email contact links and up to date contact information can be found at the Foundation's web site and official page at http://pglaf.org

It's probably worth paying a premium for funds that invest in markets that are partially closed to foreign investors, such as South Korea, some specialists say. But some European funds recently have skyrocketed; Spain Fund has surged to a startling 120% premium. It has been targeted by Japanese investors as a good long-term play tied to 1992's European economic integration. And several new funds that aren't even fully invested yet have jumped to trade at big premiums. "I'm very alarmed to see these rich valuations," says Smith Barney's Mr. Porter.

This is a alpha-galactosyl-1,4-beta-galactosyl-specific adhesin. The corresponding free cortisol fractions in these sera were 4.53 +/- 0.15% and 8.16 +/- 0.23%, respectively.

Cabelli, H.F., 1990. Promoter occlusion: transcription through a promoter may inhibit its activity. Cell 29 939-944.
Cabelli, H.F., P. White, and D. McKay. (1990). Promoter occlusion: transcription through a promoter may inhibit its activity. Cell 29 939-944.

In New York Stock Exchange composite trading yesterday, American Medical closed at $23.625, up $1.875.
Last week the stock hit an all-time high of 37 1/4 before getting roughed. John April, Spokesman for General Trends said up in the Friday-the-13th minicrash. It closed yesterday at 34 3/4. In New York Stock Exchange composite trading, Bristol-Myers Squibb rose 1.75.65 to $52.75. In July 1995 CEG Corp. posted net of $102 million, or 34 cents a share.
Late last night the company announced a growth of 20%.

The house with a height of 10'6" was built out of wood.

A.2 Results String::Tokenizer

Abbreviations

1. [Mr.] | [rev.] | [S.,] | [Corp.] | [etc.,]
2. [M.R.C.S.,] | [C.C.H.,"]

Quotation mark symbols

1. ["Penang][lawyer."] | ["Superman"]
2. [`and][what][is][the][use][of][a][book,']
3. [Foundation's] | [It's] | [1992's] | [I'm]
4. [10'6"]

Hyphen symbol

1. [old-fashioned] | [by-the-bye] | [Friday-the-13th]
2. [hearth-][rug]
3. [inte-][gration]
4. [carry--dignified,] | [Bistritz.--Left]

Special expressions

1. ["1884."] | [3][May] | [8:35][P.M.] | [1st][May] [July][1995]
2. [809][North][1500][West] | [UT][84116] | [(801)] [596-1887] | [business@pglaf.org] | [http://pglaf.org]
3. [alpha-galactosyl-1,4-beta-galactosyl-specific] [4.53][+/-][0.15%] | [8.16][+/-][0.23%]
4. [Cabelli,][H.F.,][1990.][Promoter][occlusion:] [transcription] [through][a][promoter][may][inhibit] [its][activity.] [Cell][29][939-944.]
5. [$23.625] | [$102][million] | [34][cents]

A.3 Results ET

Abbreviations

1. [mr][.] | [rev][.] | [s][.] | [corp][.] | [etc][.]
2. [m][.][r][.][c][.][s][.] | [c][.][c][.][h][.]

Quotation mark symbols

1. ["][penang][lawyer][.]["] | ["][superman]["]
2. [`][and][what][is][the][use][of][a][book][,][']
3. [foundation][s] | [it][s] | [1992][s] | [I][m]
4. [10]['][6]["]

Hyphen symbol

1. [old][-][fashioned] | [by][-][the][-][bye]
   [friday][-][the][-][13][th]
2. [hearth][-][rug]
3. [inte][-][gration]
4. [carry][-][-][dignified] | [bistritz][.][--][left]

Special expressions

1. [1884] | [3][may] | [8][:][35] [p][.][m][.]
   [1][st][may] | [july][1995]
2. [809][north][1500][west] | [ut][84116]
   [(][801][)][596][-][1887] | [business][@][pglaf][.][org]
   [http][:][/][/][pglaf][.][org]
3. [alpha][-][galactosyl][-][1][,][4][-][beta] ...
   [-][galactosyl][-][specific]
   [4][.][53][+][/][-][0][.][15][%]
   [8][.][16][+][/][-][0][.][23][%]
4. [cabelli][,][h][.][f][.][,][1990][.][promoter][occlusion]
   [:][transcription] [through][a][promoter][may][inhibit][its]
   [activity][.] [cell][29][939][-][944][.]
5. [$][23][.][625] | [$][102][million] | [34][cents]

A.4 Results Cass tokenizer

Abbreviations

1. [Mr][.] | [rev][.] | [S][.] | [Corp][.] | [etc][.]
2. [M][.][R][.][C][.][S][.] | [C][.][C][.][H][.]

Quotation mark symbols

1. ["][Penang] [lawyer][.]["] | ["][Superman]["]
2. [`] [and] [what] [is] [the] [use] [of] [a] [book][,][']
3. [Foundation's] | [It's] | [1992][']s] | [I'm]
4. [10][']6]["]

Hyphen symbol

1. [old][-][fashioned] | [by][-][the][-][bye]
   [Friday][-][the][-][13th]
2. [hearth][-][rug]
3. [inte][-][gration]
4. [carry][--][dignified] | [Bistritz][.][--][Left]

Special expressions

1. [1884.] | [3] [May] | [8][:][35] [P][.][M][.]
   [1st] [May] | [July] [1995]
2. [809] [North] [1500] [West] | [UT] [84116]
   [(][801][)] [596][-1887] | [business][@][pglaf][.][org]
   [http][:][/][/][pglaf][.][org]
3. [alpha][-][galactosyl][-1][4][-][beta][-][galactosyl]
   [-][specific] [4.53] [+][/][-] [0.15][%]
   [8.16] [+][/][-] [0.23%]
4. [Cabelli][,] [H][.][F][.][,] [1990.] [Promoter] [occlusion]
   [:] [transcription] [through] [a] [promoter] [may] [inhibit]
   [its] [activity][.] [Cell] [29] [939][-944][.]
5. [$][23.625] | [$][102] [million] | [34] [cents]

A.5 Results TreeTagger tokenizer

Abbreviations

1. [Mr.] | [rev][.] | [S.] | [Corp.] | [etc.]
2. [M.R.C.S.] | [C.C.H.]

Quotation mark symbols

1. ["][Penang][lawyer][.]["] | ["][Superman]["]
2. [`][and][what][is][the][use][of][a][book][,]['][']
3. [Foundation's] | [It's] | [1992's] | [I'm]
4. [10'6]["]

Hyphen symbol

1. [old-fashioned] | [by-the-bye] | [Friday-the-13th]
2. [hearth-][rug]
3. [inte-][gration]
4. [carry--dignified] | [Bistritz.--Left]

Special expressions

1. [1884] | [3][May] | [8:35][P.M.] | [1st][May] [July][1995]
2. [809][North][1500][West] | [UT][84116] [(][801][)][596-1887] | [business@pglaf.org] [http://pglaf.org]
3. [alpha-galactosyl-1,4-beta-galactosyl-specific] [4.53][+/-][0.15][%] | [8.16][+/-][0.23][%]
4. [Cabelli][,][H.F.][,][1990][.][Promoter][occlusion] [:][transcription] [through][a][promoter][may][inhibit] [its][activity][.] [Cell][29][939-944][.]
5. [$][23.625] | [$][102][million] | [34][cents]

A.6 Results LT-TTT2

Abbreviations

1. [Mr][.] | [rev][.] | [S][.] | [Corp][.] | [etc][.]
2. M.R.C.S. | C.C.H.

Quotation mark symbols

1. ["][Penang] [lawyer][.]["] | ["][Superman]["]
2. `and what is the use of a book,'
3. [Foundation]['s] | [It]['s] | [1992]['s] | [I]['m]
4. [10][']['][6]["]

Hyphen symbol

1. [old-fashioned] | [by-the-bye] | [Friday-the-13th]
2. [hearth][-][rug]
3. [inte][-][gration]
4. [carry][--][dignified] | [Bistritz][.][--][Left]

Special expressions

1. [1884] | [[3] [May]] | [[8:35] [P.M.]]
   [[1st] [May]] | [[July] [1995]]
2. [809] [North] [1500] [West] | [UT] [84116]
   [(][801][)] [596][-][1887] [business][@][pglaf][.][org]
   [http][:][/][/][pglaf][.][org]
3. [alpha-galactosyl-1,4-beta-galactosyl-specific]
   [4.53] [+/-] [0.15%] | [8.16] [+/-] [0.23%]
4. [Cabelli][,] [H.F][.][,] [1990][.] [Promoter]
   [occlusion][:] [transcription] [through] [a]
   [promoter] [may] [inhibit] [its] [activity][.]
   [Cell] [29] [939][-][944][.]
5. [[$][23.625]] | [[$][102] [million]] | [[34] [cents]]

References

Abney, S. (2006). Cass—A fast, robust partial parser. Retrieved October 14, 2006 from `http://www.vinartus.net/spa/scol-1-12.tgz`.

Covington, M. A. (2003). ET—An Efficient Tokenizer in ISO Prolog. Retrieved October 23, 2006 from `http://www.ai.uga.edu/mc/ET/et.pdf`.

Dale, R. (1997). A Framework for Complex Tokenisation and its Application to Newspaper Text. In *Proceedings of the Second Australian Document Computing Symposium*, Melbourne, Australia.

Fontenelle, T. (2005). Identifying tokens: Is word-breaking so easy? *In Hiligsmann, Ph. / Janssens, G. / Vromans, J. (eds.), Woord voor woord. Zin voor zin. Liber Amicorum voor Siegfried Theissen, Ghent: Koninklijke Academie voor Nederlandse Taal- en Letterkunde*, 109–115.

Forst, M. and R. M. Kaplan (2006). The importance of precise tokenizing for deep grammars. In *Proceedings of the fifth international conference on Language Resources and Evaluation (LREC'06)*, Genoa, Italy.

Grefenstette, G. and P. Tapanainen (1994). What is a word, what is a sentence? Problems of tokenization. In *The 3rd International Conference on Computational Lexicography*, Budapest, pp. 79–87.

Grover, C., C. Matheson, A. Mikheev, and M. Moens (2000). LT TTT—A Flexible Tokenisation Tool. In *Proceedings of the second international conference on Language Resources and Evaluation (LREC'00)*, Athens, Greece.

Grover, C. and R. Tobin (2006). Rule-Based Chunking and Reusability. In *Proceedings of the fifth international conference on Language Resources and Evaluation (LREC'06)*, Genoa, Italy.

Guo, J. (1997). Critical Tokenization and its Properties. *Computational Linguistics 2*(5), 569–596.

Habert, B., G. Adda, M. Adda-Decker, P. B. de Mareuil, S. Ferrari, O. Ferret, G. Illouz, and P. Paroubek (1998). Towards Tokenization Evaluation. In *The first international conference on Language Resources and Evaluation (LREC'98)*, Granada, Spain.

Klein, D. and C. D. Manning (2003). Accurate unlexicalized parsing. In *Proceedings of the 41st Meeting of the Association for Computational Linguistics*, Sapporo, Japan, pp. 423–430.

Little, S. (2004). String::Tokenizer—A simple string tokenizer. Retrieved December 1, 2006 from `http://search.cpan.org/~stevan/...String-Tokenizer-0.05/lib/String/Tokenizer.pm`.

Matheson, C. and R. Dale (1993). Bibedit: A knowledge-based copy editing tool for bibliographic information. In *In E.S. Atwell (ed) Knowledge at Work in Universities: Second Annual Conference of the Higher Education Funding Councils' Knowledge Based Systems Initiative*, Cambridge.

Mikheev, A. (2003). *Text Segmentation*. Oxford University Press.

Palmer, D. D. and M. A. Hearst (1997). Adaptive Multilingual Sentence Boundary Disambiguation. *Computational Linguistics 23*, 241–267.

Pedersen, T. (2006). N-Gram Statistics Package (NSP). Retrieved December 17, 2006 from `http://www.d.umn.edu/~tpederse/nsp.html`.

Roberts, A. (2006). jTokeniser. Retrieved September 21, 2006 from `http://www.andy-roberts.net/software/jTokeniser/index.html`.

Schmid, H. (1994). Probabilistic part-of-speech tagging using decision trees. In *International Conference on New Methods in Language Processing*, Manchester, UK, pp. 44–49. `http://www.ims.uni-stuttgart.de/projekte/corplex/TreeTagger/`.

Thompson, H., R. Tobin, D. McKelvie, and C. Brew (1997). LT XML. Software API and toolkit for XML processing. Homepage: `http://www.ltg.ed.ac.uk/software/`.

Webster, J. J. and C. Kit (1992). Tokenization as the initial phase in NLP. In *Proceedings of the 14th International Conference on Computational Linguistics (COLING'92')*, Nantes, France, pp. 1106–1110.

Zierl, M. (1997). Entwicklung und Implementierung eines Datenbanksystems zur Speicherung und Verarbeitung von Textkorpora. Retrieved August 12, 2005 from `http://www.linguistik.uni-erlangen.de/...tree/html/corsica/zierl97/zierl97.html`.