A Rule-Based Style and Grammar Checker

Daniel Naber

Diplomarbeit
Technische Fakultät, Universität Bielefeld

Datum: 28.08.2003

Betreuer:
Prof. Dr.-Ing. Franz Kummert, Technische Fakultät
Dr. Andreas Witt, Fakultät für Linguistik und Literaturwissenschaft

Contents

1 Introduction **3**

2 Theoretical Background **4**
2.1 Part-of-Speech Tagging 5
2.2 Phrase Chunking 6
2.3 Grammar Checking 7
2.3.1 Grammar Errors 8
2.3.2 Sentence Boundary Detection 9
2.4 Controlled Language Checking 10
2.5 Style Checking 12
2.6 False Friends 12
2.7 Evaluation with Corpora 13
2.7.1 British National Corpus 14
2.7.2 Mailing List Error Corpus 15
2.7.3 Internet Search Engines 15
2.8 Related Projects 16
2.8.1 Ispell and Aspell 16
2.8.2 Style and Diction 17
2.8.3 EasyEnglish 17
2.8.4 Critique 18
2.8.5 CLAWS as a Grammar Checker 18
2.8.6 GramCheck 18
2.8.7 Park et al's Grammar Checker 19
2.8.8 FLAG 19

3 Design and Implementation **20**
3.1 Class Hierarchy 20
3.2 File and Directory Structure 21
3.3 Installation 23
3.3.1 Requirements 23
3.3.2 Step-by-Step Installation Guide 23
3.4 Spell Checking 25
3.5 Part-of-Speech Tagging 25
3.5.1 Constraint-Based Extension 26
3.5.2 Using the Tagger on the Command Line 27
3.5.3 Using the Tagger in Python Code 28
3.5.4 Test Cases 29
3.6 Phrase Chunking 29
3.7 Sentence Boundary Detection 30
3.8 Grammar Checking 31
3.8.1 Rule Features 31
3.8.2 Rule Development 32
3.8.3 Testing New Rules 33
3.8.4 Example: *Of cause* Typo Rule 34
3.8.5 Example: Subject-Verb Agreement Rule 35
3.8.6 Checks Outside the Rule System 36
3.9 Style Checking 38
3.10 Language Independence 38
3.11 Graphical User Interfaces 39
3.11.1 Communication between Frontend and Backend 39
3.11.2 Integration into KWord 41
3.11.3 Web Frontend 47
3.12 Unit Testing 48

4 Evaluation Results **50**
4.1 Part-of-Speech Tagger . . . 50
4.2 Sentence Boundary Detection . . . 51
4.3 Style and Grammar Checker . . . 51
4.3.1 British National Corpus . . . 51
4.3.2 Mailing List Errors Corpus . . . 51
4.3.3 Performance . . . 52

5 Conclusion **53**

6 Acknowledgments **53**

7 Bibliography **54**

A Appendix **58**
A.1 List of Collected Errors . . . 58
A.1.1 Document Type Definition . . . 58
A.1.2 Agreement Errors . . . 58
A.1.3 Missing Words . . . 60
A.1.4 Extra Words . . . 60
A.1.5 Wrong Words . . . 61
A.1.6 Confusion of Similar Words . . . 61
A.1.7 Wrong Word Order . . . 63
A.1.8 Comma Errors . . . 63
A.1.9 Whitespace Errors . . . 63
A.2 Error Rules . . . 64
A.2.1 Document Type Definition . . . 64
A.2.2 Grammar Error Rules . . . 64
A.2.3 Style/Word Rules . . . 69
A.2.4 English/German False Friends . . . 69
A.3 Penn Treebank Tag Set to BNC Tag Set Mapping . . . 70
A.4 BNC Tag Set . . . 70
A.4.1 List of C5 Tags . . . 70
A.4.2 C7 to C5 Tag Set Mapping . . . 74

1 Introduction

The aim of this thesis is to develop an Open Source style and grammar checker for the English language. Although all major Open Source word processors offer spell checking, none of them offer a style and grammar checker feature. Such a feature is not available as a separate free program either. Thus the result of this thesis will be a free program which can be used both as a stand-alone style and grammar checker and as an integrated part of a word processor.

The style and grammar checker described in this thesis takes a text and returns a list of possible errors. To detect errors, each word of the text is assigned its part-of-speech tag and each sentence is split into chunks, e.g. noun phrases. Then the text is matched against all the checker's pre-defined error rules. If a rule matches, the text is supposed to contain an error at the position of the match. The rules describe errors as patterns of words, part-of-speech tags and chunks. Each rule also includes an explanation of the error, which is shown to the user.

The software will be based on the system I developed previously [Naber]. The existing style and grammar checker and the part-of-speech tagger which it requires will be re-implemented in Python. The rule system will be made more powerful so that it can be used to express rules which describe errors on the phrase level, not just on the word level. The integration into word processors will be improved so that errors can be detected on-the-fly, i.e. during text input. For many errors the software will offer a correction which can be used to replace the correct text with a single mouse click.

The system's rule-based approach is simple enough to enable users to write their own rules, yet it is powerful enough to catch many typical errors. Most rules are expressed in a simple XML format which not only describes the errors but also contains a helpful error message and example sentences. Errors which are too complicated to be expressed by rules in the XML file can be detected by rules written in Python. These rules can also easily be added and do not require any modification of the existing source code.

An error corpus will be assembled which will be used to test the software with real errors. The errors will be collected mostly from mailing lists and websites. The errors will be categorized and formatted as XML. Compared to the previous version, many new rules will be added which detect typical errors found in the error corpus.

To make sure that the software does not report too many errors for correct text it will also be tested with the British National Corpus (BNC). The parts of the BNC which were taken from published texts are supposed to contain only very few grammar errors and thus should produce very few warning messages when checked with this software.

There have been several scientific projects working on style and grammar checking (see section 2.8), but none are publicly available. This thesis and the software is available as Open Source software at `http://www.danielnaber.de/languagetool/`.

2 Theoretical Background

Style and grammar checking are useful for the same reason that spell checking is useful: it helps people to write documents with fewer errors, i.e. better documents. Of course the style and grammar checker needs to fulfill some requirements to be useful:

- It should be fast, i.e. fast enough for interactive use.
- It should be well integrated into an existing word processor.
- Not too often should it complain about sentences which are in fact correct.
- It should be possible to adopt it to personal needs.
- And finally: it should be as complete as possible, i.e. it should find most errors in a text.

The many different kinds of errors which may appear in written text can be categorized in several different ways. For the purpose of this thesis I propose the following four categories:

- **Spelling errors:** This is defined as an error which can be found by a common spell checker software. Spell checkers simply compare the words of a text with a large list of known words. If a word is not in the list, it is considered incorrect. Similar words will then be suggested as alternatives.
  Example: **Gemran*[1] (Ispell will suggest, among others, *German*)

- **Grammar errors:** An error which causes a sentence not to comply with the English grammar rules. Unlike spell checking, grammar checking needs to make use of context information, so that it can find an error like this:
  Harry Potter bigger ~~then~~ **than Titanic?*[2]
  Whether this error is caused by a typo or whether it is caused my a misunderstanding of the words *then* and *than* in the writer's mind usually cannot be decided. This error cannot be found by a spell checker because both *then* and *than* are regular words. Since the use of *then* is clearly wrong here, this is considered a grammar error.
  Grammar errors can be divided into structural and non-structural errors. Structural errors are those which can only be corrected by inserting, deleting, or moving one or more words. Non-structural errors are those which can be corrected by replacing an existing word with a different one.

- **Style errors:** Using uncommon words and complicated sentence structures makes a text more difficult to understand, which is seldomly desired. These cases are thus considered style errors. Unlike grammar errors, it heavily depends on the situation and text type which cases should be classified as a style error. For example, personal communication via email among friends allows creative use of language, whereas technical documentation should not suffer from ambiguities. Configurability is even more important for style checking than for grammar checking.
  Example: *But it [= human reason] quickly discovers that, in this way, its labours must remain ever incomplete, because new questions never cease to present themselves; and thus it finds itself compelled to have recourse to principles which transcend the region of experience, while they are regarded by common sense without distrust.*
  This sentence stems from Kant's Critique of pure reason. It is 48 words long and most people

[1] The asterisk indicates an incorrect word or sentence.

[2] The crossed out word is incorrect, the bold word is a correct replacement. This sentence fragment was found on the Web.

will agree that it is very difficult to understand. The reason is its length, difficult vocabulary (like *transcend*), and use of double negation (*without distrust*). With today's demand for easy to understand documents, this sentence can be considered to have a style problem.

- **Semantic errors:** A sentence which contains incorrect information which is neither a style error, grammar error, nor a spelling error. Since extensive world-knowledge is required to recognize semantic errors, these errors usually cannot be detected automatically.
  Example: *MySQL is a great editor for programming!*
  This sentence is neither true nor false – it simply does not make sense, as MySQL is not an editor, but a database. This cannot be known, however, without extensive world knowledge. World knowledge is a form of context, too, but it is far beyond what software can understand today.

I will not make a distinction between *errors* and *mistakes* in this thesis, instead I will simply use the term *error* for all parts of text which can be considered incorrect or poorly written.

Grammar (or *syntax*) refers to a system of rules describing what correct sentences have to look like. Somehow these rules exist in people's minds so that for the vast majority of sentences people can easily decide whether a sentence is correct or not. It is possible to make up corner cases which make the intuitive correct/incorrect decision quite difficult. A software might handle these cases by showing a descriptive warning message to the user instead of an error message. Warning messages are also useful for style errors and for grammar errors if the grammar rule is known to also match for some correct sentences.

2.1 Part-of-Speech Tagging

Part-of-speech tagging (POS tagging, or just tagging) is the task of assigning each word its POS tag. It is not strictly defined what POS tags exist, but the most common ones are *noun*, *verb*, *determiner*, *adjective* and *adverb*. Nouns can be further divided into singular and plural nouns, verbs can be divided into past tense verbs and present tense verbs and so on.

The more POS tags there are, the more difficult it becomes – especially for an algorithm – to find the right tag for a given occurrence of a word, since many words can have different POS tags, depending on their context. In English, many words can be used both as nouns and as verbs. For example *house* (a building) and *house* (to provide shelter). Only about 11,5% percent of all words are ambiguous with regard to their POS tags, but since these are the more often occurring words, 40% percent of the words in a text are usually ambiguous [Harabagiu]. POS tagging is thus a typical disambiguation problem: all possible tags of a word are known and the appropriate one for a given context needs to be chosen.

Even by simply selecting the POS tag which occurs most often for a given word – without taking context into account – one can assign 91% of the words their correct tag. Taggers which are mostly based on statistical analysis of large corpora have an accuracy of 95-97% [Brill, 1992]. Constraint Grammar claims to have an accuracy of more than 99% [Karlsson, 1995].

One has to be cautious with the interpretation of these percentages: some taggers give up on some ambiguous cases and will return more than one POS tag for a word in a given context. The chances that at least one of the returned POS tags is correct is obviously higher if more than one POS tag is returned. In other words, one has to distinguish between recall and precision (see section 2.7).

In the following I will describe Qtag 3.1 [Mason] as an example of a purely probabilistic tagger. Qtag is written in Java[3] and it is freely available for non-commercial use. The source code is not

[3] [Tufi s and Mason, 1998] refers to an older version that was implemented as a client/server system with the server written in C. The implementation is now completely in Java, but there is no evidence that the algorithm has changed, so the

available, but the basic algorithm is described in [Tufis and Mason, 1998]. Qtag always looks at a part of the text which is three tokens wide. Each token is looked up in a special dictionary which was built using a tagged corpus. This way Qtag finds the token's possible tags. If the token is not in the dictionary a heuristic tries to guess the token's possible POS tags by looking for common suffixes at the token's end. If the token is in the dictionary then for each possible tag two probabilities are calculated: the probability P_w of the token having the specified tag, and the context probability P_c that the tag is preceded and followed by the tags of the words to the left and to the right. A joint probability $P_{w,c} = P_w * P_c$ is then calculated and the POS tag with the highest joint probability is chosen. An example can be found in section 3.5.

Qtag itself can be used for any language, the only thing one needs is a large tagged corpus so Qtag can learn the word/tag probabilities and the tag sequence frequencies. The Qtag program is only 20 KB in size.

Constraint Grammar is an example of a purely rule-based tagger. It is not freely available, but it is described in [Karlsson, 1995]. Like Qtag it starts by assigning each word all its possible tags from a dictionary. The rules erase all tags which lead to illegal tag sequences. All the rules are hand-written, which made the development of the Constraint Grammar a time-consuming and difficult task. The result is a tagger which has an accuracy of more than 99%.

As developing rules manually is difficult, there have been attempts to learn the rules automatically (some papers are quoted in [Lindberg and Eineborg, 1999]). Brill's tagger is such an attempt [Brill, 1992]. It also starts by assigning each token all possible tags. It then tags a tagged corpus by assigning each token from the corpus its most probable tag, without taking context into account. The assigned tags are compared to the real tags and each mistagging is counted. Brill's tagger now tries to come up with rules (called *patches*) which repair these errors. A rule usually says something like "if a token has tag A, but it it followed by tag X, then make it tag B".

With 71 of these automatically built rules the system reaches an error rate of 5.1% which corresponds to a recall of 94.9%. Although Brill states that this result is difficult to compare to the results of other publications, he concludes that his rule-based tagger offers a performance similar to probabilistic taggers. One advantage of rule-based taggers is their compact representation of knowledge – 71 rules against several thousand values required by a probabilistic tagger. With today's computer power this has become less of a problem. But the smaller number of rules is also supposed to make enhancements to the system easier.

Both probabilistic and rule-based taggers need additional knowledge to approach a recall of 100%. [Lindberg and Eineborg, 1999] report promising results with adding linguistic knowledge to their tagger. Probabilistic taggers are said to have some advantages over rule-based ones: they are language independent, and there is no need to manually code rules for them. A discussion about these alleged advantages can be found in [Kiss, 2002].

One common problem is the tagging of idioms and phrases. For example, *New York* should be tagged as a noun for most applications, not as a sequence of adjective, noun. This of course is easy to achieve for many cases when the tagger is trained with a corpus which has the appropriate markup for such phrases.

2.2 Phrase Chunking

Phrase Chunking is situated between POS tagging and a full-blown grammatical analysis: whereas POS tagging only works on the word level, and grammar analysis (i.e. parsing) is supposed to build a tree structure of a sentence, phrase chunking assigns a tag to word sequences of a sentence.

Typical chunks are noun phrase (NP) and verb phrase (VP). Noun phrases typically consist of deter-

information provided in the paper should still be valid.

miners, adjectives and nouns or pronouns. Verb phrases can consist of a single verb or of an auxiliary verb plus infinitive. For example, *the dog*, *the big dog*, *the big brown dog* are all examples of noun phrases. As the list of adjectives can become infinitely long, noun phrases can theoretically grow without a limit. However, what is called noun phrase here is just an example and just like in POS tagging everybody can make up his own chunk names and their meanings. The chunks found by a chunker do not necessarily need to cover the complete text – with only noun and verb phrases, as usually defined, this is not possible anyway.

Chunking works on a POS tagged text just like POS tagging works on words: either there are hand-written rules that describe which POS tag sequences build which chunks, or a probabilistic chunker is trained on a POS tagged and chunked text. These methods can be combined by transferring the knowledge of a probabilistic chunker to rules.

As chunking requires a POS tagged text, its accuracy cannot be better than that of the POS tagger used. This is backed by the fact that even the best chunker listed on [Chunking] reaches a precision and recall of 94%, which is less than an average tagger can achieve. [Chunking] also lists many papers about chunking.

2.3 Grammar Checking

It turns out there are basically three ways to implement a grammar checker. I will refer to them with the following terms:

- **Syntax-based checking**, as described in [Jensen et al, 1993]. In this approach, a text is completely parsed, i.e. the sentences are analyzed and each sentence is assigned a tree structure. The text is considered incorrect if the parsing does not succeed.
- **Statistics-based checking**, as described in [Attwell, 1987]. In this approach, a POS-annotated corpus is used to build a list of POS tag sequences. Some sequences will be very common (for example *determiner, adjective, noun* as in *the old man*), others will probably not occur at all (for example *determiner, determiner, adjective*). Sequences which occur often in the corpus can be considered correct in other texts, too, uncommon sequences might be errors.
- **Rule-based checking**, as it is used in this project. In this approach, a set of rules is matched against a text which has at least been POS tagged. This approach is similar to the statistics-based approach, but all the rules are developed manually.

The advantage of the syntax-based approach is that the grammar checking is always complete if the grammar itself is complete, i.e. the checker will detect any incorrect sentence, no matter how obscure the error is. Unfortunately, the checker will only recognize that the sentence is incorrect, it will not be able to tell the user what exactly the problem is. For this, extra rules are necessary that also parse ill-formed sentences. If a sentence can only be parsed with such an extra rule, it is incorrect. This technique is called constraint relaxation.

However, there is a major problem with the syntax-based approach: it requires a complete grammar which covers all types of texts one wants to check. Although there are many grammar theories, there is still no robust broad-coverage parser publicly available today. Also, parsers suffer from natural language ambiguities, so that usually more than one result is returned even for correct sentences.

Statistics-based parsers, on the other hand, bear the risk that their results are difficult to interpret: if there is a false alarm error by the system, the user will wonder why his input is considered incorrect, as there is no specific error message. Even developers would need access to the corpus on which the system was trained in order to understand the system's judgment. Another problem is that someone

has to set a threshold which separates the uncommon but correct constructs from the uncommon and incorrect ones. Surely this task could be passed on to the user who would have to set some value between, say, 0 and 100. The idea of a threshold does however not really comply with the perception that sentences are – besides questions of style and constructed corner cases – usually either correct or incorrect.

Due to said problems with the other approaches a strictly rule-based system will be developed in this thesis. Unlike a syntax-based checker, a rule-based checker will never be complete, i.e. there will always be errors it does not find. On the other hand, it has many advantages:

- A sentence does not have to be complete to be checked, instead the software can check the text while it is being typed and give immediate feedback.
- It is easy to configure, as each rule has an expressive description and can be turned on and off individually.
- It can offer detailed error messages with helpful comments, even explaining grammar rules.
- It is easily extendable by its users, as the rule system is easy to understand, at least for many simple but common error cases.
- It can be built incrementally, starting with just one rule and then extending it rule by rule.

2.3.1 Grammar Errors

The number of grammar rules is extensive, even for a rather simple language like English [English G, 1981]. I will only describe very few of these grammar rules. Although English will be used for all example sentences, similar rules exist in other languages, too. The grammar rules described here are based on sentences from the corpus which violate these rules (see section A.1).

Subject-Verb Agreement In English, subject and verb have to agree with respect to number and person. For example, in **They **is** my favourite Canadian authors*[4], subject and verb disagree in number (*they* = plural, *is* = singular). In **He **am** running for president*, subject and verb disagree in person (*he* = third person, *am* = first person of *to be*).

This of course is a rather simple case. Taking the perspective of a rule-based checker, which interprets the text as a sequence of tokens with POS tags, there are several special cases:

1. Subject and verb are separated, i.e. the verb does not occur directly after the subject: **The characters in Shakespeare's Twelfth Night **lives** in a world that has been turned upside-down.*
2. The subject can be a compound subject: **Christie and Prin **is** characters from Laurence's The Diviners.*
3. Book titles are singular: **Salman Rushdie's Midnight's Children **are** my favourite novel.*

Agreement between Indefinite Article and the Following Word If the indefinite article is followed by a word whose pronunciation starts with a vowel sound, *an* has to be used instead of *a*. Software can guess a word's pronunciation by looking at its first letter. If it is one of a, e, i, o, u, the word probably starts with a vowel – but there are exceptions. Here are some examples where the a,e,i,o,u rule applies, together with the correct indefinite article:

[4] Some of these examples are taken from `http://ace.acadiau.ca/english/grammar/index.htm`.

a test, a car, a long talk

an idea, an uninteresting speech, an earthquake

Here are some exceptions:

a university, a European initiative

an hour, an honor

Tag questions (..., isn't it? etc) A tag question is often used in spoken language to obtain affirmation for a given statement. It is built by attaching a negated form of an auxiliary verb and the sentence's subject to the end of the sentence. For example, *It's warm today* becomes *It's warm today, isn't it?*. When the verb is already negated, it has to be attached in its non-negated form, as in *It wasn't very difficult, was it?*

These tag questions are also used in email communication. For native German speakers who are not yet proficient in English they are difficult to master, as their German equivalent is much easier to use – one can just attach *..., oder?* to the end of the sentence, no matter what subject and verb is used. Sometimes this is incorrectly directly translated into English, i.e. *..., or?* is attached to a sentence.

Other Errors Many other errors are technically grammar errors, but are caused by a typo. Often the error suggests that it was caused by editing existing text but missing some words:

Someone* ***suggested said* *that it worked for him after he updated to Kernel 2.4.20.*

The author of this sentence obviously wanted to replaced *said* by *suggested* but then forget to delete *said* (or vice versa).

Often similar words are mixed up and it is not possible to tell if the writer has made a typo or if he is not aware of the difference:

****Than*** *my old email is nonsense.*

Not surprisingly, the confusion between *than* and *then* also happens vice versa:

It's less controversial* ***then* *one would think.*

2.3.2 Sentence Boundary Detection

Grammaticality refers to sentences. One could argue that the following two sentences have a grammar error, because there is no agreement between the proper noun *Peter* and the personal pronoun *she*:

Peter *leaves his house.* ***She*** *feels good.*

We will not take such cases into account and simply define that this is an error on the semantic level. Instead we will focus on the question what a sentence is. Human readers have an intuitive understanding of where a sentence starts and where it ends, but it is not that simple for computers. Just splitting a string at all the places where a period occurs is not enough, as the following artificial sentences show:

This is a sentence by Mr. Smith from the U.S.A. This is another sentence... A third one; using a semicolon? And yet another one, containing the number 15.45. "Here we go!", he said.

Abbreviations, numbers and indirect speech are the problems here which make the task non-trivial for computers. [Walker et al, 2001] have evaluated three approaches to sentence boundary detection. Their focus is on automatic translation systems which require sentence boundary detection on their input. The three approaches are:

- The direct implementation into a translation system, without using a higher level of description than the words and punctuation itself. This system uses an abbreviation lexicon. The implementation is inspired by regular expressions, but everything is directly implemented in a given programming language.
- The rule-based representation of sentences as regular expressions. The regular expressions allow to encode the necessary knowledge in a declarative way. Although it is not explicitly mentioned, this method seems to use an abbreviation lexicon, too.
- The application of a machine learning algorithm which is trained on a tagged corpus. The algorithm weights several features of a potential sentence boundary like capitalization and occurrence in the abbreviation lexicon.

The evaluation by [Walker et al, 2001] shows that the machine learning algorithm offers both a precision and a recall of about 98%. The method based on regular expression yields about 96% precision and recall. The direct implementation reaches 98% precision but only 86% recall.

So even with the best sentence boundary detection, a style and grammar checker must still cope with an error rate of about 2% in the sentence boundaries. This is a bad problem for a parser which is supposed to work on those incorrectly chunked sentences, because any incorrectly added sentence boundary and any missed sentence boundary will almost always lead to unparseable input. For a rule-based system this is less of a problem: only rules which explicitly refer to sentence boundaries will be affected. These rules might incorrectly be triggered when a sentence boundary was incorrectly added, and they might remain untriggered when a sentence boundary was missed by the sentence boundary detection.

2.4 Controlled Language Checking

A controlled language is a natural language which is restricted by rules aiming at making the language simpler and thus easier to understand [Wojcik and Hoard, 1996]. In other words, a controlled language is a subset of a natural language. The most important aspect in developing a controlled language is to avoid ambiguity on all linguistic levels. A language without ambiguity has two important advantages over the common natural languages:

- It is easier to understand, especially for people who are not native speakers of the original natural language. Even for native speakers the chance of misunderstandings is reduced. This does not only make reading documents easier, it can also be vitally important, for example in documents which describe the maintenance of airplane engines. Actually AECMA Simplified English [AECMA] is used by the aerospace industries for exactly that purpose.
- It is easier to be parsed by a computer, thus being easier to be translated automatically. This promises great savings in the translation process, which is a difficult and expensive task when done completely manually.

The creation of controlled language documents can be supported by software which detects usage of unapproved terms and language constructs. [R.-Bustamente et al, 2000] describes the coverage of some of these controlled language checkers and notes that they have many goals in common with style and grammar checkers. The main difference is that when style and grammar checkers work on unrestricted text they will have to cope with unknown words and complicated sentences. Controlled language checkers work on restricted texts with a limited vocabulary of approved words. To keep word lists in a manageable size there is usually a rule which allows the use of product names even if they are not explicitly listed in the controlled language dictionary. There is no generic dictionary for anybody using controlled languages, instead every industry or even every company will need their own dictionary.

In the context of this thesis the interesting question is to what extent a style and grammar checker for a natural language can be used to check controlled language documents. Typical restrictions for controlled languages might look like this:

- **Lexical restrictions:** for example, a rule might say: use *try* only as a verb, not as a noun. This implies a semantic restriction, but since it is expressed as a syntactic restriction it can be found with a rule which triggers an error message when it encounters *try* as a noun.

- **Grammar restrictions:** for example, rule 5.4 of AECMA says: *In an instruction, write the verb in the imperative ("commanding") form.* Two example sentences are given in the AECMA documentation:
  Approved: *Remove oil and grease with a degreasing agent.*
  Not approved: *Oil and grease are to be removed with a degreasing agent.*
  This might be implemented in a grammar checker by rules which forbid the use of phrases like *is to* and *are to* followed by the passive of verbs like *remove*, *continue*, *set*.

- **Semantic restrictions:** for example, a rule might say: use *noise* only with its meaning *unwanted sound*, not as *electronic interference* (example taken from [Holmback et al, 2000, p. 125]). A style and grammar checker can only discover wrong usages like this if it has an integrated word disambiguation component. Alternatively, the checker can simply warn on every occurrence of *noise*, no matter of its meaning in the given context. This might annoy users if too many warnings are unjustified. If this specific test can be turned off, however, the warning might also be perceived as a handy reminder.

- **Style restrictions:** a rule might demand keeping sentences shorter than 20 words and to stick to one topic per sentence. The length restriction is relatively easy to check, even without a grammar checker. It is enough to check the whitespace-delimited terms in a sentence. There may be special cases like words with hyphens where it is not clear if they should be counted as one or as two words. However, this is not important as the exact number of maximum words does not matter that much – the message is: keep your sentences short. The remaining question is where a sentence starts and where it ends. Assuming that controlled language is mostly used for technical documentation and that this documentation is marked up using XML nowadays it should be easy to distinguish the different periods. For example, an algorithm might assume that a sentence ends whenever a period occurs, unless it is marked up as an abbreviation like `<abbr>etc.</abbr>`.

So it seems to be possible to implement many controlled language restrictions with a style and grammar checker for a natural language. Considering that a controlled language is a subset of a natural language, this is not surprising. Still it makes sense to develop checkers specialized for controlled language checking. The very idea of controlled languages is to use simple grammar, maybe so simple that it is possible to completely analyze each sentence automatically. With this analysis it is possible

to implement more complicated restrictions than with a rule-based checker which works on shallow analyzed text.

2.5 Style Checking

As it was mentioned, natural language grammar provides a clear distinction between sentences which are correct and those which are not. This is not the case for language style. Every writer will have to make his own decision on what style is preferred in which context. This might easily lead to an incoherent style when a document is collaboratively worked on by several people. Style checking is especially useful for these situations, as it reminds writers on a style which was decided on before. Restrictions and rules about style might cover a broad range of things:

- Choice of words: The use of foreign words might be disapproved because not everybody understands them easily. The style checker can then suggest a non-foreign replacement with the same or a very similar meaning. For example, *terra incognita* might be replaced by *unknown territory*. Similarly, it might suggest replacing words which are very uncommon by common synonyms, even if the uncommon word is not a foreign word. For example, the noun *automobile*, which occurs 230 times in the BNC might be replaced with the noun *car*, which occurs about 27,000 times in the BNC[5].

- Simplicity: The length of a sentence might be limited, leading to simpler sentence structures which are easier to understand.
- Punctuation: After a punctuation mark a space character is inserted, but no space is inserted before a punctuation mark (except the opening quote character and the opening parenthesis).
- Dates, times and numbers should be written in a certain format. For example, a date like *5/7/1999* is ambiguous to many people and it should probably be replaced by *1999-07-05* or *1999-05-07* [6].
- Contracted forms like *don't* might be disapproved and could be replaced by *do not*.
- Texts which repeat a given noun too often might sound strange. Instead, pronouns or synonyms should be used to refer to the noun. This rule clearly shows how much "good" style depends on the kind of document: using synonyms to make texts sound better is a common means used by journalists and it is taught in schools. In technical documentation, however, being clear and unambiguous has a higher priority.

Often the important aspect of style is not that some specific rule is chosen, but that one style – no matter which one – is used throughout the document or even throughout a collection of documents. For example, using different date formats in one document is confusing for the reader and looks unprofessional.

2.6 False Friends

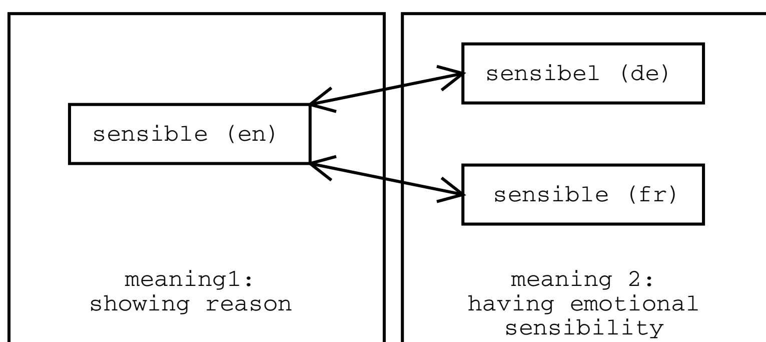
A "false friend" is a pair of words from two different languages where both words are similarly written or pronounced but carry a different meaning. Because the word sounds so familiar a learner of the language will be inclined to use the word incorrectly. Here are some English/German word pairs which are false friends:

[5] Of course one would have to search for word meanings, not just for words if one seriously wanted to interpret the results. In this very case the proportions are so clear that it does not seem to be necessary.

[6] yyyy-mm-dd is the ISO standard for dates, see `http://www.cl.cam.ac.uk/~mgk25/iso-time.html`.

- become – bekommen (become means *werden*)
- actual – aktuell (actual means *tatsächlich*)
- bald – bald (bald means *glatzköpfig*)

It is noteworthy that these words are just as misleading for native speakers of German who learn English as they are for native speakers of English who learn German. As languages often have common roots, the problem is sometimes not just between two languages. For example, *sensible* (en) / *sensibel* (de) is a false friend, as is *sensible* (en) / *sensible* (fr). But *sensibel* (de) / *sensible* (fr) is not a false friend, as the meaning is the same in German and French. This is shown in the following diagram. The arrows mark a false friend relation:



False friends are of interest in the context of this thesis because they can easily be found with a rule-based style and grammar checker. It is just as easy to suggest an alternative meaning to the user. Once the user is aware of the problem, he can turn off this specific rule so he will not get further warnings about this word – all other false friends will still show warnings until they are turned off, too.

2.7 Evaluation with Corpora

A grammar checker system can be evaluated with two common measures known from information retrieval: precision and recall. These are values between 0 and 1 (sometimes expressed as a percentage) which are defined as follows:

$$precision = \frac{real\,errors\,found}{sentences\,tagged\,as\,erroneous}$$

In other words, precision measures how many of the sentences flagged as incorrect by the software are indeed erroneous.

$$recall = \frac{real\,errors\,found}{errors\,in\,the\,text}$$

Recall measures how many of the errors in a text are found, i.e. how complete the software is. Obviously a system with precision = 1 and recall = 1 can be considered to work ”perfectly”. In practice, there is usually a tradeoff between both.

POS taggers are usually written to be robust, i.e. they cannot miss a word like a grammar checker can miss an error. Instead the tagger can return more than one tag per word, so *precision* is sometimes measured by the average number of tags per word. Then, *recall* is the probability that one of the returned tags is correct. As one can see, it is easy to develop a tagger with recall 1, it just needs to return all possible tags of a word – which obviously leads to the worst precision possible.

To develop and evaluate a style and grammar checker, one needs a corpus of unedited text, as this kind of text reflects the common input to a style and grammar checker. All errors (except for spelling errors) need to be marked up in this corpus. As such a corpus is not available, one can also work with two corpora: one corpus – which should be free of errors – is used to optimize the precision so that the system does not give false alarm too often. The other corpus – which contains only sentences with grammar errors – is used to optimize the recall so that many errors are found. Nevertheless, a significant recall/precision value must be measured with the single corpus, everything else would be too biased.

Some corpora which can be used for certain aspects of the grammar checker development will now be described.

2.7.1 British National Corpus

The British National Corpus [BNC] is a commercial corpus of British English which contains 100 million words. It was built from 1991 to 1994, so it contains texts from before 1991, but not from after 1994. About 90% of the words stem from written language, the rest stems from spoken language. The corpus contains a broad range of text types, e.g. fictional texts, technical documentation, newspaper articles etc. Because the texts are copyrighted, only parts of them (start, middle, or end) are part of the BNC.

All the BNC's texts are SGML-formatted. The markup consists of the usual meta information like author, headline and date, and it encloses paragraphs, sentences and words. Each word is assigned one of 61 tags (see section A.4). Some collocations are taken as a single word, e.g. *up to* is tagged as one preposition.

As the part of the BNC which is based on written language comes mostly from published sources, one can assume that it contains very few grammar errors. Style is, as mentioned before, mostly a matter of definition, so one cannot make statements about the presumable number of "style errors" in the BNC.

The BNC itself may not be given to people who do not have a BNC license, but its license explicitly has no restrictions on the use of the results of research with the BNC. So when developing a software system, one may use the BNC during development, but neither the BNC nor parts of it may be part of the resulting software.

Technically, the BNC comes as a set of compressed SGML data files and with software to query the data. However, this software has several drawbacks: it is a client server/system, and as the server (`sarad`) only works on Unix/Linux and the client (SARA) only works on Windows, it requires two computers even if there is only one user and there is no actual need for network access to the server. The Unix/Linux version of the server only comes with a very simple command line tool (`solve`) for querying. Furthermore, the server and this query tool are difficult to set up compared to today's standards. The installation instructions on the web are partially out of date[7].

The BNC can also be queried on the BNC web page at `http://sara.natcorp.ox.ac.uk/lookup.html`, even without a BNC license. The query language makes it possible, for example, to search for words which occur in a given part of speech and to use regular expressions. Unfortunately, the search facility is rather limited in other respects: a search for only POS tags without a specified word is not possible. A query like *dog*, which results in 7800 matches, might take 40 seconds, but only the first 50 matches are shown and there is no way to get the remaining matches. No POS annotations are displayed, and the matched words themselves are not highlighted, which makes scanning the result more difficult. During my tests, there where also several timeout and server errors so that some

[7] `http://www.hcu.ox.ac.uk/BNC/SARA/saraInstall.html`. For example, the option to hand a confi guration fi le to `sarad` is specifi ed as `-c`, whereas it must be `-p`.

queries were only answered after several tries or not at all.

As an alternative query system a software called [BNCweb] is available. It works web-based and it requires a running BNC server and a MySQL database. According to its description on the web it seems to be feature-rich, but it is not available for free.

2.7.2 Mailing List Error Corpus

So far no corpus specialized in grammar errors is publicly available[8]. Because of the importance of an error corpus for optimizing the style and grammar checker, a new corpus was developed.

The corpus contains 224 errors found mainly on international public mailing lists used for Open Source software development[9]. Most messages discuss technical issues like programming or the software development process. Many of the writers on those mailing lists are not native speakers of English. However, as the native language of people is often not obvious, this information has not been recorded. Some sentences have been shortened when it was clear that the error is completely unrelated to the part which has been removed. Other sentences have been slightly edited, e.g. to fix spelling errors which are not part of the grammar error. The distribution of errors is shown in the following table:

Category	**# of errors**	**%**
Confusion of similar words, e.g. *your* vs. *you're*	94	42.4
Agreement errors	64	28.7
Missing words	24	10.8
Extra words, e.g. **more better*	17	7.6
Wrong words, e.g. confusion of adjective/adverb, wrong prepositions	17	7.6
Wrong word order	4	1.8
Comma errors	3	1.3

The corpus data is biased in so far as only errors are listed which I noticed during normal reading of messages. In other words, no systematical search was done, and only those errors are listed which I could intuitively recognize as such. A more sophisticated approach to building an error corpus is listed in [Becker et al, 1999]. The complete corpus and a description of its XML file format can be found under section A.1.

2.7.3 Internet Search Engines

Finally, it is sometimes practical to use Internet search engines for finding given words or phrases. For example, the error corpus contains the following part of a sentence:

> **But even if it's looking fine,* ***the is*** *the problem that...*

In this fragment, *the is* is obviously an erroneous phrase. One might consider a rule which suggests *there is* as a replacement, whenever *the is* occurs. It is necessary to test if *the is* is really always incorrect. I will limit this chapter to Google, because it is the search engine which covers the largest

[8] Bill Wilson collected more than 300 ill-formed sentences, but most of the errors are just spelling errors: `ftp://ftp.cse.unsw.edu.au/pub/users/billw/ifidb`. The Cambridge Learner Corpus contains more than 5 million words and errors are annotated, but the corpus is not publicly available: `http://uk.cambridge.org/elt/corpus/clc.htm`

[9] For example: kde-core-devel@kde.org, kfm-devel@kde.org, kmail@kde.org, dev@openoffice.org. Archives for the former ones are available at `http://lists.kde.org/`, for the latter one at `http://www.openoffice.org/servlets/SummarizeList?listName=dev`.

number of pages. At the time of writing, the Google homepage claimed: *"Searching 3,083,324,652 web pages"*.

The query *"the is"* (note that the quotes are part of the query) used on Google returns 519,000 matches. Usually Google ignores very common words like *the* and *who*, but not so in phrase queries, which are carried out when the search terms are surrounded by quotes. Some of the matches are:

1. *What **the** !@#$ **is** this?*
2. *Jef Raskin - **THE Is** Not An Editor... So What Is It?*
3. *About **the IS** Associates*
4. ***The Is** Ought Problem*
5. *What **the is** this site for?*

Here one can see several reasons why *the is* might occur in a text: Match 1 contains a sequence of special characters which Google does not consider a word, so it pretends *the* and *is* are subsequent words here. Matches 2, 3 and 4 only match because Google's queries are always interpreted case-insensitive. *THE* and *IS* are names (probably acronyms) here and would not have been matched with a case-sensitive search. Finally, match 5 is an actual error, so the rule which says that *the is* is always wrong works correctly in this case. Unfortunately the huge number of matches prevents manual checks for each match, so it is not possible to use this result to prove that the rule is always correct. Still the few matches which have been checked are a good indication that the rule is useful.

Obviously Google is not useful as a replacement for a real corpus like the BNC. Firstly, Google only knows about words, not about POS tags. Google does not even offer stemming, i.e. the term *talks* will be matched only when searching for *talks*, not when searching for *talk* or *talking*. Secondly, Google can be limited to search only pages in a given language, but it cannot be told to search only pages which have been written by native speakers of English. It also cannot limit its search to categories like "technical documentation" or "oral speech"[10]. Thirdly, the Google database is constantly changing. The example given above might not be exactly reproducible anymore when you read this.

The advantage of Google is its size of some thousands of million pages. Even if just 30% of these pages are written in English, this is still much more than the BNC has to offer. Nonetheless Google is extremely fast, the *"the is"* query returned its first ten matches in 1.49 seconds[11]. Furthermore, Google is up-to-date and contains modern vocabulary, whereas the BNC collection ends in 1994. For example, the BNC only contains two occurrences of *World Wide Web*.

2.8 Related Projects

2.8.1 Ispell and Aspell

Ispell [Kuenning] and Aspell [Atkinson] are both very popular Open Source spell checkers. Most Open Source word processors make use of these programs in one way or another. For example, KWord provides an integrated interface to Ispell and Aspell. OpenOffice.org comes with MySpell, which is based on Ispell. In both cases, the user does not notice that it is Ispell/Aspell which does the real work in the background.

Spell checkers compare each word of a text to their large lists of words. If the word is not in their list, it is considered incorrect. In other words, spell checking is a very simple process which does not

[10] Although one might try to come up with queries that find only such documents, e.g. *"Al Gore speech"*.

[11] Google displays this number. Obviously it might take longer until the result appears on the user's screen because of slow network connections etc.

know anything about grammar, style or context. It is mentioned here nonetheless, because it could be considered a subset of a complete grammar checker. The integration of a spell checker is described in section 3.4.

2.8.2 Style and Diction

Style and Diction are two classical Unix commands [Style/Diction]. Style takes a text file and calculates several readability measures like the Flesch Index, the fog index, the Kincaid score and others. It also counts words, questions and long sentences (more than 30 words by default). It is not an interactive command, but it allows to specify options to print, for example, all sentences containing nominalizations or sentences with passive voice. The complete matching sentences will then be printed, without further indication where exactly the match is.

Diction takes a text and searches for certain words and phrases, for which it prints a comment. For example, the following text used as input for diction:

> *I thought you might find it interesting/useful, so I'm sending it to the list here.*

...will produce the following result:

> *I thought you* [might -> (do not confuse with "may")] *find it* [interesting -> Avoid using "interesting" when introducing something. Simply introduce it.]*/useful,* [so -> (do not use as intensifier)] *I'm sending it to the list here.*

As one can see at *so*, diction warns for *every* occurrence of certain words and gives a short statement about possible problems with the given phrase. Here are some more samples from the diction data file:

Matching phrase	Suggested replacement or comment
as a result	so
attempt	try
data is	data are
in the long run	(cliche, avoid)

The diction data file for English contains 681 entries, its German data file contains 62 entries. Except very few hardcoded exceptions like *data is / data are*, diction does not check grammar.

2.8.3 EasyEnglish

EasyEnglish is a grammar checker developed at IBM especially for non-native speakers. It is based on the English Slot Grammar. It finds errors by "exploring the parse tree expressed as a network" [Bernth, 2000]. The errors seem to be formalized as patterns that match the parse tree. Unfortunately [Bernth, 2000] does not explain what exactly happens if a sentence cannot be parsed and thus no complete tree can be built.

EasyEnglish can find wrong articles for countable/uncountable nouns (e.g. **an evidence*) and missing subject-verb agreement amongst other mistakes. It has special rules for native speakers of Germanic languages and for native speakers of Japanese. For speakers of Germanic languages it checks for overuse of the progressive form (e.g. **I'm having children*), wrong complement (e.g. **It allows to update the file* instead of *...allows updating...*) and false friends. There are other rules useful especially for speakers of Japanese. All rules are optional.

EasyEnglish does not seem to be publicly available, neither for free nor as a commercial tool.

2.8.4 Critique

Critique is a style and grammar checker that uses a broad-coverage grammar, the PLNLP English Grammar [Jensen et al, 1993, chapter 6]. It detects 25 different grammar errors from the following five categories: subject-verb agreement, wrong pronoun case (e.g. **between you and I* instead of *between you and me*), wrong verb form (e.g. **seem to been* instead of *seem to be*), punctuation, and confusion of similar words (e.g. *you're* and *your*). Furthermore, Critique detects 85 different style weaknesses, like sentences that are too long, excessive complexity of noun phrase pre-modifiers, and inappropriate wording (e.g. short forms like *don't* in formal texts).

Errors are shown to the user with the correct replacement if possible. The user can get an explanation of the problem, which is especially important for style issues for which the distinction between correct and incorrect is often not that simple. Critique can also display the parser's result as a tree. All style and grammar checks can be turned on or off independently.

For each sentence, Critique first tries to analyze the complete sentence with its parser. If the parsing succeeds, the sentence is grammatically correct and it is then checked for style errors. The style errors are found by rules that work on the parsed text. If the parsing does not succeed on the first run, some rules are relaxed. If the sentence can then for example be parsed with a relaxed subject-verb agreement rule, a subject-verb agreement error is assumed. The style checking will then take place as usual.

Interestingly, the authors of [Jensen et al, 1993, chapter 6] suggest that ideally the parser should parse any sentences, even the incorrect ones. This way all grammar and style checking could be done with the same component, namely rules that work on a (possible partial) parse tree. This however is not possible, as a grammar with few constraints will lead to many different result trees and will thus become very slow. Still, some rules have been changed from being a condition in the grammar checker to being a style rule.

Critique does not seem to be publicly available.

2.8.5 CLAWS as a Grammar Checker

CLAWS (Constituent Likelihood Automatic Word-tagging System, [CLAWS]) is a probabilistic part-of-speech tagger. [Attwell, 1987] offers a suggestion on how to use such a tagger as a grammar checker. The idea is to get the POS tags of a word and its neighbors and then check how common these sequences are. If the most common combination of tags is still below a given threshold, an error is assumed. Possible corrections can then be built by substituting the incorrect word with similarly spelled words. The substituted word that leads to the most common POS tag sequence can then be suggested as a correction.

The advantage of this probabilistic approach is that it does not require hand-written rules. It can even detect errors which were not in the corpus originally used for training CLAWS. On the other hand, a threshold needs to be found that works best for a given text. Also, only errors which are reflected in the POS tags can be found. For example, *sight* and *site* are both nouns and thus this kind of probabilistic checker will not detect a confusion between them. Also, the error message cannot be accompanied by a helpful explanation.

CLAWS is available for a fee of £750. An online demo is available on its web site. The extension for detecting grammar errors does not seem to be available.

2.8.6 GramCheck

GramCheck is a style and grammar checker for Spanish and Greek, optimized for native speakers of those languages [R.-Bustamente, 1996, R.-Bustamente, 1996-02]. It is based on ALEP (Advanced

Language Engineering Platform, [Sedlock, 1992]), a development environment for linguistic applications. ALEP offers a unification-based formalism and a graphical user interface that lets linguists develop new grammars.

GramCheck detects non-structural errors with a constraint relaxation technique. In its unification based grammar, Prolog code tries to unify features. The code also builds the corrections, depending on which relaxation was necessary to parse the input text. Structural errors are detected by error patterns which are associated with a correction pattern.

GramCheck can detect errors in number and gender agreement, and incorrect omission or addition of some prepositions. It can detect style problems like abusive use of passive and gerunds and weaknesses in wording.

Neither GramCheck nor ALEP are publicly available.

2.8.7 Park et al's Grammar Checker

[Park et al, 1997] describe a grammar checker which is optimized for students who learn English as a second language. Students' essays have been analyzed to find typical errors.

The checker is based on a Combinatory Categorial Grammar implemented in Prolog. In addition to the broad-coverage rules, error rules have been added that are applied if all regular rules fail. One of the error rules might for example parse a sentence like **He leave the office* because the subject-verb agreement is relaxed in that rule. All error rules return a short error message which is displayed to the user. No correction is offered, as this would degrade the learning effect. The user interface is a text field on a web page in which the text can be typed and submitted.

The checker detects missing sentence fragments, extra elements, agreement errors, wrong verb forms and wrong capitalization at the beginning of a sentence. It is not publicly available.

2.8.8 FLAG

FLAG (Flexible Language and Grammar Checking, [Bredenkamp et al, 2000, Crysmann]) is a platform for the development of user-specific language checking applications. It makes use of components for morphological analysis, part-of-speech tagging, chunking and topological parsing.

FLAG detects errors with so-called trigger rules indicating the existence of a potential problem in the text. For potentially incorrect sentences confirmation rules are called which carry out a more complicated analysis. There are confirmation rules which advise that there is actually no problem, and others that advise that there is indeed an error. Each rule carries a weight, and if more than one rule matches, these weights are added up. Based on the final weight, FLAG then decides whether the rule really matches and thus whether there is an error in the sentence.

This two-step approach helps increase the system's speed, as the trigger rules are rather simple and can easily be checked, whereas the more complicated and thus slower confirmation rules only need to be called for potentially incorrect sentences.

All of FLAG's rule are declarative. It also knows terminology rules and can generally work for any language. So far, only a few rules for German have been implemented. The addition of a significant number of grammar rules is only listed in the "Future Work" section in [Bredenkamp et al, 2000].

FLAG is implemented in Java and C++ and has a graphical user interface for checking texts. FLAG is not publicly available.

3 Design and Implementation

My old style and grammar checker was written in Perl [Naber]. It relied on Eric Brill's part-of-speech tagger [Brill, 1992], which is implemented in C. The Perl script normalized and formatted the text input so that it could be used as input to Brill's tagger. The tagger's output was then parsed again and the style and grammar rules were applied. This mixture of languages lead to an implementation which emphasized implementation details and neglected the larger structure of the program. As Perl requires a rather obscure way of object-oriented programming, the Perl script used only traditional, non object-oriented programming techniques. The tagger required a C compiler, which does not exist by default on e.g. Windows systems. Extending and improving a C program is also much more complicated than it is for a program written in a modern scripting language.

To improve the design, maintainability and extensibility of the style and grammar checker, both the tagger and the checker have been re-implemented in Python. Python's main properties are:

- It is system independent, i.e. it runs at least on Unix/Linux, Windows and Mac.
- It supports object-oriented programming techniques and it makes use of object-orientation in its own standard library.
- It allows for fast application development thanks to its built-in support for common data types like lists, dictionaries (hash tables) and a powerful standard library which supports e.g. regular expressions, object serialization, and XML parsing.
- It supports Unicode in a natural way, i.e. input strings are decoded (interpreted) in a given encoding and can then be accessed as Unicode strings. Output strings are encoded in a given encoding.
- It is implicitly typed, i.e. a programmer will not have to define the type of a variable, but the variable will have a type nonetheless. The type of a variable can be changed, as can be seen in this example from the interactive Python interpreter:
  ```
  >>> a = 5.4
  >>> type(a)
  <type 'float'> # 'a' is a float
  >>> a = "hello"
  >>> type(a)
  <type 'str'> # now 'a' is a string
  ```
  This dynamic typing, which is less strict than in compiled languages like C++ and Java, may lead to situations where problems occur at runtime instead of at compile time. This problem can, to a certain extent, be avoided by the use of unit tests, as described in section 3.12.

3.1 Class Hierarchy

The structure of the central classes is shown in the UML diagram in figure 1. The classes which contain unit tests are not part of the diagram. Also, attributes have no type information because Python is implicitly typed. All attributes and methods are public as Python does not support private attributes or methods. `TextChecker` is the main class. It instantiates a `Rules` object so it can access all rules. `TextChecker`'s `check()` method takes a text string and returns an XML string (see below). `check()` first calls `SentenceSplitter`'s `split()` method to detect the sentence boundaries in the input text. Then, each sentence is tagged with `Tagger`'s `tagText()` method, and chunks are detected using `Chunker`'s `chunk()` method. Now, `check()` tests whether each rule matches using the rule's `match()` method which takes the sentence, the tags, and the chunks.

The result of `match()` – a list of `RuleMatch` objects – is appended to the list containing all errors for all sentences. Once all rules are checked, the list of all errors is returned as a list and additionally as an XML string.

`TextChecker.py` can also be called from command line to check a plain text file. It offers options so that any rule and feature can be turned on or off separately. All options are listed with the `TextChecker.py --help` command. The output is an XML encoded list of errors with `<errors>` as the root element. Unlike the [Duden Ling. Engine] for example, which also returns XML, the original text is not repeated, only the errors are displayed. For example, the following output complains about an incorrect comparison from character 21 to character 25 in the original text and suggests *than* as a replacement:

```
<errors>
  <error from="21" to="25"><message>Comparison
    requires <em>than</em>.</message></error>
</errors>
```

3.2 File and Directory Structure

The following files and directories are part of the style and grammar checker. Their installation and implementation will be covered in the next sections:

- `data/abbr.txt`: pre-defined abbreviations to improve sentence boundary detection
- `data/words`: word/tag probability data
- `data/seqs`: tag sequence probability data
- `kword/`: patches for KWord
- `snakespell-1.01/`: the Snakespell module, allows access to Ispell via Python
- `rules/`: style and grammar rules in XML format and its DTD
- `python_rules/`: style and grammar rules written in Python
- `socket_server.py`: a server which provides access to the style and grammar checker via a local TCP socket
- `client.py`: a test script to check whether `socket_server.py` is running
- `config.py`: the configuration file, needed to set the path where the style and grammar checker is installed
- `TextCheckerCGI.py`: a CGI script to access the style and grammar checker via a web form
- `query.py`: a CGI script to query BNC files in XML format
- `tag.py`: a script to train the tagger and to tag texts
- `TextChecker.py`: the main module which implements the `TextChecker` class. Can also be called from the command line to check text files.
- `Tagger.py`, `Rules.py`, `Tools.py`, `TagInfo.py`, `SentenceSplitter.py`, `Chunker.py`: Python modules which are used by the style and grammar checker backend. These files cannot be called directly.

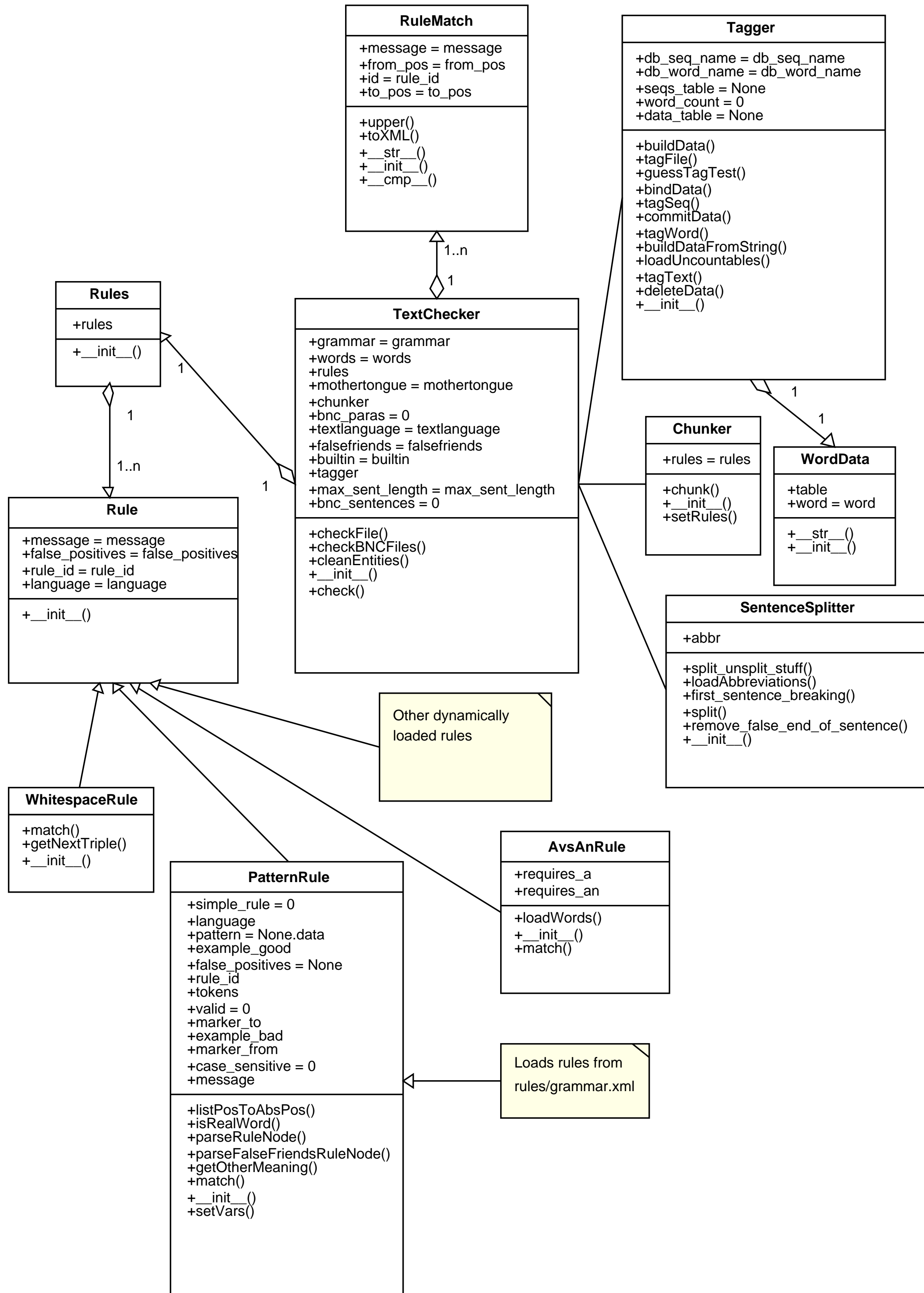


Figure 1: UML class diagram of the backend

- TaggerTest.py, SentenceSplitterTest.py, ChunkerTest.py, RulesTest.py, TextCheckerTest.py: Test cases for the modules above. These files can be called to run the unit tests.

- README, COPYING: short documentation and copyright information

3.3 Installation

3.3.1 Requirements

The style and grammar checker backend needs the following software. The listed versions numbers are those versions which have been tested. Later versions should work without problems, earlier versions might fail:

- Python 2.3c2[12] (http://www.python.org)

To make use of the checker integration into KWord, the following software is required:

- KWord 1.3beta1 source code (http://www.koffice.org), which requires at least[13]:
  - kdelibs 3.1 (http://www.kde.org)
  - Qt 3.1 (http://www.trolltech.com)

Both the checker backend and the frontend have been developed and tested under Linux only. The backend should also run on Windows and Mac without problems. The frontend is integrated into KWord, which is only available on KDE, which in turn only runs on Unix/Linux[14]. The backend can be connected via standard sockets as they are used for Internet communication, so it should not be a problem to integrate it into word processors on Windows or Mac.

3.3.2 Step-by-Step Installation Guide

First, the style and grammar checker backend needs to be installed:

1. Unpack the program archive in your home directory:
   ```
   > tar xvzf languagetool-1.0.tar.gz
   > cd languagetool
   ```
   Set the installation directory in the BASEDIR option in config.py. Now test the checker with a command like this:
   ```
   > ./TextChecker.py test.txt
   ```
   test.txt should be a short test which contains a grammar error, for example: *Peter's car is bigger then mine.* The output should be an XML encoded error message like this (whitespace has been added here for better readability):
   ```
   <error from="15" to="26">
   <message>Comparison requires <em>than</em>.</message>
   </error>
   ```

[12]Python 2.2.x seems to have a bug in its minidom implementation which sometimes prevents the error rules from being correctly parsed.

[13]Obviously a C++ compiler and the usual tools like make are also required. This list does not contain the obvious requirements.

[14]There is a project which tries to port KDE to Windows: http://kde-cygwin.sourceforge.net/

2. Optionally, you may want to install the CGI frontend. This requires a web server with support for CGI scripts. Just set a link from your web server's CGI directory to the CGI script[15]:
   ```
   > ln -s /full/path/to/TextCheckerCGI.py \
   /srv/www/cgi-bin/TextCheckerCGI.py
   ```
   Then call `http://localhost/cgi-bin/TextCheckerCGI.py` in your browser and submit a test sentence.

Now the KWord sources can be installed. One should first try to make KWord work without any patches. This takes a bit longer because some things need to be compiled twice, but it makes it easier to spot the source of problems:

1. Once all requirement (Qt, kdelibs, ...) are fulfilled, unpack the KOffice 1.3beta1 archive in your home directory, configure and compile it:
   ```
   > tar xvjf koffice-1.2.90.tar.bz2
   > cd koffice-1.2.90
   > ./configure --prefix=/usr/local/
   ```
   To save compile time, you can optionally remove some programs from the `TOPSUBDIRS` line in the `Makefile`: karbon, kchart, kdgantt, kformula, kivio, koshell, kounavail, kpresenter, kspread, kugar, kexi
   Now compile the source code:
   ```
   > make
   > make install
   ```
2. Check whether KWord starts correctly:
   ```
   > /usr/local/bin/kword
   ```

Now the patches which make KWord use the style and grammar checker need to be applied:

1. Apply the patches and copy the modified files:
   ```
   > cd lib/kotext
   > patch <~/languagetool/kword/kword.diff
   > cp ~/languagetool/kword/koBgSpellCheck.cc ./
   > cp ~/languagetool/kword/koBgSpellCheck.h ./
   > cd ../..
   > cp -r ~/languagetool/kword/language tools/
   ```
2. Compile and install the modified files and KWord (which also requires a recompile due to the changes):
   ```
   > make install
   ```
3. In a different shell, start the checker server:
   ```
   > ./socket_server.py
   ```
4. Set the `LANGUAGETOOL` environment variable and restart KWord:
   ```
   > export LANGUAGETOOL=~/languagetool/
   > /usr/local/bin/kword
   ```
   Make sure that the menu item Tools->Spellcheck->Autospellcheck is activated. Type in this test sentence:
   Peter's car is bigger ***then*** *mine, and this* ***isa*** *spelling error.*

[15]The Apache web server may be configured to deny access to links for security reasons. The linked script will not work in that case. If security is not an issue (e.g. for local testing), this can be changed with this option in `httpd.conf`: `Options +FollowSymLinks`

After a short time, a red line should appear underneath the *isa* typo, and a blue line should appear under the incorrect *then*. Correct the errors and the lines should disappear almost immediately.

3.4 Spell Checking

Users might be confused by a software which does something as complicated as grammar and style checking, but which does not include something as simple as spell checking. Although spell checking should happen before the grammar checker gets a text, it makes sense – from the user's point of view – to integrate both into one program.

Snakespell[16] is a Python module which allows accessing a local Ispell installation via Python. Its `check()` method takes a string and returns the number of spelling errors in this string. The errors can then be queried with the `getPositions()` method, which provides access to Ispell's list of possible corrections (i.e. words which are written similar to the erroneous word).

Snakespell is included in the `languagetool/snakespell-1.01` directory, so it does not have to be installed separately.

3.5 Part-of-Speech Tagging

The POS tagger used by the style and grammar checker is a probabilistic tagger based on Qtag (see section 2.1) with a rule-based extension. The POS tagger is implemented in the `Tagger` class in `Tagger.py`. The most important methods are `buildData(filename)`, which takes the filename of a BNC file and learns tag and tag sequence probabilities, and `tagText(text)`, which is used to tag plain text.

`buildData()` loads the BNC file and uses a regular expression to find all its words and their assigned tags. These word/tag tuples are put into a list and handed to the `addToData()` method. This method builds a dictionary (the Python term for *hash table*) which maps each word to a list of possible tags. Each tag carries a value which represents the tag's probability for this word. For example, the dictionary might contain a mapping `word_to_tag['walk'] = [(VB, 0.6), (NN, 0.4)]`. This mapping means that the word *walk* occurs as a verb (VB) with a probability 0.6, and as a noun (NN) with a probability 0.4.

Two other dictionaries map 2-tag-sequences to probability values. For example, the dictionary might contain a mapping `sequence[(DET, AJ0)] = 0.1`. This means that the probability that `DET` is followed by `AJ0` is 0.1. Another dictionary contains the opposite direction, e.g. `AJ0` preceded by `DET`. The `bindData()` method finally serializes all dictionaries and saves them to disk.

`tagText()` is used to tag any text. First it splits the text at whitespace characters. For each element of the list, except whitespace elements, all possible tags are looked up using the `tagWord()` method. There are special cases like *of course*, which are treated as one word by the BNC. So for each element in the list, it is first checked whether it is in the list of known words when the following word is appended to it. If so, both words are merged to one word before given to the `tagWord()` method.

If a word is not known, its tag is guessed according to some simple rules:

1. If the word starts with a capital letter, assume a proper noun (NN0).
2. If the word ends in *dom, ment, tion, sion, ance, ence, er, or, ist, ness,* or *icity,* assume a singular noun (NN1).
3. If the word ends in *ly*, assume an adverb (AV0).

[16] Available at `http://www.scriptfoundry.com/modules/snakespell/`.

4. If the word ends in *ive, ic, al, able, y, ous, ful,* or *less*, assume an adjective (AJ0).
5. If the word ends in *ize, ise, ate,* or *fy*, assume the infinitive form of a verb (VVI).

The suffixes used in these rules have been collected on the web[17], where some suffixes like *en* can indicate either an adjective (e.g. *wooden*) or a verb (e.g. *soften*), so they are not used. As soon as one of the rules matches, the guess takes place and the other rules are ignored. Thus the order of rules is important, e.g. the ending *ly* has to be checked first, as there is also an ending *y*.

Next, the tag sequences' probabilities of the current word's tag and its neighbors are looked up. As some words can have more than one tag and the most probable tag has not yet been chosen, all combinations of all tags are checked. As mentioned, the tagger always looks at a word and both its neighbors, i.e. it looks at a three-token-sequence. As an example, assume the tagger is currently looking at the sequence *like fat food*. First, these word probabilities will be looked up (these vales are just examples, not the real values):

like	PRP=0.7, VVI=0.3
fat	NN1=0.6, AJ0=0.4
food	NN1=1

`tag.py` allows to look up these values manually for debugging and testing purposes with a command like this: `./tag.py --wordtag food`

Now all possible POS tag combinations are looked up:

PRP NN1 NN1	0.1
VVI NN1 NN1	0.02
PRP AJ0 NN1	0.2
VVI AJ0 NN1	0.05

The tag and sequence probabilities are multiplied:

```
P(like/PRP) * P(PRP NN1 NN1) = 0.7 * 0.1  = 0.07
P(like/VVI) * P(VVI NN1 NN1) = 0.3 * 0.02 = 0.006
P(like/VVI) * P(VVI AJ0 NN1) = 0.3 * 0.2  = 0.06
P(like/PRP) * P(PRP AJ0 NN1) = 0.7 * 0.05 = 0.035
```

In this case, the probability that PRP should be assigned to *like* is highest. The algorithm will advance to the next token. Dummy entries are inserted at the end (and also at the start, but we ignore that in this example) of the string, so that again a 3-token-window exists, which is now: *fat food Dummy*

The probability calculation starts again, so that each token finally carries three combined probabilities, one for each position in the three token window. These probabilities are summed up, and the tag with the highest probability is selected.

3.5.1 Constraint-Based Extension

A probabilistic tagger can easily be trained with a POS-tagged corpus. But the tagger's results will completely depend on this corpus, which makes it difficult to understand the tagger's result if it is not correct. So it might prove useful to add rules which take precedence over the result of the probabilistic

[17]For example at `http://depts.gallaudet.edu/englishworks/reading/suffixes.html`.

part of the tagger. These rules have to be developed manually, but their effect is easily understood, which makes changing and extending them simple.

The tagger implemented here features the `applyConstraints()` method, which takes the current word, its context, and all possible POS tags for the current word. It can remove one or more of the possible POS tags, so that the probabilistic tagger will only select a tag from the reduced list. Obviously, if all but one tag is removed, the probabilistic tagger will choose the only remaining tag, ignoring its probability.

Technically the `applyConstraints()` method has access to all possible tags of the current word and their probabilities. It could add other tags or modify the probabilities as well, even though this is not the idea of constraints.

3.5.2 Using the Tagger on the Command Line

The `tag.py` script can be used to train the tagger and to tag text files. This is only necessary during development. Data files with frequency information are part of the style and grammar checker, so there is usually no need for the user to build his own data files, unless he wants to port the style and grammar checker to a new natural language for which a tagged corpus is available.

To train the tagger on a tagged corpus, the `tag.py` command is called like this:

```
./tag.py --build A0*
```

This will take all files which match the pattern `A0*` and will add all word and POS frequency information to the files `words`, `seqs1`, and `seqs2` in the `data` directory. If one wants to start from scratch, those files need to be deleted manually before calling the command. The input data needs to be in SGML format with elements like this (this is the format of the BNC Sampler):

```
<w NNB>Mr <w NP1>Maxwell <w VBDZ>was <w II>at ...
```

To use the now trained tagger to tag a text file, `tag.py` is called like this:

```
./tag.py --tag test.txt
```

For a text file

```
This is a stoopid test.
```

the result looks like this:

```
<!-- Statistics:
count_unknown = 1 (20.00%)
count_ambiguous = 1 (20.00%)
-->
<taggedWords>
 <w term="This" type="DT0">This</w>  <w> </w>
 <w term="is" type="VBZ">is</w>  <w> </w>
 <w term="a" type="ZZ0">a</w>  <w> </w>
 <w term="stoopid" type="unknown">stoopid</w>  <w> </w>
 <w term="test" type="NN1">test</w>  <w>. </w>
</taggedWords>
```

Words for which the tag selection was ambiguous are counted, as are those words for which no POS tag was found in the lexicon.

When `tag.py` is called with an SGML file instead of a text file, it checks if the file contains `<w>` elements. If it does, the SGML file is assumed to contain a BNC text. `tag.py` then enters a special mode which compares the tagger's POS tags with those of the BNC. For each mismatch, it will print a warning:

```
*** tag mismatch: got work/NN1, expected work/['VVI']
```

Here, the tagger assigned *work* the tag `NN1`, whereas the BNC labeled it as `VVI`. In this mode, the statistics are printed as well. This is very useful to improve the tagger, for example by adding constraints, as described in section 3.5.1.

3.5.3 Using the Tagger in Python Code

Here is an example which shows how the tagger can be used from Python code:

```
tagger = Tagger.Tagger("data/tw", "data/t1", "data/t2")
tagger.deleteData()
tagger.bindData()
tagger.buildDataFromString("The/DET tall/ADJ man/NN")
```

First, a `Tagger` object is created. The arguments refer to file names which will be used to save the frequency information acquired in training. Then, all previous data saved in the given files is deleted, i.e. all knowledge about words and their possible tags is removed. Finally, a minimal corpus – just one phrase – is used to train the tagger. This minimal corpus is a text string with pairs in the `word/tag` format.

Now that the tagger is trained, it can be used to tag new text:

```
res = tagger.tagText("The man was tall")
print res
```

This script's output will be the following list of tuples, where the first item is the token and the second item is the most probable POS tag for that token[18]:

```
[('The', 'DET'),
(' ', None),
('man', 'NN'),
(' ', None),
('was', 'unknown'),
(' ', None),
('tall', 'ADJ')]
```

All words which were part of the small training corpus are assigned their correct POS tag, other tags are marked `unknown`. Whitespace has no tags.

[18] Actually, the output also contains another item in each tuple which is just a normalized form of the original word. This normalized form is identical to the original word in this example and has thus been left out.

3.5.4 Test Cases

The tagger's learning and tagging, including its potential to guess unknown words, is covered by test cases in `TaggerTest.py` (see 3.12 for a general description of test cases). Some of these test cases will be explained here.

The following test case makes sure that the token's context is taken into account when assigning tags:

```
r = self.tag("""The/DET fat/NN1 is/VB hot/AJ0
  The/DET fat/AJ0 guy/NN1
  A/DET man/NN1 used/VBD fat/NN1""",
  "A fat man")
self.assertEqual(r, [('A', 'DET'), ('fat', 'AJ0'),
  ('man', 'NN1')])
```

`self.tag()` is a helper method which takes two arguments: the first one is a training corpus, the second one is the text to tag. The method makes sure that all previous data is deleted, so only the corpus from the first argument is used. A list of tuples is returned. The test case makes sure that *fat* is tagged as `AJ0` here, although it appears twice as `NN1` in the corpus, and only once as `AJ0`. The reason is that the sequence `DET NN1 NN1`, which would appear if *fat* was tagged as `NN1`, never occurs in the corpus. *fat* as `AJ0` leads to the sequence `DET AJ0 NN1`, which does occur and this reading is thus preferred.

Other tests make sure the guessing for unknown words works. For example, unknown words with a capital first letter are assumed to be proper nouns (see section 3.5):

```
tag = tagger.guessTagTest("Großekathöfer")
self.assertEqual(tag, 'NP0')
```

There also is a test which makes sure that words whose tag cannot be guessed are not tagged at all:

```
tag = tagger.guessTagTest("verboten")
self.assertEqual(tag, None)
```

3.6 Phrase Chunking

The phrase chunking implemented by the style and grammar checker is rule-based. The `Chunker` class' `chunk()` method takes a POS annotated text and returns a list of (`from`, `to`, `chunk`) tuples. `from` and `to` are the start and end position of a chunk, `chunk` is the chunk's name at that position.

The `Chunker` class reads its rules from `data/chunks.txt`. Entries in this file look like this:

```
NP: NN1 NN1
NPP: NN1 NN2
```

These example entries define noun phrases (NP) and plural noun phrases (NPP). Usually a noun phrase is considered to consist of a determiner, one or more adjectives (optional), and one or more nouns. In this example, what is called noun phrase is just a sequence of two nouns. This makes it possible to add error rules which find determiner-noun agreement errors. These error rules can be used to find an error in the following sentence:

You can measure a **baseball teams quality by other non subjective means, ...*

baseball teams is recognized as a plural noun phrase. The system also has an error rule "a" _NPP, so that the determiner *a* followed by a plural noun phrase is considered to be an error. Actually, the error in this case is the missing apostrophe, the correct phrase is *a baseball team's quality*.

In order to find the longest phrase the rules in `chunks.txt` need to be ordered so that the longest patterns come first, because the chunker uses the first match it finds and then stops.

3.7 Sentence Boundary Detection

As mentioned before, grammar checking works on sentences. Furthermore it is possible to specify rules which refer explicitly to the beginning or end of a sentence. There is also a style rule which considers very long sentences bad style. For all of this, it is necessary to split the text into sentences. Usually the text has no semantic markup, so sentence boundaries are not marked up either.

The sentence detection implemented in this system is based on the Perl module Lingua::EN::Sentence version 0.25[19]. This module has been ported to Python, source code comments and test cases have been added (the Perl version has incomplete source code documentation, and no test cases at all). The new module is called `SentenceSplitter.py`, the test cases are contained in `Sentence-SplitterTest.py`.

The algorithm to find sentence borders is divided into the following steps:

1. Insert sentence borders at positions which contain a punctuation character followed by white-space.
2. Remove those sentence borders which where added in step 1, but which are actually no sentence borders. For example, abbreviations which end in a period are fixed in this step. A list of abbreviations is loaded from `data/abbr.txt`.
3. Once more add sentence borders, for example at positions which contain *no.* (which could be the abbreviation for *number*) but which are not followed by a number.

All these insertions and deletions are performed using regular expressions. The algorithm basically implements some rules and knows about a list of exceptions (the abbreviations). The porting from Perl to Python was rather easy, as Python regular expressions are compatible to Perl's. The syntax, however, differs quite a lot. A regular expression substitution which adds sentence boundaries at all delimiters which are followed by a whitespace looks like this in Perl:

```
$text =~ s/($PAP\s)/$1$EOS/g;
```

The variable `$PAP` is defined as a regular expression which matches any of the punctuation characters period, question mark or exclamation mark, optionally followed by a closing parenthesis or a closing quote. `$EOS` contains a special character which signals a sentence boundary. The same substitution in Python looks like this:

```
text = re.compile("(%s\s)" % PAP).sub("\\1%s" % EOS, text)
```

Since Python does not provide special operators for regular expressions, the code becomes longer.

Here are some short example strings which the checker can correctly split into sentences. These strings are used in the unit tests in `SentenceSplitterTest.py`. The number in parenthesis is the number of sentences which the checker splits the string into:

[19] Available at `http://search.cpan.org/dist/Lingua-EN-Sentence/`.

- *This is e.g. Mr. Smith, who talks slowly... But this is another sentence.* (2)
- *Mrs. Jones gave Peter $4.5, to buy Chanel No 5. He never came back.* (2)
- *Don't split strings like U.S.A. please.* (1)
- *"Here he comes!" she said.* (1)
- *They met at 5 p.m. on Thursday.* (1)

3.8 Grammar Checking

Most grammar errors which the system will recognize are expressed as rules in `grammar.xml` in the `rules` directory. The rule system used is an enhanced version of the system developed before [Naber, chapter 5.2]. Since many rules depend on POS tags, the rules have to be adapted if a different tagger with a different tag set is used. This is the case here: the old system's tagger knows 35 tags – the Penn Treebank tag set –, the new tagger knows 61 tags – the BNC C5 tag set (there is also the BNC C7 tag set with 137 tags, which is used for the BNC Sampler). Despite its much larger number of tags, the new tag set is not simply a superset of the old one: Some tags from the Penn Treebank tag set have more than one equivalent in the BNC tag set, i.e. the BNC tag set is more detailed in these cases. In other cases, more than several tags from the Penn Treebank tag set map to just one tag from the BNC tag set, i.e. the BNC tag set is less detailed for these tags. So in order to use the existing rules with the new tag set, a mapping needed to be developed manually (see section A.3 for the complete mapping).

3.8.1 Rule Features

Rules are expressed in XML format. The DTD can be seen in section A.2.1. The `<rule>` element's parts will be described in the following, together with examples of the according XML representation:

- `id` and `name` attribute: The `id` attribute is used internally, for example to name the rules which should be active and those which should be disabled. It needs to be unique and it may not contain special characters except the underscore. The `name` attribute is used as the name of the rule, as it is visible in the graphical user interface that lets users turn rules on or off. Example:
  ```
  <rule id="SOME_EXTEND" name="Confusion of extend/extent">
  ...</rule>
  ```
- `<pattern>` sub element: This is the most important part of a rule, as it describes what text is incorrect. The pattern is a sequence of words, POS tags, and/or chunks. If this sequence is found in the to-be-checked text, the matching part is considered erroneous. The pattern's parts are separated by whitespace. Any string inside quotation marks is considered a word, any string that starts with an underscore is a chunk name and everything else is considered a POS tag. A pattern has a `lang` attribute, so that it will only be applied to text of that language. As the text language is not automatically detected, it is the user's task to set it, e.g. with the `--textlanguage` option in case `TextChecker.py` is used. The `mark_from` and `mark_to` attributes define what part of the original sentence is marked as incorrect. Both values default to 0, so that the incorrect part is as long as the pattern. A value of 1 for `mark_from` moves the beginning of the incorrect area one token to the right. Similar, a value of -1 for `mark_to` will move the end of the incorrect area one token to the left. Example:
  ```
  <pattern lang="en" mark_from="1">CRD "ore"</pattern>
  ```
  `CRD` is the POS tag which is used to mark up cardinal numbers. So with this pattern, in a text like **Type in one **ore** more words* only *ore* will be marked as incorrect, thanks to the value of

mark_from.
Technically the parts of the pattern are regular expression, so it is possible to use ".*" to refer to any word, or to refer to any kind of verb with V.. Also, the pipe symbol can be used to represent alternatives, as in "(does|did)" (please note that in these examples, the quotes are part of the pattern). The caret character can be used for negation, e.g. ^(DT0) will match any POS tag except DT0 (a determiner). Special tags are SENT_START and SENT_END that match, respectively, the beginning or the end of a sentence. By default, the words in a rule are matched case-insensitive. This can be changed with the case_sensitive="yes" attribute.

- <message> sub element: This is the user-visible error message. It is supposed to be displayed together with the incorrect sentence to let the user know what exactly is wrong about the input sentence. If some text of the message is inside an <em> element, this text will become a clickable link in the checker's user interface. The part of the input sentence which is marked as incorrect will be replaced by the link's text. Example:
  ```
  <message>Did you mean <em>extent</em> ("extent" is a noun,
  "extend" is a verb)?</message>
  ```
  Similar to regular expressions, the \n syntax can be used to refer to parts of the incorrect sentence: \1 will display the first word marked as incorrect, \2 the second one and so on. This can be used to offer a correction to word order errors.
- <error_rate> sub element: This is the number of warnings when using this rule on text from the BNC. It is described in section 4.3.1.
- <example> sub elements: There is at least one example sentence which is correct and another example sentence which violates the rule. Example:
  ```
  <example type="correct">It is, to a certain extent.</example>
  <example type="incorrect">It is, to a certain extend.</example>
  ```

Rules can be combined in the <rulegroup> element. The id attribute is then moved from the <rule> element to the <rulegroup> element. Only a complete rule group can be turned on or off, not its individual rules. This is useful for a more concise list of rules, especially in the graphical user interface.

3.8.2 Rule Development

Here is a short recipe for rule development:

1. Take an error (for example from the error corpus in section A.1) which is not yet detected by the style and grammar checker.
2. Use the erroneous word and the words from its context to make a rule, e.g. **were are* → *we are.*
3. If possible, generalize the rule by using the words' POS tags instead of the words themselves.
4. Check if the rule pattern might also appear in correct sentences. This can be quickly done as described in the following section and with the corpora described in section 2.7. If too many correct sentences are matched, revoke step 3 and try again.

The next section will describe how to test rules. Then some examples will be given to illustrate rule development in detail.

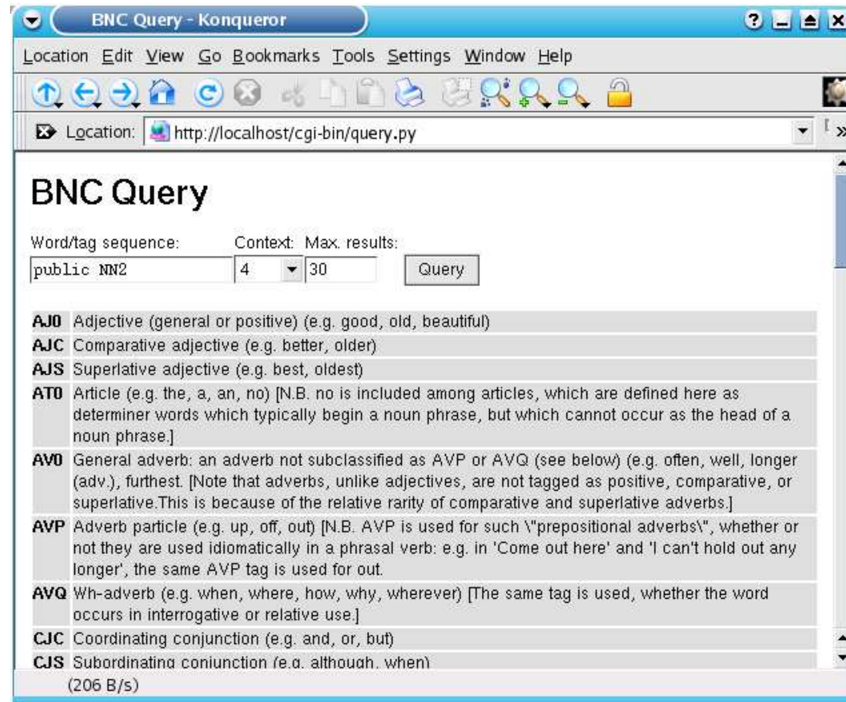


Figure 2: Starting a BNC query in the web interface

3.8.3 Testing New Rules

Each new rule should be checked with real life sentences to make sure it is not triggered by sentences which are in fact correct. As described in section 2.7, the BNC is suitable for this.

To partly overcome the problems with the available BNC query software (see section 2.7.1) a simple web based query script named `query.py` has been developed. Unlike a real client it does not access the data via a server but it loads the data files directly from disk. To make this less error prone the BNC SGML files have to be converted to XML files first, for example using James Clark's `s2x` tool[20].

Because no index is used, the search speed depends on the number of files which have to be searched. For example searching for *public NN2* (the word *public* followed by a plural noun) takes 13 seconds to find the first 30 matches. When a file `<filename>` is queried for the first time, the `query.py` script saves all necessary data to a file `<filename>.pickle`, which is only 1/3 of the size of the original XML file. Although this is technically not an index, it speeds up searching tremendously. About 1400 sentences per second can be searched thanks to this optimization[21].

Figure 2 shows a screenshot of the query page with its input fields and a list which describes all the BNC's tags. Figure 3 shows the result of this query, with the matching words printed in bold.

`query.py` only supports simple queries. For more advanced rules, which make use of e.g. negation, `TextChecker.py` needs to be used on the command line:

```
./TextChecker.py -c -l0 --grammar=RULE_ID --words=None \
  --builtin=None /data/bnc/A/
```

[20] `s2x` is also known as `sgml2xml`. It is part of Clark's parser SP (available at `http://www.jclark.com/sp/`), which also contains `nsgmls`.

[21] These figures where measured on an AMD Athlon 900 MHz with 256 MB RAM.

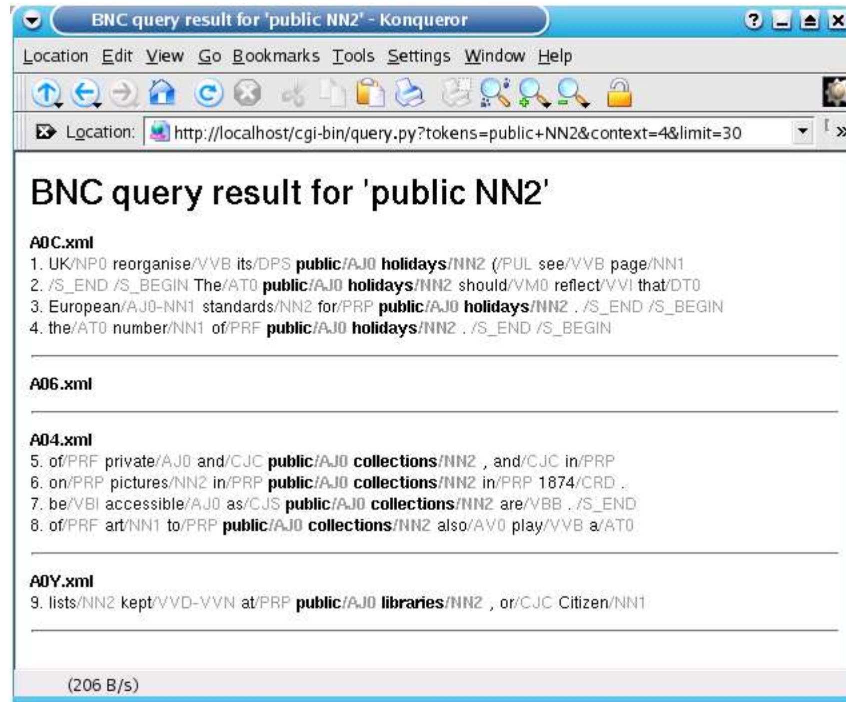


Figure 3: Result of the BNC query

It will check all BNC SGML files in the `/data/bnc/A/` directory. The `-c` switch tells the checker that it is going to work on SGML files, so any markup is removed before the text is checked. The `-l0` option sets the maximum sentence length to unlimited, so that no warnings about long sentences will appear. The `--grammar=RULE_ID` option will deactivate all grammar rules except the given ones (a list of rules may also be provided, separated by commas). Finally, all built-in rules and style checking rules are deactivated by `--builtin=None` and `--words=None`. The command will print out XML code with the error message each time the rule matches.

3.8.4 Example: *Of cause* Typo Rule

I will explain the development of an error rule with an error from the error corpus (see section A.1):

> *****Of* ***cause*** *there is much more to see in the respective regions.*

A trivial way to turn this into a rule is to take the complete sentence as a pattern. This way, if exactly the same sentence occurs again, its error will be recognized. Obviously that is not what one wants, as the error only seems to affect the *of cause* phrase, which could directly be used as a pattern.

Searching a 10% subset of the BNC for the phrase *of cause* gives 5 matches, which are correct phrases like *a relationship of cause and effect*. In comparison, the phrase *of course* gives 918 matches. The idea is now to improve the *of cause* pattern so that it only matches for real errors. The *of cause and effect* phrase can be excluded by extending the pattern so that it matches *of cause*, but only if it is not followed by *and*: `"of" "cause" ^"and"`. This way the correct sentences will not match, but the original error is still recognized.

Now an error message is needed which explains the user what the problem is. As there might still be cases where the rule matches although the sentence is correct, it is a good idea to phrase the message as a suggestion. The message should obviously contain the correct replacement *course*. Now the complete rule in XML looks like this:

```
<rule id="OF_CAUSE"
  name="Confusion of 'of cause/of course'">
  <pattern lang="en" mark_from="1" mark_to="-1">
    "of" "cause" ^"and"</pattern>
  <message>Did you mean "of <em>course</em>"
    (=naturally)?</message>
  <error_rate warnings="0" sentences="75900" />
  <example type="correct">The law of cause and
    effect.</example>
  <example type="correct">Of course there is much more
    to see.</example>
  <example type="incorrect">Of cause there is much more
    to see.</example>
</rule>
```

3.8.5 Example: Subject-Verb Agreement Rule

As a more sophisticated example, the subject-verb agreement rule does not simply work like pattern matching on words. It can for example find the error in the following sentence:

> **The baseball team* ***are*** *established.*

The problem with this sentence is that there is a third-person singular noun phrase *baseball team* but the verb is not in third-person form. Thus a rule is required that matches third-person noun phrases followed by a verb not in third-person form. The example sentence is tagged like this:

```
The baseball team are established.
AT0 NN1      NN1  VBB VVN
```

The sequence of two `NN1`s is recognized as a singular noun phrase (see section 3.6). The verb *are* is tagged as `VBB` (present tense of *be*, except *is*), so a pattern rule that finds this problem is `_NP VBB`. The underscore simply indicates the reference to a chunk. This way chunk names and POS tag names will not be confused.

Other verb forms which are incorrect here can be added so that the XML rule finally looks like this:

```
<rule id="_NP_INF" name="Subject/Verb agreement">
  <pattern lang="en">_NP (VDB|VDI|VHB|VHI|
                         VBI|VBB|VVI|VVB)</pattern>
  <message>Use a singular verb for singular
    nouns.</message>
  <error_rate warnings="248" sentences="75900" />
  <example type="correct">A baseball team <em>is</em>
    established.</example>
  <example type="incorrect">A baseball team <em>are</em>
    established.</example>
</rule>
```

The relatively high value in the `error_rate`'s `warnings` attribute is caused by some correct sentences that are marked as incorrect. For example, the rule also matches questions like *When will the baseball team be established?*

These kind of agreement checks depend on a tag set which is powerful enough to distinguish the different verb forms.

3.8.6 Checks Outside the Rule System

Some error checks cannot be expressed in the rule system, or they cannot be expressed with rules in an efficient way. Currently there are four of these checks implemented outside the rule system:

- The sentence length check warns if a sentence contains more than a given number of words, 30 words being the default limit. This check is outside the rule system since ”more than n words” cannot easily be expressed as a rule. It would require a rule which explicitly matches a sequence of e.g. 30 words of any content. Such a rule would be difficult to configure, as a user surely just wants to enter a number.
- The determiner check warns if *a* is used instead of *an* when the next word's pronunciation starts with a vowel. Vice versa, it warns if *an* is used if the next word does not start with a vowel. The reason that this check is not in the rule system is that it uses a heuristic which checks the next word's first character (see section 2.3.1). Also, the list of exceptions might grow quite large so it is easier to manage if it is in a separate file, not in the XML rule file.
- The word repeat check warns if a word is followed by the same word. This check is outside the rule system because the rule patterns cannot refer to other parts of the pattern.
- The whitespace check warns if there is no space character after these punctuation characters: `.,?!:;`
  It also warns if there is a space character before these characters. There are some exceptions, e.g. for numbers like *5,000*. This check is outside the rule system because rules cannot refer to whitespace.

As these four checks are implemented in Python, they can be extended regardless of any limitations of the rule system. Obviously programming knowledge is required in order to make changes to these checks or in order to add other checks which cannot be expressed by rules. An alternative solution for implementing these checks would have been to extend the power of the rule system. In the end, this would have meant to implement a new programming language which could be used inside rule patterns. This obviously does not make much sense, as Python is already available as a powerful, clean and easy-to-learn programming language.

Python can dynamically execute code created at runtime, so it can also create objects whose name is not known when a Python program is developed. This feature has been used to simplify the addition of new Python rules, i.e. rules outside the rule-system. Any Python file (i.e. any file that ends in `.py`) from the `python_rules` directory is dynamically imported at runtime using the following code:

```
exec("import %s" % filename)
try:
  exec("dynamic_rule = %s.%s()" % (filename, filename))
except AttributeError:
  raise InvalidFilename(filename)
```

Here, the `exec()` statement is used to import code from a given file. Then, the rule object will be created in the `dynamic_rule` variable with another `exec()` call. This will fail if the class in that object has a different name than the file it is saved in. In that case, an exception will be thrown and the script will stop with a helpful error message.

Implementing a new Python rule is very simple: A new Python file needs to be added to the python_-rules directory – none of the existing code needs to be changed. The rule from the new file will automatically be used when the checker is started the next time. The new class needs to implement at least the __init__() and match() methods. __init__() takes no argument, match() takes a list of annotated text, a list of chunks, and a position number. It returns a list of RuleMatch objects, which is empty if no error was found.

The word repeat rule is implemented like this and I will use it as an example (some trivial code is left out to keep the example simple). The WordRepeatRule class is derived from the Rule class. As the Rule class is in the Rules.py file, which is in a different directory, the Rules class has to be imported first. sys.path is extended so that the parent directory is searched for the Rules.py file. The __init__() method is the constructor, it calls the super class' constructor with WORD_-REPEAT as a unique ID for this rule and with a short string that describes the rule:

```
sys.path.append("..")
import Rules

class WordRepeatRule(Rules.Rule):

  def __init__(self):
      Rules.Rule.__init__(self, "WORD_REPEAT", \
        "A word was repeated", 0, None)
      return
```

match() takes a sentence as a list of tagged words, where each list element consists of the original word, a normalized word (usually the same as the original word), and a POS tag. Furthermore a list of chunks is available, but it is not needed for this check. The position_fix paramater will be used to calculate the errors' positions in the text:

```
  def match(self, tagged_words, chunks, position_fix=0):
      matches = []
```

Each word in the list of tagged words is saved to org_word, the next word is saved to org_word_-next. The whitespace between these two words is ignored:

```
      while 1:
          if i >= len(tagged_words)-2:
              break
          org_word = tagged_words[i][0]
          org_word_next = tagged_words[i+2][0]
          text_length = text_length + len(org_word)
```

Whitespace is skipped:

```
          if tagged_words[i][1] == None:
              i = i + 1
              continue
```

Now the two words are compared after being converted to lowercase. If the two words are identical, the list of matches is extended by a new RuleMatch element. It contains this rule's ID, the from/to position which describes where the error occurs and a message that will be displayed to the user:

```
            if org_word.lower() == org_word_next.lower():
                matches.append(Rules.RuleMatch( \
                    self.rule_id, \
                    text_length+position_fix, \
                    text_length+ \
                    len(org_word_next)+position_fix, \
                    "You repeated a word. Maybe you " + \
                    "should remove one of the words?", \
                    org_word))
            i = i + 1
```

Finally, the list of all errors is returned:

```
        return matches
```

3.9 Style Checking

Style checking comprises whitespace checking (see the previous section) and a number of error rules from `rules/words.xml`. The default rules implement these checks (also see section A.2.3):

- A check for short forms like *don't* and *we'll* which suggests the use of the long form like, respectively, *do not* or *we will*.
- A rule which checks if a sentence is started with *Or*, as this might be considered bad style.
- A rule which complains about the often ambiguous use of *billion*, which means *million million* in British English and *thousand million* in American English.

As mentioned before, good style strongly depends on the text type and personal preference. That is why the number of default rules for style checking is rather limited. Professional users will need to write their own rules, adapted to their personal needs.

3.10 Language Independence

Obviously it is desirable to have style and grammar checking which is language independent. But it is equally obvious that portions of the software, like the error rules, have to be language dependant. So here is a list with parts of the software which are language independent and those which are not:

- The stochastic part of the tagger is completely language-independent. Any language for which a tagged corpus exists can be used for training the tagger.
- The tagger's built-in rules, e.g. the guessing of unknown words' tags, which increase the taggers accuracy, only work for English input.
- The file with pre-defined abbreviations which improve sentence boundary detection only contains English abbreviations.
- Only English error rules are part of the checker. However, each rule has a `lang` attribute so that rules for other languages can easily be added without confusion (all current rules use `lang="en"`). The checker will ignore all rules which do not refer to the language of the text which should be checked.

- The user interface (see next section) itself is English, but it can of course check texts in different languages if the backend has been extended accordingly. As with all KDE software, translations for the interface can be added using standard Open Source tools like KBabel, a tool that helps translating the user-visible strings in a program. This is possible even without recompiling the checker.
- Both Python (backend) and Qt (frontend) support Unicode, so it should even be possible to extend the checker to languages whose character set cannot be represented with 8-bit characters (however, this has not been tested).

3.11 Graphical User Interfaces

A style and grammar checker takes text as input and returns a list of errors as output. How exactly the errors are presented to the user depends on the application using the style and grammar checker, thus it makes no sense to bloat the style and grammar checker with its own graphical user interface.

In other words, the style and grammar checker should be usable from a broad range of programs. To facilitate this, the checker's features must be accessible to software which is written in a language other than Python (the language the checker is written in). The checker is a backend program running in the background waiting for input, the application provides the graphical user interface (GUI) and is the frontend.

3.11.1 Communication between Frontend and Backend

The fronted/backend communication must fulfill two requirements:

1. It must be language independent, because the backend is written in Python but most other complex applications are written in languages like Java or C++.
2. It must be non-blocking (asynchronous), i.e. the frontend application must not wait for the backend to return, because the backend needs some time to analyze a sentence and it is not acceptable for the user interface to "freeze" during this time.

Usually matching a sentence against the pre-defined rules is rather fast (see section 4.3.3). But as the number of rules is not limited, and as all rules are independent, there is no upper limit for the time needed to check a text. The rules' independence here means that even if one rule matches, all other rules will still be checked, so that independent errors in a sentence can be found in only one pass. Actually the required check time is O(n*m), with n being the number of rules and m being the text length.

The simplest way to integrate one program into another one – without the need for both programs to be written in the same language – is to just let one program call the other one as an external command. In this case, this is not feasible, as the checker has a rather long start up time. This is caused by the large data files which have to be loaded first and the rules which have to be parsed from XML files. The measured startup time is about 4.1 seconds on an AMD Athlon 900 MHz with 256 MB RAM. This is too much, as the checker should be called for each word while the sentence is being typed. This way it can find errors immediately and not just when the sentence ends.

So the checker should start only once and then wait in the background for new text to be checked. The checker should, in other words, behave like a server. This has been implemented in the `socket_server.py` script, which uses the `TextChecker` class to do the actual work. The server process makes use of network programming techniques by binding itself to a socket on port 50100. Python includes the `socket` class for high-level access to sockets. The relevant parts of the `socket_server.py` script will be explained in the following:

A socket class is created and bound to port 50100 on the local machine (IP address 127.0.0.1). The port number is rather arbitrary[22], but the client must of course use the same port to get access to the server. If the port binding fails because some other program – maybe another instance of `socket_server.py` – is running on the port already, Python will do the error handling automatically and throw an exception. As we cannot really do anything about the problem, we do not catch the exception:

```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
name = "127.0.0.1"
port = 50100
s.bind((name,port))
```

Now we listen to connections from clients which wish to talk to the server. A `TextChecker` object is created which will do all the real work:

```
s.listen(1)
checker = TextChecker.TextChecker(grammar_cfg, ...)
while 1:
  conn, addr = s.accept()
```

For security reasons, we only accept connections from the local machine. The checker code should be audited for potential security issues before this restriction is removed:

```
  if addr[0] != "127.0.0.1":
    conn.close()
    continue
```

The client's data is received, the `checkWords` function (see below) is called with this data and the result is sent back to the client. Then the connection is closed again:

```
  while 1:
    data = conn.recv(limit)
    l.append(data)
    if not data or len(data) < limit:
      break
  data = str.join(”, l)
  check_result = checkWords(checker, data)
  conn.send(check_result)
  conn.close()
```

The `checkWords()` function takes the `TextChecker` object and the client's input. It saves all results in an XML string which is returned at the end of this method:

```
def checkWords(checker, words):
  result = '<result>'
```

[22] On Linux/Unix systems, only the root user can bind to ports below 1024. To avoid conflicts, one should use a port number which is not yet used for new applications. Which port numbers are commonly used can be check at `http://www.iana.org/assignments/port-numbers`.

An `iSpell` object is created which gives access to a local installation of Ispell via Python code. The `check()` method takes a string and returns the number of spelling errors found:

```
ispell = iSpell()
r = ispell.check(words)
if r > 0:
```

For each spelling error, the XML result is extended with an `<error>` element which contains the original word, its position and a comma-separated list of suggested corrections:

```
for mistake in ispell.getMistakes():
  for p in mistake.getPositions():
    result = '%s<error from="%d" to="%d" word="%s"
                 corrections="%s"/>' %  ...
```

Finally, the style and grammar checker is called and its XML result is added to the spelling errors found above. The `<result>` element is closed and the XML string which now lists all spelling, style and grammar errors is returned:

```
(rule_matches, res, tags) = checker.check(words)
result = result + '</result>\n'
return result
```

3.11.2 Integration into KWord

KWord is an Open Source word processor which is part of KOffice, an office suite for KDE[23]. KDE again is an Open Source desktop environment developed in C++. All KDE applications make use of Qt, a very powerful C++ class library. Qt provides all common widgets to write modern graphical user interfaces, and it also comes with many convenience classes which add an easy-to-use object oriented layer on low-level operations.

KWord was chosen as an example of how to integrate the style and grammar checker into an existing word processing application. KWord already supports on-the-fly spell checking (also called online spell checking sometimes), i.e. it marks unknown words with a red zig-zag line during typing. The underline disappears as soon as the word gets corrected. The same feature is desirable for style and grammar checking. KWord's clean design, its manageable size (compared to other Open Source word processors like OpenOffice.org's Writer) and its use of the powerful Qt library make this task feasible.

On-the-fly spell checking in KWord is separated into its own class, `KoBgSpellCheck`. This class makes use of the `KoSpell` class, which then again really checks a paragraph using the Aspell library[24]. Because of the clean separation of classes, only a few files have to be changed to integrate the style and grammar checker. Most changes are done in `koBgSpellCheck.cc` and `koBgSpellCheck.h`, which are located at `lib/kotext/` in the KOffice source tree.

The spell checking already works on paragraphs instead of single words. The Aspell library splits the paragraph into words and looks for errors. However, the fact that complete paragraphs are handed

[23] These are the homepages of these projects: `http://www.koffice.org`, `http://www.kde.org`, `http://www.trolltech.com` (Qt)

[24] Aspell is an enhanced Ispell. It comes not only in form of a standalone program, but also in form of a library which can be used by other programs. Its homepage is `http://aspell.sourceforge.net/`.

over to the spell checker is due to speed concerns, not because Aspell actually uses this context. The spell checking is driven by a self-repeating timer event which starts a new check every second. Only modified paragraphs are checked. Text will be checked no matter if it was typed in via keyboard, pasted from somewhere else or loaded from an external file. As all this already works for simple spell checking, we do not need to take special care of it for the style and grammar checker.

The changes which are necessary to replace the spell checker by a style and grammar checker – which also checks for spelling errors – are limited to rather a few places, of which the most important ones will be explained in the following. Only the new implementation will be described, not the old one which only performed spell checking. The class name `KoBgSpellCheck`, which is now a bit misleading, has been kept to minimize the number of files that need to be touched.

`KoBgSpellCheck::KoBgSpellCheck()`

This is the constructor method of the `KoBgSpellCheck` class. It will create a `QSocket` object. The `QSocket` provided by Qt offers a simple way to access sockets:

```
m_socket = new QSocket();
```

The socket can be connected to so-called slots, which are common C++ methods which are called if something happens at the socket. In this case, the `slotError()` method will be called if there is an error at the socket, and the `slotReadyRead()` method will be called when information has arrived at the socket:

```
connect(m_socket, SIGNAL(error(int)),
  this, SLOT(slotError(int)));
connect(m_socket, SIGNAL(readyRead()),
  this, SLOT(slotReadyRead()));
```

This slot mechanism automatically makes the program asynchronous, as it will not wait for the socket to return information. Instead, the program works as usual and the connected methods are called just when they actually receive data to work with.

`KoBgSpellCheck::spellCheckNextParagraph()`

This method hands the text information of one paragraph to the server. It connects to a socket on the local machine on port 50100. It then sends the text to that socket with a trailing newline added. This method will return immediately, i.e. it does not wait until something is returned by the server. When something is returned, the slots which have been connected in the constructor are called instead. This way KWord is responsive and can be used as usual even if the server takes some seconds to return a result:

```
m_socket->connectToHost("127.0.0.1", 50100);
QTextStream os(m_socket);
os << text << "\n";
```

`KoBgSpellCheck::slotError(int err)`

If the server cannot be connected, this method will be called. If the error number suggests a serious problem, like ”Host not found”, an error message is printed.

`KoBgSpellCheck::slotReadyRead()`

This method is called whenever a new response has arrived at the socket. First, it saves pointers to the current paragraph in KWord. If the paragraph is – for some reason – not available, it returns to avoid crashes caused by null pointers:

```
KoTextObject *fs = m_bgSpell.currentTextObj;
if ( !fs ) return;
KoTextParag *parag = m_bgSpell.currentParag;
if ( !parag ) return;
```

Next, the socket is read line by line and the lines are appended to a `QString` variable, which is the Qt class for strings. It can handle Unicode strings and it provides the expected methods, for example to get the string length or find substrings inside the string:

```
QString s;
while( m_socket->canReadLine() ) {
  QString t = m_socket->readLine();
  s = s + t;
}
```

A regular expression is constructed which will find reported spelling errors:

```
QRegExp rx("<error from=\"([^\"]*)\"
  to=\"([^\"]*)\" word=\"([^\"]*)\"
  corrections=\"([^\"]*)\"/>");
rx.setMinimal(true);
int offset = 0;
```

The received string is matched against the regular expression to find all spelling errors:

```
while( (offset = rx.search(s, offset)) != -1 ) {
  int pos_from = rx.cap(1).toInt();
  int pos_to = rx.cap(2).toInt();
  KoTextStringChar *ch = parag->at(pos_from);
  KoTextFormat format(*ch->format());
  format.setMisspelled(true);
```

The error position is marked as misspelled in the document:

```
  parag->setFormat(pos_from, pos_to-pos_from,
        &format, true, KoTextFormat::Misspelled);
  offset += rx.matchedLength();
}
```

Now the same is repeated for style and grammar errors. As they have a slightly different XML format this requires a different regular expression:

```
QRegExp rx2("<error from=\"([^\"]*)\"
  to=\"([^\"]*)\">(.*)</error>");
rx2.setMinimal(true);
offset = 0;
while( (offset = rx2.search(s, offset)) != -1 ) {
  int pos_from = rx2.cap(1).toInt();
  int pos_to = rx2.cap(2).toInt();
  KoTextStringChar *ch = parag->at(pos_from);
  KoTextFormat format(*ch->format());
  format.setStyleGrammarWarning(true);
  parag->setFormat(pos_from, pos_to-pos_from,
      &format, true, KoTextFormat::Misspelled);
  offset += rx2.matchedLength();
}
```

Finally, KWord is informed that the paragraph will need to be re-rendered so the zig-zag lines which indicate errors will become visible:

```
parag->setChanged(true);
m_bgSpell.needsRepaint = true;
```

The zig-zag underlines which indicate a spelling error are red. Style and grammar errors should be indicated by a different color, so a new text style `StyleGrammarWarning` was added to `kotextformat.h`, additionally to the existing `Misspelled` text format. `DrawParagStringInternal` from `kotextparag.cc` was modified so that spelling errors are still underlined in red, but style and grammar errors are underlined in blue:

```
if( format->isMisspelled() ) {
  painter.setPen( QPen( Qt::red, 1 ) );
} else if( format->isStyleGrammarWarning() ) {
  painter.setPen( QPen( Qt::blue, 1 ) );
}
```

The screenshot in figure 4 shows an erroneous sentence fragment in which the error – *then* instead of *than* – is recognized while the sentence is being typed in. This causes the fragment to be underlined in blue. There is another blue underline under the word *the*, because it is the 26th word in the sentence and the style checker was configured to warn when a sentence becomes longer than 25 words. The red underlines mark potentially misspelled words.

All the source code changes are not directly part of KWord, but of the `kotext` library used by KWord. Because of this, changes which are not binary compatible require a recompilation of KWord, not only of the kotext library. Binary compatibility is defined like follows [Ettrich]:

> A library is binary compatible, if a program linked dynamically to a former version of the library continues running with newer versions of the library without the need to recompile.

[Ettrich] also lists source code modifications which cause a library to become binary incompatible, like adding a member variable to a class. If this is the case with `kotext` and KWord does not get recompiled, undefined behavior will follow: KWord might crash unreproducibly, or variables might contain random values.

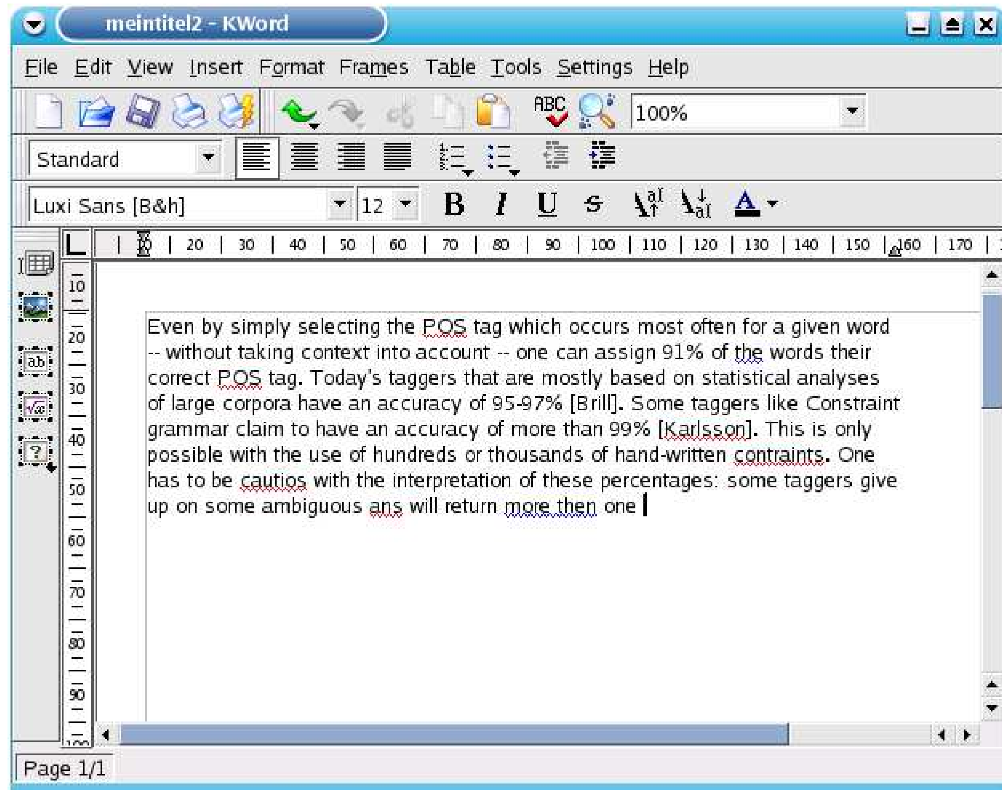


Figure 4: On-the-fly grammar and spell checking in KWord

Obviously, the user interface should not only mark errors, it should also be able to offer corrections. For this, the user needs to select the incorrect sentence and then use the *Language Tool* menu entry from the context menu (the menu that opens up when the right mouse button is clicked). The error dialog displayed then is shown in figure 5. The user can click the linked word in the *Warnings* area to replace the incorrect word with the suggested replacement. The text will then be re-checked immediately. If *OK* is clicked, the corrected text will appear in KWord. The error dialog also contains a *Configure* button which opens the configuration dialog shown in figure 6.

The error dialog and its configuration dialog are implemented as a so-called KDataTool. A KDataTool is a class derived from the `KDataTool` class. It is automatically integrated into KWord's context menu, so that no changes to the KWord code are necessary. Instead, the KDataTool resides in the `tools/language` directory.

`main.cc` implements the `run()` method, which is called by KWord when the according context menu item is selected. The selected text is given to the `run()` method. This method connects to the checker via a socket, just like the on-the-fly checker does. It then displays the dialog with the erroneous text and suggested corrections and waits for user input.

`ConfigDialog.cc` implements the configuration dialog. It reads the checker's XML rule files to know which rules are available and then offers an option to turn each rule on or off. To know where the rule files are, the environment variable `LANGUAGETOOL` must point to the checker's installation directory.

Currently KWord only supports KDataTools which work on plain text. Because of this limitation, any formatting like bold, italics etc will get lost when the user has corrected the text in the error dialog and clicks *OK*. This limitation is supposed to be removed in one of the next KWord releases.

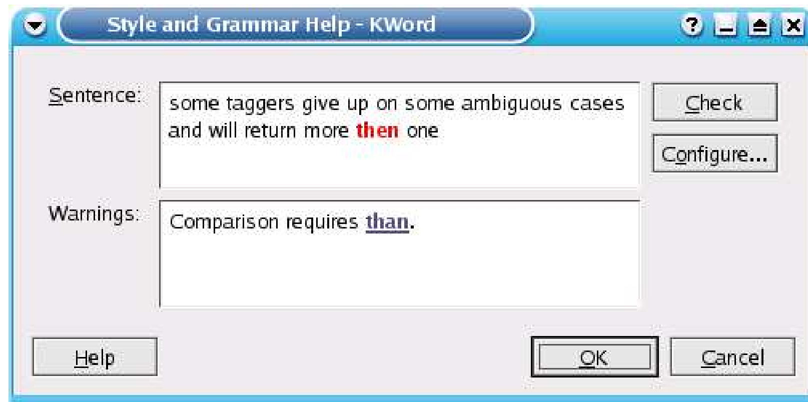


Figure 5: The checker offering a correction in KWord

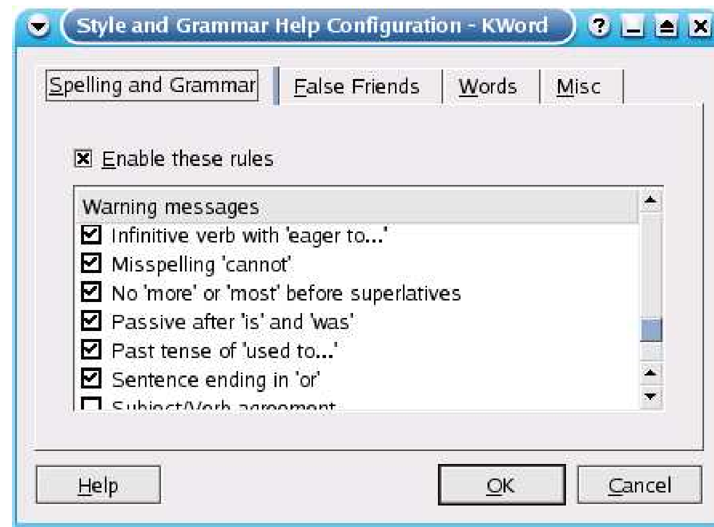


Figure 6: The checker's configuration dialog

Figure 7: The style and grammar checker web interface

3.11.3 Web Frontend

To demonstrate the style and grammar checker online, a web frontend (`TextCheckerCGI.py`) has been developed. This way people can test the software without installing it. The web frontend consists of a text field and some check boxes which, for example, let users select if the result should be displayed with POS tags, as shown in the screenshot in figure 7. Text can be entered into the text field and it will be checked for errors when the *Check Text* button is pressed. On the result page the given text will be displayed again, with erroneous words printed in a bold red font, as shown in figure 8. The error message which explains the error is inserted right after the error, using a blue font. If the option to display POS tags was activated, each word will be followed by its POS tag. The POS tags will be displayed in their short form, e.g. *PNP* (personal pronoun, see section A.4). As the short form is difficult to understand, a click will show a message box with an explanation of this tag. If Javascript is not activated, it is not possible to show a message box so the explanation is shown on a new page instead.

Technically the web frontend is not a client in the way KWord is one. That is because the web frontend script does not contact a server[25], instead it uses the Python classes directly. This is no problem as the web frontend itself is written in Python. So all it needs to do is create a `TextChecker` object and call its `check()` method. The result is then formatted as HTML text, using colors which should make the output easily readable.

The script is a so-called CGI script. CGI scripts are called by a web server, they receive their input via standard input and environment variables and typically print out HTML. Like any CGI script, the checker is started each time the form in the web page is submitted. As mentioned, starting the checker takes some seconds because it needs to load large data files. This could be improved by using a server process which is queried by the checker CGI script.

Another limitation, besides startup speed, is the limited number of options. The false friend and style

[25] Of course the browser contacts the web server, but this is the general principle of the web and in no way specifi c to the style and grammar checker web frontend, so we will not discuss it here.

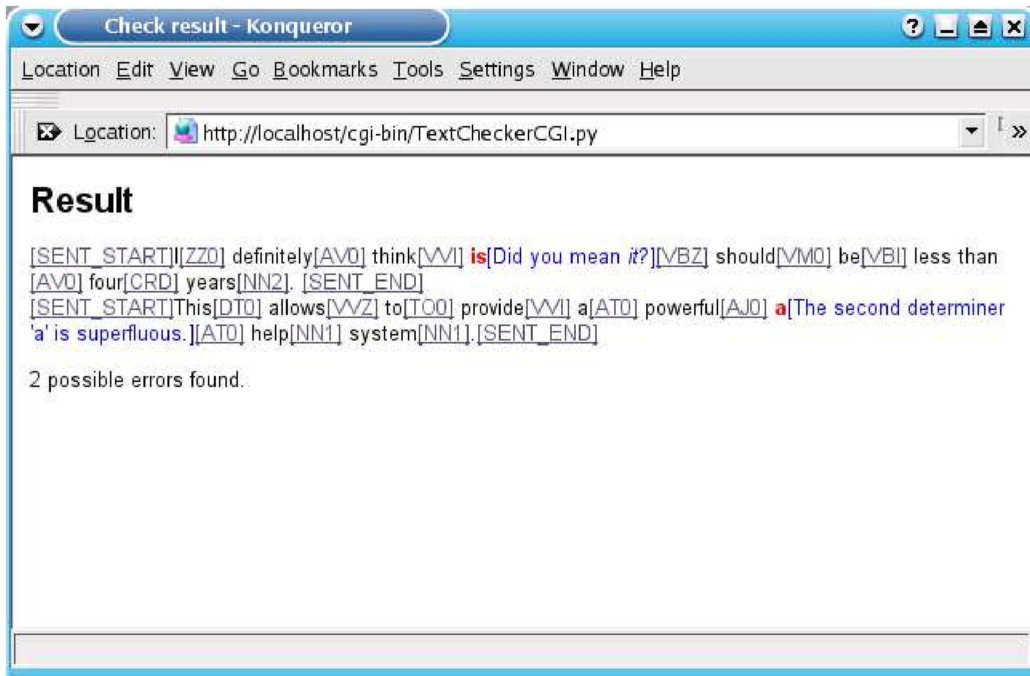


Figure 8: The web interface result page

rules cannot be activated independently. The sentence length limit is set to a constant value of 30 words, but it can be disabled completely. There are no technical reasons for these limitations – as the web interface is mainly intended for demonstration purposes a clean interface is more important than a large number of features.

3.12 Unit Testing

A software which takes text as input and generates other text as output can easily be tested by automatic test cases, also known as unit tests. Python provides the `unittest` module which is a framework for unit tests, similar to the [JUnit] framework from Java, which became popular with Extreme Programming.

Each unit test calls at least one method from the software to be tested and compares the method's result with the expected result. This expected result must of course be defined once by the developer who writes the test case. Typically the methods tested are those which are part of the public API of the class.

For example, a method `pow()` which computes n^i could be tested like this:

```
c = Calculator()
self.assertEqual(c.pow(0, 1), 0)
self.assertEqual(c.pow(1, 0), 1)
self.assertEqual(c.pow(2, 2), 4)
self.assertEqual(c.pow(3, 10), 59049)
```

This test case should be run after any change in the implementation of the `Calculator()` class. If the actual results differ from the expected results, the test will stop with an error message. This way one can optimize or clean up the implementation and still be sure that it works, even without manual testing. Once all the important cases, especially corner cases like 1^0, are covered by a test case, unit

testing is even more powerful than manual testing. In other words, unit testing is a good way to avoid regressions.

As it requires clearly defined input and output in form of common data types (numbers, strings, etc), it cannot easily be used for testing graphical user interfaces. Thus, here it is used to test the backend of the application. The test cases are in separate files, namely in `ChunkerTest.py`, `RulesTest.py`, `SentenceSplitterTest.py`, `TaggerTest.py`, and `TextChecker-Test.py`.

4 Evaluation Results

The style and grammar checker in its current state consists of 54 grammar rules, 81 false friend pairs, 5 style rules and 4 built-in Python rules. All the following information refers to this rule set, with all rules activated, unless stated otherwise.

Usually an evaluation takes place by measuring precision and recall values. Unfortunately, this is not possible for the grammar checker in this case. On the one hand is a corpus that only contains sentences with errors (my error corpus, see section A.1), on the other hand is a corpus that contains very few errors (the BNC, see section 2.7.1). What would be required for meaningful precision/recall values is a corpus of yet unedited text with all errors marked up, but such a thing is not yet publicly available.

4.1 Part-of-Speech Tagger

The BNC itself was automatically tagged with [CLAWS], so it has an error-rate of 1.5% and 3.3% of the words have more than one tag, i.e. they are not unambiguously resolved. This makes the BNC unsuitable for the evaluation of a new tagger. The BNC Sampler instead has been manually checked, and all ambiguous tags have been resolved, one can thus assume an almost 100% correct assignment of tags. The BNC Sampler contains one million words of written English, which should be enough to train and test the tagger. The other million words of spoken English in the BNC Sampler have not been used in this evaluation.

Not only have the errors been fixed in the BNC Sampler, it also uses a more detailed tag set called C7. This tag set contains 140 tags (not counting punctuation tags), much more than the 61 tags of the C5 tag set used for the BNC. More tags mean a more difficult job for a tagger, so the C7 tag set has been mapped to the smaller C5 tag set both when training the tagger and when running it on the test texts. This mapping does not introduce any problems, as C5 is a subset of C7. This way any C7 tag can be mapped to its C5 counterpart unambiguously (see section A.4.2).

The BNC Sampler texts have been divided into a training set (90%), and a test set (10%). After being trained on the training set, the tagger correctly assigns a POS tag to 93.05% of the words in the test set, with no ambiguities left. The test has been repeated three times with different training and test sets, the quoted number is the average accuracy of these three runs. One of the three test runs will now be described more detailed:

- 72,014 (93.58%) of the tokens were assigned their correct POS tag. Each token was assigned exactly one tag, so recall is 0.9358 and precision is 1.
- 38,085 (49.49%) of all tokens had only one tag, so the process of assigning them a tag was trivial for the tagger because context was not relevant.
- 3,635 (4.72%) tokens were not in the tagger's lexicon, thus their tag had to be guessed. The guesser's result was correct in 49% of the cases.
- It took 40 seconds to tag 73,319 tokens, so the tagger works at a rate of about 1,800 tokens/second on the test system.

The overall result of about 93% recall is worse than that of other taggers, as described in section 2.1. The reason might be the broad coverage of the training and testing corpus. Tagging a corpus which only covers one domain – like the popular Wall Street Journal Corpus – is a simpler task, also because the number of unknown words is smaller.

4.2 Sentence Boundary Detection

The sentence boundary detection has been checked with 57,585 sentences from the written-language part of the BNC Sampler. Three files have been excluded from the evaluation because they contain texts for which it is not clear where the sentences start and end[26]. The result is a recall of 96.0% and a precision of 97.0%. In other words, 96% of all sentence boundaries have been found, and of all detected positions that are supposed to be a sentence boundary 97% are indeed sentence boundaries.

This is already close to the best results reported in section 2.3.2. There are some cases concerning colons and semicolons where the markup of the BNC Sampler and the sentence boundary detection disagree about what constitutes a sentence. It should be possible to fix these cases to further improve accuracy.

The results can be reproduced with the `SentenceSplitterEval.py` script. It will read a BNC Sampler file, find all paragraphs and then find all sentences inside those paragraphs. The sentence boundary positions are saved to a list, all markup is removed and the string is then split by the `SentenceSplitter` module. The detected boundary positions are compared to the boundary positions from the BNC Sampler markup so that precision and recall can be calculated.

4.3 Style and Grammar Checker

4.3.1 British National Corpus

The number of errors in proof-read text can be checked for large amounts of text using the `--check` option of `TextChecker.py`, as described in section 3.8.3. This has been done for all grammar rules, and the results have been made part of the rule XML file. For example, the rule with ID `THE_A` contains this element:

```
<error_rate warnings="16" sentences="75900" />
```

This means that the checker claimed 16 errors in 75,900 sentences for that rule. As the BNC's texts have been proof-read, one can assume that many of these 16 errors are false alarms. There are several reasons why a false alarm might occur: the text was incorrectly split into sentences, a word has been assigned an incorrect part-of-speech tag, or the rule simply is not strict enough and triggers too often. The only reason for not checking a larger number of sentences is that the checker is slow on such large amounts of text.

Each rule (or group of rules) contains this `<error_rate>` element. The number of potential false alarms varies from 0 to 250 per 75,900 sentences. This information could be used to offer only those rules to the user that have a low rate of false alarms. Some rules also have a value `all` for the `sentences` attribute. This means that they have been checked on the whole BNC, using the BNC online search, as described in section 2.7.1. This is only possible with simple rules that just list a sequence of words.

4.3.2 Mailing List Errors Corpus

From the list of collected errors (section A.1), the checker correctly identifies 42 errors, not counting any false friend warnings. In comparison, Microsoft Word 2000™ correctly identifies 49 of the errors. Unlike MS-Word, my checker identifies the error as soon as the pattern occurs in the text, i.e. even during typing. MS-Word waits until the sentence is finished and only then analyzes it.

[26] These files contain a collection of prayers (filename `gx0`) which lack punctuation or instructions with many enumerations (`eap`, `g2r`) for which it is not clear whether every item makes up a separate sentence.

Also, MS-Word does not complain about more than one error per sentence, even if the errors are independent.

4.3.3 Performance

The checker's startup time on my system[27] is 4.1 seconds. This has been been checked with the empty file `test.txt` and the following command:

```
time ./TextChecker.py --mothertongue=de \
  --textlang=en test.txt
```

No time has been put into improving this startup time, as a server script (see section 3.11.1) has been developed so that a long startup time is not a problem.

Once the checker is running, the number of sentence checks per second depends on the number of activated rules and the length of the sentences. The following command checks one file from the BNC with all available rules:

```
time ./TextChecker.py -c --mothertongue=de \
  --textlang=en /data/bnc/A/A0/A00
```

With this setting, about 27 sentences per second are checked.

[27] AMD Athlon 900 MHz, 256 MB RAM, Python 2.3c2

5 Conclusion

In this thesis, a style and grammar checker has been developed and has been tested on real-world errors. A rule set has been developed that detects several of these typical errors.

New error rules can easily be added to the checker by adding them to an XML file. For each error, the rule can provide a helpful description of the problem and correct and incorrect example sentences. Even errors that cannot be expressed by the rule system can easily be added without modifying the checker's source code: a Python file that implements the rule just needs to be put into a special folder to be used automatically.

Thanks to a clear separation between backend and frontend, it is easy to develop graphical user interfaces which interact with the backend. A simple web interface with a limited number of options has been written and the checker has been integrated into KWord. A checker server process running in the background is fast enough for interactive use of the checker in KWord, so that errors can be underlined during typing. A configuration dialog allows enabling and disabling each error rule independently.

The backend and the command line tool of the checker are easy to install, they only require Python and no extra modules. The CGI frontend additionally requires a webserver. The KWord integration currently requires installation of KWord from source code so that the necessary patches can be applied.

The rule-based approach does not require a complicated parser and allows for a step-by-step development model which should also make it relatively easy to adapt the checker to other languages. The part-of-speech tagger can be trained on a tagged corpus, and new error rules can be developed as needed.

6 Acknowledgments

I would like to thank my supervisors Prof. Dr.-Ing. Franz Kummert and Dr. Andreas Witt for making it possible for me to work on such an interesting and practically oriented subject. I would also like to thank Hannes Fehr for proofreading this thesis and finding errors which my style and grammar checker does not know yet.

7 Bibliography

References

[AECMA] *AECMA Simplified English*, `http://www.aecma.org/Publications/SEnglish/senglish.htm`, 2003-04-16

[Aston, 1998] Guy Aston, Lou Burnard: *The BNC Handbook: Exploring the British National Corpus with SARA*, Edinburgh Univ. Press, 1998

[Atkinson] Kevin Atkinson: *GNU Aspell*, `http://aspell.sourceforge.net/`, 2003-04-02

[Attwell, 1987] Eric Atwell, Stephen Elliott: *Dealing with ill-formed English text*, in: The computational analysis of English, Longman, 1987

[Becker et al, 1999] Markus Becker, Andrew Bredenkamp, Berthold Crysmann, Judith Klein: *Annotation of Error Types for German News Corpus*, Proceedings of the ATALA workshop on Treebanks, Paris, 1999

[Bernth, 2000] Arendse Bernth: *EasyEnglish: Grammar Checking for Non-Native Speakers*, Proceedings of the Third International Workshop on Controlled Language Applications (CLAW00), Association for Computational Linguistics, April 29-30, Seattle, Washington, pp. 33-42, 2000

[BNC] British National Corpus, `http://www.hcu.ox.ac.uk/BNC/`, 2003-05-03

[BNCweb] *BNCweb: a web-based interface to the British National Corpus*, `http://homepage.mac.com/bncweb/home.html`, 2003-04-13

[Bredenkamp et al, 2000] Andrew Bredenkamp, Berthold Crysmann, Mirela Petrea: *Looking for errors: A declarative formalism for resource-adaptive language checking*, Proceedings of the 2nd International Conference on Language Resources and Evaluation (LREC-2000), May 31 - June 2, pp. 667-673, Athens, Greece, 2000

[Brill, 1992] Eric Brill: *A Simple Rule-Based Part Of Speech Tagger*, Proceedings of ANLP-92, 3rd Conference on Applied Natural Language Processing, Trento, Italy, 1992

[Chunking] *Computational Natural Language Learning (CoNLL-2000): Chunking*, `http://cnts.uia.ac.be/conll2000/chunking/`, 2003-07-29

[CLAWS] *CLAWS part-of-speech tagger for English*, `http://www.comp.lancs.ac.uk/ucrel/claws/`, 2003-07-15

[Crysmann] Berthold Crysmann: *FLAG: Flexible Language and Grammar Checking*, `http://flag.dfki.de/`, 2003-05-01

[Duden Ling. Engine] *Die Duden-Linguistic-Engine*, `http://www.duden.de/sprachtechnologie/linguistic_engine.html`, 2003-07-31

[English G, 1981] *English G Grammatik*, Cornelsen, 1981

[Ettrich] Matthias Ettrich: *Binary Compatibility Issues With C++*, `http://developer.kde.org/documentation/library/kdeqt/kde3arch/devel-binarycompatibility.html`, 2003-05-04

[Google] Google: *The Basics of Google Search*, `http://www.google.com/help/basics.html`, 2003-04-27

[Harabagiu] Sanda Harabagiu: *Topics in Natural Language Processing*, `http://engr.smu.edu/~sanda/ut/Lecture2-nlp.ps.gz`, 2003-05-01

[Holmback et al, 2000] Heather Holmback, Lisbeth Duncan, Philip Harrison: *A Word Sense Checking Application for Simplified English*, Proceedings of the Third International Workshop on Controlled Languages Applications, CLAW2000, Seattle

[Jensen et al, 1993] Karen Jensen, George E. Heidorn, Stpehen D. Richardson (Eds.): *Natural language processing: the PLNLP approach*, Kluwer Academic Publishers, 1993

[JUnit] *JUnit, Testing Resources for Extreme Programming*, `http://www.junit.org`, 2003-08-27

[Karlsson, 1995] Fred Karlsson (ed.): *Constraint grammar: a language independent system for parsing unrestricted text*, Mouton de Gruyter, 1995

[Kiss, 2002] Tibor Kiss: *Anmerkungen zur scheinbaren Konkurrenz von numerischen und symbolischen Verfahren in der Computerlinguistik*, in: Gerd Willée, Bernhard Schröder, Hans-Christian Schmitz (eds.): *Computerlinguistik. Was geht - was kommt? - Computational Linguistics Achievements and Perspectives*, Gardez! Verlag, pp. 163-171, Sankt Augustin, 2002

[Kuenning] Geoff Kuenning: *International Ispell*, `http://fmg-www.cs.ucla.edu/fmg-members/geoff/ispell.html`, 2003-04-02

[Leech] Geoffrey Leech: *A brief users' guide to the grammatical tagging of the British National Corpus*, `http://www.hcu.ox.ac.uk/BNC/what/gramtag.html`, 2003-02-03

[Lindberg and Eineborg, 1999] Nikolaj Lindberg, Martin Eineborg: *Improving part of speech disambiguation rules by adding linguistic knowledge*, Proceedings of the Ninth International Workshop on Inductive Logic Programming, LNAI Series 1634, Springer, 1999

[Mason] Oliver Mason: *Qtag – a portable POS tagger*, http://web.bham.ac.uk/O.Mason/software/tagger/, 2003-05-24

[Naber] Daniel Naber: *Entwicklung einer Software zur Stil- und Grammatikprüfung für englische Texte*, http://www.danielnaber.de/languagetool/, 2002-06-10

[Park et al, 1997] Jong C. Park, Martha Palmer, and Gay Washburn: *An English Grammar Checker as a Writing Aid for Students of English as a Second Language*, Conference on Applied Natural Language Processing (ANLP), Descriptions of System Demonstrations and Videos, Washington, D.C., USA, March, 1997

[Penn Treebank] *The Penn Treebank Project*, http://www.cis.upenn.edu/~treebank/home.html, 2003-04-04

[R.-Bustamente, 1996] Flora Ramírez Bustamante, Fernando Sánchez León: *GramCheck: A Grammar and Style Checker*, Proceedings of the 16th International Conference on Computational Linguistics, Copenhagen, 1996

[R.-Bustamente, 1996-02] Flora Ramírez Bustamante, Fernando Sánchez León: *Is Linguistic Information Enough for Grammar Checking?*, Proceedings of the First International Workshop on Controlled Language Applications, CLAW '96, Katholieke Universiteit Leuven, Leuven, Belgique, pp. 216-228, 1996

[R.-Bustamente et al, 2000] Flora Ramírez Bustamante, Thierry Declerck, Fernando Sánchez León: *Towards a Theory of Textual Errors*, in: Proceedings of the Third International Workshop on Controlled Languages Applications, CLAW2000, Seattle

[Sedlock, 1992] David Sedlock: *An Overview of the Advanced Language Engineering Platform*, http://www.nicklas.franken.de/das/papers/alep_overview_paper/overview/overview.html, 1992-07

[Style/Diction] *Style and Diction*, http://www.gnu.org/software/diction/diction.html, Updated 2002-08-22

[Tufis and Mason, 1998] Dan Tufis, Oliver Mason: *Tagging Romanian Texts: a Case Study for QTAG, a Language Independent Probabilistic Tagger*, Proceedings of the First International Conference on Language Resources & Evaluation (LREC), pp. 589-596, Granada (Spain), 28-30 May 1998

[Walker et al, 2001] Daniel J. Walker, David E. Clements, Maki Darwin, Jan W. Amtrup: *Sentence Boundary Detection: A Comparison of Paradigms for Improving MT Quality*, In Proceedings MT Summit VIII. Santiago de Compostela, Spain, 2001

[Wojcik and Hoard, 1996] Richard H. Wojcik, James E. Hoard: *Controlled Languages in Industry*, in: Ronald A. Cole, Joseph Mariani, Hans Uszkoreit, Annie Zaenen, Victor Zue (Eds.): *Survey of the State of the Art in Human Language Technology*, chapter 7.6, `http://cslu.cse.ogi.edu/HLTsurvey/`, 1996

A Appendix

A.1 List of Collected Errors

This chapter contains the complete error corpus of 224 sentences which was collected mostly on international mailing lists. See section 2.7.2 for more information about how this corpus was compiled.

The categories used to structure the list are rather ad-hoc and not related to the way the checker works, their only use is to make the list more easily readable for human readers. A few sentences appear in more than one category because they contain two different errors: in each appearance of the sentence, errors have been fixed so that exactly one error is left.

All sentences are saved in an XML file. The XML markup specifies the errors by marking their position in the sentence and by specifying a modification which corrects the sentence. This correction consists of removing words, inserting words, or replacing words. Any combination of these three kinds of corrections is also possible. Making the necessary correction explicit makes it easier for human readers to understand the errors, since after looking at many incorrect sentences it can become quite difficult to see *what* the error is.

For example, the erroneous sentence **It reach 1.0 soon.* is marked up like this:

<s>It <insert>will</insert> reach 1.0 soon.</s>

The sentences appear in no specific order, they are enumerated so that it is easier to refer to specific sentences. For better readability the list of errors in the following sections does not make use of XML. Instead, text inside the `<insert>` element is printed in **bold** and text inside the `<remove>` element is ~~crossed out~~. Consequently, the `<replace>` element is represented as crossed out text followed by bold text which replaces the crossed out text.

A.1.1 Document Type Definition

The following Document Type Definition (DTD) is used to mark up the erroneous sentences in `errors.xml`, which is part of the style and grammar checker software:

```
<!ELEMENT errors (category)*>
<!ELEMENT category (s)*>
<!ATTLIST category name CDATA #REQUIRED>
<!ELEMENT s (#PCDATA|remove|insert|replace)*>
<!ATTLIST s type CDATA #IMPLIED>
<!ELEMENT remove (#PCDATA)>
<!ELEMENT insert (#PCDATA)>
<!ELEMENT replace (#PCDATA)>
<!ATTLIST replace by CDATA #REQUIRED>
```

A.1.2 Agreement Errors

1. Carsten Burghardt ~~have~~ **has** actually already implemented the folder selection.
2. Ms. Zaheda resigned SUN, and currently Mr. Eric is the ~~people~~ **person** in charge of this.
3. That project is my degree thesis, was ~~wrote~~ **written** 2 years ago and never touched ever since.
4. Someone else ~~suggest~~ **suggested** to add this to linux/drivers/usb/storage/unusual_devs.h.
5. ...for developers ~~which~~ **who** are interested in issues related to KDE's file...
6. I want you to read it and to give me a price ~~of~~ **for** ~~create~~ **creating** a Perl script that ~~fit~~ **fits** the requirements in that project, the perl script looks like e-commerce but it's very simple.

7. The report must also discuss the problems faced while doing the assignment, how the problems were ~~overcame~~ **overcome**, and the lessons learned.
8. The Web site also ~~contain~~ **contains** fi elds for the user to enter their details: ...
9. Or did that rule ~~changed~~ **change** while I was on vacation?
10. Does each version of MySql ~~has~~ **have** binary AND src version for installation?
11. And what about the ones who ~~doesn't~~ **don't** have wireless?
12. This is a set of changes that is well ~~test~~ **tested** by myself and others.
13. Anything about UPB ~~was~~ **has** already ~~wrote~~ **been written**.
14. JFYI, KMail now ~~support~~ **supports** S/MIME through a plugin.
15. ~~Follow~~ **Following** Eike's advice, now I am ~~join~~ **joining** the maillist.
16. If you mirror kde-www on kdemirror.de, ~~you has only to~~ **you only have to** change your local confi g.php
17. ...which is right in many ~~way~~ **ways**,...
18. I now have ~~understand~~ **understood** that it is the IMAP-server I'm using.
19. I might ~~forgot~~ **forget** to send it then.
20. Where can I ~~found~~ **fi nd** a fi x?
21. Re: Can't ~~Preventing~~ **prevent** exploitation with rebasing
22. ~~Does~~ **Did** the problem ~~has appeared~~ **appear** after an update of KDE or not?
23. You will ~~lost~~ **lose** all your changes!
24. I have not ~~say~~ **said** that is a MySpell problem I would like only to signal it ...
25. For maximum ability to nail spambots, consider using a cgi directory that is not ~~names~~ **named** "cgi-bin".
26. Doing so we have ~~create~~ **created** a my.cnf in mysql datadir containing:
27. Click on a folder and then move the mouse may ~~selects~~ **select** wrong folder
28. Display a column in the list of folders which ~~show~~ **shows** the number of unread messages per folder.
29. When using javascript menus, the pop-up menus are ~~showed~~ **shown** at the back...
30. The patch has only ~~be~~ **been** applied to KOffi ce.
31. If it ~~doesn't be~~ **isn't** stable/mature enough before 3.1 fi nal, I will....
32. ...but Dirk ~~tell~~ **told** me to go ahead ;)
33. I fi xed buglist.cgi and the show_activity.cgi and ~~send~~ **sent** in the patch.
34. Well, as long as there is no link to both of these, ~~I~~ **it** doesn't affect the problem we wanted to address.
35. ...since it was noticed that Sebastian also had ~~make~~ **made** a suggestion about it.
36. Has anyone ~~considering~~ **considered** writing interface test suites...
37. I ~~produces~~ **produce** all the arguments...
38. We need to ~~found~~ **fi nd** a common layer...
39. ~~I'm~~ **I** do not run a mix of OS...
40. I do not run a mix of OS and it ~~are~~ **is** a long time ago...
41. Why ~~does~~ **do** I still install a not used eventFilter ;)
42. That ~~doesn't was~~ **wasn't** my intention.
43. Have no probs with your stuff, but why ~~does~~ **do** we need an undock button?
44. I am sure others ~~has~~ **have** a list as long as mine.
45. ...but I hope in a few ~~year~~ **years** it will...
46. ...which does not ~~has~~ **have** 50% threshold.
47. It is updated every few ~~day~~ **days**.
48. What would ~~happend~~ **happen**, if someone writes...
49. Why ~~does~~ **is** kmail HEAD afraid of using kdelibs HEAD?
50. The emails of this form should ~~goes~~ **go** to this email address: ...
51. We would be glad if this little helper would be able to help other ~~developer~~ **developers**.
52. You can ~~found~~ **fi nd** it on my web page.
53. Did you maybe ~~missed~~ **miss** the kconf_update run that updates the signature confi g?
54. No, the indexes should be ~~build~~ **built** automatically.
55. I ~~improves~~ **improved** it a bit...
56. 3.0.4 is mostly ~~know~~ **known** for miscompiling aRts, and the fi x was too large to merge.
57. IIRC you told me a long time ago GnuPG 1.0.7 was to be ~~release~~ **released** the "next weekend".

58. I had a look into this and the dialog didn't ~~got~~ **get** deleted.
59. Does somebody ~~knows~~ **know** more?
60. I solved this by ~~move~~ **moving** all folders to IMAP, which was a task for months.
61. I solved this by moving all folders to IMAP, which was a task for ~~month~~ **months**.
62. I always get these ~~error~~ **errors**.
63. It ~~do~~ **does** ~~has~~ **have** some bugs though.
64. In chap. 7, it ~~discussing~~ **discusses** using signals and slots to create a little xml parser, which then displays a tree list view.

A.1.3 Missing Words

1. It **will** reach 1.0 soon.
2. I **am** just surprised though that Opera developers are of such high opinion about their Opera browser.
3. The result of the work on NPTL, the new thread implementation, can **be** checked out.
4. The language patch has **been** added to http://www.danielnaber.de/perlfectsearch/.
5. I'll defi nitely step by at the KDE booth, maybe we can **make** arrangements there.
6. I think~~,~~ it's ready **to** replace cvs2pack.
7. So please if the admin did not add **an** image, let the script ~~to~~ take this image automatically.
8. If they are already up to date, then I **thought** it is enough to replace the date and the version to acknowledge that they have been verifi ed.
9. That dump can then **be** used to fi gure out just what went wrong.
10. That **is** why we will add a function to the token defi nitions.
11. Wouldn't you **go** mad if I fork your program, modifi y it and come up to all others with it to trash your original program out of kde without one single personal email?
12. ...we should be able to let this structure live on for **a** year.
13. I don't **think** KDE eV yet owns any domain names.
14. So, could we have at least the option of threading just by subject and ignoring **???** completely.
15. These user unfriendly message boxes have **been** the reason to leave windows95 seven years ago ;-)
16. Being annoyed by today's debian-security OFF TOPIC ranting, I noticed my wish for **this** feature, ...
17. For local fi les with a known **???**, simply create a KFileItem and pass it to the other constructor.
18. ...test if more **or** less everything works...
19. In KDE 2.2.0 and KDE 2.2.1 we **had** a bug that caused KMail...
20. Of **course** this isn't a general solution.
21. ...but I hope in **a** few ~~year~~ **years** it will...
22. I volunteer to create **a** testcase then.
23. Kyoto was **the** oldest capital city in Japan.

A.1.4 Extra Words

1. I discussed the contest with Lukas Tinkl so here is the very latest revision of the ~~much~~ famous (!) Koffi ce Icon Contest.
2. Someone ~~suggested~~ said that it worked for him after he updated to Kernel 2.4.20.
3. Selecting ~~a~~ folders to be synchronized, ...
4. But, probably, it will be ~~more~~ better to set the default language of Automatic Detection by...
5. As I said, we only have ~~a~~ very few weeks left and have to march towards that date very effi ciently.
6. Actually, the more I think about it, I like Daniel's proposal ~~the~~ best.
7. I must ~~to~~ try it.
8. I'm hoping that bugzilla.kde.org ~~that it~~ will be found to be flexible enough...
9. There is ~~a small~~ **some** research done on what effects has the computer on education.
10. We can always slip it by ~~for~~ 2-3 weeks by adding another release candidate
11. Kyoto was the ~~most~~ oldest capital city in Japan.
12. You can fi nd it among ~~with~~ my patches on my web page.

13. I ~~will~~ wrote to an e-mail you told me.
14. This will be ~~a~~ much work.
15. If you ~~are~~ use SMTP to send your messages, ...
16. You must ~~to~~ fill the Document Info.
17. Can somebody remember how the fix ~~have~~ worked?

A.1.5 Wrong Words

1. Sorry, but the backtrace is ~~complete~~ **completely** useless.
2. With the ~~currently~~ **current** system load I don't think that will be successfull.
3. In which case is Ruby ~~most~~ **more** powerful than Python?
4. The person that claims for the authory of the TWO "PhpNuke SQL Injection" has discovered them at the same time ~~than~~ **as** me and exploits them _exactly_ the same way.
5. ...well it's ~~easy~~ **easier** now than it was before...
6. The patch has only been applied ~~for~~ **to** KOffice.
7. Sorry, I have to correct ~~me~~ **myself**.
8. Only during the ~~week end~~ **weekend** I ...
9. ...since it was noticed that Sebastian also had made ~~an~~ **a** suggestion about it.
10. I guess this isn't easily ~~possibly~~ **possible**.
11. Surely not before KDE 3.1 (think of translations etc.), ~~or~~ **right**?!?
12. I already did that, but it does not work, ~~too~~ **either**.
13. I changed one behaviour with my original commit, which is not consistent with EXCEL ~~too~~ **either**.
14. This doesn't belong here, ~~too~~ **either**.
15. KDE League ~~has not~~ **does not have** enough man power ~~too~~ **either** and cannot do as much as could be possible.
16. Are you planning to add non-English documentation ~~either~~ **too**?
17. I'll think a bit more ~~on~~ **about** this and ...

A.1.6 Confusion of Similar Words

1. Thanks for ~~you~~ **your** interest in our service.
2. It's a bit more ~~then~~ **than** 15 min. for me.
3. ~~Than~~ **Then** my old email is nonsense.
4. I volunteer to create a testcase ~~than~~ **then**.
5. It is also harder to maintain stability in the code base with small changes that just fix the bugs ~~then~~ **than** to reorganise often.
6. It's less controversial ~~then~~ **than** one would think :)
7. We show that our system achieves state of the art performance in text chunking with less computational cost ~~then~~ **than** previous systems.
8. Starting over just so we can integrate GNOME and KDE apps fully is a waste of time and more work ~~then~~ **than** just perfecting the apps we have.
9. If you click on the same column more ~~then~~ **than** once, ...
10. I am not wiser ~~that~~ **than** you but I try to listen to others when it becomes...
11. I'd rather keep all the syntax highlighters in one file until it gets ~~to~~ **too** big.
12. Ok, I wrote this to you, that you don't have to do ~~to~~ **too** much research for your KDE-CVS digest...
13. This is not ~~to~~ **too** close at the Linuxtag, so that there are not two events which have to be organized at the same time and it doesn't overlap ~~to~~ **too** much with school holidays.
14. This will perform a commit whenever the cache gets ~~to~~ **too** full.
15. ...but now it's getting ~~to~~ **too** complicated and time-consuming for me.
16. Ideally we should have one database, not ~~too~~ **two**.
17. Someone has ~~too~~ **to** keep it up to date.
18. Indexing the whole directory would take ~~to~~ **too** long.

19. The identity you are using ~~the~~ **to** write the current message needs to...
20. ...but also easily uses all available fi le handlers (on Linux systems) if you have ~~to~~ **too** many folders.
21. If the application + FS interaction leads to contention, then you have ~~to~~ **two** choices to optimize:...
22. Note: the links in this display ~~to~~ **do** not work yet.
23. Unhide can be triggered by moving ~~to~~ **the** mouse to a corner.
24. I think ~~your~~ **you're** confused.
25. I want to see ~~you~~ **your** printer.
26. Are you in a position where ~~you~~ **your** company will let you contribute patches...
27. Thanks again for your help and good luck for ~~you~~ **your** KDE Edu submission.
28. The biggest user benefi t in this is that ~~you~~ **your** local disk usage is almost zero.
29. Maybe Ingo should just have reverted ~~you~~ **your** damned dump.
30. Where's ~~you~~ **your** point?
31. You should ask ~~you~~ **your** admin.
32. Thanks again ~~fore~~ **for** your help and good luck for your KDE Edu submission.
33. I work for a publishing house and ~~were~~ **we** are in the process of conceptualizing various books on Linux.
34. ~~Of~~ **If** you disable this option, ...
35. The next test will tell us ~~it~~ **if** the usb <-> scsi emulation is working correctly, run this on the console.
36. ~~Is~~ **If** our licenses are compatible, would you care to take over integrating a new thesaurus into OOo as well?
37. You may CC me if you ~~thing~~ **think** you must...
38. So I ~~thing~~ **think** we will...
39. Aethera ~~use~~ **used** to use KParts.
40. Please don't ~~us~~ **use** dispose!
41. But I think ~~it~~ **it's** serious:...
42. If the cafeteria is open (which ~~is~~ **it** wasn't the last day at GUADEC 3).
43. The Kopete team has decided ~~its~~ **it's** time we should move to kdenetwork if we want to be included in KDE 3.2.
44. It is more correct to say that the AEGYPTEN branch has landed *on* ~~it's~~ **its** HEAD ;-)
45. ~~Off~~ **Of** course not.
46. KHTML is just the only part of KDE I know ~~off~~ **of** that uses automasking.
47. ... - though there is not much you can ~~by~~ **buy** for $0.02 let alone 0.02 euro ;-)
48. $1k would not be enough to ~~by~~ **buy** the beer for the people who develop The GIMP.
49. You can discuss ~~then~~ **the** warnings and determine if any of them are valid.
50. We can make sure it works and ~~the~~ **then** just have to change the link.
51. Thanks for the ~~responds~~ **response**.
52. Or ~~way~~ **was** that a very recent change in Qt?
53. ...~~there for~~ **therefore** we should try to fi x...
54. Or we just ~~through~~ **throw** away all bugs and start from scratch :-)
55. I have ~~no~~ **now** introduced a yPosition() function...
56. We think it helps fi nding bottlenecks and so ~~one~~ **on**.
57. I'm ~~nut~~ **not** sure... ;-)
58. Please ~~not~~ **note** that saying "Open Source" does not mean very much.
59. ~~At~~ **Add** to that retarded corporate intranets with Windows DNS and mail servers.
60. Ah, I ~~new~~ **knew** you would say that.
61. But be warned that caching and downloading ~~a~~ **are** two completely different things.
62. I can be pretty ~~dump~~ **dumb** when it comes to such things...
63. Your best ~~change~~ **chance** to meet some of them...
64. ~~How~~ **Who** else has to be involved in this process?
65. But even if it's looking fi ne, ~~the~~ **there** is the problem that for lots of calls you...
66. I ~~though~~ **thought**...
67. I ~~a~~ **am** not wiser than you but I try to listen to others when it becomes...
68. Jeremy Hylton noticed that in ceval that ~~their~~ **there** is a test of whether...

69. Unfortunately ~~their~~ **there** are some problems with your patch:
70. There are two things I know about branches: ~~the~~ **they** always introduce bugs and inconsistencies.
71. I guess simply turning this ~~of~~ **off** isn't easily possible.
72. I have been out with the ~~flew~~ **flu** for the past 3 days.
73. You could put the antenna of your radio to this applet and ~~here~~ **hear** the sound...
74. The initial data needs to be under ~~and~~ **an** Open Source lincense.
75. For example, it is highly unlikely ~~than~~ **that** any system will learn human dialogue performance from dialogue transcripts...
76. No surprise, since the trash folder is the only ~~on~~ **one** for me that automatically expires.
77. Searching is really simple: Type in one or more words into ~~to~~ **the** search field and click "Search" (or press Return).
78. Searching is really simple: Type in one ~~ore~~ **or** more words...
79. The initial OOo thesaurus implementation was a real hack job basically thrown together by me to fill a ~~whole~~ **hole** and get myself up to speed with how lingucomponents interact with OOo and what it was like to write cross-platform code using the SAL abstraction layer.
80. Now, this is ~~were~~ **where** my ignorance sets in.
81. It is nice ~~the~~ **to** read this.
82. You see everything at ~~one~~ **once** without clicking in the list.
83. I believe you ~~half~~ **have** to turn on ExecCGI for the public_html dir.
84. As you probably know, my native language ~~in~~ **is** Russian, plus I talk some other Cyrillic-based languages.
85. You can get my photo ~~hear~~ **here**: ...
86. If the last word ~~in~~ **is** incorrect, it gets underlined.
87. We really ~~where~~ **were** popular...
88. I think in ~~cause~~ **case** coolo didn't do it already we can include...
89. Yes it is, to a certain ~~extend~~ **extent**.
90. Going in Settings/SpellChecker and change the language ~~their~~ **there** is not so cool ;)
91. Of ~~cause~~ **course** there is much more to see in the respective regions.
92. This point is ~~mood~~ **moot** as there is no active/passive membership status.
93. ~~The~~ **Then** it won't compile...
94. If you don't ~~active~~ **activate** the OpenPGP plugin then KMail will use the built-in inline OpenPGP by default.

A.1.7 Wrong Word Order

1. Why should Sebastian waste his time for changing that "details" on his design if we ~~not even can~~ **cannot even** agree that the left side menu will make it for the content pages?
2. ...and btw., ~~think you~~ **do you think** the www/kde-ev should move to...
3. I can give you ~~more a detailed~~ **a more detailed** description of what's necessary.
4. ~~Does someone can~~ **Can someone** reproduce what I described before?

A.1.8 Comma Errors

1. Please select a template<remove>,</remove> that matches the csv file.
2. I don't think<remove>,</remove> this is the job of the KDE League reps.
3. Thanks to everyones reponses on this<insert>,</insert> really useful to know.

A.1.9 Whitespace Errors

1. In which case is Ruby more powerful than Python<remove> </remove>?

A.2 Error Rules

A.2.1 Document Type Definition

This DTD is used for the error rule description by the following files in the `rules` directory: `grammar.xml`, `false_friends.xml`, `words.xml`.

```
<!ENTITY % Languages "(en|de|fr)">
<!ELEMENT rules (rule|rulegroup)*>
<!ELEMENT rulegroup (message?,error_rate?,rule+)>
<!ATTLIST rulegroup id ID #REQUIRED>
<!ATTLIST rulegroup name CDATA #IMPLIED>
<!ELEMENT rule ( pattern,
   ((message?,error_rate?,example,example+) |
   (translation+,message?,example*)) )>
<!ATTLIST rule id ID #IMPLIED>  <!-- well,
   required but not if there's a group -->
<!ATTLIST rule level (0|1|2|3|4|5|6|7|8|9|10) #IMPLIED>
<!ATTLIST rule name CDATA #IMPLIED>
<!ELEMENT pattern (#PCDATA)>
<!ATTLIST pattern lang %Languages; #REQUIRED>
<!ATTLIST pattern case_sensitive (yes|no) #IMPLIED>
<!ATTLIST pattern mark_from CDATA #IMPLIED>
<!ATTLIST pattern mark_to CDATA #IMPLIED>
<!ELEMENT translation (#PCDATA)>
<!ATTLIST translation lang %Languages; #REQUIRED>
<!-- message shown to the user if a rule matches: -->
<!ELEMENT message (#PCDATA|em)*>
<!-- percentage of sentences tagged as wrong in BNC: -->
<!ELEMENT error_rate EMPTY>
<!-- number of (real or unreal) warnings when testing
   BNC sentences: -->
<!ATTLIST error_rate warnings CDATA #IMPLIED>
<!-- number of BNC sentences used to get 'warnings': -->
<!ATTLIST error_rate sentences CDATA #IMPLIED>
<!ELEMENT example (#PCDATA|em)*>
<!ATTLIST example type (correct|incorrect|
   triggers_error) #REQUIRED>
<!-- used to explain 'triggers_error': -->
<!ATTLIST example reason CDATA #IMPLIED>
<!-- emphasis of the most significant part: -->
<!ELEMENT em (#PCDATA)>
```

A.2.2 Grammar Error Rules

The error rules listed here can be found in `rules/grammar.xml`:

1. **Pattern:** "(a|an|one)" _NPP
   Message: For plural nouns, use the plural article *the* .
   Example: *A* baseball teams were successful.

2. **Pattern:** _NP (VDB|VDI|VHB|VHI|VBI|VBB|VVI|VVB)
   Message: Use a singular verb for singular nouns.
   Example: A baseball team *are* established.

3. **Pattern:** "(does|did)" ".*" "can"
   Message: Did you mean *can \2* ?
   Example: *Does someone can* reproduce that?

4. **Pattern:** "more" "(a|an)" AJ0
   Message: Did you mean *a more \3* ?
   Example: I can give you *more a detailed* description of what's necessary.

5. **Pattern:** "think" "you" "(a|an)"
   Message: Did you mean *are* ?
   Example: Some would think you *a* fortunate man.

6. **Pattern:** "(some|certain)" "extend"
   Message: Did you mean *extent* ("extent" is a noun, "extend" is a verb)?
   Example: Yes it is, to a certain *extend* .

7. **Pattern:** "is" "were"
   Message: Did you mean *where* ?
   Example: Now, this is *were* my ignorance sets in.

8. **Pattern:** CRD "ore"
   Message: Did you mean *or* ?
   Example: Type in one *ore* more words.

9. **Pattern:** "the" "only" "on"
   Message: Did you mean *the only one* ?
   Example: The trash folder is the only *on* for me that works.

10. **Pattern:** "their" "(is|are)"
    Message: Did you mean *there* ?
    Example: He noticed that *their* is a new test.

11. **Pattern:** SENT_START "I" "(a|an)"
    Message: Did you mean *am* (first person of "to be")?
    Example: I *a* not sure.

12. **Pattern:** "I" "new"
    Message: Did you mean *knew* (past of "to know")?
    Example: Ah, I *new* you would say that.

13. **Pattern:** "please" "not" "that"
    Message: Did you mean *note* ?
    Example: Please *not* that saying "Open Source" does not mean very much.

14. **Pattern:** "nut" (AV0|AJ0)
    Message: Did you mean *not* ?
    Example: I'm *nut* sure.

15. **Pattern:** "and" "so" "one"
    Message: Did you mean *and so on* (="etc.")?
    Example: We think it helps finding bottlenecks *and so one* .

16. **Pattern:** "through" "away"
    Message: Did you mean *throw* ?
    Example: Or we just *through* away all bugs.

17. **Pattern:** "or" "way" "(it|that|this)"
    Message: Did you mean *was* ?
    Example: Or *way* that a very recent change in Qt?

18. **Pattern:** AT0 "responds"
    Message: Did you mean *response* ?
    Example: Thanks for the *responds* .

19. **Pattern:** "(think|know)" "off"
    Message: Did you mean *of* ?
    Example: It's the only part of KDE I know *off* that uses automasking.

20. **Pattern:** "do" XX0 "us"
    Message: Did you mean *use* ?
    Example: Please don't *us* dispose.

21. **Pattern:** VVB "to" "use"
    Message: Use the past tense to build a sentence with "to use".
    Example: Aethera *use* to use this.

22. **Pattern:** "(I|you|he|she|they|we)" "things?"
    Message: Did you mean *think* or *thinks* ?
    Example: I *thing* that's a good idea.

23. **Pattern:** "were" VBB
    Message: Did you mean *where* or *we* ?
    Example: *Were* are in the process of implementing this.

24. **Pattern:** VBZ VBD
    Message: Did you mean *it* ?
    Example: ...which *is* wasn't the last day.

25. **Pattern:** "fore" DPS
    Message: Did you mean *for* ?
    Example: Thanks again *fore* your help.

26. **Pattern:** "not" "too" (SENT_END|NN1|NN0|NN2)
    Message: Did you mean *two* ?
    Example: Ideally we should have one database, not *too* .

27. **Pattern:** "your" (VVD)
    Message: Did you mean *you're* or *you are* ?
    Example: I think *your* confused.

28. **Pattern:** AJC "that"
    Message: Did you mean *than* ?
    Example: I am not wiser *that* you.

29. **Pattern:** "(less|more)" (AJ0|NN1|NN0) "then"
    Message: Did you mean *than* ?
    Example: It's less controversial *then* one would think.

30. **Pattern:** AT0 AT0
    Message: *Remove* the second determiner.
    Example: *The the* thing is this.

31. **Pattern:** "than" SENT_END
    Message: Did you mean *then* ?
    Example: I volunteer to create a testcase *than* .

32. **Pattern:** "of" "cause" ^"(and|to)"
    Message: Did you mean "of *course* " (=naturally)?
    Example: Of *cause* all these things are relative.

33. **Pattern:** "eager" "to" (VBB|VBD|VBG|VBN|VBZ|VDB|VDD|VDG|VDI|VDN|VDZ|VHB|VHD|VHG|VHI| VHN|VHZ| VM0|VVB|VVD|VVG|VVN|VVZ)
    Message: Use the base form of the verb with *eager to* .
    Example: I'd be eager to *trying* out your patch.

34. **Pattern:** "or" SENT_END
    Message: Don't end a sentence with *or* . If you want to ask for confirmation, use *isn't it* , *don't you* etc.
    Example: I think we should meet soon, *or* ?

35. **Pattern:** "a" "much" NN1
    Message: You maybe should omit the article *a* .
    Example: This will be *a much* work.

36. **Pattern:** "(more|most)" AJS
    Message: *Remove* 'most' or 'more' when you use the superlative.
    Example: Kyoto is the *most oldest* city.

37. **Pattern:** "is" "(should|could)"
    Message: Did you mean *it* ?
    Example: I definitely think *is* should be less than four years.

38. **Pattern:** "the" AJ0 AT0
    Message: The determiner 'a' might be superfluous.
    Example: This allows to provide the powerful *a help* system.

39. **Pattern:** "a" AJ0 "(a|the)"
    Message: The determiner 'a' might be superfluous.
    Example: This allows to provide a powerful *a help* system.

40. **Pattern:** ^"(has|will|must|could|can|should|would|does|did)" "he" (VVI|VVB)
    Message: Third person singular noun requires verb with '-s'.
    Example: Then he *look* at the building.

41. **Pattern:** "did" (VVD|VVG|VVN|VVZ)
    Message: 'Did' requires base form of verb.
    Example: Peter did *went* to the cinema.

42. **Pattern:** ^(DT0) VM0 (VVD|VVG|VVN|VVZ)
    Message: Modal verbs like 'will'/'might'/... require base form of verb.
    Example: John will *goes* home.

43. **Pattern:** NP0 VM0 NN2
Message: Modal verbs like ’will’/’might’/... require base form of verb.
Example: John will *walks* home.

44. **Pattern:** "is" (VVI|VVZ)
Message: Passive requires verb with ’-ed’.
Example: He is *go* home.

45. **Pattern:** "was" (VVI|VVZ)
Message: Passive requires verb with ’-ed’.
Example: The ball was *catch* by John.

46. **Pattern:** "as" (AJ0|AV0) "(like|than|then)"
Message: Comparison is written "as (adjective) *as* ".
Example: This house is as big *like* mine.

47. **Pattern:** AT0 "from"
Message: Did you mean *form* ?
Example: Type in the *from* on the web page.

48. **Pattern:** "can" "not"
Message: Can + not is written as *cannot* .
Example: You *can not* use the keyboard to select an item.

49. **Pattern:** "(more|less)" "(then|as)"
Message: Comparison requires *than* .
Example: The tagger will return more *then* one tag.

50. **Pattern:** (AJC) "(then|as)"
Message: Comparison requires *than* .
Example: This house is bigger *then* mine.

51. **Pattern:** "its" (CJC|AT0|DT0|EX0|PRF|PRP|VM0|POS|PNP|AV0|AVP|TO0|DTQ|PNQ|DTQ|AVQ)
Message: Did you maybe mix up *it’s* (=*it is*) and *its* (=possessive pronoun)?
Example: *Its* where people walk around.

52. **Pattern:** "its" (VVI|VVZ|VVG)
Message: Did you maybe mix up *it’s* (=*it is*) and *its* (=possessive pronoun)?
Example: *Its* going to be difficult.

53. **Pattern:** "its" "(a|the)"
Message: Did you maybe mix up *it’s* (=*it is*) and *its* (=possessive pronoun)?
Example: *Its* a good opportunity.

54. **Pattern:** "no" (VM0|VVI|VVD|VVN)
Message: Did you maybe mix up *no* (negation) and *now* (=at this moment)?
Example: There *no* will be a park.

55. **Pattern:** "no" (CJC|DT0|AT0|EX0|PRF|PRP|VM0|NP0|AVP|DTQ|PNQ|AVQ)
Message: Did you maybe mix up *no* (negation) and *now* (=at this moment)?
Example: There’s *no* a new building.

56. **Pattern:** "no" "(were|was|been|be|is)"
Message: Did you maybe mix up *no* (negation) and *now* (=at this moment)?
Example: *No* is the right time.

A.2.3 Style/Word Rules

The style rules listed here can be found in `rules/words.xml`:

1. **Pattern:** ".*" "n't"
   Message: Suggestion: use a long form like *do not* instead of *don't* for formal texts.
   Example: *Don't* open the box unless you've read the instructions.

2. **Pattern:** ".*" "'ll"
   Message: Suggestion: use a long form like *\1 will* instead for formal texts.
   Example: *He'll* open the box.

3. **Pattern:** ".*" "'ve"
   Message: Suggestion: use a long form like *\1 have* instead for formal texts.
   Example: Questions *we've* asked some people.

4. **Pattern:** ".*" "'re"
   Message: Suggestion: use a long form like *\1 are* instead for formal texts.
   Example: What *We're* Reading

5. **Pattern:** SENT_START "or"
   Message: Suggestion: don't start a sentence with *or* .
   Example: *Or* one could think about other approaches.

6. **Pattern:** "billion"
   Message: Suggestion: use *thousand million* instead of billion to avoid ambiguities.
   Example: Feds "Misplaced" $17.3 *Billion* Last Year

A.2.4 English/German False Friends

The rules listed here can be found in `rules/false_friends.xml`:

also/also, actual|actually/aktuell, advocate/Advokat, argument/Argument, art/Art, bald/bald, become/bekommen, billion/Billion, blame/blamieren, box/Box, brave/brav, chef/Chef, concur/konkurrieren, consequent|consequently/konsequent, curious/kurios, decent/dezent, dome/Dom, engagement/Engagement, etiquette/Etikett, eventually/eventuell, evergreen/Evergreen, exceptional|exceptionally/ausnahmsweise, familiar/familiär, fast/fast, formula/Formular, fantasy/Fantasie, fraction/Fraktion, gasoline/Gas, garage/Garage, gift/Gift, gross/groß, gym/Gymnasium, handy/Handy, high school/Hochschule, confession/Konfession, labor/Labor, last night/letzte Nacht, list/List, None/live, local/Lokal, map/Mappe, marmalade/Marmelade, meaning/Meinung, menu/Menü, murder/Mörder, mist/Mist, natural|naturally/natürlich, notice/Notiz, objective/Objektiv, ordinary/ordinär, overhear/überhören, overwind/überwinden, paragraph/Paragraph, photograph/Fotograf, pension/None, preservative/Präservativ, prospect/Prospekt, prove/prüfen, public/Publikum, realize/realisieren, rentable/rentabel, receipt/Rezept, Roman/Roman, sensible/sensibel, sea/See, serious/seriös, site/Seite, spare/sparen, spend/spenden, star/Star, stern/Stern, sympathetic/sympatisch, sympathy/Sympathie, tablet/Tablett, oversight/Übersicht, oversee/übersehen, pass/Pass, unsympathetic/unsympatisch, undertaker/Unternehmer, warehouse/Warenhaus

A.3 Penn Treebank Tag Set to BNC Tag Set Mapping

Penn Treebank Tag	Description	BNC Tag
CC	coordinating conjunction	CJC
CD	cardinal number	CRD
DT	determiner	DT0, AT0
EX	existential there	EX0
FW	foreign word	UNC
IN	preposition/subordinating conjunction	PRF, PRP
JJ	adjective	AJ0
JJR	adjective, comparative greener	AJC
JJS	adjective, superlative	AJS
LS	list marker	CRD / -
MD	modal	VM0
NN	noun, singular or mass	NN0, NN1
NNS	noun plural	NN2
NNP	proper noun, singular	NP0[1]
NNPS	proper noun, plural	NP0[1]
PDT	predeterminer	DT0
POS	possessive ending	POS
PRP	personal pronoun	PNP[1]
PRP$	possessive pronoun	PNP[1]
RB	adverb	AV0[1]
RBR	adverb, comparative	AV0[1], AJC[1]
RBS	adverb, superlative	AV0[1], AJS[1]
RP	particle	AVP
TO	to	TO0
UH	interjection	UNC
VB	verb, base form	VVI[2]
VBD	verb, past tense	VVD[2]
VBG	verb, gerund/present participle	VVG[2]
VBN	verb, past participle	VVN[2]
VBP	verb, sing. present, non-3d	VVI[1,2]
VBZ	verb, 3rd person sing. present	VVZ[2]
WDT	wh-determiner	DTQ
WP	wh-pronoun	PNQ
WP$	possessive wh-pronoun	DTQ
WRB	wh-abverb	AVQ

[1] This BNC tag occurs more than once in the right column. This means that information is lost when the mapping is applied in the direction Penn Treebank tag set $\longrightarrow$ BNC tag set.

[2] The verbs *do* and *have* have special tags.

A.4 BNC Tag Set

A.4.1 List of C5 Tags

The following list is taken from [Leech]:

Each tag consists of three characters. Generally, the first two characters indicate the general part of speech, and the third character is used to indicate a subcategory. When the most general, unmarked category of a part of speech is indicated, in general the third character is 0. (For example, AJ0 is the tag for the most general class of adjectives.)

- **AJ0** Adjective (general or positive) (e.g. good, old, beautiful)
- **AJC** Comparative adjective (e.g. better, older)
- **AJS** Superlative adjective (e.g. best, oldest)
- **AT0** Article (e.g. the, a, an, no) [N.B. no is included among articles, which are defined here as determiner words which typically begin a noun phrase, but which cannot occur as the head of a noun phrase.]
- **AV0** General adverb: an adverb not subclassified as AVP or AVQ (see below) (e.g. often, well, longer (adv.), furthest. [Note that adverbs, unlike adjectives, are not tagged as positive, comparative, or superlative. This is because of the relative rarity of comparative and superlative adverbs.]
- **AVP** Adverb particle (e.g. up, off, out) [N.B. AVP is used for such "prepositional adverbs", whether or not they are used idiomatically in a phrasal verb: e.g. in 'Come out here' and 'I can't hold out any longer', the same AVP tag is used for out.
- **AVQ** Wh-adverb (e.g. when, where, how, why, wherever) [The same tag is used, whether the word occurs in interrogative or relative use.]
- **CJC** Coordinating conjunction (e.g. and, or, but)
- **CJS** Subordinating conjunction (e.g. although, when)
- **CJT** The subordinating conjunction that [N.B. that is tagged CJT when it introduces not only a nominal clause, but also a relative clause, as in 'the day that follows Christmas'. Some theories treat that here as a relative pronoun, whereas others treat it as a conjunction. We have adopted the latter analysis.]
- **CRD** Cardinal number (e.g. one, 3, fifty-five, 3609)
- **DPS** Possessive determiner (e.g. your, their, his)
- **DT0** General determiner: i.e. a determiner which is not a DTQ. [Here a determiner is defined as a word which typically occurs either as the first word in a noun phrase, or as the head of a noun phrase. E.g. This is tagged DT0 both in 'This is my house' and in 'This house is mine'.]
- **DTQ** Wh-determiner (e.g. which, what, whose, whichever) [The category of determiner here is defined as for DT0 above. These words are tagged as wh-determiners whether they occur in interrogative use or in relative use.]
- **EX0** Existential there, i.e. there occurring in the there is ... or there are ... construction
- **ITJ** Interjection or other isolate (e.g. oh, yes, mhm, wow)
- **NN0** Common noun, neutral for number (e.g. aircraft, data, committee) [N.B. Singular collective nouns such as committee and team are tagged NN0, on the grounds that they are capable of taking singular or plural agreement with the following verb: e.g. 'The committee disagrees/disagree'.]
- **NN1** Singular common noun (e.g. pencil, goose, time, revelation)
- **NN2** Plural common noun (e.g. pencils, geese, times, revelations)
- **NP0** Proper noun (e.g. London, Michael, Mars, IBM) [N.B. the distinction between singular and plural proper nouns is not indicated in the tag set, plural proper nouns being a comparative rarity.]

- **ORD** Ordinal numeral (e.g. first, sixth, 77th, last) . [N.B. The ORD tag is used whether these words are used in a nominal or in an adverbial role. Next and last, as "general ordinals", are also assigned to this category.]
- **PNI** Indefinite pronoun (e.g. none, everything, one [as pronoun], nobody) [N.B. This tag applies to words which always function as [heads of] noun phrases. Words like some and these, which can also occur before a noun head in an article-like function, are tagged as determiners (see DT0 and AT0 above).]
- **PNP** Personal pronoun (e.g. I, you, them, ours) [Note that possessive pronouns like ours and theirs are tagged as personal pronouns.]
- **PNQ** Wh-pronoun (e.g. who, whoever, whom) [N.B. These words are tagged as wh-pronouns whether they occur in interrogative or in relative use.]
- **PNX** Reflexive pronoun (e.g. myself, yourself, itself, ourselves)
- **POS** The possessive or genitive marker 's or ' (e.g. for 'Peter's or somebody else's', the sequence of tags is: NP0 POS CJC PNI AV0 POS)
- **PRF** The preposition of. Because of its frequency and its almost exclusively postnominal function, of is assigned a special tag of its own.
- **PRP** Preposition (except for of) (e.g. about, at, in, on, on behalf of, with)
- **PUL** Punctuation: left bracket - i.e. (or [
- **PUN** Punctuation: general separating mark - i.e. . , ! , : ; - or ?
- **PUQ** Punctuation: quotation mark - i.e. ' or "
- **PUR** Punctuation: right bracket - i.e.) or]
- **TO0** Infinitive marker to
- **UNC** Unclassified items which are not appropriately classified as items of the English lexicon. [Items tagged UNC include foreign (non-English) words, special typographical symbols, formulae, and (in spoken language) hesitation fillers such as er and erm.]
- **VBB** The present tense forms of the verb BE, except for is, 's: i.e. am, are, 'm, 're and be [subjunctive or imperative]
- **VBD** The past tense forms of the verb BE: was and were
- **VBG** The -ing form of the verb BE: being
- **VBI** The infinitive form of the verb BE: be
- **VBN** The past participle form of the verb BE: been
- **VBZ** The -s form of the verb BE: is, 's
- **VDB** The finite base form of the verb DO: do
- **VDD** The past tense form of the verb DO: did
- **VDG** The -ing form of the verb DO: doing
- **VDI** The infinitive form of the verb DO: do
- **VDN** The past participle form of the verb DO: done
- **VDZ** The -s form of the verb DO: does, 's
- **VHB** The finite base form of the verb HAVE: have, 've
- **VHD** The past tense form of the verb HAVE: had, 'd
- **VHG** The -ing form of the verb HAVE: having
- **VHI** The infinitive form of the verb HAVE: have
- **VHN** The past participle form of the verb HAVE: had

- **VHZ** The -s form of the verb HAVE: has, 's
- **VM0** Modal auxiliary verb (e.g. will, would, can, could, 'll, 'd)
- **VVB** The finite base form of lexical verbs (e.g. forget, send, live, return) [Including the imperative and present subjunctive]
- **VVD** The past tense form of lexical verbs (e.g. forgot, sent, lived, returned)
- **VVG** The -ing form of lexical verbs (e.g. forgetting, sending, living, returning)
- **VVI** The infinitive form of lexical verbs (e.g. forget, send, live, return)
- **VVN** The past participle form of lexical verbs (e.g. forgotten, sent, lived, returned)
- **VVZ** The -s form of lexical verbs (e.g. forgets, sends, lives, returns)
- **XX0** The negative particle not or n't
- **ZZ0** Alphabetical symbols (e.g. A, a, B, b, c, d)

Total number of grammatical tags in the BNC Basic tag set: 61

A.4.2 C7 to C5 Tag Set Mapping[28]

C7 tag	C5 tag	C7 tag	C5 tag	C7 tag	C5 tag
APPGE	DPS	NN2	NN2	RGT	AV0
AT	AT0	NNA	NNS	RL	AV0
AT1	AT0	NNB	NNS	RP	AVP
BCL	AV0	NNJ	NN0	RPK	AVP
CC	CJC	NNJ2	NN2	RR	AV0
CCB	CJC	NNL1	NP0	RRQ	AVQ
CS	CJS	NNL2	NP0	RRQV	AVQ
CSA	CJS	NNO	CRD	RRR	AV0
CSN	CJS	NNO2	NN2	RRT	AV0
CST	CJT	NNT1	NN1	RT	AV0
CSW	CJS	NNT2	NN2	TO	TO0
DA	DT0	NNU	NN0	UH	ITJ
DA1	DT0	NNU1	NN1	VB0	VBB
DA2	DT0	NNU2	NN2	VBDR	VBD
DAR	DT0	NP	NP0	VBDZ	VBD
DAT	DT0	NP1	NP0	VBG	VBG
DB	DT0	NP2	NP0	VBI	VBI
DB2	DT0	NPD1	NP0	VBM	VBB
DD	DT0	NPD2	NN2	VBN	VBN
DD1	DT0	NPM1	NP0	VBR	VBB
DD2	DT0	NPM2	NN2	VBZ	VBZ
DDQ	DTQ	NULL	NULL	VD0	VDB
DDQGE	DTQ	PN	PNI	VDD	VDD
DDQV	DTQ	PN1	PNI	VDG	VDG
EX	EX0	PNQO	PNQ	VDI	VDI
FO	UNC	PNQS	PNQ	VDN	VDN
FU	UNC	PNQV	PNQ	VDZ	VDZ
FW	UNC	PNX1	PNX	VH0	VHB
GE	POS	PPGE	PNP	VHD	VHD
IF	PRP	PPH1	PNP	VHG	VHG
II	PRP	PPHO1	PNP	VHI	VHI
IO	PRF	PPHO2	PNP	VHN	VHN
IW	PRP	PPHS1	PNP	VHZ	VHZ
JJ	AJ0	PPHS2	PNP	VM	VM0
JJR	AJC	PPIO1	PNP	VMK	VM0
JJT	AJS	PPIO2	PNP	VV0	VVB
JK	AJ0	PPIS1	PNP	VVD	VVD
MC	CRD	PPIS2	PNP	VVG	VVG
MC1	CRD	PPX1	PNX	VVGK	VVG
MC2	CRD	PPX2	PNX	VVI	VVI
MCGE	CRD	PPY	PNP	VVN	VVN
MCMC	CRD	RA	AV0	VVNK	VVN
MD	ORD	REX	AV0	VVZ	VVZ
MF	CRD	RG	AV0	XX	XX0
ND1	NN1	RGQ	AVQ	ZZ1	ZZ0
NN	NN0	RGQV	AVQ	ZZ2	ZZ0
NN1	NN1	RGR	AV0		

[28]I received this mapping via email from Dr. Paul Rayson, Lancaster University.

Erklärung

Ich erkläre, dass ich diese Diplomarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Bielefeld, den ....

Unterschrift