
Eberhard-Karls-Universität Tübingen
International Studies in Computational Linguistics
Bachelor of Arts Thesis
Course: Finite State Methods in Natural Language Processing
Lecturer: Dale Gerdemann

Analysis of German Compounds Using Weighted Finite State Transducers

Torsten Marek
c/o Michael Lantsch
Kemptener Straße 15
81475 München
torsten.marek@gmx.net

Hiermit versichere ich, dass ich die vorgelegte Arbeit selbstständig und nur mit den angegebenen Quellen und Hilfsmitteln einschließlich des WWW und anderer elektronischer Quellen angefertigt habe. Alle Stellen der Arbeit, die ich anderen Werken dem Wortlaut oder dem Sinne nach entnommen habe, sind kenntlich gemacht.

Contents

1. Introduction	4
1.1. Composition in German	4
1.2. Computational problems	7
1.3. Applications of Compound Splitting in NLP	8
1.4. Goals	8
2. Literature review	8
2.1. Simple lexicon-based approach	9
2.2. Letter n-grams	9
2.3. Parallel corpora	10
2.4. Finite state techniques	11
3. Compound analysis with wFSTs	12
3.1. Weighted finite state automata	12
3.2. Preparing the training data	15
3.3. The compound splitter	19
3.4. Structural analysis	21
3.5. Annotated walkthrough	23
3.6. Evaluation	25
4. Conclusion	28
5. References	30
A. Instructions on experiments	32
A.1. Prerequisites & preparations	32
A.2. Splitting compounds	32
B. Storage format descriptions	35
B.1. Condensed format	35
B.2. XML format	36
C. Training program code	37

1. Introduction

1.1. Composition in German

Composition is one of the most productive word formation processes in German and can be used to create new words of each lexical category. Most often, compounds are nouns, which means that the rightmost word of the compound is a noun like in *Polizei+auto* (police car) or a nominalized word, like in *Zähne+knirschen* (teeth grinding), where *Knirschen* (grinding) is derived from a verb. Other possibilities are adjectives, *hauch+dünn* (very thin, sheer) or verbs like *tot+schlagen* (to strike dead). While no other word categories are allowed in rightmost position (heads), others can function as so-called modifiers, such as adverbs, prepositions or bound lexical prefixes like *bio-*. A detailed discussion and comparison of word category combinations in compounds is contained in Olsen (2000).

German compounds, as well as compounds in Dutch, English, Danish, Finnish and Greek etc. can be analyzed as binary-branching trees. Morphosyntactic features (word category, inflectional paradigm) of the compound are always inherited from the rightmost word, which is also called the head of the compound. The contribution from the modifier is purely semantic. Once the compound is created, all morphosyntactic processes will happen at the head, i.e. the plural of the compound *Haus+tür* (house door) is *Haus+türen.PL* and not **Häuser+tür.PL*.

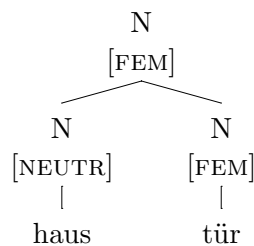


Figure 1: Simple compound

There are no restrictions on the complexity of the head or the compound, although very long words are usually dispreferred. The head can be complex as in *Computerfachzeitschrift* (professional computer magazine) in figure 2 as well as the modifier, for example in *Fahrradkurierjacke* (bike messenger coat) in figure 3. Still, in the case of the complex head, the rightmost word designates category, gender and morphosyntactic features of the whole compound. As visible in the figures, the features are taken from the head at every branch in the compound.

Though many compound constructions have been lexicalized over the years, even more are ad-hoc constructions. Still, in the course of this work, we will consider any word that can

synchronously be analyzed as a compound as one, and split it into its maximally small parts.

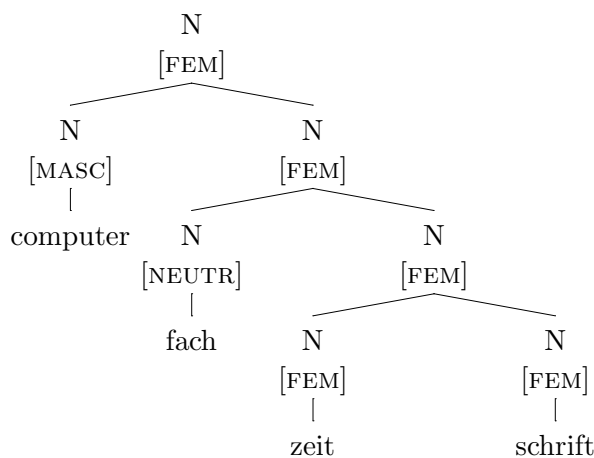


Figure 2: Complex heads

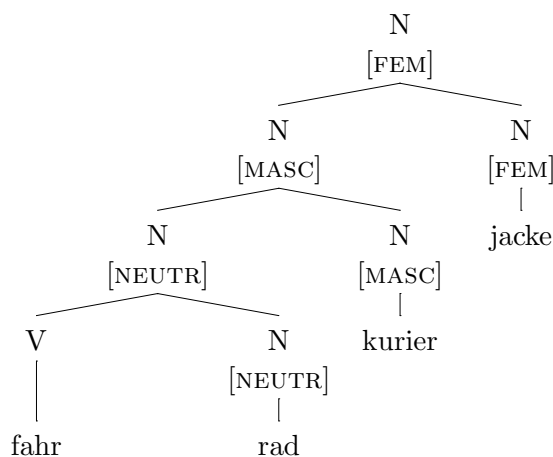


Figure 3: Complex modifiers

Instead of drawing trees whenever talking about the structure of a compound, bracketed expressions are used. The bracketed variants of the compounds in figures 1, 2 and 3 are (*haustür*), (*computer (fach (zeit schrift))*) and (*((fahr rad) kurier) jacke*). While this notation is less readable, it allows for a more compact presentation and greatly reduces workload, since no trees have to be set.

German compounds normally are written as a single orthographic unit, opposed to English compounds, where single words often are delimited by whitespace. Some compounds can contain

a separating dash between two or more of its parts. Some examples for dash insertion are:

Acronyms: *CD-Laufwerk* (CD drive)

New/uncommon foreign words: *Thread-Bibliothek* (threading library).

But: *Archivierungssoftware* (archiving software).

Proper names: *Lufthansa-Pressesprecher* (Lufthansa press speaker)

But: *Deutschlandlied* (name of Germany's national anthem)

Other hyphenations are left to the judgement of the author, *Festplattenzugriff* (hard disk access) can also be written as *Festplatten-Zugriff*. These spelling alternatives may not be confused with words like *Schritt-für-Schritt-Anleitung* (step-by-step guide), where the modifier *Schritt-für-Schritt* can not be analyzed as a compound, but is really a syntactic structure that has been glued together to be used as a single word.

Linking morphemes

The word form of the modifier used during composition is the nominative singular for nouns, the verbal root for verbs (*rennen* → *renn-* (to read)) and the uninflected form for adjectives. In some cases, when the modifier is a noun or a verb, an additional morpheme (the so-called linking morpheme, German *Fugenelement*) may be attached to it. An example is the plural suffix *-e* in *Hund|e+hütte* (dog house). The percentage of modifiers receiving a non-empty linking morpheme in noun-noun compounds ranges from 32% in Marek (2005) to 52% in Langer (1998).

Fuhrhop (1996) gives a comprehensive overview of the linking morpheme phenomenon in German. She divides linking morphemes into two categories: the older, paradigmatic morphemes and the younger, non-paradigmatic ones. Paradigmatic morphemes mimic the nominative plural, as in the example above, or the genitive singular like in *Anwalt|s+kanzlei* (law firm). Sometimes, the linking morpheme cannot be identified as a genitive singular synchronously, like in *Schmerz|ens+schrei* (cry of pain). In this case, the paradigm of the word has undergone changes, but the compound already had been lexicalized before that change happened and thus kept the now outdated form.

The only non-paradigmatic linking morpheme is *-s*. It attaches to all feminine nouns created with derivational suffixes like *-ung*, *-heit* or *-keit*, but also to words carrying the Latin suffix *-ion*. An example is *Bildung|s+system* (educational system), where **Bildungs* is no valid form of *Bildung*.

Linking morphemes attach only to nominal modifiers, with the exception of the schwa syllable *-ə*, which attaches to some verbs with monosyllabic roots. One such verb is *laden* (to load). If it

is used as a modifier like in *Lad|e+gerät* (charging device), it always appears with this particular linking morpheme.

Langer (1998) identifies a total number of 68 different linking morphemes in CISLEX (Langer et al., 1996). On the one hand, this includes very rare plural suffixes like *-ien* in *Prinzip|ien+reiter* (doctrinaire). On the other hand, the author considers a linking morpheme affixation that occurs together with a truncation of the root to be different from the sole affixation. Thus, several linking morphemes are counted more than one time, with different suffixes dropped from the modifier. In our training data and in this work, we distinguish between a total of 12 distinct morphemes, listed in table 1.

morpheme	frequency	example
∅	66.581%	<i>Feuer+wehr</i> (fire brigade)
-s	18.525%	<i>Gehalt s+scheck</i> (paycheck)
-n	6.977%	<i>Straße n+verkehr</i> (road traffic)
-en	3.228%	<i>Stern en+karte</i> (star map)
-e	1.797%	<i>Werb e+texter</i> (ad writer)
∅-dropping	1.732%	<i>Quell(e)+text</i> (source code)
-es	0.575%	<i>Tod es+opfer</i> (casualty)
-er	0.470%	<i>Ei er+schale</i> (egg shell)
-ns	0.082%	<i>Name ns+raum</i> (namespace)
-ens	0.016%	<i>Herz ens+wunsch</i> (heart’s desire)
stem umlaut	0.015%	<i>Mängel+liste</i> (list of deficiencies)
-nen	0.002%	<i>Königin nen+witwe</i> (queen widow)

Table 1: Linking morpheme distribution

1.2. Computational problems

One problem when dealing with compounds computationally is the problem of over-segmentation. In these cases, a compound split produces several analyses, which may all be more or less plausible. The following examples are taken from our training corpus:

- *Stadt+plan+ersatz* (town map substitute) vs. *Stadt+planer+satz* (town planner sentence)
- *Konsument|en+beschimpfung* (consumer insulting) vs. *Konsum+ente|n+beschimpfung* (consumption duck insulting)
- *Puffer+bau+stein* (buffer module) vs. *Puff+erbau+stein* (brothel building brick)
- *gelb+rot* (yellow-red) vs. *Gel+brot* (gel bread)
- *Angel+ernte* (fishing rod harvest) vs. *Angelernte* (semi-skilled person, from *anlernen* (to train))

An additional source of errors are derivative suffixes that can also function as words, like *-ei/Ei* (egg) and *-ion/Ion* (ion). Schiller (2005) mentions some more common errors. Among them is semantic ambiguity of single word segments, which we will not discuss further in this work, because we lack semantic information.

Another problem regarding compound usage is the selection of linking morphemes in natural language generation. For one modifier, different linking morphemes can be inserted in different compounds, like *Kind|er+arzt* (pediatrician) and *Kind+prozess* (child process) (Baayen, 2003; Marek, 2005). This problem is beyond the scope of this paper.

1.3. Applications of Compound Splitting in NLP

Compound splitting has successfully been used to improve the quality of example-based machine translators (EBMT) (Hedlund et al., 2002; Brown, 2002), predicting the head of a compound based on one or more modifiers in typing aids (Baroni et al., 2002) and to lower the number of out-of-vocabulary words in speech recognition systems (Spies, 1995). Other applications include decomposition of non-lexicalized compounds for spelling correction, information extraction/retrieval or word-level text alignment between languages that have different compounding mechanisms.

1.4. Goals

In this work, we will try to create a program that is able to split compounds into their single parts, based on a manually annotated training corpus with split compounds. The program should also try to find a bracketing for each compound, again based on a training corpus. The program will use weighted finite state transducers to find optimal splits and bracketings. We will see that the compound splitter works with high accuracy around 99%, while the structural analysis still has an acceptable accuracy around 96%.

2. Literature review

All compound splitting algorithms treated in the literature review use large knowledge resources like monolingual lexicons (Monz and de Rijke, 2001), monolingual corpora (Larson et al., 2000) or bilingual, parallel corpora (Brown, 2002; Koehn and Knight, 2003). The approach by Schiller (2005) combines a lexicon based approach with word frequency counts from several monolingual corpora. Except for Larson et al. (2000), every algorithm uses a lazy learning technique and relies on the availability of a lexicon with compound parts at application and, optionally, training

time. All articles treat compound splitting for German, although most of them could be used with other languages as well.

2.1. Simple lexicon-based approach

Monz and de Rijke (2001) use compound splitting to improve the accuracy of their monolingual IR system. In their strategy, they ignore the possibility of over-segmentation and only support two of the (non-empty) linking morphemes. The algorithm goes through a possibly complex noun from left to right and tries to find the current prefix in the monolingual lexicon. If the string can be found, the algorithm recurses on the suffix of the noun, possibly with a linking morpheme stripped from the beginning of the remaining string. If a sequence of words and linking morphemes can be found that consumes all characters of the input string, the word is recognized as a compound and no further analysis will be made. If no split can be found, but the whole noun itself appears in the lexicon, the word is recognized as a non-complex noun.

The authors of this paper point out several weaknesses in their approach. The algorithm does not handle all linking morphemes (although it could easily be expanded to do that), but also fails to handle more complex morphological phenomena like final-*ə* dropping or stem allomorphy (umlauts). Still, they completely ignore the possibility of over-segmentation. Thus, a word like *Metallindustrie* might be split into *Met+all+industrie* (mead outer space industry) instead of the semantically vastly more plausible split *Metall+industrie* (metal industry), if the lexicon contains the words needed for the former analysis.

The authors evaluate their compound splitter based on the number of compound parts it could analyze correctly. When the compound splitter is applied to all nouns (compounds and non-complex words), precision is 86.1% and recall 86.0%. If the compound splitter is only used for “real” compounds, precision drops to 49.3% and recall to 49.0%. This is the lowest result of all reviewed papers including our own experiments.

2.2. Letter n-grams

Larson et al. (2000) propose an algorithm to split compounds into words based on the counts of letter n-grams, which the authors claim is able to cope with linking morphemes as well as final-*ə* dropping.

First, they retrieve all word tokens from a large corpus and count all bigrams and trigrams at word boundaries. To split a word, for each letter n-gram the number of words that start and end with it are looked up in the corpus. Then, to find places for good splits, the difference between every two adjacent positions for both start and end n-grams are computed. If for a start and end n-grams a local maximum of the count differences is found at two adjacent positions, a word

boundary is inserted.

Unfortunately, the authors do not give any figures about accuracy, precision or recall of their compound splitter, because they are mainly interested in recombining the compound units to enlarge their lexicon for speech recognition.

2.3. Parallel corpora

Brown (2002)

Brown (2002) uses a parallel corpus of German and English article titles from medical journals. The algorithm proposed in his paper relies on the fact that German compounds often correspond to a series of single words in an English text, and that some or all of the words may be cognates. This assumption is justified when dealing with medical texts, where most of the words go back to Greek or Latin roots and thus differ only little between English and German. The authors give a number of correspondences between letters and sequences of letters to turn an English word into its German cognate and vice versa. The algorithm then tries to split a German compound based on its English translation.

For a compound candidate in German, a pair of English words that are similar enough according to the letter correspondences is searched. Given such a pair can be found, the algorithm tries to split the German compound in such a way that the parts are maximally similar to the English words, again based on the letter correspondences. As a result, the German word *Hormontherapie* can correctly be split into *Hormon+therapie* based on its corresponding pair of English words *hormone therapy* (Brown, 2002, p. 4).

In an extension, one or both of the English words can be translated into non-cognate German words, based on a bilingual dictionary extracted from the corpus. Again, similarity is measured on the (partially) translated pair.

For evaluation, the aforementioned titles from articles in medicine journals were used. The error rate is 1% if no translations are allowed (baseline figure), and raises to 4.6% and 7.6% if one or both English words can be translated prior to splitting. Although the number of errors raises, it has to be noted that if two translations are allowed, two times as many compounds as in the baseline case can be split at all, and still 1.5 times more with one only translation. Unfortunately, the authors use a sample of merely 500 German compound types for evaluation, which is small compared to their training corpus. This compound decomposition algorithm was also added to the author's MT system, and the number of German words without an English translation fell by 24.4%.

Koehn and Knight (2003)

The method for compound splitting described in Koehn and Knight (2003) is also based on finding one-to-one correspondences between parts of a German compound and English content words. First, a German compound is split into two words in every possible way. Between those two words, the linking morpheme *-s* or *-es* may occur. To find out if any of these splits are correct, a bilingual translation lexicon is created from a word-aligned parallel corpus. As an example, the German word *Polizei+auto* (police car) will have a split *Polizei+auto*, where the direct translations of both parts match each other in both languages. An erroneous split like *Angel+ernte* from section 1.2 will be ruled out, because none of the words *angling rod* or *harvest* can be found on the English side.

The authors also propose an extension to handle words that are translated differently depending on whether they appear as a freestanding word or as a compound modifier. Additionally, only a limited number of part-of-speech categories is allowed to appear in compounds.

In their evaluation, the best splitter, using the parallel corpus and the POS restrictions, has a precision of 93.8%, a recall of 90.1% and an accuracy of 99.1%. The remaining errors are, according to their claims, due to the lack of more training material. This coincides with our findings in section 3.6.

2.4. Finite state techniques

Schiller (2005) describes an approach to splitting compounds using weighted finite state transducers (wFSTs). In contrast to the other approaches, the algorithm from this paper does not create all possible splits for a compound itself, but uses the output from Xerox's morphological analyzer for German. This analyzer creates all splits for a compound and is thus susceptible to some of the errors shown in Section 1.2.

To select the correct split from the list of all possible ones, the single parts receive a weight based on their frequency in lists of manually decomposed compounds. These lists were extracted from several German corpora and together contain more than 240,000 compounds. The author also claims that nearly always the split with the least number of single words will be the most likely interpretation. This claim seems to hold true generally, and we have not been able to find a compound that can be split in a way that the split with more parts appears to be more likely to a human reader. Even if one such compound would exist, it would not invalidate the author's theory in general, because one compound would not have a large impact on the overall quality of the compound splitter. Only if it was possible to find a productive process or a series of such compounds, this theory would be weakened.

To make the algorithm prefer splits with less parts, the weight of each part is modified in such

a way that the weight of a single word is always more favorable than the weight of two parts together. Schiller (2005) uses the real semiring (cf. section 3.1), thus two weights are multiplied with each other. To achieve this, the probability of one part is restricted to the range $[0.5..0.7]$, because $\forall x \in \mathbb{N} \setminus \{1\}, 0.7^x < 0.5$.

The author uses two large training lists, a newspaper corpus and a collection of medical texts, and two evaluations lists, a corpus with texts of a major German weekly news magazine and a corpus with texts from several sources. The precision and recall measures vary with regard to training and evaluation corpora being used. Precision ranges from 89% to 98%, recall is always between 98% and 99%.

The technique used in Schiller (2005) has also been applied to the problem of word segmentation in Chinese texts (Sproat et al., 1994; Pereira et al., 1994). Chinese sentences are generally written without any spacing between the single words and thus exhibit some of the same problems of over-segmentation that are encountered when splitting German compounds. Basically, the lexicon is a weighted automaton containing sufficiently many Chinese words, or in the compound splitter case, parts of German compounds. Input I is composed with the closure over the lexicon D and the path with the best weight is chosen as the correct interpretation.

$$\text{bestpath}(I \circ D^*)$$

In the case of German compounds, this schema is modified to acknowledge the differences between heads and modifiers. For instance, a linking morpheme following a modifier needs to be recognized. Thus, input I is composed with the concatenation of the modifiers M and the heads H .

$$\text{bestpath}(I \circ [M^+ H])$$

The lexicon might also contain some productive prefixes (Sproat et al., 1994) to recognize words that did not appear in the lexicon or training corpus.

3. Compound analysis with wFSTs

3.1. Weighted finite state automata

In a weighted finite state automaton, each transition and final state is extended to have a weight. Each string in the language of the automaton thus has a weight, which is computed from the weights of the arcs that are visited during the automaton traversal and the weight of its final state.

One of the first applications of weighted finite state transducers as described in Mohri et al. (1996) or Pereira et al. (1994) has been speech recognition, where they were used to model the conversion from a series of sounds to phonemes, to words and, eventually, to sentences. In this paper, we will use the same technique as described in section 2.4.

When the automaton is traversed, the weights of the visited arcs and the final state are combined according to a certain *semiring*. A *semiring* $(A, \oplus, \otimes, \bar{0}, \bar{1})$ consists of the domain A , two binary operators, for addition (\oplus) and multiplication (\otimes) of weights and the neutral elements $\bar{0}, \bar{1} \in A$ for both operations ($\bar{0} \oplus a = a$ and $\bar{1} \otimes a = a$). The following constraints must be satisfied as well:

- $\bar{0}$ destroys any value when used with multiplication: $\bar{0} \otimes a = \bar{0}$
- Both operators are commutative: $a \oplus b = b \oplus a \wedge a \otimes b = b \otimes a$
- Multiplication rules over addition: $a \otimes (b \oplus c) = a \otimes b \oplus a \otimes c$

A semiring which satisfies these constraints is the *real* semiring $(\mathbb{R}^+, +, \cdot, 0, 1)$. 0 is the neutral element for addition ($a + 0 = a$), and 1 for multiplication ($1 \cdot a = a$), and multiplication rules over addition ($a(b + c) = ab + ac$), which has already been mentioned in section 2.4.

Another well-known semiring, which we are going to use in this paper, is the *tropical* semiring $(\mathbb{R}^+, \min, +, +\infty, 0)$. It satisfies all constraints:

$$\begin{aligned}
 \min(a, +\infty) &= a && \text{(neutral elements)} \\
 a + 0 &= a && \\
 \min(0, a) &= 0 && \text{(destruction)} \\
 \min(a, b) &= \min(b, a) && \text{(commutativity)} \\
 a + b &= b + a && \\
 a + \min(b, c) &= \min(a + b, a + c) && \text{(associativity)}
 \end{aligned}$$

The advantage the tropical has over the real semiring is that no number underflow can take place when using probabilities. With the real semiring, a probability might get too small to be represented by a floating point data type, when many numbers between 0 and 1 are multiplied. To be able to use probabilities with the tropical semiring, they are transformed with the formula

$$a' = -\log(a) \tag{1}$$

The resulting numbers are now interpreted as weight, where the lowest weights is the best, and two subsequent weights are added together rather than multiplied.

To illustrate the two different semirings and to show their equality, we imagine a language with the words AB, ABC, CDE, D and E with the respective probabilities 0.2, 0.25, 0.05, 0.25 and 0.25. The automaton to split up strings is

$$[D \ \varepsilon : +]^+ D$$

with D containing all words and their weights. At each word boundary, a plus sign will be inserted. The net using the probabilities converted by formula 1 is shown in figure 4. Here, only the first part of the automaton is actually printed ($D \ \varepsilon : +$), because the other part does not include anything new.

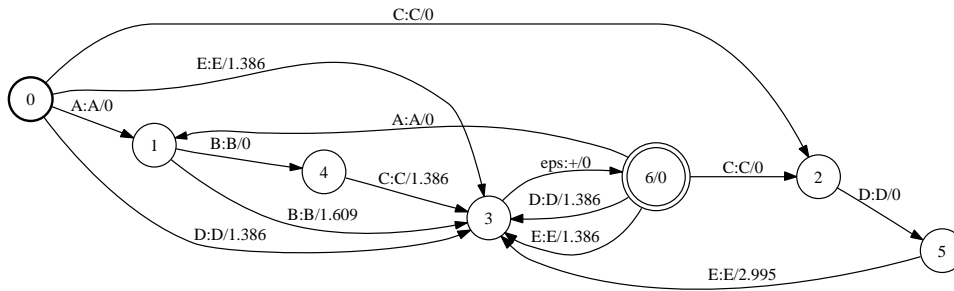


Figure 4: Toy language splitter

To find an optimal split for the string ABCDE, we compose its acceptor with our splitter automaton and receive the results in table 2. As we can see, the split ABC+D+E has a higher probability (or lower weight) than the other split. Thus, it will be selected as the correct split.

output	tropical	real	explanation
AB+CDE	4.6052	0.01	$-\log(0.2 \cdot 0.05) = -(\log(0.2) + \log(0.05)) = 4.6052$
ABC+D+E	4.1589	0.015626	$-\log(0.25^3) = -3 \log(0.25) = 4.1589$

Table 2: Equal weights

The automaton we are going to create to split German compounds will follow the same general outline and differ only in details like unknown word handling and markup of linking morphemes.

In this work, we are going to use the AT&T FSM LibraryTM by Mehryar Mohri, Fernando C. N. Pereira and Michael D. Riley¹ (FSMtools), which offers full support for creating and manipulating weighted finite state acceptors and transducers through a collection of CLI programs. In order to use the library more conveniently, we wrote a number of thin Python wrappers around the programs, along with classes to create weighted acceptors and transducers. The code example shows the program that creates the splitter for the little toy language, as seen in figure 4.

¹<http://www.research.att.com/~fsmtools/fsm/>

```

l = [("AB", .2), ("ABC", .25), ("CDE", .05),
     ("D", .25), ("E", .25)]

D = Automaton()
startstate = D.newState()
for word, weight in l:
    endstate = D.acceptWord(begin, startstate, weight=-log(weight))
    D.makeFinal(endstate)

D.finalize()

rmepsilon(
    concat(
        closure(
            concat(D, WordBoundaryInserter()),
            allowEmpty=False
        ),
        D
    )
).writeto("toylang.fsm")

```

The definition of automata is definitely and unavoidably more verbose than with specialized languages like XFST, SFST or LEXtools. While it would be possible to define the "+" as a binary concatenation operator, the unary Kleene star and plus operators can not be realized in a Python environment, because it is not possible (as in many other programming languages) to modify the arity of existing operators or define completely new operators.

In contrast to other finite state toolkits, FSMtools does not automatically determinize and minimize automata or remove ε -transitions. Therefore, when an intermediate automaton is saved as for later usage with the `writeto` method, it will be determinized and minimized. For a transducer, the "Encoded minimization" technique from the FSMtools manual² is used.

3.2. Preparing the training data

Text and knowledge resources

To create the compound splitter automaton, a large corpus of manually decomposed and checked compounds is needed. As the text source, we chose the German computer magazine *c't*³, which appears bi-monthly and is targeted to medium professional computer users. We used all texts

²<http://www.research.att.com/~fsmtools/fsm/man4/fsmintro.1.html>

³<http://www.heise.de/ct/>

from the issues 01/2000 to 13/2004 with a total of 117 magazines and more than 20,000 A4 pages of text.

The text archives can be bought on CD or DVD⁴, which contain the magazine articles as HTML files. After extracting the text from the HTML files and converting it to UTF-8, the corpus was 107 MiB big and contained 15 million word tokens according to the UNIX tool `wc`.

The next step was to delete all lower-case words to drop the majority of non-substantives, except for those at sentence beginning. To filter out function words, we created a stop word list based on the members of closed-list word classes in the STTS⁵. This left us with a list of 378,846 compound and non-compound words.

To get a list of non-compound words, we used the German lexicon in CELEX-2 (Baayen et al., 1995), release 2.5. The lexicon was still using the old German orthography, so we converted it half-automatically to the new orthography (mostly issues of `ß` vs. `ss`). Then we extracted a list of 33,457 nouns, verbs and adjectives from the list of German lemmas, using the information about word parts in the morphological database (GML) and converting it to the correct orthographic form using our updated version of the orthographic database (GOL). Unfortunately, the lexicon is quite outdated already and additionally contains a number of errors, so it was enriched with missing words or words that occur frequently in the text corpus. Also, adverbs that can be used as compound modifiers, like *innen* (inside), were added to this list. Additionally, the information from the database of orthographic word forms (GOW) was used to identify valid word forms of compound heads.

Compound splitting

We wrote a first compound splitter in Python that would try to create all possible splits for a compound and if there was more than one possible split, let a human decide which analysis to take. This very rudimentary splitter supports the following list of morphological phenomena:

- all linking morphemes from tables 1 and 3
- non-standard nominative suffixes (like *Ministerium*, *Ministerien* (ministry, ministries))
- productive derivational prefixes (*ent-*, *ver-* for verbs) and suffixes (like *-ung* for verbs, *-in* for nouns)
- stem umlauting with and without suffixation (*Magen*, *Mägen* (stomach, stomachs))
- dropping of word-final *ə* (like *Zellkern* (cell core), *Zelle+kern*)
- additional inflectional suffixes following the head

⁴<http://www.emedia.de/\%4016qs6RPmXPQ/bin/cd.pl>

⁵<http://www.sfs.uni-tuebingen.de/Elwis/stts/Wortlisten/WortFormen.html>

If all linking morphemes were allowed to occur after every noun without restriction, the splitter would make errors like breaking *Kursstand* (price level) into **Kurs|s+tand* (approx. price trumpery). To fix errors like these, linking morphemes were restricted to appear only in certain left contexts (written as Perl-style regular expressions), as listed in table 3.

left context	morphemes
C	ens, es
$C ei$	er
$V [rl]$	n
$C (au ei)$	en, en
$O e$	s
in	nen
$name$	ns

Symbols

C:consonant
V:vowel
O:obstruent

Table 3: left contexts for linking morphemes

If a compound had several possible splits, the program referred to the already split compounds and tries to find the best split based on compound part frequencies. It also favors the split with the least parts. If the splitter can not make a good guess, it will ask a human annotator to make that decision. This way, only about 12,000 compound splits actually needed human assistance.

The half-automatichal compound splitting extracted 158,657 compounds from the word list. As expected, this list contained a lot of errors, a lot of them were discovered and fixed while working on the several versions of the finite state compound splitter. The number of wrong splits or non-compound words should by now be below 3%, although we do not have exact figures on that.

Structural annotation

Each compound also has a structural analysis. This annotation work was also done with a Python program. For a compound with three parts, the bracketing program tried to find the correct analysis based on the frequencies of the trivially analyzed two-part compounds. If no choice could be made, a human had to decide. Compounds with more than three parts were annotated only by humans. The distribution of compounds of different lengths is given in table 6 on page 19. Again, compared to the actual size of the corpus, only a small number of around 8,000 compounds needed manual disambiguation.

Even for humans, finding a correct bracketing for a structure is not always easy and sometimes even impossible. Leaving the trivial case of two-part compounds aside, not all structurings are as clear as in the case of *Mittelwertsberechnung* (computation of the mean), where the only semantically possible structuring is *((mittel wert) berechnung)*, since *Mittelwert* (mean) is clearly

a compound, and the whole compound refers to the computation (*Berechnung*) of just that value. In the case of *Staatssicherheitsdienst* (National security service, the political police of the former GDR), it can either mean “service for national security”, (*(staat sicherheit) dienst*), or “security service of the state”, (*staat (sicherheit dienst)*), as opposed to another security service. In cases like *Adressverwaltungsprogramm* (address management program), the question is whether the compound should describe a management program for addresses, (*adresse (verwaltung programm)*), or a program for address management, (*(adresse verwaltung) programm*). In doubt, we normally decided for the complex modifier instead of the complex head. While we think that this is a sane default, the correct interpretation for compounds with semantically ambiguous parts can only be found by directly looking at the source context.

Corpus structure

Table 4 contains several examples from the condensed compound corpus, which is the output of the annotation programs. The format of the compound splits is described in Appendix B.

word	split	structure
Druckfestigkeit	druck{V,N}+festigkeit{N}	(druck festigkeit)
Lochkartensystem	loch{N,V}+karte n{N}+system{N}	((loch karte) system)
Insolvenzmasse	insolvenz{N}+masse{N}	(insolvenz masse)
Abfragezeitrum	abfrag e{V}+zeit{N}+raum{N}	(abfrag (zeit raum))

Table 4: Condensed compound splits

A converter script merges both split and structural analysis into an XML file, whose format is also described in appendix B. During this conversion, prepositions (category PREP) and prefixes (category PFX) will be joined with the next lexical word to the right. If a former compound now only consists of one word after that, it will be dropped from the corpus. This action of course enlarges the number of word types. But prepositions and prefixes always directly attach to the next word on the right, only very seldomly can they attach to a complex head. The decision to not treat prepositions as compound parts in their own right was done very early during the work on this paper, and further research should be done with the prepositions detached from their right neighbors. Additionally, all word forms were dropped and only compound types will remain. The size of the corpus and some more statistics are contained in table 5. The corpus is then split into a training and an evaluation corpus for 10-fold cross-validation. Technical details about and instructions for running the conversion process are given in appendix A.

Counts		Lengths			
Compounds:	121,597		min.	max.	avg.
Characters:	1,763,041	parts / compound	2	6	2.16
Tokens:	265,131	characters / compound	5	40	14.50
Types:	13,060	characters / part	2	20	6.65
Lemmas:	10,609				

Table 5: Compound corpus statistics

no. of parts	frequency
2	82.900%
3	16.193%
4	0.875%
5	0.030%
6	0.002%

Table 6: Distribution of parts per compound

3.3. The compound splitter

The first part of compound analysis is the compound splitter. It will try to find an optimal split for a given compound and also mark up possible linking morphemes, but it will not change the string otherwise, for instance convert words to lemmas etc. The compound *Bundestagsabgeordneter* (member of the German parliament) will be marked up to *Bund|es+tag|s+abgeordneter*.

The program that creates the splitter automaton reads in all compounds from the training corpus. It counts frequencies of words and word/linking morpheme combinations. These count frequencies are then converted into probabilities and converted using formula 1. The weight each word receives is

$$W(w) = -\log\left(\frac{C(w)}{N}\right) \quad (2)$$

where $C(w)$ is the absolute count of the word w and N is the absolute number of compound parts in the training corpus. The weight for each linking morpheme attaching to a modifier is computed with the formula

$$W(l) = -\log\left(\frac{C(l_w)}{C(w)}\right) \quad (3)$$

where $C(l_w)$ is the absolute count of the linking morpheme l appearing to the right of the word w . If the word only appears with one linking morpheme (mostly \emptyset), no additional weight

will be given, since $-\log(1) = 0$. If a word never appears as a modifier but only as a head, there are several options:

- Do not use the word as a modifier at all
- Use the word, but only with the \emptyset -morpheme as linking morpheme
- Use the word, with the linking morphemes that similar-ending words have
- Use the word, with the default probabilities for linking morphemes (cf. table 1)

In the evaluation, we will modify the training algorithm to try out some of these alternatives and see if there are differences in the performance of the algorithm.

The structure of the splitter automaton is already given in the simple example in section 3.1, we only have to modify it to allow different lexicons for heads and modifiers to support the insertion of linking morphemes.

$$[M \ \varepsilon: +]^+ \ H$$

The lexicon M contains all words with appropriate linking morpheme acceptors. Figure 5 shows an excerpt from this automaton. It contains only the word *Rind* (cow, bull), which has the weight 10.18 and can appear with the three linking morphemes \emptyset (*Rind+fleisch* (beef)), *-er* (*Rind|er+wahn* (mad cow disease)) and *-s* (*Rind|s+leder* (calf leather)). Between *Rind* and the linking morphemes, a boundary symbol will be inserted, except the linking morpheme is \emptyset . The lexicon H contains all words from the training corpus with their respective probabilities.

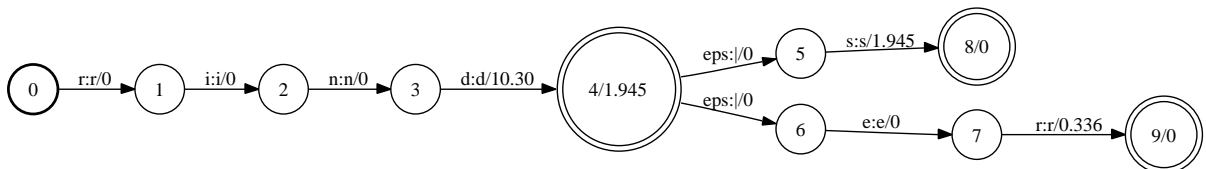


Figure 5: Linking morpheme boundary inserter

In contrast to Schiller (2005), the compound splitter does not prefer analyses with less splits by default. In practice, the split with the least parts will nearly always be selected as the correct analysis (cf. section 2.4), but in theory, it is possible to create an optimal split which has not the least parts. In our experiments, we found out that putting extra weight on the insertion of word boundaries and thus preferring less splits neither harms nor helps splitting performance. In order to keep the weighting as clean as possible, it has not been included in the final version.

Unknown word handling

An analysis of the compound corpus shows that for each possible training-evaluation split, the average number of unknown compound part types in the evaluation corpus is 325.90. Each evaluation corpus is around 12,150 compounds large, thus in average, 337.20 compounds will contain at least one unknown part. Since the compound splitter is not able to handle unknown words at all, one unknown part will result in a complete failure of the analysis. Thus, 2.78% of the compounds could not be analyzed without the splitter actually making any mistake. There are several possibilities to upgrade the compound splitter with unknown word handling. One is to include productive derivative suffixes directly into the lexicon, a technique which has successfully been used by Sproat et al. (1994). Still, our splitter lexicon is not by far as complete as theirs, and most of the words are not derived from already known words, but form completely new lexicon entries. Another possibility is to create a model for German words and unite it with the modifier and head lexicons. This approach has the drawback of drastically increasing the number of possible splits and thus rendering the number of analyses for a single compound returned by the splitter unusable, but in order to handle unknown words, we had to choose this option.

Our simple word model consists of the three first and last characters of each word, weighted after their occurrence in the training corpus. Between the prefix and the suffix, every other word character is allowed to occur. Though this word model is overly simplifying and non-accurate, the error rate made by our splitter is far lower than the 2.78% of compounds that could not be analyzed at all without it. The training program already extracts the linking morpheme distributions for word suffixes, so linking morphemes of unknown words can be recognized as well.

The unknown word model has the general structure

$$[P \ ?^*] \circ [?\ ?^* \ S] \circ [C^* \ V^+ \ C^*]$$

with P and S containing all prefixes/suffixes, C all consonants and V all vowels. The last expression ensures that all words generated by this model have at least one vowel. Additionally, the recognition of unknown prefixes and suffixes can be activated at an additional cost. It is also possible to put an additional weight on the recognition of unknown words. In the evaluation, we will find out which model works best.

3.4. Structural analysis

As described in section 3.2, every compound has a bracketed structure. However, since a bracketing is always recursive, it is not possible to handle them with finite state automata. In order

to be able to make a structural analysis of a compound, we “flatten” these structures. In the description of the formalism, we will use the word “atom” to refer to compound parts.

The relation between adjacent compounds and atoms is encoded using the four operators =, <, > and –.

operator	bracketed	flat
=	(atom atom)	atom = atom
<	(atom compound)	atom < compound
>	(compound atom)	compound > atom
–	(compound compound)	compound – compound

Table 7: structuring operators

Using the rules in table 7, each bracketing can unambiguously be converted into a flat structure. The word *Brennstoffzellenhersteller* (fuel cell producer) has the bracketed structure $((brenn\ stoff)\ zelle)\ hersteller$, from which the flat structure $brenn = stoff > zelle > hersteller$ is derived. In this case, the inversion of the flattening is unambiguous. In the case of *Lieblingsfernsehserie* (favorite TV series) with the structure $(liebling\ ((fern\ seh)\ serie))$, the flat structure is $liebling < fern = seh > serie$. If this is to be converted back into a bracketed structure, the result depends on the binding power of the two angle operators, which we will discuss later.

Before any structural operators can be inserted, all linking morphemes have to be removed and all words converted into their lemma forms, if applicable. The training corpus will be scanned for words with lemmas that diverge from the surface form, and an automaton will be created to do this conversion. The lemmatization automaton for a single word is

$$[L .p. ?*] (|:\varepsilon [?:\varepsilon]^+) +:+$$

The operator *.p.* is the priority union operator as described in Beesley and Karttunen (2003), and *L* contains all necessary surface-to-lemma conversions.

In the training script, we convert all bracketed structures into their flat counterparts and count the occurrences of the four structural operators after **and** before each word. This is done to find both left-to-right selectional preferences as well as right-to-left ones. After that, three automata are created. All the automata contain the same words, only the places where structural operators can be inserted vary. They depend on the current word’s position inside the compound, as listed in table 8.

As with the splitter automaton, the insertions are weighted according to their frequency in the training corpus. As visible, for each word boundary, two operators will be inserted. Therefore,

position	name	before	after
left border	F	-	replace “+” with operator
right border	L	insert operator	-
anywhere else	N	insert operator	replace “+” with operator

Table 8: Insertions of structural operators

the output is composed with an automaton I that only accepts those analyses where all two consecutive operators are of the same type. This automaton will then drop the second operator, as it is superfluous.

Since the words in these automata come from the same training corpus as in the splitter automaton, the same words are missing, and the input from the splitter automaton might contain unknown words. A complex model like in the splitter is not needed. Analogous to the automata with the known words, three automata that accept any sequence of characters and insert all relational operators at the correct places are priority-united with the automata that contain the known words. They are marked with the subscript letter u . The final form of the automaton for structural analysis is

$$[[F .p. F_u] [N .p. N_u]^* [L .p. L_u]] \circ I$$

The output from the structural analysis is always ambiguous and may contain invalid structurings. To find a correct analysis, the strings in the output language of the analysis are sorted by their weight and parsed with a parser for infix expressions, which has been enriched with constraints on operand types, e.g. the $=$ operator may only have atoms to both of its sides. The analysis with the lowest weight that can correctly be parsed is taken as the correct one. To resolve the ambiguities discussed earlier, a binding hierarchy of the operators has to be defined. Clearly, the operator $=$ binds most, and $-$ least. The asymmetric angle operators $<$ and $>$ lie inbetween those two, and we will evaluate which order leads to the best results.

3.5. Annotated walkthrough

In this section, we will describe how the several automata from the former sections work together. We will take the word *Farbtintenverbrauch* (color ink consumption) and apply it to each stage of the analysis step by step, explaining what the automata do. For this walkthrough, the first nine parts of the compound corpus of the have been used as training data. When other parts are used, the weights will vary.

Splitting

Before the word can be split, it will be lowercased, since the automata only contain lowercase characters. This reduces its alphabet size, and we do not gain anything from distinguishing case, since we are not interested if our compound is a noun or not. Then, a simple acceptor for the word will be created and composed with the splitter model from section 3.3. With unknown word handling switched off, the splitter outputs two possible analyses:

1. **farb+tinte|n+verbrauch**, weight=20.9745
2. **farb+tinte|n+verb+rauch**, weight=32.6939

The *bestpath* function correctly chooses the first analysis as the correct one.

Lemmatization

The lemmatization automaton removes all linking morphemes. There is also another phenomenon in this compound: the word *Farbe* (color) has lost its final syllable, which is introduced again. The output of the lemmatizer is **farbe+tinte+verbrauch**.

Structuring

The output of the lemmatizer is composed with the automaton from section 3.4, which inserts all possible structural operators in every combination and removes the word boundary symbols. This intermediation stage will contain invalid analyses like **farbe=<tinte>-verbrauch*, which are filtered out. The best two analyses are

1. **farbe=tinte=verbrauch**, weight=0.7671
2. **farbe=tinte>verbrauch**, weight=4.7566

followed by a lot more possible or impossible structurings.

Finding a good structure

In the last step, a program reads in the whole language and sorts it by the weight of the output. Then, it tries to parse each expression, starting with the one having the lowest weight. If that structure is not parsable (as is *farbe=tinte=verbrauch* in this case), it will continue with the other expressions and take the first one that can successfully be parsed as the correct analysis. In this example, the correct analysis is *((farbe tinte) verbrauch)*.

3.6. Evaluation

Splitter

In the evaluation of the compound splitter, we will test several different configurations for the training program. Due to time restraints, it was not possible to run every experiment for all training/evaluation splits. Instead, each was executed on three different parts of the corpus, whose numbers are given in the last column of each table. The evaluation figures are averaged over all the experiments. The first column of each table contain an ID for the automaton configuration. The second column (labeled "base") contains the ID of the configuration each automaton is built on. This way, the full configuration of each automaton is specified.

The values used to compute precision and recall are $P_g(c)$, which is the set of parts of the compound c as given in the training data, and $P_s(c)$, which is the set of parts returned by the splitter. $G(c)$ is the size of the intersection $P_g(c) \cap P_s(c)$ and thus the number of parts the splitter could correctly analyze. E is the set of compounds used for evaluation, and the function $I(x)$ maps the truth values $\{True, False\}$ to the numbers $\{1, 0\}$. For the tables, P , R , F and A are multiplied with 100 to get percentage figures.

$$\begin{aligned}
 \textit{precision } P &= \frac{\sum_{c \in E} G(c)}{\sum_{c \in E} |P_s(c)|} \\
 \textit{recall } R &= \frac{\sum_{c \in E} G(c)}{\sum_{c \in E} |P_g(c)|} \\
 F &= \frac{2PR}{P + R} \\
 I(x) &= \begin{cases} 1 & \text{if } x = \text{True} \\ 0 & \text{if } x = \text{False} \end{cases} \\
 \textit{accuracy } A &= \frac{\sum_{c \in E} I(P_g(c) = P_s(c))}{|E|}
 \end{aligned}$$

Table 9 contains the baseline values for an unweighted and a weighted compound splitter. Heads that did not occur as modifiers are not used as such in this model, and the unknown word model is not included

For the next experiments, the weighted automaton M2 will be used. In table 10, the impact of using words as modifiers that appear in head position only is measured. In the first experiment, only the \emptyset -morpheme is allowed as a linker. In the second one, the linking morpheme probabilities are guessed from the last three characters of the word (cf. section 3.3).

exp.	base	weights	P%	R%	F	A%	parts
M1	-	no	97.8802	96.1637	0.9701	95.8440	2, 9, 10
M2	M1	yes	97.9811	96.2628	0.9711	95.9509	

Table 9: Weight inclusion/exclusion

exp.	base	linking morphemes	P%	R%	F	A%	parts
M3	M2	\emptyset only	98.3170	96.8254	0.9756	96.5953	1, 5, 7
M4	M2	similar-ending words	98.4680	97.1282	0.9779	96.9051	

Table 10: Allow all words as modifiers

In the last experiment, we will see what difference the inclusion of the unknown word model makes. Unknown word handling is added to both M2 and M4. Early experiments have shown that the accuracy drops dramatically if there is no additional cost for the recognition of unknown words. This is due to the fact that an unknown word has no upper length limit if it is in modifier position. If a compound has more than two parts, the unknown word model will glue the leading parts into one word, because one unknown word (with the weight from prefix and suffix) can have less weight than two weighted words. For three words, the unknown word is nearly always cheaper to create. Because of that, the creation of an unknown word receives an additional cost of 100, making sure that unknown words are only created when needed.

exp.	base	P%	R%	F	A%	parts
M5	M2	99.2829	99.2580	0.9927	99.1009	3, 4, 8
M6	M4	99.3513	99.3951	0.9937	99.2270	

Table 11: Splitter with unknown word handling

These last experiments show that the current unknown word model is best when it is used least, and more known words to give a measurable increase in precision recall.

The most frequent problem of the unknown word model is that it consumes more than one word when the compound actually has more parts, like in *Ehrendoktorwürde* (honorary doctor), which is wrongly analyzed as *Ehre|n+doktorwürde*. The words *Doktor* and *Ehre* are known, but the weight for *Doktor* plus the weight for *Würde* as an unknown word is larger than taking *Doktorwürde* as one larger unknown word with the weights of the prefix *dok* and the suffix *rde*. Another problem appears if an unknown part has a known word as substring: since unknown words are strongly dispreferred, the known substring will be recognized as a word, and the remaining parts will become shorter unknown words.

Structure

Through the experiments on the compound splitter, we know that the upper limit of accuracy the structure analysis can reach is around 99.2% (cf. table 11, A% of M6), because a wrong split will unavoidably turn into a wrong structure. To see the influence of these errors, we will also make an experiment where we use the structure automaton on manually decomposed compounds.

The leftmost column of table 12 contains the binding hierarchy of the relational operators, with binding power decreasing from left to right. The third column shows where the input (the already split compounds) for the structuring comes from, either the annotated data (S2, S4) or from the best compound splitter automaton (S1, S3). Only the accuracy is used in the evaluation of the structure analysis.

exp.	binding hierarchy	compound source	A%	parts
S1	=, <, >,-	splitter M6	96.0252	2, 6, 9
S2		training data	97.0477	
S3	=, >, <,-	splitter M6	95.9622	
S4		training data	96.9847	

Table 12: Structure analysis evaluation

We see that the accuracy values are around 1% better when the gold splits are used, which is more than predicted by the error rate of the splitter. Additional errors are introduced because the evaluation of normal compound splits does not check for correctly marked-up linking elements, in order to be comparable to the figures from the papers in the literature review, which did not handle linking morphemes either. When linking morphemes are not correctly recognized, the compound parts can not be reduced to their lemmas and thus are unknown words to the structure automaton, which explains the additional errors.

As visible, giving the operator < a higher binding than > improves the accuracy a bit. Still, both solutions are unsatisfactory, because the analysis $w < w = w > w$ will always become $((w (w w)) w)$ with the first set and $(w ((w w) w))$ with the second. Further research should be conducted on this formalism. One possibility might be to introduce a second set of asymmetric operators which has less binding power than the current one, or try to learn long-distance relationship / selectional preferences in compounds.

4. Conclusion

In this paper, we have created a highly accurate finite-state compound analyzer for German. The compound splitter handles most of the phenomena occurring in German compounding, most prominently linking morphemes, umlauts in the stem and alternate nominative suffixes. With weights deduced from word frequencies in a training corpus, it is also possible to prevent the problem of producing semantically more difficult or impossible compound splits (so-called over-segmentation).

In order to analyze the tree-like structure of a compound using finite state techniques, we proposed a formalism to convert a bracketed structure into a flat string of words with operators determining the relation between words or parts of a compounds.

For training and evaluation, we decomposed and bracketed a list of more than 120,000 compounds extracted from a German computer magazine. While most of the work was done using a preliminary version of the compound splitter, a considerable amount of compounds needed manual intervention. Afterwards, additional corrections were done on the corpus, reducing the amount of errors like wrong splits, erroneously split words etc. From this data, we created weighted finite state transducers for splitting and structuring a compound using the AT&T finite state toolkit. For evaluation, we used 10-fold cross-validation to avoid overfitting the analyzers to the test data. Because of the time consumption of the analysis, we did not run every experiment with every possible split, but only on three parts each. The experiments showed that the best splitter was able to analyze 99.23% of the compounds in our evaluation data correctly, with unknown words being the biggest source for wrong analyses. Full analysis of a compound (splitting and assigning a bracketing) succeeded in 96.03% of all cases, using the best splitter and structural analyzer models. The errors of the splitter do have an impact on the structural analysis, since a wrong split unavoidably turns out into a wrong bracketing. But mainly, the structure model suffers from the comparatively small number of compounds longer than three parts (cf. table 6).

Directions for further work

As already mentioned in the evaluation, the compound splitter will definitely benefit from a better model for unknown German words and a better mechanism for including the unknown word model into the splitter lexicon. For the unknown word model, it would be possible to extract letter 2- or 3-grams from the training corpus and try to create a weighted automaton from them. Instead of n-grams, it is also possible to use syllables identified by a hyphenation algorithm for German words.

Another possible extension to the splitter automaton is the inclusion of high-frequent deriva-

tive suffixes for certain word categories to further minimize the number of out-of-vocabulary words.

For the structural analysis, it might be beneficial to identify long-distance relationships in compounds between two non-consecutive parts. Further studies should be made on the direction and importance of selections, whether left-to-right selections should receive more or less weight than right-to-left ones, and whether preferences of a head are more or less important than preferences of a modifier. To investigate this question, more compounds with more than two parts are needed. Given the already high quality of the compound splitter, creating a new and bigger compound corpus than the one created and used in this paper should be less work.

As a direct and much-needed application of the splitter, one might think about enriching a free spell checker with the ability to split more complex, non-lexicalized German compounds, whose parts may or may not be in the spell checking dictionary. Unfortunately, the finite-state toolkit used in this paper is neither open nor free, and thus usage of a similar toolkit (like the RWTH toolkit) or a rewrite of the algorithm might be required, while retaining the original ideas.

A by-product of this paper is the considerable amount of annotated compounds. This list will be published on the author's homepage (as will be the code from this paper) in the hope that other researchers will benefit from the work already done.

5. References

- Baayen, R. Harald (2003). *Probabilistic approaches to morphology*. In Bod, Rens, Hay, Jennifer, and Jannedy, Stefanie, eds., *Probabilistic Linguistics*. Cambridge, MA: MIT Press.
- Baayen, R. Harald, Piepenbrock, R., and Gulikers, L. (1995). *The CELEX Lexical Database (Release 2) [CD-ROM]*. Philadelphia, PA: Linguistic Data Consortium, University of Pennsylvania.
- Baroni, M., Matiasek, J., and Trost, H. (2002). *Predicting the components of German nominal compounds*. In van Harmelen, F., ed., *Proceedings of the 15th European Conference on Artificial Intelligence (ECAI 2002)*. Amsterdam, Netherlands: IOS Press.
- Beesley, Kenneth R. and Karttunen, Lauri (2003). *Finite State Morphology*. Stanford, CA: CSLI Publications.
- Brown, Ralf D. (2002). *Corpus-Driven Splitting of Compound Words*. In *Proceedings of the Ninth International Conference on Theoretical and Methodological Issues in Machine Translation (TMI-2002)*. Keihanna, Japan.
- Fuhrhop, Nanna (1996). *Fugenelemente*. In Lang, Ewald and Zifonun, Gisela, eds., *Deutsch - typologisch*. Berlin / New York: de Gruyter.
- Hedlund, T., Keskustalo, H., Pirkola, A., Airio, E., and Järvelin, K. (2002). *Utacir @ CLEF 2001 - Effects of Compound Splitting and N-gram Techniques*. In Peters, C., Braschler, M., Gonzalo, J., and Kluck, M., eds., *Second Workshop of the Cross-Language Evaluation Forum (CLEF 2001), Revised Papers*.
- Koehn, Philipp and Knight, Kevin (2003). *Empirical Methods for Compound Splitting*. In *Proceedings of EACL 2003*. Budapest, Hungary.
- Langer, Stefan (1998). *Zur Morphologie und Semantik von Nominalkomposita*. In *Tagungsband der 4. Konferenz zur Verarbeitung natürlicher Sprache (KONVENS 98)*.
- Langer, Stefan, Maier, Petra, and Oesterle, Jürgen (1996). *CISLEX - an Electronic Dictionary for German. Its Structure and a Lexicographic Application*. In *Proceedings COMPLEX 96*.
- Larson, Martha, Willett, Daniel, Köhler, Joachim, and Rigoll, Gerhard (2000). *Compound splitting and lexical unit recombination for improved performance of a speech recognition system for German parliamentary speeches*. In *6th Int. Conference on Spoken Language Processing (ICSLP)*. Beijing, China.

-
- Marek, Torsten (2005). *Predicting Linking Elements in German Compounds: A Stochastic Approach*.
URL <http://diotavelli.net/files/tmarek-linkmorphemes.pdf>
- Mohri, Mehryar, Pereira, Fernando, and Riley, Micheal (1996). *Weighted Automata in Text and Speech Processing*. In *Proceedings ECAI-96, Workshop on Extended Finite State Models of Language*. Budapest, Hungary.
- Monz, Christof and de Rijke, Maarten (2001). *Shallow morphological analysis in monolingual information retrieval for Dutch, German and Italian*. In Peters, C., Braschler, M., Gonzalo, J., and Kluck, M., eds., *Second Workshop of the Cross-Language Evaluation Forum (CLEF 2001), Revised Papers*.
- Olsen, Susan (2000). *Composition*. In Booij, Gert, Lehmann, Christian, and Mugdan, Joachim, eds., *Morphologie: ein internationales Handbuch zur Flexion und Wortbildung*. Berlin / New York: de Gruyter.
- Pereira, Fernando, Riley, Micheal, and Sproat, Richard (1994). *Weighted Rational Transductions and their Application to Human Language Processing*. In *ARPA Workshop on Human Language Technology*.
- Schiller, Anne (2005). *German Compound Analysis with wfsc*. In *Proceedings of the Fifth International Workshop of Finite State Methods in Natural Language Processing (FSMNLP 2005)*. Helsinki, Finland.
- Spies, Marcus (1995). *A Language Model for Compound Words in Speech Recognition*. In Pardo, J., Enriquez, E., Ortega, J., Ferreiros, J., Marcias, J., and Valverde, F., eds., *Proceedings of the 4th Conference on Speech Communication and Technology (EUROSPEECH 95)*. Universidad Politecnica, Madrid, Spain.
- Sproat, Richard, Shih, Chilin, Gale, William, and Chang, Nancy (1994). *A Stochastic Finite-State Word-Segmentation Algorithm for Chinese*. In Kaufmann, Morgan, ed., *32nd Annual Meeting of the Association for Computational Linguistics*. New Mexico State University, Las Cruces, New Mexico.

A. Instructions on experiments

A.1. Prerequisites & preparations

Although the programs should in principle run on any platform where both Python and FSMtools are supported, it has only been tested on Linux. We tried to avoid writing platform-dependent code, but subtle differences in handling shell commands might lead to problems under Win32 operating systems; after all, the script are based on calling external programs.

To execute the Python programs, at least Python 2.4 is needed, version 2.3 will **not** work. The code has also successfully been tested with Python 2.5.

The following extension modules for Python are used in this work:

PyParsing: A parsing framework for Python by Paul T. McGuire, included in the code & data download

URL: <http://pyparsing.sourceforge.net/>

cElementTree: An object-oriented data model for XML trees by Frederik Lundh, included in Python 2.5

URL: <http://effbot.org/zone/celementtree.htm>

Both FSMtools⁶ and LEXtools⁷ are needed. FSMtools contains the binaries for creating and manipulating finite state automata, but unfortunately lacks a tool to extract the language of an automaton. This is done using the binary `lex fsm strings` from LEXtools. All binaries have to be in directories contained in the search path for binaries (environment variable `PATH`), so that they can be executed by the Python programs.

The annotated data and the code used to create the automata are available at the URL <http://diotavelli.net/files/wfst-splitter.tar.bz2>

When the file is extracted, a directory named `wfst-splitter` is created. Table 13 lists the files in this directory, along with a short description of their contents.

A.2. Splitting compounds

To convert the condensed training data into the XML files used by the training and evaluation programs, execute

```
$ ./create_ccorpus
$ ./splitccorpus -p1
```

Depending on speed and memory of the computer, this might take a while. After that, three new files will have been created in the subdirectory `data/`: the complete corpus `ccorpus.xml`,

⁶<http://www.research.att.com/~fsmtools/fsm>

⁷<http://www.research.att.com/~alb/lextools>

Programs

<code>analyze_compound</code>	Analyze a single compound
<code>create_ccorpus</code>	Convert condensed into XML format
<code>evaluate</code>	Run an evaluation
<code>splitccorpus</code>	Split the XML corpus into training and evaluation part
<code>train</code>	Create the automata from the training data

Modules

<code>ccorpus_utils.py</code>	Parsers for the condensed format
<code>files.py</code>	Filenames
<code>fsm.py</code>	FSMtools bindings and automaton classes
<code>nets.py</code>	Simple, reusable automata
<code>pyparsing.py</code>	Python parsing framework by Paul T. McGuire
<code>statistics.py</code>	Misc. statistical functions
<code>structure.py</code>	Conversion between flat and bracketed structure
<code>symbols.py</code>	Automaton alphabet, symbol names

Data files

<code>COPYING</code>	Copyright and license statements
<code>data/alpha.syms</code>	Automaton alphabet
<code>data/ccformat.dtd</code>	DTD for the XML corpus
<code>data/ccorpus.txt</code>	Annotated compounds
<code>data/structures.txt</code>	Compound structures

Table 13: Files

the training corpus `cc-train.xml` and the evaluation corpus `cc-eval.xml`.

To create the automata, run

```
$ ./train
```

which will output some status information and timings. Here, for the first time the external binaries will be used. If errors like

```
/bin/sh: fsmcompile: command not found
Traceback (most recent call last):
  File "./train", line 357, in ?
...
IOError: [Errno 32] Broken pipe
close failed: [Errno 32] Broken pipe
```

occur, please check if the FSMtools binaries are executable and can be found by the scripts. Putting them into the same directory as the script might not work if “.” is not contained in `PATH`.

After that, the compound analyzer is ready and can be tried out the with the script `analyze_compound`:

```
$ ./analyze_compound Bundestagsabgeordneter  
bund|es+tag|s+abgeordneter<126.0480>
```

```
$ ./analyze_compound -f Bundestagsabgeordneter  
((bund tag) abgeordneter)<133.6444>
```

By default, a compound will be split only. A full analysis, resulting in a bracketing, is done if the switch `-f` is provided.

To run a full evaluation, the program `evaluate` is used. Analogous to `analyze_compound`, the script will normally only create compound splits, unless the switch `-f` is used. The evaluation program will output each error done by the splitter, along with the analysis from the evaluation corpus. It will also print statistics and timings at the end.

```
$ ./evaluate -l 250  
E: a:entreißdieb+stahl, g:entreiß+diebstahl, err=1, c=96, err%=1.0417%  
E: a:höhen+verstell+bar, g:höhen+verstellbar, err=2, c=204, err%=0.9804%  
E: a:klang+zauber+ei, g:klang+zauberei, err=3, c=231, err%=1.2987%  
E: a:fachter+minus, g:fach+terminus, err=4, c=233, err%=1.7167%
```

```
Correct analyses: 98.4000% of 250 tests  
Precision: 98.5267  
Recall: 98.8909  
F: 0.9871  
Evaluation took 5.4914 s, 21.9654 ms / compound
```

```
$ ./evaluate -fl 100  
E: a:(funk (netz gruppe)), g:((funk netz) gruppe), err=1, c=11, err%=9.0909%  
E: a:((spitze übertragung) wert), g:(spitze (übertragung wert)), err=2, c=84, err%=2.3810%  
E: a:(entreißdieb stahl), g:(entreiß diebstahl), err=3, c=96, err%=3.1250%  
E: a:((fax deck) blatt), g:(fax (deck blatt)), err=4, c=100, err%=4.0000%
```

```
Correct analyses: 96.0000% of 100 tests  
Evaluation took 3.1946 s, 31.9456 ms / compound
```

Most scripts do come with command line switches to control their behaviour, help on these switches is always provided when they are called with the `--help` option. The source code files contain additional documentation in comments.

B. Storage format descriptions

B.1. Condensed format

The file `data/ccorpus.txt` contains all compounds in a condensed format. For each compound, all information is preserved to convert it either into lemmas-only or into the original surface form, covering all phenomena like linking morpheme, alternate nominative suffixes, dropped final syllables or stem umlauts.

Each line in the file contains the lowercased original surface form and the split compound, divided by a single whitespace character. The format for a split compound is:

```
CompoundSplit = Part "+" Part { "+" Part };
Part = Root [ "," NomSuffix] [ "," LinkMorpheme | "(" InflexionSuffix ")" ]
      "{" Category [ "," Category ]}";

NomSuffix = Char { Char };
Root = Char | "A" | "O" | "U" { Char | "A" | "O" | "U" };
LinkMorpheme = Char { Char };
InflexionSuffix = Char { Char };
Category = "A" | "V" | "N" | 'ADV' | 'PREP' | 'PFX';
(* omitted: Char is any character from the set {a..z, ä, ö, ü, ß} *)
```

The condensed representation of the word *Farbtintenverbrauch* from the walkthrough in section 3.5 is:

```
farbtintenverbrauch farb,e{N}+tinte|n{N}+verbrauch{N}
```

All characters between `,` and `|` (the linking morpheme boundary) are added to the surface form when the lemma is created. All characters between linking morpheme boundary and the start of the category set are removed from it. In some case, these two processes are combined, like in *Geschichtsverfälschung* (history falsification):

```
geschichtsverfälschung geschicht,e|s{N}+verfälschung{N}
```

An uppercase O, U or A in the root mean that this character is an umlaut in the surface, but an unaccented character in the lemma, like in *Bücherregal* (book shelf):

```
bücherregal bUch|er{N,V}+regal{N}
```

Each word has at least one category, but usually does have more, although normally only one category is correct. The correct category for each word is supposed to be in the set of categories, but they are not checked at all. The training script makes no use of them, and they are simply dropped during conversion to the XML format.

The format has been created to be as condensed as possible, while retaining all information. Also, it had to be editable with a minimal amount of keystrokes (which is not true for most XML formats). Regular expressions for parsing this format are in the module `ccorpus_utils.py`.

For each distinct compound, there is a bracketing in `data/structure.txt`, which contains only the lemmas. The entries for the above three examples are

```
((farbe tinte) verbrauch)
(geschichte verfälschung)
(buch regal)
```

During conversion, the compounds in the two files are associated by their lemma-normalized forms, which can be computed from both representations.

B.2. XML format

All other programs after the conversion use the XML corpus exclusively. There is a DTD for the XML format, although the XML parser used in the code is non-validating.

```
<!ELEMENT ccorpus (compounds)>
<!ELEMENT compounds (compound+)>
<!ELEMENT compound (cpart+, structure)>
<!ELEMENT cpart (lemma, surface, sfx)>
<!ELEMENT lemma (#PCDATA)>
<!ELEMENT surface (#PCDATA)>
<!ELEMENT sfx (#PCDATA)>
<!ELEMENT structure (#PCDATA)>

<!ATTLIST compounds count NMTOKEN #REQUIRED>
<!ATTLIST compound id ID #REQUIRED
length NMTOKEN #REQUIRED>
<!ATTLIST cpart id ID #REQUIRED
type (head | mod) #REQUIRED>
```

The following example is a corpus containing only the well-known word *Farbtintenverbrauch*.

```
<ccorpus>
  <compounds length="1">
    <compound id="c100894" length="3">
      <cpart id="c100894_p0" type="mod">
        <lemma>farbe</lemma>
        <surface>farb</surface>
        <sfx/>
      </cpart>
      <cpart id="c100894_p1" type="mod">
        <lemma>tinte</lemma>
```

```

    <surface>tinten</surface>
    <sfx>n</sfx>
  </cpart>
  <cpart id="c100894_p2" type="head">
    <lemma>verbrauch</lemma>
    <surface>verbrauch</surface>
    <sfx/>
  </cpart>
  <structure>((o o) o)</structure>
</compound>
</compounds>
</ccorpus>

```

In this format, the lemma and the surface form are already precomputed, as well as the linking morpheme. The attribute `type` of the `cpart` nodes may either contain "mod", which means that this compound part is in the modifier position somewhere, or "head", for the head of the compound. Also, the structure of the compound is included, although the lemmas have been dropped to save space.

C. Training program code

The following listing shows the code of the training script, which creates the automata discussed in sections 3.3 and 3.4. This script is the most important piece of code created for this paper, all other scripts and modules are straightforward (evaluation, application) and contain code that is only loosely related to the actual problem (automaton classes, FSMtools bindings).

```

#!/usr/bin/env python2.4
# -*- coding: utf-8 -*-
from __future__ import division

import sys
import time

from fsm import *
from nets import *
from statistics import *
from files import *
from symbols import *
from ccorpus_utils import getCompounds
from structure import parseBracketed

CONTEXT_WIDTH = 3
UNKNOWN_WORDS = True
USE_HEADS_AS_MODS = True

```

```

USE_HEADS_WITH_LINKERS = True

def merge_max(from_, to):
    for key, value in from_.iteritems():
        to[key] = max(value, to[key])

class Relations(object):
    """Class for counting occurrences of relational symbols"""
    max_before = dict.fromkeys(RELATION_SYMBOLS, 0)
    max_after = dict.fromkeys(RELATION_SYMBOLS, 0)

    def __init__(self):
        self.before = {}
        self.after = {}

    def count_before(self, sym):
        count(self.before, sym)

    def count_after(self, sym):
        count(self.after, sym)

    def normalize(self):
        normalize_nlog(self.before)
        normalize_nlog(self.after)

        merge_max(self.before, self.max_before)
        merge_max(self.after, self.max_after)

UnknownWord = None
def insertStructureSymbols(word, parts, rels):
    """Insert a word with the relational symbols at the correct
    places into the automaton parts"""
    first_part, next_part, last_part = parts

    may_insert = {
        first_part: bool(rels.after),
        next_part : bool(rels.before) and bool(rels.after),
        last_part : bool(rels.before)
    }

    word_begins = {
        first_part: first_part.begin,
        next_part : next_part.newState(),
        last_part : last_part.newState()
    }
    word_ends = {
        first_part: first_part.newState(),
        next_part : next_part.newState(),

```

```

    last_part : last_part.newState()
}

for rel_sym in RELATION_SYMBOLS:
    for idx, p in enumerate(parts):
        if idx > 0 and rel_sym in rels.before and may_insert[p]:
            p.insert(p.begin, word_begins[p], rel_sym, rels.before[rel_sym])

        if idx < 2 and rel_sym in rels.after and may_insert[p]:
            p.createArc(word_ends[p], p.newFinalState(),
                WORD_BOUNDARY, rel_sym, weight = rels.after[rel_sym])

for p in parts:
    if may_insert[p]:
        if word is UnknownWord:
            p.accept(word_begins[p], word_ends[p], wordchars)
            p.accept(word_ends[p], word_ends[p], wordchars)
        else:
            p.acceptWord(word_begins[p], word, to_= word_ends[p])

if may_insert[last_part]:
    last_part.makeFinal(word_ends[last_part])

def appendLinkers(linkerdict, automaton, from_state):
    """Append linker boundaries and linkers to a state in the
    automaton"""
    for linker in linkerdict:
        if linker != "":
            lstart = automaton.newState()
            automaton.insert(from_state, lstart, LINKER_BOUNDARY)
            if linker.islower():
                e = automaton.acceptWord(lstart, linker, weight = linkerdict[linker])
            else:
                e = automaton.newState()
                automaton.insert(lstart, e, linker, weight = linkerdict[linker])
            automaton.makeFinal(e)
        else:
            automaton.makeFinal(from_state, linkerdict[linker])

def SurfaceToLemma(to_lemma):
    """Create the surface->lemma conversion automaton"""
    print "lemmas"
    lemma_exceptions = Automaton()
    begin = lemma_exceptions.newState()
    for surface, lemma in to_lemma.iteritems():
        current = begin
        for i_sym, o_sym in izip(chain(surface, repeat(EPSILON)), lemma):

```

```

        next = lemma_exceptions.newState()
        lemma_exceptions.createArc(current, next, i_sym, o_sym)
        current = next
    lemma_exceptions.makeFinal(next)
lemma_exceptions.finalize()
return priority_union(lemma_exceptions, QmarkStar())

def IdenticalRelationSymbols():
    """Create the automaton that ensures only identical adjacent
    relation symbols"""
    ident_rel_syms = Automaton()
    begin = ident_rel_syms.newFinalState()
    ident_rel_syms.accept(begin, begin, wordchars)
    for rel_sym in RELATION_SYMBOLS:
        first = ident_rel_syms.newState()
        ident_rel_syms.accept(begin, first, rel_sym)
        ident_rel_syms.delete(first, begin, rel_sym)
    ident_rel_syms.finalize()
    return ident_rel_syms

def makeStructureAutomaton(word_rels, to_lemma):
    lemma = SurfaceToLemma(to_lemma)

    print "structure (known words)"
    parts = [SafeEntryAutomaton() for x in range(0, 3)]
    for word, rels in word_rels.iteritems():
        rels.normalize()
        insertStructureSymbols(word, parts, rels)
    for p in parts:
        p.finalize()

    print "structure (unknown words)"
    unknown_parts = [SafeEntryAutomaton() for x in range(0, 3)]
    unknown_rels = Relations()
    unknown_rels.before = Relations.max_before
    unknown_rels.after = Relations.max_after

    insertStructureSymbols(UnknownWord, unknown_parts, unknown_rels)
    for p in unknown_parts:
        p.finalize()

    print "Assemble structure automaton"
    first_part, next_part, last_part = parts
    first_unknown, next_unknown, last_unknown = unknown_parts

    compose(

```

```

minimize_encoded(
  concat(
    closure(
      concat(
        lemma,
        LinkMorphemeRemover(),
        WordBoundaryAcceptor()
      ),
      allowEmpty = False
    ),
    lemma
  )
),
minimize_encoded(
  concat(
    priority_union(
      first_part,
      first_unknown
    ),
    closure(
      priority_union(
        next_part,
        next_unknown
      ),
      allowEmpty = True
    ),
    priority_union(
      last_part,
      last_unknown
    ),
  )
),
IdenticalRelationSymbols()
).writeto(HIERARCHY_AUT, "const_input_indexed")

def createUnknownWordModel(prefixes, suffixes, all_linkers, with_linkers):
    UNKNOWN_WORD_WEIGHT = 100

    # prefixes
    prefix_net = Automaton()
    PFXBEG = prefix_net.newState()

    for pfx in prefixes:
        end = prefix_net.acceptWord(PFXBEG, pfx, weight = prefixes[pfx])
        prefix_net.makeFinal(end, UNKNOWN_WORD_WEIGHT)

    X = (max(prefixes.itervalues()) + 1) / CONTEXT_WIDTH
    b = PFXBEG

```

```

for i in range(0, CONTEXT_WIDTH):
    n = prefix_net.newState()
    prefix_net.accept(b, n, wordchars, weight = X)
    b = n
prefix_net.makeFinal(b, UNKNOWN_WORD_WEIGHT)
prefix_net.finalize()

#suffixes
suffix_net = Automaton()
SFXBEG = suffix_net.newState()
for sfx in suffixes:
    end = suffix_net.acceptWord(SFXBEG, sfx, weight = suffixes[sfx])

    if not with_linkers or sfx not in all_linkers:
        suffix_net.makeFinal(end)
    else:
        appendLinkers(all_linkers[sfx], suffix_net, end)

X = (max(suffixes.itervalues()) + 1) / CONTEXT_WIDTH
b = SFXBEG
for i in range(0, CONTEXT_WIDTH):
    n = suffix_net.newState()
    suffix_net.accept(b, n, wordchars, weight = X)
    b = n

suffix_net.makeFinal(b)
suffix_net.finalize()

return compose(
    SimpleWordModel(),
    concat(
        prefix_net,
        QmarkStar()
    ),
    concat(
        QmarkStar(),
        suffix_net,
    )
)

def makeSplitterAutomaton(words, word_linkers, prefixes, suffixes, all_linkers):
    print "splitter (known words)"
    heads = Automaton()
    mods = Automaton()
    H_BEGIN = heads.newState()
    M_BEGIN = mods.newState()
    # insert all words into both automata
    for word in words:

```

```

heads.makeFinal(heads.acceptWord(H_BEGIN, word, weight = words[word]))

if word in word_linkers:
    m_end = mods.acceptWord(M_BEGIN, word, weight = words[word])
    appendLinkers(word_linkers[word], mods, m_end)
else:
    if USE_HEADS_AS_MODS:
        m_end = mods.acceptWord(M_BEGIN, word, weight = words[word])
        mods.makeFinal(m_end)
heads.finalize()
mods.finalize()

if UNKNOWN_WORDS:
    print "splitter (unknown words)"
    heads = union(
        heads,
        compose(
            LimitLength(13),
            createUnknownWordModel(prefixes, suffixes, all_linkers, False)
        )
    )

    mods = union(
        mods,
        createUnknownWordModel(prefixes, suffixes, all_linkers, True)
    )

    print "Assemble splitter automaton"
    concat(
        closure(
            concat(
                mods,
                WordBoundaryInserter()
            ),
            allowEmpty = False
        ),
        heads,
    ).writeto(SPLITTER_AUT, "const_input_indexed")

prefixes = {}
suffixes = {}
linkers = {}
mod_l = {}
c_parts = {}

missing_linkers = set()

word_rels = {}
to_lemma = {}

```

```

start_time = time.time()
print "reading training data..."
for compound in getCompounds(TRAIN_CCORPUS):
    structure = parseBracketed(compound.findtext("structure"))
    words = list(compound.findall("cpart/lemma"))

    rels = structure.relations()
    for idx, rel_sym in enumerate(rels):
        word_rels.setdefault(words[idx].text, Relations()).count_after(rel_sym)
        word_rels.setdefault(words[idx+1].text, Relations()).count_before(rel_sym)

    for cpart in compound.findall("cpart"):
        surface = cpart.findtext("surface")

        sfx = cpart.findtext("sfx")

        if sfx and sfx.islower():
            surface = surface[:-len(sfx)]

        count(c_parts, surface)

        if cpart.attrib["type"] == "head":
            if surface not in mod_l:
                missing_linkers.add(surface)
        else:
            count2(mod_l, surface, sfx)
            missing_linkers.discard(surface)

        if cpart.findtext("lemma") != surface:
            to_lemma[surface] = cpart.findtext("lemma")

    # collect word beginnings and endings
    if len(surface) > CONTEXT_WIDTH:
        count(prefixes, surface[:CONTEXT_WIDTH])
        count(suffixes, surface[-CONTEXT_WIDTH:])
        if cpart.attrib["type"] == "mod":
            count2(linkers, surface[-CONTEXT_WIDTH:], sfx)

# convert absolute counts into weights
normalize_nlog(prefixes)
normalize_nlog(suffixes)
normalize2_nlog(linkers)
normalize2_nlog(mod_l)
normalize_nlog(c_parts)

# add linkers for head-only words
if USE_HEADS_WITH_LINKERS:
    for word in missing_linkers:

```

```
sfx = word[-CONTEXT_WIDTH:]
if sfx in linkers:
    mod_l[word] = linkers[sfx]

del missing_linkers

makeSplitterAutomaton(c_parts, mod_l, prefixes, suffixes, linkers)
makeStructureAutomaton(word_rels, to_lemma)

print "Training took %.2f seconds" % (time.time() - start_time)
```