Maschineller Erwerb von Wortklassifikationsregeln

Schriftliche Arbeit zur Erlangung der Würde einer
Magistra Artium
im Fach
Linguistische Datenverarbeitung/Computerlinguistik
an der
Universität Trier

vorgelegt von
Sandra Kübler
Brunnenstr. 3
54317 Gusterath

Trier, im Oktober 1995

Inhaltsverzeichnis

Einleitung 1

1. Grundlagen auf den Gebieten der Wortklassifizierung und des Maschinellen Lernens 3

1.1. Ein kurzer Überblick über die Entwicklung auf dem Gebiet der Wortklassifizierung 3

1.2. Maschinelles Lernen 4

2. Die transformationsbasierte fehlergesteuerte Wortklassifizierung 8

2.1. Erstellung der kontext-freien Regeln 10

2.2. Erstellung der kontext-sensitiven Regeln 14

2.3. Bewertung des Systems 15

2.4. Unterschiede zum Vorgänger- bzw. Nachfolge-Modell 17

2.5. Änderungen des Systems zur Anpassung auf das Taggen eines deutschsprachigen Korpus 19

3. Ergebnisse aus der Anwendung des transformationsbasierten fehlergesteuerten Tagging auf die deutsche Sprache 21

3.1. Die durchgeführten Tests 21

3.2. Ergebnisse aus dem Test mit 1.000 Sätzen 22

3.3. Unterschiede bei 2.000 Sätzen 25

3.4. Eine Auswahl fehlerhaft getaggter Sätze 26

4. Transformationsbasiertes fehlergesteuertes Tagging im Vergleich zu anderen Ansätzen 28

4.1. Regelbasierte Ansätze 29

4.1.1. Computational Grammar Coder 29

4.1.2. TAGGIT 31

4.1.3. THE TAGGER 33

4.1.4. Das ASCOT Tagging System 34

4.2. Probabilistische Ansätze 36

4.2.1. CLAWS 36

4.2.2. VOLSUNGA 40

4.2.3. Churchs Stochastic Parts Program 42

4.2.4. Das Hidden-Markov-Modell von Cutting et al. 44

4.2.5. POST 46

4.3. Neuere Ansätze 47

4.3.1. Das konnektionistische Taggingsystem von Benello et al. 47

4.3.2. NETgrams 49

4.3.3. Hindles Tagger 51

4.3.4. ENGCG 53
4.3.5. Der constraint-basierte Ansatz von Chanod und Tapanainen 55
4.3.6. TAKTAG 57
4.4. Vergleich mit Brills System 58
4.4.1. Wie hoch ist die Erfolgsquote? 59
4.4.2. Ist das System ein vollständiges Taggingsystem? 59
4.4.3. Können unbekannte Wörter systematisch behandelt werden? 60
4.4.4. Wird ein Lexikon verwendet? 61
4.4.5. Welche Informationen müssen dem System bereitgestellt werden? 61
4.4.6. Ist ein bereits getaggtes Textstück nötig? 62
4.4.7. Ist die im System gewonnene linguistische Information einsehbar? 63
4.4.8. Wird eine Lemmatisierung durchgeführt? 63
4.5. Zusammenfassung 64
5. Fazit 65
Anhang A: Das Tagset 66
Anhang B: Das Programmlisting 70
Anhang C - Die aus 1.000 Sätzen gelernten kontext-sensitiven Regeln 139
Anhang D - Die aus 2.000 Sätzen gelernten kontext-sensitiven Regeln 154
Literaturverzeichnis 161

Einleitung

Bis vor einigen Jahren hat die automatische Wortklassifizierung (part-of-speech tagging) auf dem Gebiet der Computerlinguistik nur eine untergeordnete Rolle gespielt. In der Grundlagenforschung auf den Gebieten der Syntax und der Semantik sah man diese Information als gegeben an. Und da üblicherweise nur kleine Datenmengen, z.B. zum Testen eines Parsers, verwendet wurden, war es kein Problem, diese Daten manuell zu taggen.
Grundlegend geändert hat sich die Situation erst, als nicht mehr nur Grundlagenforschung betrieben wurde, sondern die ersten anwenderorientierten Systeme angegangen wurden. Diese sollten nicht nur zu Forschungszwecken verwendet werden, sondern waren für den Markt bestimmt. Auf allen Gebieten jedoch, sei es nun die automatische Übersetzung, das Information Retrieval oder die Spracherkennung, war eine Taggingkomponente unumgänglich. Selbst eine effiziente Rechtschreibkontrolle oder ein Textverarbeitungssystem mit Spracheingabe kommt ohne eine automatische Wortklassifizierung nicht aus, wenn nicht eine weitaus aufwendigere Grammatikkomponente vorhanden ist: In (1) und (2) z.B. kann ohne die Information über die Wortart nicht entschieden werden, ob das Wort "verfahren" als Substantiv verwendet wird, also groß geschrieben werden muß, oder ob es als Verb gebraucht ist und daher klein geschrieben werden muß.

(1) Das Verfahren steht noch aus.
(2) Wie wir hier verfahren müssen, ist noch nicht geklärt.

Die Grenze zwischen Tagging- und Grammatikkomponente ist jedoch nicht scharf umrissen definiert, da im oben beschriebenen Beispiel die Taggingkomponente grammatikalische Information verwendet. Eine Grammatikkomponente dagegen ist auf eine vorhergehende Taggingkomponete angewiesen, da sonst eine große Anzahl sehr präziser Grammatikregeln nötig wäre, die Abhängigkeiten von Wort- und nicht von Tagsequenzen beschreiben.
Die Aufgabe der automatischen Wortklassifizierung ist es jedoch nicht nur, die Wortarten, wie z.B. Substantiv, Adjektiv, Verb, Konjunktion, etc., zu ermitteln. Es werden vielmehr komplexe Merkmalstrukturen (Tags) verwendet, die außer der Information über

die Wortart zusätzliche Information über Numerus, Kasus, Tempus und ähnliches enthalten. Diese zusätzliche Information ist z.B. in der Maschinellen Übersetzung wichtig. Soll z.B. der Satz (3) ins Englische übersetzt werden, muß bekannt sein, welches der zwei Objekte das direkte und welches das indirekte ist. Dies läßt sich nur aus der Kasus-Information der Objekte ermitteln.

(3) Er übergab den Mann seinen Mitarbeitern.

Einen weiterer Grund für das zunehmende Interesse an der automatischen Wortklassifizierung stellten auch die ersten, in den 70er Jahren erstellten, maschinenlesbaren, großen Korpora dar. Ein fehlerfrei getaggtes Korpus ist Voraussetzung zu allen weiteren Verwendungsarten. Hierbei wurde zum ersten Mal deutlich, daß eine automatische Wortklassifizierung der manuellen in Schnelligkeit und Konsistenz überlegen ist. Ein Experiment von Marcus et al. (1993, 319), in dem manuelles Taggen mit manueller Postedition verglichen wird, zeigt, "that manual tagging took about twice as long as correcting, with about twice the inter-annotator disagreement rate that was about 50% higher."

In dieser Arbeit soll erst ein kurzer Überblick über die Gebiete der Wortklassifizierung und des maschinellen Lernens gegeben werden (Kap. 1). Dann wird der Ansatz der transformationsbasierten fehlergesteuerten Wortklassifizierung (Transformation-Based Error-Driven Tagging) von Brill (1992, 1993, 1994) vorgestellt und für die Verwendung für deutschsprachige Korpora angepaßt (Kap. 2). Hierbei handelt es sich um ein regelbasiertes System, bei dem die Regeln im Gegensatz zu den bisher vorhandenen Systemen nicht manuell erarbeitet und dem System vorgegeben werden; das System erwirbt die Regeln vielmehr selbst anhand von wenigen Regelschemata aus einem kleinen bereits getaggten Lernkorpus. In Kapitel 3 werden die Ergebnisse aus der Anwendung des Systems auf Teile eines deutschsprachigen Korpus dargestellt. In Kapitel 4 schließlich werden andere Taggingsysteme vorgestellt und mit dem System von Brill (1993) anhand von acht Kriterien verglichen.

1. Grundlagen auf den Gebieten der Wortklassifizierung und des Maschinellen Lernens

In diesem Kapitel soll kurz ein Überblick über die beiden grundlegenden Gebiete gegeben werden, auf denen das System von Brill (1992, 1993, 1994) aufbaut: das Gebiet der Wortklassifizierung und das des Maschinellen Lernens.

1.1. Ein kurzer Überblick über die Entwicklung auf dem Gebiet der Wortklassifizierung

Die ersten Systeme zur automatischen Wortklassifizierung stammen aus den 60er Jahren und waren regelbasiert (vgl. auch Kap. 5.1). Zu dieser Gruppe gehören die Systeme von Klein und Simmons (1963) sowie von Greene und Rubin (1971). Da die Regeln für solche Systeme in mühevoller Kleinarbeit und ohne konzeptuelle Grundlage erstellt wurden, waren die Fehlerquoten trotz der großen Menge an eingegebener Information sehr hoch (> 10%). Die Verbesserung der Resultate durch Anpassen der Regeln führte zu Flip-Flop-Problemen. D.h., die Fehler, die durch das Anpassen der Regeln verhindert werden sollten, wurden durch andere Fehler ersetzt, die durch Nebenwirkungen von einzelnen Regeln oder durch unerwünschte Wechselwirkungen von mehreren Regeln entstanden.
Eine Verbesserung ergab sich erst durch die Einführung von probabilistischen Modellen, wie z.B. dem Hidden-Markov-Modell (vgl. auch Kap. 5.2.). Ein Grund, warum diese Modelle sich einer großen Beliebtheit auf dem ganzen Gebiet des Natural Language Processing (NLP) erfreuen, liegt darin, daß sich die benötigten statistischen Angaben automatisch berechnen lassen. Auch die niedrige Fehlerquote (< 3%) spricht für die Verwendung von solchen Modellen in der automatischen Wortklassifizierung. Nachteile dagegen ergeben sich aus der großen Menge der benötigten Daten zum Trainieren des Systems, aus der Unfähigkeit der Systeme, idiomatische Ausdrücke zu erkennen, aus der schlechten Portabilität des Systems auf andere Textsorten und aus der unzugänglichen linguistischen Information in Form von Übergangswahrscheinlichkeiten. Zu der Gruppe

der probabilistischen Systeme zählen z.B. diejenigen von Garside (1987), Church (1988), DeRose (1988), de Marcken (1990) und Weischedel et al. (1993).
Neuere Ansätze verwenden neuronale Netze (Nakamura, Shikano1989, Benello et al. 1989), ein regelbasiertes Verfahren unter Einbeziehung von syntaktischem Wissen (Voutilainen 1995), ein constraint-basiertes Verfahren (Chanod, Tapanainen 1995), etc. (vgl. auch Kap. 5.3.).
Einen weiteren neuen Ansatz stellen maschinelle Lernverfahren (Hindle 1989, Brill 1992, 1993, 1994) dar, bei denen die Wortklassifizierungsregeln nicht eingegeben, sondern vom System gelernt werden; weiterhin existiert auch ein hybrides Verfahren (Lee et al. 1995) mit einer probabilistischen Komponente und einer Komponente zum Lernen von Regeln, wobei die Lernkomponente von Brill (1992) als Grundlage dient.

1.2. Maschinelles Lernen

Wie bei vielen Begriffen aus der künstlichen Intelligenz und der Computerlinguistik kann der Begriff 'Maschinelles Lernen' zwar ohne weiteres intuitiv erklärt werden, es ist jedoch schwierig, eine formale Definition zu geben. Die wohl bekannteste Definition stammt von Simon (1983, 28): "Learning denotes changes in the system that are adaptive in the sense that they enable the system to do the same tasks or tasks drawn from the same population more efficiently and more effectively the next time." Es wird jedoch deutlich, daß die Definition nicht ausreicht. "Diese Definition ist aus zwei Gründen kritisiert worden: sie deckt auch solche Phänomene ab, die man üblicherweise nicht als Lernen bezeichnet, und sie deckt nicht alle dem Lernen zugerechneten Phänomene ab." (Morik 1993, 248), d.h. sie ist einerseits zu weit und andererseits zu eng gefaßt. Auch die zufällige Verwendung besserer Hilfsmittel würde z.B. unter Lernen gefaßt, während der Erwerb von Wissen, das für eine gestellte Aufgabe nicht relevant ist, nicht darunter fällt. Andere Ansätze definieren Lernen über den Erwerb expliziten Wissens, den Erwerb von Fähigkeiten, die Theorie- und Hypothesenbildung (vgl. Cohen, Feigenbaum 1982, 326-327), das Konstruieren oder Verändern von Repräsentationen von Erfahrungen oder Interaktionen mit der Umwelt (vgl. Morik 1993, 249). Doch auch diese Definitionen decken nicht den gesamten Bereich des Lernens ab.

Eine der Schwierigkeiten beim Erstellen der Definition für Maschinelles Lernen besteht darin, daß sehr viele verschiedene Aspekte des Lernens berücksichtigt werden müssen. Diese Aspekte können auch als verschiedene Arten des Lernens verstanden werden. Die wichtigsten Kategorien sind wie folgt:

- rote learning
- Lernen durch Deduktion
- Lernen durch Unterweisung
- Lernen durch Analogie
- Lernen durch Induktion

'Rote learning' bedeutet, daß dem System Information in einer Weise zugeführt wird, die es direkt verarbeiten kann. Das System muß die Informationen also nicht verstehen oder interpretieren, es muß nur in einer bestimmten Situation die entsprechenden Informationen abrufen. Beim Lernen durch Deduktion wird vom System ebenfalls keine 'neue' Information erworben: es wird nur von einer Regel auf den Einzelfall abgeleitet.

Bsp. Aus der Regel

X ist ein Mensch → X ist sterblich

kann abgeleitet werden:

Axel ist ein Mensch → Axel ist sterblich.

Beim Lernen durch Unterweisung muß das System ebenfalls von einer allgemeinen Anweisung auf die spezielle Handlung schließen. Aber "The system must understand and interpret the high-level knowledge and relate it to what it already knows." (Cohen, Feigenbaum 1982, 333). Diese Umformung wird Operationalisierung genannt. Das Lernen durch Analogie hat zum Ziel, Wissen aus einem bekannten Bereich auf einen neuen zu übertragen, indem strukturelle Ähnlichkeiten zwischen Ausgangs- und Zielbereich erkannt und daraus eine Abbildung zwischen Elementen des Ausgangs- und des Zielbereichs entwickelt wird. Das Lernen durch Induktion stellt in gewisser Hinsicht eine Umkehrung der Deduktion dar: Es wird aus dem Einzelfall der Allgemeinfall hergeleitet. Da bei dieser Methode jedoch schon ein einziges Gegenbeispiel (z.B. ein schwarzer Schwan nach hundert weißen) die Induktion zunichte machen kann, hat diese Art des

Schließens einen stark hypothetischen Charakter. Das Lernen durch Induktion kann weiterhin in drei Unterkategorien eingeteilt werden: das Lernen anhand von Beispielen, das Lernen durch Beobachtung und Entdeckung, und das 'conceptional clustering'. Ziel des Lernens anhand von Beispielen ist es, eine möglichst spezielle Beschreibung zu finden, die alle positiven Beispiele abdeckt, aber keines der negativen. Diese Art von Lernen wird auch Lernen durch Suche genannt (vgl. Morik 1993, 258). Beim Lernen durch Beobachtung und Entdeckung werden durch kontrollierte Veränderungen der Umgebung Schlüsse auf die Gesetzmäßigkeiten gezogen. Beim conceptual clustering schließlich sollen anfangs ungeordnete Elemente in geordnete Klassen einer Taxonomie aufgeteilt werden (vgl. Abb. 1): Die Einteilung wird aufgrund eines Distanz- oder Ähnlichkeitsmaßes vorgenommen.

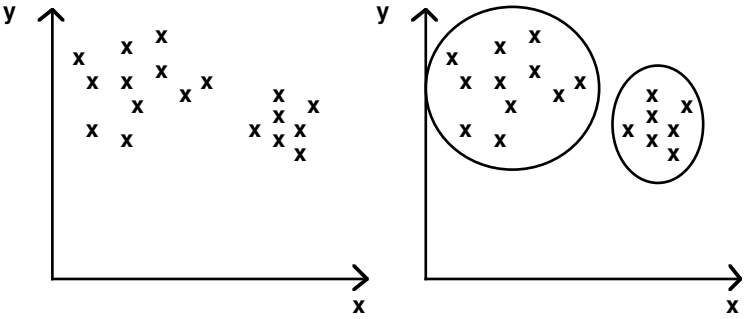

Abbildung 1

Beim Versuch, Brills System (1992, 1993, 1994) in diese Lernkategorien einzuordnen, ergeben sich jedoch Probleme. Das Lernverfahren weist eine gewisse Ähnlichkeit mit deduktivem Lernen auf: Aus den allgemeinen Regelschemata müssen die einzelnen Regeln entwickelt werden. Im Gegensatz zu einer Deduktion jedoch, sind diese Regelschemata nicht allgemein gültig, die Aufgabe des Systems ist es vielmehr, aufgrund der Lernkorpora die besten Variablenbelegungen für diese Schemata zu finden, so daß eine gültige Aussage entsteht. Dieses Merkmal weist auf induktive Mechanismen im Lernverfahren hin. Der oben angesprochene hypothetische Charakter dieser Art des Schließens wird dadurch vermieden, daß eine Bewertungsmöglichkeit für die vom System vorgeschlagenen Regeln besteht, da die Lernkorpora bereits getaggt sind, so daß entschieden werden kann, welche Regel das Korpus am besten beschreibt. Man kann Brills System also als gemischt deduktiv-induktives System beschreiben.

Eine anders geartete Form von Lernen findet sich in neuronalen Netzen, auch konnektionistische Modelle genannt. Neuronale Netze sind informationsverarbeitende Systeme, deren Aufbau sich an die Gehirnstruktur von Säugetieren anlehnt, allerdings in sehr vereinfachter Form. Sie bestehen aus einer großen Anzahl von Zellen und gerichteten Verbindungen zwischen diesen Zellen. In solchen Systemen wird Information durch die Aktivierung bestimmter Zellen über die Verbindungen von der Eingabeschicht über verdeckte Schichten zur Ausgabeschicht weitergeleitet. Der Lernvorgang in neuronalen Netzen besteht in der Einstellung der Gewichte der Zellverbindungen, so daß die Impulse verstärkt oder gehemmt werden. Die Einstellung erfolgt durch eine große Menge von Trainingsmustern, die so lange in das Netz eingegeben werden, bis die Gewichtsmatrix so angepaßt ist, daß die Trainingsmuster richtig ihren Klassen zugeordnet werden. Anders als bei der künstlichen Intelligenz, die die Informationsverarbeitung als die Manipulation von Symbolen definiert, besagt das Paradigma des Konnektionismus, "daß Informationsverarbeitung als Interaktion einer großen Anzahl einfacher Einheiten (Zellen, Neuronen) angesehen wird, die anregende oder hemmende Signale an andere Zellen senden. Symbole werden üblicherweise nur implizit dargestellt durch das Aktivierungsmuster der Einheiten (verteilte Repräsentation)." (Zell 1994, 26).

2. Die transformationsbasierte fehlergesteuerte Wortklassifizierung

Das größte Problem, das sich bei wissens- oder regelbasierten Ansätzen zur automatischen Wortklassifizierung stellt, ist die umfassende Modellierung des benötigten linguistischen Wissens. Probabilistische Ansätze umgehen dieses Problem, indem sie kein linguistisches Wissen verwenden. Statt dessen werden Übergangswahrscheinlichkeiten zwischen den Tags verwendet, um die wahrscheinlichste Sequenz von Tags zu ermitteln. Die Übergangswahrscheinlichkeiten werden im allgemeinen aus großen, bereits getaggten Korpora berechnet. Probleme ergeben sich bei diesem Ansatz vor allem bei idiomatischen Ausdrücken, die dann durch explizit gespeichertes linguistisches Wissen vor- oder nachbearbeitet werden müssen.

Bei der transformationsbasierten fehlergesteuerten Wortklassifizierung (Brill 1992, 1993, 1994) wird das zur Wortklassifizierung nötige Wissen anhand von vorgegebenen Regelschemata, die kaum linguistisches Wissen enthalten, und von kleinen Mengen von bereits richtig getaggtem Text automatisch gelernt. Das in den Regelschemata enthaltene Wissen besteht darin, daß dem System ein Konzept von Suffix und Präfix, ebenso wie von idiomatischen Ausdrücken, bereitgestellt wird, das auch die maximale Länge der Affixe in einer bestimmten Sprache enthält. Außerdem wird durch die Regelschemata Wissen über Wortbildungsmechanismen (Präfixbildung, Suffixbildung, etc.) zur Verfügung gestellt. Die weitere Beschreibung dieser Methode orientiert sich an Brill (1993), da die Dissertation die umfassendste Beschreibung enthält. Das System, wie bei Brill (1993) beschrieben, besteht aus zwei Phasen (vgl. Abb. 2). Beide Teilkomponenten des Systems verwenden die Methode des transformationsbasierten fehlergesteuerten Lernens. In der ersten Phase werden kontext-freie Regeln gelernt, die das wahrscheinlichste Tag für jedes Wort ermitteln. Für das Training werden ein kleines bereits getaggtes und ein größeres ungetaggtes Korpus benötigt. Als Nebeneffekt der ersten Phase wird ein Lexikon angelegt, das für alle Wörter aus dem getaggten Korpus das wahrscheinlichste Tag enthält. In der zweiten Phase werden kontext-sensitive Regeln gelernt, die mittels Informationen aus dem Kontext eines Worts das Tag des Worts in dieser Situation bestimmen. Dazu wird ein kleines bereits getaggtes Korpus benutzt. Aufgrund des in Phase 1 erstellten Lexikons und der zwei Sets von Regeln können dann weitere beliebige Texte getaggt werden. Da nur zwei kleine getaggte Korpora verwendet werden, ist die Erstellung der Lernkorpora wesentlich weniger arbeitsintensiv als bei den Korpora für pro-

babilistische Systeme. So kann das System ohne großen Aufwand für verschiedene Textsorten oder Tagsets trainiert werden. Das in den Regelschemata enthaltene linguistische Wissen ist im Gegensatz zu den frühen regelbasierten Systemen nicht spezifisch, sondern sehr allgemein gehalten. Daraus ergibt sich der Vorteil, daß das Wissen dem Wissen eines Muttersprachlers entspricht, der nicht in der Lage ist, sämtliche Regelhaftigkeiten der Wortklassifizierung aufzuzählen, statt dessen aber die zur Klassifizierung notwendigen Kriterien zu benennen.

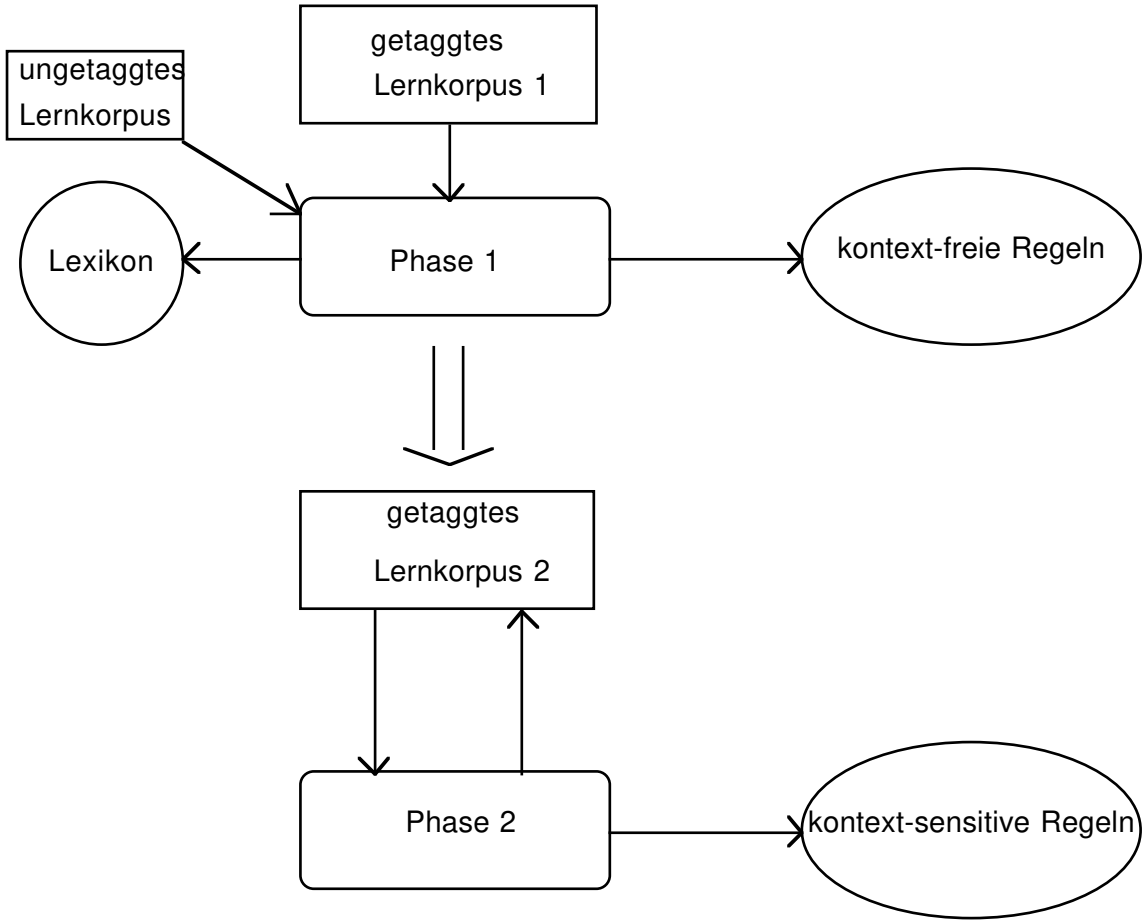

Abbildung 2

Die Methode des transformationsbasierten fehlergesteuerten Lernens ist nicht nur auf die automatische Wortklassifizierung beschränkt, sie wurde auch zur Disambiguierung von Präpositionalphrasenanbindungen und zum syntaktischen Parsen angewandt (Brill 1993). Einen ähnlichen Ansatz verfolgen Su et al. (1992), um ein automatisch bestimmbares

Maß zur Qualitätsbeurteilung von automatischen Übersetzungssystemen zu erhalten. Dazu wird ein vom System übersetzter Text mit dem gewünschten Endprodukt verglichen und die Anzahl der nötigen Nachbearbeitungsschritte ermittelt. Ist das System parametergesteuert, kann über dieses Maß in wiederholten Durchläufen die optimale Parameter-Einstellung gefunden werden.

2.1. Erstellung der kontext-freien Regeln

In der kontext-freien Phase soll aufgrund eines kleinen getaggten Lernkorpus das wahrscheinlichste Tag für jedes Wort aus dem Lernkorpus gefunden werden. Dies ließe sich ohne großen Aufwand mit einer Statistik erledigen. Da das Lernkorpus jedoch sehr klein ist, kann mit einer Statistik nicht sichergestellt werden, daß später im zu taggenden Korpus allen Wörtern ein Tag zugeordnet werden kann. Diese Aufgabe übernehmen in diesem System die kontext-freien Regeln, die mit Hilfe der Regelschemata erstellt werden. Regelschemata sind Anweisungen, ein Tag zu ändern, wenn ein bestimmtes Merkmal vorhanden ist. Das Ausgangs- und das Zieltag ebenso wie das Merkmal bestehen in dem Schema aus Variablen, die aufgefüllt werden müssen, um eine Regel zu erhalten. Das System geht dabei so vor, als ob das Lernkorpus nur mit Default-Tags annotiert wäre und erst getaggt werden müßte. Folglich ist jedes Wort im Korpus zu diesem Zeitpunkt mit zwei Tags annotiert: mit dem schon vorhandenen (richtigen) Tag und dem Tag, das das System per Default vergeben hat. Im nächsten Schritt werden mittels der Regelschemata alle Regelkandidaten ermittelt, die sich durch das Auffüllen der Variablen der Schemata ergeben. Das Ausgangstag ergibt sich aus dem vom System vergebenen Tag, das Zieltag aus dem bereits vorhandenen (richtigen) Tag. Die letzteren werden abgesehen von der Instanziierung der Zieltag-Variable nicht verwendet. Sie dienen allerdings im folgenden Schritt dazu, die Validität der Regelkandidaten zu bewerten: Durch einen Vergleich des vom Regelkandidaten vorgeschlagenen Tags mit dem richtigen Tag wird überprüft, ob der Kandidat eine positive oder negative Veränderung bewirkt. Die Ergebnisse der Überprüfung für alle Wörter, auf die die mögliche Regel zutrifft, ergeben zusammengenommen die Bewertung des Kandidaten. Der Kandidat mit der besten Bewertung ergibt die erste Regel, die dann auf das Korpus angewandt wird.

D.h. für alle Wörter, auf die die Regel zutrifft, wird das Default-Tag durch das Zieltag der Regel ersetzt.
Die Regelschemata haben die Form:

Ändere das Tag von X zu Y, wenn *Test* erfüllt ist.

oder

Ändere das Tag zu Y, wenn *Test* erfüllt ist.

Test kann nach Brill (1993, 63) folgende Form haben:

- Wenn das Löschen des Präfixes x, mit $|x| \leq 4$, ein Wort ergibt.
- Wenn die ersten k Buchstaben des Wortes x sind, mit $k \leq 4$.
- Wenn das Löschen des Suffixes x, mit $|x| \leq 4$, ein Wort ergibt.
- Wenn die letzten k Buchstaben des Worts x sind, mit $k \leq 4$.
- Wenn das Hinzufügen der Buchstabenfolge x, mit $|x| \leq 4$, als Suffix ein Wort ergibt.
- Wenn das Hinzufügen der Buchstabenfolge x, mit $|x| \leq 4$, als Präfix ein Wort ergibt.
- Wenn das Wort Z jemals unmittelbar rechts/links vom Wort erscheint.
- Wenn der Buchstabe W im Wort vorhanden ist.

Die Tests, die das Abtrennen oder Hinzufügen eines Suffixes bzw. Präfixes betreffen, sind erfüllt, wenn das neu entstandene Wort in dem großen, ungetaggten Korpus gefunden wird. Dieses Korpus dient nur der Überprüfung der Worthypothesen. Da nur ein ungetaggtes Korpus für die Tests benötigt wird, ist trotz der Größe keine weitere Bearbeitung nötig. Bei den Nachbarwörtern werden nur die 200 häufigsten Wörter betrachtet, um den Suchaufwand zu verringern.
Die Regelschemata enthalten keine linguistische Information (wenn man davon absieht, daß die maximale Länge von Affixen enthalten ist), sie geben lediglich die Stelle im Wort, die untersucht wird, sowie die Art der Untersuchung an.
Damit aus diesen Regelschemata Regeln werden, müssen die Variablen X, Y, W, Z, und x aufgefüllt werden. Dies geschieht folgendermaßen: Im Anfangszustand werden alle Wörter im Lernkorpus mit einem Default-Tag (z.B. [substantiv, singular, nominativ])[1]

[1] Die hier benutzten eckigen Klammern dienen zur Kennzeichnung eines Tags. Diese Notationskonvention wird auch im folgenden benutzt.

versehen. Danach werden auf jedes Wort alle Regelschemata angewandt. Gelingt der Test des Regelschemas, wird daraus ein Regelkandidat erstellt, der außer der Information aus dem Schema auch das Ausgangstag, das Zieltag, und das Affix oder das Wort enthält, für das der Test gelang. Das Zieltag ergibt sich aus dem im Lernkorpus vorgegebenen Tag.

Bsp. Ein möglicher Regelkandidat für das Schema

> Ändere das Tag von X zu Y, wenn der Buchstabe W im Wort vorhanden ist.

wäre:

> Ändere das Tag von [substantiv, singular, nominativ] zu [substantiv, plural, nominativ], wenn der Buchstabe 'ä' im Wort enthalten ist.

Damit wäre das Wort "Häuser" mit dem vorgegebenen Tag [substantiv, plural, nominativ] abgedeckt. Außer dem oben genannten Kandidaten werden jedoch auch für alle anderen Buchstaben des Wortes entsprechende Kandidaten erstellt.

Durch die oben beschriebene datengesteuerte Vorgehensweise wird sichergestellt, daß nur solche Regeln entstehen, für die das Affix auch wirklich im Lernkorpus vorkommt. D.h. für das folgende Regelschema z.B. werden nur für die Präfixe Regeln erstellt, die im Lernkorpus vorhanden sind:

> Ändere das Tag von X zu Y wenn die ersten k Buchstaben des Wortes x sind.

Dadurch verringert sich der Suchaufwand beträchtlich.
Brill (1993, 35) nennt diese Regeln auch Transformationen, da sie den Transformationen von Chomsky (1965) in Form und Wirkung ähnlich sind: Der Test stellt die Strukturbeschreibung der Transformation dar, die restliche Regel die Strukturveränderung. Eine Transformation bildet eine syntaktische Beschreibung der Tiefenstruktur auf eine syntaktische Beschreibung der Oberflächenstruktur ab, eine Regel von Brill (1993) bildet eine Merkmalsstruktur eines Worts auf eine andere ab. Da jedoch immer nur ein Tag verändert wird, sind die Regeln von Brill (1993) weniger komplex.
Wenn alle Regelkandidaten ermittelt sind, muß festgestellt werden, welcher Kandidat die größte positive Veränderung auslöst. Hierbei darf das Ergebnis jedoch nicht von der Auftretenshäufigkeit eines Worts abhängig sein. "To avoid this problem, success is measured on a per-type basis instead of a per-token basis. This means that the effect of a

transformation on the tagging of the word *the* will be given equal consideration as the effect on the word *upside-down*." (Brill 1993, 64).
Eine Regel R, die das wahrscheinlichste Tag von X zu Y ändert, wenn der Test Z gelingt, erhält folgende Wertung:

$$Score(R) = \sum_{W} \frac{Freq(W,Y) - Freq(W,X)}{Freq(W)} * Z(W) * Aktuell(W,X)$$

Dabei ist W ein Wort aus dem Lernkorpus, Freq(W) die Auftretenshäufigkeit von W und Freq(W, Y) die Anzahl , wie oft W im Lernkorpus mit Y getaggt ist,

$$Z(W) = \begin{cases} 1 & wenn \quad Test \quad Z \quad gelingt \\ 0 & sonst \end{cases}$$

und

$$Aktuell(W,X) = \begin{cases} 1 & wenn \quad W \quad mit \quad X \quad annotiert \quad ist \\ 0 & sonst \end{cases}$$

Diese Funktion mißt die Verbesserung, die eine Regel für jedes Wort bewirkt. Der Regelkandidat, der die beste Bewertung bekommt, wird dann in die Liste der Regeln aufgenommen und auf das Korpus angewandt. Danach wird die Prozedur mit dem so entstandenen Korpus als Anfangszustand durchgeführt (vgl. Abb. 3). So wird jedesmal der Regelkandidat ausgewählt, der die meisten positiven Veränderungen bewirkt. Die Prozedur wird so lange wiederholt, bis keine Regelkandidaten mehr gefunden werden, die eine positive Bewertung haben bzw. deren Bewertung einen Schwellenwert übersteigt.

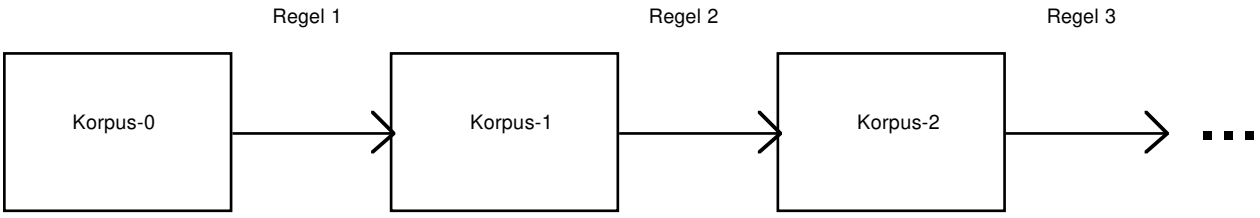

Abbildung 3

2.2. Erstellung der kontext-sensitiven Regeln

Nachdem für die Wörter aus dem ersten Lernkorpus das wahrscheinlichste Tag gefunden wurde, muß jetzt für jedes einzelne Wort im Kontext entschieden werden, welches Tag in diesem Kontext das richtige ist. Dazu wird wieder dieselbe Methode wie zur Erstellung der kontext-freien Regeln benutzt. Als Startzustand dient diesmal das zweite Lernkorpus, das wieder so behandelt wird, als ob es noch nicht getaggt sei. Die bereits vorhandenen (richtigen) Tags werden wie auch in der ersten Phase nur zur Bewertung der Regelkandidaten und zur Instanziierung der Zieltag-Variable herangezogen. Der Startzustand in dieser Phase ergibt sich durch die Anwendung der kontext-freien Regeln auf das zweite Lernkorpus.
In der kontext-sensitiven Phase wird das folgende Regelschema verwendet:

Ändere das Tag von X zu Y, wenn *Test* zutrifft.

Brill (1993) verwendet folgende *Tests*:

- Wenn das vorhergehende/nachfolgende Wort mit dem Tag Z annotiert ist.
- Wenn das vorhergehende Wort mit dem Tag W und das nachfolgende Wort mit dem Tag Z annotiert ist.
- Wenn die vorhergehenden/nachfolgenden zwei Wörter mit W und Z getaggt sind.
- Wenn eins der zwei vorhergehenden/nachfolgenden Wörter mit Z getaggt ist.
- Wenn eins der drei vorhergehenden/nachfolgenden Wörter mit Z getaggt ist.
- Wenn das Wort, das zwei weiter vorne/hinten steht, mit Z getaggt ist.

Diese Schemata werden wieder auf alle Wörter im Korpus angewandt, wodurch Regelkandidaten ermittelt werden. Im Gegensatz zu der ersten Phase wird die Bewertung hier aufgrund der tatsächlichen Anzahl der Wörter und nicht unabhängig davon vorgenommen:

$$Score(R) = \sum_{W} (Freq(W,Y) - Freq(W,X)) * Z(W) * Aktuell(W,X)$$

Mit Hilfe dieser Bewertungsfunktion wird wieder der beste Regelkandidat ausgesucht, in die Liste der kontext-sensitiven Regeln aufgenommen und auf das Korpus angewandt.

Ebenso wie in der ersten Phase wird dies so lange wiederholt, bis keine Regelkandidaten mehr mit positiver Bewertung bzw. mit einer Bewertung, die über dem Schwellenwert liegt, gefunden werden.

2.3. Bewertung des Systems

Brill (1993) sieht das transformationsbasierte fehlergesteuerte Taggen als Alternative zu den bis dahin weit verbreiteten probabilistischen Taggingsystemen. Obwohl sein Ansatz keine Verbesserung der Erfolgsquote erreicht, bietet er trotzdem einige Vorteile gegenüber den probabilistischen Ansätzen: Der wohl wichtigste Unterschied liegt in der geringen Menge an linguistischer Information, die der transformationsbasierte fehlergesteuerte Taggingansatz benötigt. Wo andere Systeme eine vordefinierte Matrix an Übergangswahrscheinlichkeiten erwarten, die u.U. noch manuell optimiert werden müssen, begnügt sich Brills (1993) System mit einer kleinen Anzahl von Regelschemata, aus denen es sich die nötige linguistische Information erarbeitet. Auch das getaggte Korpus, das zu diesem Lernprozeß benötigt wird, ist signifikant kleiner als diejenigen, mit denen die probabilistischen Systeme trainiert werden (vgl. 5.4.6.). Bedingt durch diese Faktoren läßt sich das Wortklassifizierungssystem ohne Aufwand auf verschiedenen Textsorten oder Tagsets trainieren. Auch die Umstellung auf eine andere Sprache bedeutet nur, daß zusätzliche Regelschemata angegeben werden müssen. Weitere Vorteile ergeben sich daraus, daß die Regeln nicht in Form von Zahlen sondern in formaler Schreibweise abgelegt werden, so daß sie später noch einsehbar und weiter verwendbar sind. Außerdem ist durch den regelbasierten Lernansatz auch die systematische Behandlung von unbekannten Wörtern möglich.
Probleme könnten sich daraus ergeben, daß Brills (1993) System direkt mit den flektierten Wortformen arbeitet und folglich keine Lemmatisierung durchführt. Wird das System als Teilkomponente in einem größerem NLP-System verwendet, muß die Lemmatisierung spätestens vor der Semantikkomponente durchgeführt werden, da sonst die Rückführung der Wortformen auf die Sememe schwierig wird. Es wäre jedoch interessant festzustellen, ob sich die transformationsbasierte fehlergesteuerte Methode auch bei der Lemmatisierung einsetzen ließe.

Weitere Schwachpunkte liegen in der Beschränkung des Systems auf lokale Phänomene und in der Auswahl der Regeln aus der Menge der Kandidaten, da bei Sprachen mit freier Wortstellung der lokale Kontext eines Wortes nicht in dem Maße wie bei einer Sprache mit fester Wortstellung determiniert werden kann. Ebenso wie die probabilistischen Systeme betrachtet die transformationsbasierte fehlergesteuerte Wortklassifizierung nur lokale Phänomene. Zwar ist der lokale Kontext mit 3 Wörtern zu jeder Seite des zu bearbeitenden Wortes größer als bei probabilistischen Systemen, aber auch damit lassen sich nicht-lokale Phänomene nicht erfassen, wie (4) zeigt:

(4) Er sah das Auto um die Ecke biegen, mit dem er gestern zusammengestoßen war.

Hier kann für das Wort 'dem' das Genus nur entschieden werden, wenn die Beziehung zu 'Auto' hergestellt wird.
Roche und Schabes (1995) dokumentieren weiterhin, daß Brills (1992) System zwar in bezug auf Algorithmuskomplexität und Speicherkomplexität sehr effizient arbeitet, daß das Zeitverhalten aber durchaus noch verbessert werden kann. Sie führen das ineffiziente Verhalten auf zwei Gründe zurück: "The first reason for inefficiency is the fact that an individual rule is compared at each token of the input, regardless of the fact that some of the current tokens may have been previously examined when matching the same rule at a previous position." (Roche, Schabes 1995, 230). Wenn also in der Sequenz CDCCA nach der Teilsequenz CCA gesucht wird, muß das D zweimal betrachtet werden, einmal wird es mit dem zweiten Symbol der Teilsequenz verglichen und im nächsten Schritt mit dem ersten. "The second reason for inefficiency is the potential interaction between rules." (Roche, Schabes 1995, 230). So kann die Änderung eines Tags, die eine Regel bei einem Wort bewirkt, durch eine nachfolgende Regel aufgehoben werden. Eine der ersten Regeln, die Brills (1993) System in der kontext-freien Phase lernt, ist diejenige, daß das Tag von [substantiv, singular] auf [substantiv, plural] geändert wird, wenn das betreffende Wort auf -s endet. In einem späteren Schritt wird das Tag bei Wörtern, die auf -ss enden, wieder auf [substantiv, singular] zurückgesetzt.
Roche und Schabes (1995) schlagen daher vor, die kontext-sensitive Regelmenge[2] als einen deterministischen endlichen Transducer, also einem endlichen Automaten mit

[2] Sie beziehen sich auf die 92er Version, in der noch keine kontext-freie Komponente vorgesehen ist, vgl. auch 3.4.

einem Eingabe- und einem Ausgabeband, zu implementieren, wodurch ein lineares Zeitverhalten, unabhängig von der Anzahl der Regeln, erreicht werden kann. Das Eingabeband dieses Transducers entspricht in diesem Fall der Tagsequenz vor Anwendung der Regel, das Ausgabeband der Tagsequenz danach. Dieser endliche Transducer wird in vier Schritten erstellt: Im ersten Schritt wird jede Regel in einen endlichen Transducer umgewandelt. Im zweiten Schritt werden die Transducer in global operierende Transducer überführt; diese arbeiten nicht nur die Teilsequenz des Regelinhalts ab, sondern sind in der Lage, das ganze Wort zu verarbeiten. Im dritten Schritt werden alle Transducer zu einem einzigen kombiniert, und im vierten Schritt wird dieser Transducer in einem deterministischen überführt. Dies ist nicht für jeden nicht-deterministischen Transducer möglich, doch Roche und Schabes (1995) führen den formalen Beweis, das Brills (1992) Tagger in einen deterministischen Transducer überführt werden kann.
Der hier beschriebene Transducer erreicht dieselbe Erfolgsquote wie Brills (1992) System, besitzt dieselbe Kompaktheit, erreicht jedoch die fast zehnfache Geschwindigkeit anderer Systeme, wie Tests mit Brills (1992) System und Churchs (1988) probabilistischem Tagger (vgl. 5.2.3.) gezeigt haben. Es muß jedoch berücksichtigt werden, daß diese Methode nicht in der Lernphase, sondern erst bei der Anwendung der bereits gelernten Regeln verwendet werden kann. Trotzdem ist sie nützlich bei der Verwendung des Tags in NLP-Systemen, wobei die Lernphase im voraus erledigt wird, und dann die gelernten Regeln in Form eines endlichen Transducers in das NLP-System integriert werden.

2.4. Unterschiede zum Vorgänger- bzw. Nachfolger-Modell

Das erste Modell (Brill 1992) war eine wesentlich einfachere Version des oben vorgestellten Systems (Brill 1993). Die wichtigste Änderung betrifft die erste Phase: Im 92er Modell wird in der ersten Phase zuerst das wahrscheinlichste Tag für jedes Wort durch eine Statistik, die auf einem großen Korpus basiert, ermittelt. Bei Wörtern, die im Lernkorpus nicht enthalten sind, versucht eine Prozedur aufgrund der Endung des Worts auf die Wortart zu schließen; sonst wird angenommen, daß es sich um ein Substantiv handelt. Auf die so getaggten Wörter werden dann drei Regelschemata angewandt, die den Wortkontext (also nicht den Tagkontext) des betreffenden Worts, seine lexikalischen

Eigenschaften und die lexikalischen Eigenschaften von Wörtern in der Umgebung des Worts als Kriterien benutzen. Hier wird auch schon in der ersten Phase auf den Kontext des Worts zugegriffen, aber im Gegensatz zu der zweiten Phase sind es nicht die Tags, sondern die Wörter selbst und ihre lexikalischen Eigenschaften, die betrachtet werden.
Die Unterscheidung zwischen den zwei Phasen besteht bei diesem Modell folglich nicht zwischen kontext-frei und kontext-sensitiv, sondern zwischen Wortmerkmalen und Tagmerkmalen. Auch in der zweiten Phase gibt es einige Unterschiede zum 93er Modell: Zum einen gibt es zwei Regelschemata mehr, die eine mögliche Großschreibung des Worts oder seines Vorgängers betrachten. Außerdem werden die Schemata nicht auf alle Wörter angewandt, statt dessen wird eine Liste erstellt mit Einträgen der Form <X, Y, n>, die die Anzahl n angeben, wie oft ein Wort das Tag X erhalten hat, obwohl es im Lernkorpus mit Y annotiert war. Dann wird für jeden dieser Einträge entschieden, "which instantiation of a template from the prespecified set of patch templates results in the greatest error reduction." (Brill 1992, 113). Danach wird wie im 93er Modell der beste Kandidat ausgewählt und angewandt.
Die Neuerungen im 94er Modell betreffen die zweite Phase: Im 93er Modell gab es keine Möglichkeit, Abhängigkeiten zwischen zwei Wörtern darzustellen, da die Regelschemata, die im 92er Modell dafür zuständig waren, aufgegeben wurden. Diese Schwachstelle wurde im 94er Modell behoben. Diesmal wurden die nötigen Regelschemata jedoch nicht in der ersten, sondern in der zweiten Phase verwendet, außerdem wurden auch noch Abhängigkeiten zwischen einem Tag und einem Wort berücksichtigt. Dafür wurden folgende Regelschemata zusätzlich eingeführt:

Ändere das Tag von X zu Y:

- Wenn das vorhergehende/nachfolgende Wort W ist.
- Wenn das Wort, das zwei Positionen weiter vorne/hinten steht, W ist.
- Wenn eins der zwei vorhergehenden/nachfolgenden Wörter W ist.
- Wenn das aktuelle Wort W und das vorhergehende/nachfolgende Wort V ist.
- Wenn das aktuelle Wort W ist und das vorhergehende/nachfolgende Wort mit T getaggt ist.

Dadurch ist das System in der Lage, auch den lexikalischen Kontext zu berücksichtigen, was ihm einen Vorteil über die probabilistischen Systeme gibt, die aufgrund ihres Aufbaus nicht dementsprechend modifiziert werden können.
Als Beispiele dafür, daß diese Schemata auch wirkungsvoll eingesetzt werden, gibt Brill (1994, 724) folgende Regeln an, die beim automatischen Taggen des Penn Treebank Korpus (vgl. Marcus et al. 1993) gelernt wurden:

° Ändere das Tag von [präposition] zu [adverb], wenn das Wort zwei Positionen weiter rechts "as" ist.

° Ändere das Tag von [verb, nicht-dritte Person, singular, präsens] zu [verb, grundform], wenn eins der zwei vorhergehenden Wörter "n't" ist. (Anm.: Die Kurzform von "not" wird in diesem Korpus als eigenständiges Wort betrachtet.)

2.5. Änderungen des Systems zur Anpassung auf das Taggen eines deutschsprachigen Korpus

Als eine der interessanten Eigenschaften seines Systems nennt Brill u.a. die Tatsache, daß "There is very little linguistic knowledge, and no language-specific knowledge built into the system." (1993, 12). Dies relativiert er allerdings später, als er zu den kontext-freien Regeln anmerkt: "The templates could be extended to handle languages with infixes by allowing a transformation such as: *Change a tag if the character string X appears internal to a word*." (Brill 1993, 63). Hierbei wird deutlich, daß die Wortbildungsmechanismen einzelner Sprachen durchaus berücksichtigt werden sollten. Dies läßt sich jedoch dadurch umgehen, daß alle bekannten Wortbildungsmechanismen in Regelschemata implementiert werden. Ist ein Schema in einer Sprache nicht anwendbar, ergibt sich als Konsequenz, daß zu diesem Schema keine Regel gelernt wird. Brill (1993, 133) erklärt hierzu: "The only possible adverse affect [...] is that it could result in finding a local maximum during learning which blocks the application of other useful transformations, thereby resulting in an overall degradation in performance." Das kann jedoch auch problemlos bemerkt und behoben werden.
Aus diesen Systemeigenschaften ergibt sich, daß kaum Änderungen im System vorgenommen werden müssen. Zum einen muß die Affixlänge von 4 vergrößert werden, da

das Deutsche auch längere Präfixe und Suffixe kennt (z.B. 'zusammen-' in 'zusammenkommen'). Außerdem muß auch, wie von Brill vorgeschlagen (s.o.), die Wortbildung durch Infigierung berücksichtigt werden. Schließlich wird in das System noch ein Regelschema integriert, mit dem ein weiterer Wortbildungsmechanismus im Deutschen, die Ablautbildung, behandelt werden kann. Dazu muß dem System das Konzept des Vokals zur Verfügung gestellt werden. Das Regelschema sieht folgendermaßen aus:

> Ändere das Tag von X zu Y, wenn durch das Ersetzen eines Vokals im Wort durch den Vokal x ein Wort entsteht.

Inwieweit sich diese Veränderungen positiv auf das Ergebnis auswirken, soll dadurch geklärt werden, daß das Korpus einmal mit der Original-Version von Brill (1993) und einmal mit der an die deutsche Sprache angepaßten Version getestet wird (Ergebnisse vgl. Kap. 4).

3. Ergebnisse aus der Anwendung des transformationsbasierten fehlergesteuerten Tagging auf die deutsche Sprache

Die in diesem Kapitel beschriebenen Tests wurden mit Texten aus einem Korpus mit Artikeln der 'taz' (Berliner Tageszeitung) durchgeführt. Die zwei Lernkorpora mit jeweils 1.000 Sätzen stammen aus Artikeln vom Januar 1990, die 3.000 Sätze des Testkorpus aus Artikeln vom Dezember 1990 und die 48.000 Sätze des ungetaggten Korpus schließlich aus Artikeln vom Juli 1991.
Die zwei Lernkorpora und das Testkorpus wurden in zwei Stufen getaggt: Zuerst wurde jedes Wort in dem lexikalischen System Celex© nachgeschlagen und mit sämtlichen, dort vermerkten Tags annotiert. Hierbei wurde ein Tagset von 170 Tags verwendet (siehe Anhang A). Im zweiten Schritt wurde dann jedes Wort manuell disambiguiert. Außerdem wurde die satzinitiale Kapitalisierung rückgängig gemacht. Im ungetaggten Korpus mußten nur die bei der Konvertierung der Daten übersehenen Formatierungszeichen entfernt werden.

3.1. Die durchgeführten Tests

Der geplante Test des Systems, wie in Brill (1993) beschrieben, wurde mit den in 2.5. beschriebenen Anpassungen für die deutsche Sprache implementiert (vgl. Anhang B). Der Test dieses Systems erwies sich jedoch als faktisch unmöglich, da unverhältnismäßig hohe CPU-Zeiten bei der Bearbeitung des kontext-freien Teils des Systems eine Terminierung des Prozesses nicht zuließen, zumal auch verschiedene Male Rechnerabschaltungen bzw. -fehlfunktionen den Neustart des Prozesses nötig machten. Auch eine Zwischenspeicherung der Ergebnisse führte nicht zu einer Verbesserung der Situation, da erst nach zwei Tagen CPU-Zeit die Hälfte der Wörter abgearbeitet war, die zum Lernen der ersten kontext-freien Regel bearbeitet werden müssen. Die mangelnde Geschwindigkeit ist durch die große Anzahl der Regelkandidaten bedingt, die auf den - zumindest im Vergleich zum Englischen - eher synthetischen Sprachbau zurückgeführt werden kann. Statt dessen wurden zwei weitere Tests durchgeführt, die auf den kontext-freien Teil des Systems nach Brill (1993) verzichteten: Für beide Tests wurde, in Anlehnung an Brills

92er System, der kontext-sensitive Teil des 93er Systems mit einer Initialisierungskomponente kombiniert, die für jedes Wort das Tag ermittelt, mit dem es im Lernkorpus am häufigsten annotiert ist. Im Gegensatz zu Brills 92er System wurden also zwei Regelschemata weniger verwendet (vgl. 2.4). Die durch die Initialisierung gewonnene Information wurde in einem Lexikon gespeichert, so daß sie für die spätere Wortklassifizierung des Testkorpus verwendet werden kann. Beim ersten Test bestand das Lernkorpus aus 1.000 Sätzen, beim zweiten (Vergleichs-)Test wurde die Zahl der Sätze auf 2.000 erhöht. Durch die Ermittlung der ersten 300 Regeln sollte festgestellt werden, ob sich die Performanz der kontext-sensitiven Regelfindung durch die Vergrößerung des Lernkorpus verbessern läßt.

3.2. Ergebnisse aus dem Test mit 1.000 Sätzen

Beim ersten Test wurde das zweite Lernkorpus mit 1.000 Sätzen verwendet. Basierend auf diesem Korpus lernte das System 700 Regeln (vgl. Anhang C). Die ersten 20 Regeln sind in Tabelle 1 aufgeführt.

Ausgangs-tag	Zieltag	Nachbar 1	Nachbar 2	Wo?	Wie?	Anzahl 1	Anzahl 2
Swgd:R	Smn:R	nS:M		hinten	oder	2	1
Pmwsd:R	Sma:R	aS:M		hinten	oder	3	1
Swna:R	Pmwsna:R	nP:M		hinten	oder	3	1
X:P	X:D	X:I		hinten	genau	1	1
Ssna:R	Smn:R	nS:M		hinten	oder	2	1
Swna:R	Pmwsna:R	nP:F		hinten	oder	3	1
nS:M	dS:M	X:PR		vorne	oder	2	1
Smg:R	Ssg:R	gS:N		hinten	oder	3	1
Ssna:R	Ssna:O	3SIE:V		hinten	genau	1	1
Swgd:R	Pmwsg:R	gP:M		hinten	oder	2	1
Swna:R	Pmwsna:R	nP:N		hinten	oder	3	1
Swna:R	Pmwsna:R	aP:N		hinten	oder	2	1
nS:M	aS:M	Pmwsd:R		vorne	oder	3	1
Ssd:R	Smd:R	dS:M		hinten	oder	3	1
Pmwsd:R	Sma:R	aS:M		hinten	oder	2	1
Smd:R	Ssd:R	dS:N		hinten	oder	2	1
Swgd:R	Smn:O	X:I		vorne	genau	1	1
Swna:R	Pmwsna:R	o5:A		hinten	oder	2	1
Swgd:R	Pmwsg:R	gP:F		hinten	oder	2	1
Swna:R	Pmwsna:R	aP:M		hinten	oder	3	1

Tabelle 1

Die Spalte 'Ausgangstag' gibt hier das zu ersetzende Tag an, 'Zieltag' das neue Tag und 'Nachbar 1' und 'Nachbar 2' den Kontext, der vorhanden sein muß, damit die Regel Anwendung findet. 'Wo?' gibt an, ob der Kontext vor dem Wort, dahinter oder davor und dahinter berücksichtigt werden soll, 'Anzahl 1' definiert die Größe des Kontextes, der berücksichtigt werden soll und 'Anzahl 2' die Anzahl der Wörter, die aus diesem Kontext betrachtet werden. 'Wie?' setzt schließlich fest, ob ein bestimmtes Wort aus dem Kontext betrachtet wird (= 'genau'), oder ein beliebiges (= 'oder').
Die erste Regel besagt z.B., daß das Tag [Swgd:R] durch das Tag [Smn:R] ersetzt werden soll, wenn eins (= Anzahl 2) der beiden (= Anzahl 1) nachfolgenden (= Wo?) Wörter mit [nS:M] (= Nachbar 1) annotiert ist.
Im Vergleich mit den ersten zwanzig Regeln, die Brills 93er System für das Englische gelernt hat, (vgl. Brill 1993, 87) fällt auf, daß die Regeln für das Deutsche am Anfang vor allem Änderungen bei Artikeln bewirken. Von den ersten 20 Regeln beschäftigen sich allein 16 mit Artikeln. Im Gegensatz dazu wurden in Brills 93er System zu Anfang vor allem Verb-Regeln gelernt.
Im folgenden Schritt wurden die bisher gelernten 700 Regeln auf das Testkorpus angewandt. Zuerst wurden die Wörter des Testkorpus eingelesen; falls sie im Lexikon enthalten waren, wurden sie mit dem dort enthaltenen Tag annotiert, sonst erhielten sie das Default-Tag. Von den insgesamt 46.201 Wörtern waren 13.436 nicht im Lexikon enthalten, was einen Prozentsatz von 29,1% unbekannter Wörter ergibt. Damit liegt er signifikant über den von Brills (1993) System für den englischen Korpus ermittelten 18,7%.
Nach der Initialisierung waren 41,0% der Wörter, d.h. 18.937 Wörter, falsch getaggt. Brill (1992, 113), berichtet dagegen von einer Fehlerquote von 7.9%.
Nachdem die Initialisierung erfolgt war, wurden die 700 kontext-sensitiven Regeln auf das Testkorpus angewandt. Abb. 4 zeigt den Grad der Korrektheit in Abhängigkeit von der Anzahl der bereits angewendeten Regeln.

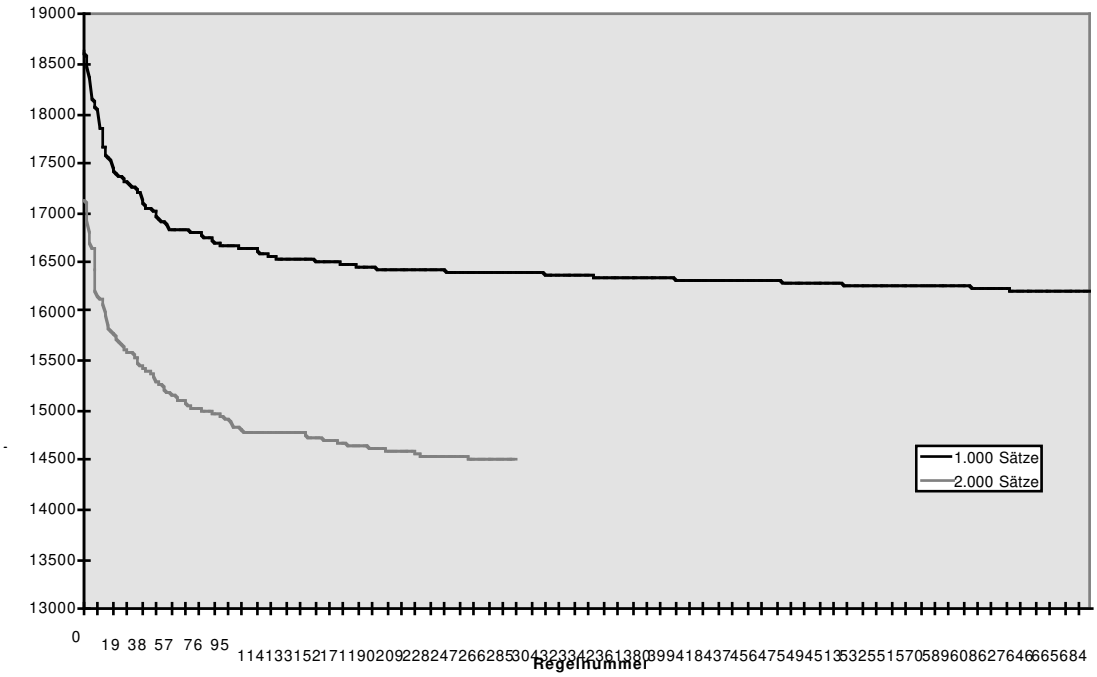

Abbildung 4

Die Zahl der falsch getaggten Wörter senkt sich durch die Anwendung der Regeln von 18.937 auf 16.174. Dies ergibt insgesamt eine Fehlerquote von 35,0%. Von den 16.174 falsch getaggten Wörtern waren 3.930 aus dem Lernkorpus bekannt, 12.244 Wörter dagegen waren im Lernkorpus nicht enthalten. Dies ergibt einen Korrektheitsgrad von 88,0% für bekannte Wörter und den Korrektheitsgrad von 8,9% für unbekannte Wörter. Die signifikantesten Verbesserungen ergeben sich jedoch bei der Anwendung der ersten 90 Regeln. Mit diesen Regeln wird eine Fehlerquote von 36,2% erreicht, mit Hilfe der restlichen 610 Regeln senkt sich die Quote lediglich um weitere 1,2%.
Aus diesen Ergebnissen wird deutlich, daß für die deutsche Sprache die Verwendung der kontext-freien Phase unumgänglich ist, da in dieser Phase auf lexikalische Information getestet wird, was bei der Klassifizierung von unbekannten Wörtern bedeutende Verbesserung verspricht. Durch die kontext-freie Phase sollte für unbekannte Wörter, wenn schon nicht das richtige Tag, so doch wenigstens ein Tag bestimmt werden, bei dem zumindest die richtige Wortklasse besitzt. Dies kann von der kontext-sensitiven Phase nicht geleistet werden, da dort eher lokale syntaktische Phänomene zum Lernen benutzt werden.

3.3. Unterschiede bei 2.000 Sätzen

Im zweiten Test wurde zum Lernen ein Lernkorpus mit 2.000 getaggten Sätzen verwendet. Der Lernvorgang wurde abgebrochen, nachdem 300 Regeln gelernt waren, da diese Regeln nur zum Vergleich mit den im ersten Test gelernten dienen sollten. Diese Regeln wurden anschließend auf das Testkorpus angewandt, das auch schon im ersten Test benutzt wurde.

Ausgangs-tag	Zieltag	Nachbar 1	Nachbar 2	Wo?	Wie?	Anzahl 1	Anzahl 2
Swgd:R	Smn:R	nS:M		hinten	oder	2	1
Pmwsd:R	Sma:R	aS:M		hinten	oder	3	1
nS:M	dS:M	X:PR		vorne	oder	2	1
X:P	X:D	X:I		hinten	genau	1	1
Ssna:R	Smn:R	nS:M		hinten	oder	2	1
Swna:R	Pmwsna:R	nP:M		hinten	oder	3	1
Swna:R	Pmwsna:R	nP:F		hinten	oder	3	1
Pmwsd:R	Sma:R	nS:M		hinten	oder	3	1
nS:M	aS:M	Sma:R		vorne	oder	3	1
Ssna:R	Ssna:O	3SIE:V		hinten	genau	1	1
Smd:R	Ssd:R	dS:N		hinten	oder	2	1
Smg:R	Ssg:R	gS:N		hinten	oder	3	1
Swgd:R	Pmwsg:R	gP:M		hinten	oder	3	1
Swna:R	Pmwsna:R	o5:A		hinten	oder	2	1
Swgd:R	Smn:O	X:I		vorne	genau	1	1
nS:N	dS:N	X:PR		vorne	oder	3	1
nS:M	dS:M	Smd:R		vorne	oder	2	1
Swna:R	Pmwsna:R	nP:N		hinten	oder	2	1
Swna:R	Pmwsna:R	aP:N		hinten	oder	2	1
nS:F	dS:F	X:P	Swgd:R	vorne	genau	2	2

Tabelle 2

In Tabelle 2 sind wiederum die ersten 20 gelernten Regeln aufgeführt. Es zeigt sich, daß z.T. Regeln gelernt werden, die im ersten Test entweder gar nicht oder erst sehr viel später gelernt wurden. Betrachtet man jedoch den Kurvenverlauf in Abb. 4, zeigt sich, daß die Unterschiede in den Regeln bzw. in der Reihenfolge der Regeln keine erkennbare Veränderung ergeben. Mit einem Lernkorpus von 1.000 Sätzen wird nach 300 Regeln eine Verbesserung der Fehlerquote um 4.7% erreicht, bei 2.000 Sätzen erzielt dieselbe

Anzahl von Regeln eine Verbesserung von 6,4%. Der größte Unterschied ergibt sich aus der Vergrößerung des Lexikons: Während im ersten Test die unbekannten Wörter einen prozentualen Anteil von 29,1% ausmachten, waren es im zweiten Test nur noch 24,0%. Als Folge daraus ergab sich beim zweiten Test nach der Initialisierung des Testkorpus eine Fehlerquote von 37,8%. Eine derartige Verbesserung läßt sich jedoch sinnvoller durch die Ergänzung des Systems mit einem systematisch erstellten Lexikon erreichen.

3.4. Eine Auswahl fehlerhaft getaggter Sätze

Hier soll eine zufällige Auswahl fehlerhaft getaggter Sätze gezeigt werden. Sie wurde der Ausgabedatei des ersten Tests entnommen. Korrekt getaggte Wörter sind hier nur mit einem Tag annotiert, falsch getaggte werden hervorgehoben in folgender Form gezeigt: Wort|richtiges Tag|System-Tag.

1. noch|X:D **werden|13PIE:V|i:V Kontaktadressen|nP:F|nS:M** in|X:P der|Swgd:R **UdSSR|dS:oF|nS:M** gesucht|pA:V ,|X:I denen|Pmwsd:O **die|Pmwsna:R|Swna:R InitiatorInnen|nP:MF|nS:M** die|Pmwsna:R guten|o5:A Gaben|aP:F **pers\"onlich|o0:A|X:D** \"uberreichen|i:V wollen|13PIE:V .|X:I

2. von|X:P den|Pmwsd:R **weltweit|X:D|nS:M gesch\"atzten|o5:A|nS:M** acht|0:U bis|X:C zehn|0:U Millionen|dP:F **HIV-Infizierten|dP:M|nS:M** sind|13PIE:V rund|X:D ein|Ssna:R Drittel|nS:N ,|X:I von|X:P den|Pmwsna:R **1.3|0:U|nS:M** Millionen|dP:F **Erkrankten|dP:M|gP:M** etwa|X:D **300.000|0:U|nS:M** Frauen|nP:F .|X:I

3. ist|3SIE:V es|n:O wirklich|X:D **realistisch|o0:A|nS:M** ,|X:I in|X:P **Ru\"sland|dS:oN|nS:M** heute|X:D von|X:P Hunger|dS:M zu|X:D sprechen|i:V ?|X:I

4. dies|Ssna:O **geschah|13SIA:V|nS:M per|X:P|nS:M** Hardth\"ohen-Erla\"s|aS:M zun\"achst|X:D \"uber|X:P **die|Pmwsna:R|Swna:R Kreiswehrersatz\"amter|aP:N|nS:M** (|X:I **KWEAs|aP:aN|nS:M**)|X:I in|X:P **Westdeutschland|dS:oN|nS:M** ,|X:I weil|X:C **die|Pmwsna:R|Swna:R Wehr-**

beh\"orden|nP:F|nS:M in|X:P Berlin|dS:oN ,|X:I zwei|0:U **KWEAs|nP:aN|nS:M** in|X:P den|Pmwsd:**R \"ostlichen|o5:A|nS:M Stadtteilen|dP:M|nS:M Lichtenberg|dS:oN|nS:M** und|X:C Pankow|dS:oN ,|X:I sich|a:O noch|X:D im|X:PR Aufbau|dS:M **befinden|13PIE:V|nS:M** .|X:I

5. da|X:D werde|13SKE:V zwar|X:D auch|X:C **gestritten|pA:V|nS:M** ,|X:I aber|X:C wenn|X:C es|n:O **einem|d:O|Ssd:R dreckig|X:D|o0:A** geht|3SIE:V ,|X:I kann|3SIE:V man|0:O sich|a:O auf|X:P die|Swna:R Familie|aS:F **verlassen|i:V|nS:M** .|X:I

6. vor|X:P dem|Smd:R Treffen|dS:N hatte|13SIA:**V Tutu|nS:pM|nS:M gewarnt|pA:V|nS:M** ,|X:I da\"s|X:C Politiker|nP:M ,|X:I die|Pmwsna:O eine|Swna:R **Beteiligung|aS:F|nS:F verweigern|13PIE:V|nS:M** ,|X:I als|X:C **Feinde|nP:M|nS:M** der|Swgd:R Befreiung|gS:F **verurteilt|pA:V|nS:M** werden|i:V w\"urden|13PKA:V .|X:I

7. **das|Ssna:O|Ssna:R bedeutet|3SIE:V|pA:V** aber|X:D ein|Ssna:R **\"uberaus|X:D|nS:M ungesundes|o8:A|nS:M Wirtschaften|aS:N|nS:M** .|X:I

8. weil|X:C **prominente|o4:A|nS:M PDS-Besucher|nP:M|nS:M** nicht|X:D mehr|X:D \"uberall|X:D **willkommen|o0:A|nS:M** sind|13PIE:V ,|X:I **bedankt|3SIE:V|nS:M** sich|a:O **Modrow|nS:pM|gS:pM** bei|X:P der|Swgd:R **Belegschaft|dS:F|nS:M** ,|X:I da\"s|X:C ich|n:O bei|X:P **euch|d:O|nS:M** sein|i:V **durfte|13SIA:V|nS:M** .|X:I

9. alle|4:O **H\"andler|nP:M|nS:M** h\"atten|13PKA:V ein|Ssna:R **Strafverfahren|aS:N|nS:M** wegen|X:P **Versto\"ses|gS:M|nS:M** gegen|X:P das|Ssna:R **Urheberrecht|aS:N|nS:M** zu|X:D erwarten|i:V .|X:I

10. **Stone|nS:pM|nS:M** ,|X:I der|Smn:O zehn|0:U Jahre|aP:N lang|X:D mit|X:P seinem|7:O **Drehbuch|dS:N|nS:M hausieren|i:V|nS:M** ging|13SIA:V ,|X:I ist|3SIE:V selbst|X:O Vietnam-Veteran|nS:M .|X:I

4. Transformationsbasiertes fehlergesteuertes Tagging im Vergleich zu anderen Ansätzen[3]

In diesem Kapitel sollen andere Ansätze im Tagging vorgestellt und mit dem System von Brill (1993) verglichen werden. Der Vergleich wird in 4.4. anhand der folgenden acht Kriterien vorgenommen:

- Welche Erfolgsquote hat das System?
- Ist das System ein vollständiges Taggingsystem?
- Können unbekannte Wörter systematisch behandelt werden?
- Wird ein Lexikon verwendet?
- Welche Informationen müssen dem System bereitgestellt werden?
- Ist ein bereits getaggtes Textstück nötig, und wenn ja, wie groß muß es sein?
- Ist die im System gewonnene linguistische Information einsehbar?
- Wird eine Lemmatisierung durchgeführt?

Soweit nicht anders vermerkt, sind diese Systeme zur Wortklassifizierung der englischen Sprache erstellt worden.

[3] In diesem Kapitel wurden die verschiedenen Tagbenennungen der einzelnen Autoren beibehalten. Zur besseren Übersicht wurden die Tags jedoch einheitlich in eckige Klammern gesetzt.

4.1. Regelbasierte Ansätze

Wie schon in Kapitel 1 beschrieben, waren die ersten Tagging-Systeme regelbasiert. An dieser Stelle sollen die Systeme von Klein und Simmons (1963) sowie von Greene und Rubin (1971) vorgestellt werden, die beide Kontextregeln (context frame rules)[4] verwenden. Die Systeme von Brodda (1982) und von Akkerman et al. (1987) dagegen sind weniger als automatische Taggingsysteme konzipiert, sondern eher als interaktive.

4.1.1. Computational Grammar Coder

Das System

Das erste Taggingsystem, das in der Literatur beschrieben wird, ist der Computational Grammar Coder (CGC) von Klein und Simmons (1963). Motiviert war dieser Ansatz durch die Suche nach einer Alternative zu den großen Lexika, die bis dahin ausschließlich verwendet wurden, und die zu dieser Zeit noch wegen der begrenzten Speicherkapazität der damaligen Rechner auf Magnetbändern gespeichert werden mußten. Das Ziel war daher, die linguistische Information in Regeln zu verarbeiten, um einerseits "the labor of constructing a very large dictionary" (Klein, Simmons 1963, 335) zu vermeiden und andererseits eine Möglichkeit zur Bearbeitung unbekannter Wörter zu haben. CGC wurde in einem System zum Beantworten von Fragen, namens proto-synthex (Simmons et al. 1962) eingesetzt.
Klein und Simmons (1963) verwenden in ihrem System 30 verschiedene Tags, wobei Nomen, Verben und Adjektive nicht weiter unterteilt werden; Nomen und Verben erhalten allerdings zusätzlich noch eine Numerusangabe. Im CGC durchlaufen die Wörter bis zu sechs voneinander unabhängige Tests. Der erste Schritt besteht aus einer Lexikonsuche. CGC verwendet mehrere kleine Lexika, die zum einen ca. 400 Funktionswörter und eine Liste von Interpunktionszeichen enthalten, außerdem ca. 1.500 "nouns, verbs and adjectives that are exceptions to the computational rules ..." (Klein, Simmons 1963, 339). Im zweiten Test werden großgeschriebene, nicht-satzinitiale Wörter mit den Tags

[4] Kontextregeln sind Regeln, die für einen Kontext nicht-ambiger Tags die möglichen Tagsequenzen angeben.

[noun] und [adjective] versehen. Test 3 und 4 sind Suffixtests: Im dritten Test wird auf Singular- bzw. Pluralendungen und auf Partizipendungen (-ing, -ed) geprüft. Die nicht flektierten, d.h. lemmatisierten Wörter werden dann zu Test 4 weitergegeben, wo auf Derivationsendungen bis zu einer Länge von 5 Buchstaben geprüft wird.

Bsp. Das Wort "nationalities" wird in Test 3 als Plural erkannt, als "nationality" an Test 4 weitergegeben und dort wegen der Endung -ity als [noun] getaggt.

Im letzten Test schließlich werden Kontextregeln verwendet, um die bis zu diesem Zeitpunkt ambig gebliebene Wörter zu disambiguieren. Kontextregeln werden angewandt, wenn das System eine Sequenz von einem oder mehreren ambigen Tags entdeckt, die durch eindeutig getaggte Wörter begrenzt sind. Die Regeln geben sämtliche Tagsequenzen an, die zwischen den beiden eindeutigen Tags im Englischen möglich sind. Diese Regelinformation wird mittels 'pattern matching' dazu verwendet, die bereits in den vorhergehenden Tests gefundenen Alternativen einzuschränken.

Bsp. Für die Tagsequenz $[article]\begin{bmatrix} adjective \\ verb \end{bmatrix}\begin{bmatrix} noun \\ adjective \end{bmatrix}[verb]$ gibt es folgende Kontextregeln, d.h. zwischen [article] und [verb] sind folgende Kombinationen möglich: [adjective] [noun]; [noun] [adverb]; [noun] [noun]. Durch das pattern matching zwischen Tagsequenz und Regeln bleibt nur die Kombination [adjective] [noun] übrig.

Es wurden allerdings nur Tagsequenzen mit höchstens drei ambigen Tags berücksichtigt. Insgesamt gibt es rund 500 solcher Regeln.

Kritik

Die oben erwähnten Kontextregeln wurden "empirically derived by hand analysis of a sample of Golden Book Encyclopedia text." (Klein, Simmons 1963, 342), bis 90% des Textes durch die Regeln abgedeckt waren. Die restlichen Einträge sollten automatisiert werden. Dies birgt die Gefahr, daß die Regeln nicht systematisch erstellt werden, sondern

nach der trial-and-error-Methode. Dadurch kann nicht sichergestellt werden, daß alle Phänomene abgedeckt werden. Außerdem zieht eine Änderung des Tagsets notwendigerweise die Änderung der Regeln nach sich. Dadurch ist die Anpassung des Systems auf verschiedene Textsorten sehr aufwendig.
Der Test des Systems mit einigen Seiten der Enzyklopädie ergab, daß 90% des Textes korrekt und eindeutig getaggt wurde. Klein und Simmons (1963) berichten, daß sich auch die Befürchtung nicht bewahrheitet hatte, daß viele ambige Tagsequenzen sich weiter als über drei Wörter hinziehen würden. DeRose (1988) kritisiert jedoch, daß durch die wenig spezifischen Tags die Ambiguität reduziert wird und daß in einem größeres Textkorpus sowohl längere ambige Tagsequenzen als auch Tagsequenzen mit geringer Häufigkeit aufgetreten wären. Hierbei kann ergänzt werden, daß das Aufnehmen all dieser Tagsequenzen - falls es überhaupt möglich ist, diese empirisch zu ermitteln - den Umfang des Systems jedoch so erweitern würde, daß kaum ein Unterschied zu einem Lexikonansatz bestehen würde. Außerdem können bei Veränderungen der Regeln, um Fehler zu beheben, Flip-Flop-Probleme auftreten, d.h. diese Veränderungen haben unter Umständen auch Wechselwirkungen mit anderen Regeln, die nicht sofort sichtbar werden, die aber zu anderen Fehlern führen können.

4.1.2. TAGGIT

Das System

TAGGIT (Greene, Rubin 1971) war das erste Taggingsystem, das auf ein großes Textkorpus angewandt wurde. Es wurde benutzt, um das Brown Korpus (Brown University Standard Corpus of Present-day American English) zu taggen.
TAGGIT verwendet 86 Tags, die mit gewissen Abwandlungen auch in späteren Taggingsystemen (vgl. CLAWS und VOLSUNGA) benutzt wurden. Anders als bei Klein und Simmons (1963) besitzt TAGGIT zwei Stufen, in der ersten werden alle möglichen Tags für ein Wort gesucht, in der zweiten Stufe wird dann die Disambiguierung geleistet. Das eigentliche Taggen in der ersten Stufe geschieht ähnlich wie bei CGC. Erst wird in einem Ausnahmenlexikon gesucht, das ca. 3.000 Wörter, darunter alle Funktionswörter, enthält. Danach werden Sonderfälle wie Zusammenziehungen, kapitalisierte Wörter, etc.

behandelt. Zum Schluß werden die Wortendungen mit einer Liste von Suffixen verglichen. Wörter, denen bis dorthin noch kein Tag zugewiesen worden ist, bekommen schließlich die Tags [NN], [VB] und [JJ] zugewiesen. Die Disambiguierungskomponente basiert auf 3300 Kontextregeln der Form:

W X ? Y Z → A

oder

W X ? Y Z → nicht A

Hierbei steht das Fragezeichen für das ambige Wort, W, X, Y und Z für eindeutig getaggte Wörter aus dem Kontext des Wortes, wobei aber nicht immer alle Variablen belegt sein müssen. Der vom System betrachtete Kontext besteht folglich aus bis zu zwei eindeutig getaggten Wörtern vor und/oder hinter dem Wort. A gibt das Tag an, das dem Wort '?' zugeordnet werden muß, falls es unter den möglichen Tags vorhanden ist, bzw. bei der zweiten Regelvariante, das aus der Menge der möglichen Tags entfernt werden muß. Die Kontextregeln sind in einer Reihenfolge angeordnet, so daß die allgemeineren Regeln zuerst angewandt werden. Führen sie zu einer Disambiguierung, werden die spezielleren Regeln blockiert.
Ursprünglich war auch für TAGGIT ein Regelformat geplant, das eine Reihe von bis zu drei ambigen Wörter berücksichtigen könnte; bei der Erstellung der Regeln, die für das Disambiguieren eines kleinen Teils des Korpus benötigt wurden, stellte sich jedoch heraus, "that a sequence of two or three ambiguities rarely occured more than once in a given context." (Greene, Rubin 1971, 32). Daher wurde auf ein solches Format verzichtet.

Kritik

Greene und Rubin (1971) geben für TAGGIT den Prozentsatz der richtig getaggten Wörter mit 77% an. Dies ist wesentlich niedriger als der Prozentsatz von Klein und Simmons (1963), allerdings sind die Ergebnisse nur bedingt vergleichbar, da der CGC nur anhand von kleinen Textmengen getestet wurde, während mit TAGGIT immerhin die eine Million Wörter des Brown Korpus automatisch getaggt wurden.
Prinzipiell können hier jedoch z.T. dieselben Kritikpunkte wie bei Klein und Simmons (1963) angeführt werden: Auch hier wurden die Kontextregeln basierend auf einer

kleinen Textmenge erstellt, wodurch nicht sichergestellt werden kann, daß alle Phänomene damit beschrieben werden. Auch hier können Flip-Flop-Probleme auftreten, und die Änderung des Tagsets zieht ebenfalls eine Änderung der Regeln nach sich. Außerdem können auch Sequenzen von ambigen Wörtern zum Scheitern der Disambiguierung führen. Marshall (1987, 43) demonstriert dies am Beispiel des folgenden Satzes:

(4) Henry likes stew.

In (4) können 'likes' und 'stew' sowohl Pluralsubstantiv oder 3. Person Singular Verben sein. Die Disambiguierungsregeln, die diese Ambiguitäten auflösen könnten, können nicht angewandt werden, da beide Wörter ambig sind, und deshalb als Regelkontext nicht verwendbar sind.

4.1.3. THE TAGGER

Das System

THE TAGGER (Brodda 1982) gehört nur bedingt in die Gruppe der regelbasierten Tagger. Von seiner Zielsetzung her ist es eher ein interaktives Anwenderprogramm, das den Linguisten bei der Wortklassifizierung unterstützen soll. Es verwendet jedoch ein Lexikonsystem, das den Anwender mit Vorschlägen unterstützt. Durch den Lexikonaufbau können nicht nur ganze Wörter mit ihren Tags eingetragen werden, sondern es besteht auch die Möglichkeit, Präfixe und Suffixe mit den entsprechenden Tags einzutragen. Das Lexikon besteht aus Ersetzungsregeln der Form

xxxx* yyy* C

wobei xxxx* ein Wort oder ein Affix ist, yyy* das zugehörige Tag, und C eine Zahl zwischen 0 und 3 ist, die angibt, ob es sich bei dem Eintrag um ein Wort (3), ein Präfix (1) oder um ein Suffix (2) handelt oder ob der Eintrag sowohl als Wort als auch als Affix verwendet werden kann (0).
Brodda (1982, 95) erklärt außerdem, daß das System in begrenztem Maß lernfähig ist, "the program also learns from the behaviour of the analyzing linguist, so that it will successively perform better." Das Lernverhalten ergibt sich aus Neueinträgen von Vor-

schlägen des Anwenders in das Lexikon. Soweit aus dem Artikel zu erkennen ist, geschieht die jedoch weder automatisch, wie bei einem Lernsystem zu erwarten wäre, noch besteht eine Möglichkeit des Systems, Angaben des Anwenders in Regeln zu generalisieren.

Kritik

Leider macht Brodda (1982) keine Angaben, inwieweit das System die Aufgabe des Anwenders erleichtert oder wie oft richtige Informationen geliefert werden, bzw. wie zuverlässig die Informationen der Affixregeln sind. Das Fehlen derartiger Angaben kann dadurch erklärt werden, daß der Autor offensichtlich nicht so sehr am Mechanismus des Tagging, sondern eher an grundlegenden Problemen der Datenverarbeitung - wie die Koordinierung von Ein- und Ausgabe oder die Vor- und Nachteile verschiedener Dateiformate - interessiert ist. Grundsätzlich kann jedoch kritisiert werden, daß der Anwender zwar die Möglichkeit hat, neue Regeln einzuführen, es ist jedoch nicht möglich, die globalen Auswirkungen einer solchen Regel auf das Taggen des restlichen Textes zu überprüfen. Da auch kein direkter Zugriff des Anwenders auf das Lexikon möglich ist, kann es zu einer starken Redundanz innerhalb des Lexikons kommen, wenn spezielle ebenso wie allgemeinere Regeln zur Behandlung eines Phänomens aufgenommen wurden. Dies führt zu einem ineffizienten System, das die Aufgabe des Anwenders u. U. noch erschwert.

4.1.4. Das ASCOT Tagging System

Das System

ASCOT , das 'Automated Scanning System for Corpus Oriented Tasks', (Akkerman et al. 1987) ist ein als interaktives System konzipiertes Taggingprogramm. Es verfügt nur über eine Komponente, in der den Wörtern alle möglichen Tags zugewiesen werden, die Disambiguierung erfolgt manuell. Die Zuweisung der Tags findet in zwei Phasen statt: zuerst wird im Lexikon Aslex nachgeschlagen, ob das Wort dort vorhanden ist

(= straight hit). In der zweiten Phase, dem sogenannten Reroute, wird durch das Abtrennen der einzelnen Schichten von Affixen versucht, auf einen Lexikoneintrag zu stoßen (= indirect hit). Wird so ein Eintrag gefunden, wird das Wort mit der Information des

Lexikoneintrags und der Information über die Flexion oder Derivation versehen. ASCOT leistet demzufolge eine Lemmatisierung. Der Durchlauf durch die zweite Phase wird dem Benutzer freigestellt, wenn in der ersten Phase ein Lexikoneintrag gefunden wurde. Ein Verzicht auf die zweite Phase kann jedoch dazu führen, daß nicht alle Lesarten eines Worts gefunden werden, wenn das Wort sowohl als Grundform als auch als abgeleitete Form interpretiert werden kann.

Bsp. 'moped' als Nomen ist im Lexikon zu finden; damit es auch als Partizip der Vergangenheit erkannt wird, muß Reroute ebenfalls durchlaufen werden.

Kritik

Da auf eine maschinelle Disambiguierung verzichtet wurde, können für ASCOT keine Erfolgsquoten angegeben werden. Die Autoren begründen diesen Schritt wie folgt: "it will be in the interest of overall modularity and clarity to be able to trace the exact contribution of each component to the analytic process. Also it will be easier to locate and deal with, problem-areas, and in the course of time facilitate shifting portions of the analytic burden from one component to another." (Akkerman et al. 1987, 182). Diese Argumentationsweise kann jedoch entkräftet werden, wenn nicht nur das Endergebnis aus vorläufigem Tagging und Disambiguierung vom System präsentiert wird, sondern wenn auch die Teilergebnisse zugänglich sind.

4.2. Probabilistische Ansätze

In diesem Kapitel werden diverse probabilistische Systeme vorgestellt, bei denen das richtige Tag eines Worts durch Berücksichtigung eines minimalen Kontext mittels Übergangswahrscheinlichkeiten gefunden wird. Probabilistische Systeme sind i.a. in zwei Phasen aufgeteilt, in der ersten Phase werden alle möglichen Tags für ein Wort durch Lexikonsuche oder Suffixanalyse gesucht, in der zweiten findet eine Disambiguierung statt. In diesen Fällen bezieht sich die Charakterisierung als probabilistisches System auf die Disambiguierungsphase.

4.2.1. CLAWS

Das System

CLAWS, das 'Constituent-Likelyhood Automatic Word-Tagging System', (Atwell, Leech, Garside 1984, Garside 1987, Marshall 1987, Atwell 1987) wurde ebenso wie TAGGIT entwickelt, um die automatische Wortklassifizierung eines Korpus, d.h. des LOB-Korpus (Lancaster-Oslo/Bergen Korpus of British English) zu übernehmen (für einen parallelen Ansatz vgl. Eeg-Olofsson 1985, 1987). CLAWS ist ein gemischt probabilistisch-regelbasiertes System, wobei durch die regelbasierte Komponente alle möglichen Tags für ein Wort ermittelt werden und die probabilistische Komponente die Disambiguierung leistet. In CLAWS wurden die Erkenntnisse aus dem TAGGIT-Projekt weiter verwendet und verbessert. CLAWS verwendet das Tagset von TAGGIT, nur bei Eigennamen und bei Pronomen wurden feinere Unterschiede gemacht, so daß insgesamt 133 verschiedene Tags unterschieden werden. CLAWS besteht außerdem aus fünf Phasen, die hintereinander angewandt werden: Pre-Editing, WORDTAG, IDIOMTAG, CHAINPROBS und Post-Editing. Aufgabe der Pre-Editing Phase ist es, den laufenden Text in eine Wortliste umzuwandeln, wobei unwichtige Information unterdrückt, satzinitiale groß geschriebene Wörter - wo nötig - umgewandelt, Kontraktionen auseinandergezogen und die Interpunktionszeichen getaggt werden.
WORDTAG weist den Wörtern kontext-unabhängig ein oder mehrere Tags zu. Dazu verwendet es ein Lexikon mit ca. 7200 Worteinträgen und eine Menge von Regeln. Ein

Wort wird zuerst im Lexikon nachgeschlagen, das alle Funktionswörter, die häufigsten Nomen, Verben und Adjektive, sowie Ausnahmen enthält, die in den Regeln nicht berücksichtigt werden. In einem zweiten Schritt werden alle Numerale und Formeln getaggt, danach Wörter mit Bindestrichen, wobei auf das Vorhandensein bestimmter Wortteile geprüft wird.

Bsp. Wenn die Tags eines Wortteils [past participle] enthalten, weise dem Wort das Tag [adjective] zu, z.B. 'self-employed'. (nach Garside 1987, 38)

Trifft keiner dieser Fälle zu, werden Suffixlisten verwendet. Hierbei wird noch zwischen groß geschriebenen Wörtern und klein geschriebenen unterschieden. Die Suffixlisten sind der Länge nach geordnet, so daß erst auf die längsten Suffixe getestet wird. Das Suffix -s wird abgetrennt und das restliche Wort mittels Lexikon und Suffixlisten getaggt. Scheitert auch dieser Test, wird das Wort per Default mit [noun], [verb] und [adjective] bzw. bei groß geschriebenen Wörtern mit [proper noun] getaggt.
IDIOMTAG sucht nach bestimmten Wortfolgen, bzw. Wort-Tagfolgen, für die mittels eines begrenzten Kontext die Anzahl der möglichen Tags eingegrenzt werden kann. Hierzu wird eine Liste von 150 Phrasen durchsucht und eine gefundene Wortsequenz entsprechend getaggt. In dieser Phase werden auch Wortgruppen wie 'because of' oder 'at once' behandelt. Da sie als syntaktische Einheit agieren, sollten sie nicht einzeln getaggt werden. Daher bekommt das erste Wort der Gruppe das entsprechende Tag zugewiesen, die restlichen Wörter ein sogenanntes ditto-Tag.
Die probabilistische Komponente CHAINPROBS leistet die Disambiguierung der bis jetzt ambig gebliebenen Wörter. Die Erprobung von TAGGIT hatte ergeben, daß 80% der Regeln nur einen minimalen Kontext von einem Wort vor oder hinter dem ambigen Wort benutzten. Diese Beobachtung "suggested that the task of disambiguating words in the LOB corpus might be approached through a statistical analysis of co-occuring tags in the already-tagged Brown Corpus, which would yield a transition matrix showing the probability of any one tag following any other." (Marshall 1987, 45). Die Übergangswahrscheinlichkeitsmatrix wurden aus der ersten Hälfte des Brown Korpus ermittelt. Wenn zwei w_1 und w_2 aufeinanderfolgende Wörter sind und w_1 korrekt mit A getaggt ist, wird die Wahrscheinlichkeit, daß B das korrekte Tag für w_2 ist, mit folgender Formel berechnet:

$$\text{prob}(w_2, B) = \frac{freq(AB)}{freq(A) \times freq(B)}$$

wobei freq(AB) die Wahrscheinlichkeit ist, mit der B im Korpus auf A folgt, und freq(x) die absolute Häufigkeit von (x) ist. Dieser Wert gibt die Stärke der Bindung zwischen A und B an. Dieses Wahrscheinlichkeitsmaß berücksichtigt nur die Bindung zweier aufeinander folgender Tags, trotzdem ist dies nicht der Regelfall wie bei TAGGIT. Bei Folgen von mehreren ambiger Tags, eingeschlossen von eindeutig getaggten Wörtern, werden alle möglichen Kombinationen der möglichen Tags untersucht und diejenige mit dem besten Wert wird dann bevorzugt. Die Gesamtwahrscheinlichkeit einer solchen Folge von ambigen Wörtern ergibt sich dabei aus dem Produkt der Wahrscheinlichkeiten aller Tags geteilt durch die Summe der Wahrscheinlichkeiten aller Pfade:

$$prob(X_j) = \frac{val(X_j)}{\sum_{i=1}^{n} val(X_i)}$$

mit X_j als j-ter Sequenz, val(X_j) als Produkt der Wahrscheinlichkeiten der einzelnen Tags der Sequenz und n als der Anzahl der verschiedenen Sequenzen. Da die Interpunktionszeichen immer eindeutig getaggt sind, sind die ambigen Sequenzen i.a. nicht länger als 10 Wörter. Außerdem können auch die Wort-Tag-Kombinationen direkten Einfluß auf die Auswahl des besten Pfades nehmen: Die Tags für bestimmte Wörter sind mit sogenannten Seltenheitsmarkierungen versehen, dabei wird zwischen 'selten' und 'sehr selten' unterschieden. Für jede solche Markierung wird die Wahrscheinlichkeit der Sequenz auf die Hälfte (für 'selten') bzw. auf ein Achtel (für 'sehr selten') heruntergekürzt.
Um jedoch auch eine Bewertungsmöglichkeit, unabhängig von der Korrektheit der umliegenden Tags, zu haben, berechnet CHAINPROBS außerdem noch die 'additive' Wahrscheinlichkeit, daß ein Tag korrekt ist; sie wird berechnet aus dem Quotienten aus der Summe aller Sequenzen, die das Tag für das fragliche Wort wählen, und der Summe aller möglichen Sequenzen:

$$\text{prob}(w_j, Z) = \frac{\sum_{i=1}^{n} falls \quad tag_j \quad in \quad X_i \quad Z \quad ist, \quad dann \quad val(X_i) \quad sonst \quad 0}{\sum_{i=1}^{n} val(X_i)}$$

wobei Z das fragliche Tag und tag_j das j-te Tag in der Sequenz ist.
In der Ausgabe von CHAINPROBS sind sowohl die bevorzugte Tagsequenz markiert als auch für jedes Wort die Tags mit ihren additiven Wahrscheinlichkeiten. Stimmen beide

überein und überschreitet die additive Wahrscheinlichkeit des bevorzugten Tags einen Schwellenwert, dann werden die anderen Tags gelöscht, ansonsten muß in der Post-Editing-Phase entschieden werden, welches das richtige Tag ist.
Post-Editing in CLAWS geschieht manuell, in zwei Durchläufen: erst werden die restlichen Ambiguitäten aufgelöst, dann wird für jedes Wort überprüft, ob das von CHAIN-PROBS gewählte Tag das richtige ist.

Kritik

Durch die Verwendung von Übergangswahrscheinlichkeiten und die Berücksichtigung aller möglichen Sequenzen von ambigen Wörtern ist CLAWS das erste System, das in der Lage ist, mehr als ein ambiges Wort zu verarbeiten. Außerdem besitzt das System eine hohe Genauigkeit (96-97%). Ihalainen (1990) berichtet weiter, daß CLAWS auch bei der Anwendung auf englische Dialekt-Texte des Helsinki-Korpus ohne größere Probleme arbeitete, obwohl es auf Standard-Englisch trainiert wurde. Ein weiterer Vorteil des Systems liegt in der Objektivierung der verwendeten Information, da die Regularitäten, mit denen gearbeitet wird, nicht mehr auf subjektiver Basis und von Hand erstellt werden müssen. Vielmehr lassen sie sich aus einem getaggten Korpus berechnen. Ein Nachteil dabei ist jedoch, daß nicht sichergestellt ist, daß die Übergangswahrscheinlichkeiten eine wirkungsvolle Abbildung der Übergänge im zu taggenden Korpus darstellen, da diese Information abhängig von der Textsorte ist (vgl. auch Weischedel et al. 1993). Auch muß die Übergangsmatrix immer neu berechnet werden, wenn das Tagset verändert wird; dies setzt voraus, daß ein bereits getaggtes Korpus mit dem neuen Tagset vorhanden ist.
DeRose (1988, 34) kritisiert weiterhin, "CLAWS is time- and storage-inefficient in the extreme, and in some cases a fallback algorithm is employed to prevent running out of memory, ... ". Und da das System die Wahrscheinlichkeiten aller Pfade berechnet, bedeutet dies, daß "it operates in time and space proportional to the product of all the degrees of ambiguity of the words in the span." (DeRose 1988, 34). Aus diesem Grund zeigt der Algorithmus ein Zeitverhalten, das exponentiell zu der Sequenzlänge ist.

4.2.2. VOLSUNGA

Das System

VOLSUNGA (DeRose 1988) ist als System zur Disambiguierung von Tags konzipiert. DeRose (1988) verwendet als Korpus das von Greene und Rubin (1971) getaggte Brown Korpus und kann somit auf ein vollständiges Lexikon zurückgreifen. DeRose (1988, 35) begründet dies damit, daß "This problem has been repeatedly solved via suffix analysis ... and is not of substantial interest here." VOLSUNGA verwendet ein Tagset von 97 Tags, die auf dem Tagset basieren. Ebenso wie CLAWS verwendet VOLSUNGA die constituent-likelyhood Methode zur Berechnung der Übergangswahrscheinlichkeiten; auf die additiven Wahrscheinlichkeiten und die Sonderbehandlung von idiomatischen Ausdrücken wird jedoch verzichtet. Anstatt der Seltenheitsmarkierungen verwendet DeRose (1988) die 'Relative Tag Probablilities' (RTP), ein Maß, das angibt, wie häufig ein Tag ein Wort beschreibt. Dadurch wird die Abstufung deutlicher; allerdings mußten diese Werte normalisiert werden, da sonst die Übergangswahrscheinlichkeiten völlig überschattet worden wären.
Der größte Unterschied zu CLAWS besteht jedoch in der Berechnung der einzelnen Pfade von ambigen Sequenzen. Während in CLAWS alle möglichen Tagsequenzen überprüft werden, verwendet VOLSUNGA das Prinzip der dynamischen Programmierung. Dynamische Programmierung "does not refer to a particular class of optimization problems (...); rather it indicates a general procedure for decomposing a problem into a series of subproblems involving fewer variables and combining their solutions to get the solution of the original problem." (Danø 1975, 62). Bei VOLSUNGA besteht das Gesamtproblem darin, den besten Pfad durch den Graphen aus ambigen Tags zu finden. Die Teilprobleme ergeben sich aus der Suche nach dem besten Pfad bis zu den möglichen Tags eines Wortes x_i. Hierbei werden nicht mehr alle Pfade bis zum Ende verfolgt, sondern immer nur die besten beibehalten, d.h. es werden alle Pfade von allen Tags von Wort x_{i-1} zu allen Tags von Wort x_i berechnet. Dann wird für jedes Tag von x_i der beste Pfad, d.h. der Pfad mit der höchsten Wahrscheinlichkeit, beibehalten, alle anderen werden aufgegeben. Dies bedeutet, daß die Anzahl der aktuellen Pfade immer so hoch ist wie die Anzahl der möglichen Tags des gerade zu bearbeitenden Worts. Bei dem Satz in Abb. 5 z.B. muß CLAWS alle 2*3*2*2=24 eingezeichneten Pfade verfolgen. Erst wenn

alle Pfade berechnet sind, wird der beste ausgesucht und die anderen aufgegeben. VOLSUNGA berechnet alle Pfade vom einzigen Tag von 'The' zu den Tags von 'man'. Danach werden alle Pfade von den Tags von 'man' zu den Tags von 'still' berechnet und für die Tags [NN], [VB] und [RB] der jeweils beste Pfad ausgesucht. Wenn der beste Pfad zu 'still'[NN] der Pfad [AT] [NN] [NN] ist, dann wird nur dieser weiterverfolgt. Der andere mögliche Pfad [AT] [VB] [NN] wird aufgegeben. D.h. wenn die Analyse 'still' erreicht hat, gibt es bei VOLSUNGA nur drei Pfade, bei CLAWS wären es sechs. Folglich ist die Anzahl der Pfade abhängig vom Ambiguitätsgrad des zu bearbeitenden Worts, womit ein lineares Zeitverhalten des Systems erreicht ist.

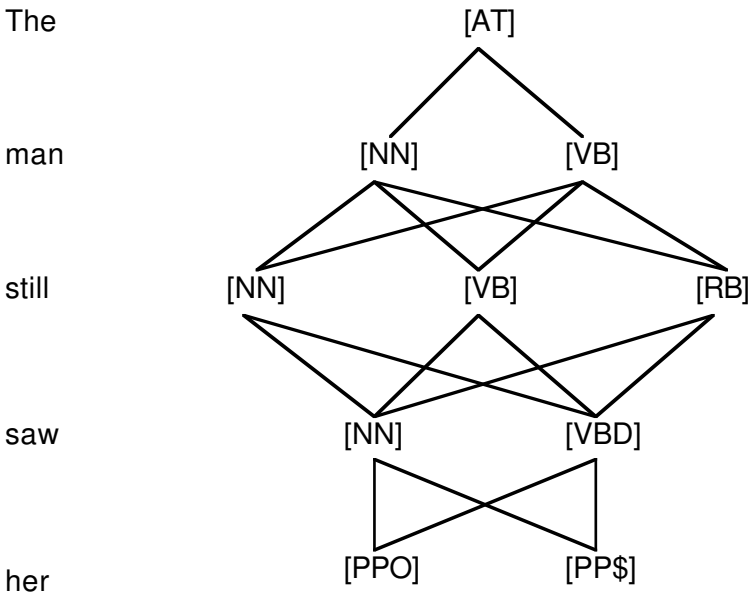

Abbildung 5 (nach DeRose 1988, 36)

Kritik

VOLSUNGA hat wie alle probabilistischen System den Vorteil, daß die linguistische Information nicht explizit angegeben werden muß, sondern aus einem bereits getaggten Korpus in Form von Übergangsregeln errechnet wird. Durch den Verzicht auf eine gesonderte Behandlung idiomatischer Ausdrücke fällt auch die Notwendigkeit für weitere

linguistische Information weg, und durch die dynamische Programmierung wird ein sehr effizientes Zeitverhalten erreicht.
DeRose (1988) gibt den Korrektheitsgrad seines Systems mit 96% an. Dies ist jedoch nur bedingt mit den anderen Systemen vergleichbar, da VOLSUNGA in mancher Hinsicht unter idealen Bedingungen getestet wurde, die bei der weiteren Anwendung des Systems auf andere Korpora nicht mehr gegeben sind: Da als Testkorpus das bereits getaggte Brown Korpus verwendet wurde, konnte ein vollständiges Lexikon verwendet werden, die RTPs konnten sehr genau bestimmt werden und die Matrix der Übergangswahrscheinlichkeiten war eine genaue Abbildung der Verhältnisse im zu taggenden Korpus. Letzteres sollte laut DeRose (1988, 37) auch für andere Texte zutreffen, da "the Corpus is comprehensive enough so that use of other input text is unlikely to introduce statistically significant changes in the program's performance."
Dies ist jedoch nicht für alle Textsorten sichergestellt.
Auch die Verwendung der RTPs ist nicht unumstritten, da selbst bei einem großen Korpus viele Wörter mit einer sehr geringen Häufigkeit auftreten, so daß die Verteilung der möglichen Tags für dieses Wort nicht sicher bestimmt werden kann.

4.2.3. Churchs Stochastic Parts Program

Das System

Ebenso wie DeRose (1988) konzentriert sich Church (1988) in seinem System auf die Disambiguierung von bereits zur Verfügung stehenden möglichen Tags und verwendet einen dynamischen Programmierungsalgorithmus mit linearem Zeitverhalten. Anders als DeRose (1988) verwendet Church (1988) jedoch ein Trigramm-Modell zur Darstellung der Übergangswahrscheinlichkeiten. Die Wahrscheinlichkeit, daß ein Tag X gegeben ist, wenn die Tags Y und Z folgen wird durch folgende Formel bestimmt:

$$\text{prob(X, YZ)} = \frac{freq(XYZ)}{freq(YZ)}$$

wobei freq(XYZ) die Häufigkeit des Auftretens der Tagsequenz XYZ darstellt, und freq(YZ) die Häufigkeit der Sequenz YZ. Die Wahrscheinlichkeit eines Pfads von möglichen Tags wird durch das Produkt dieser kontextuellen Wahrscheinlichkeiten und der

lexikalischen Wahrscheinlichkeiten, die DeRose (1988) RTPs nennt, gebildet. Allerdings berechnet Church (1988) die lexikalischen Wahrscheinlichkeiten nicht direkt über die Verteilung der möglichen Tags eines Wortes im Korpus. Die Tagverteilung bildet die Grundlage der Berechnung, es werden jedoch auch Möglichkeiten berücksichtigt, die im Korpus nicht vorkommen. Zu diesem Zweck wird für jedes Wort in einem konventionellen Lexikon nachgeschlagen, welche möglichen Tags es gibt, und für jedes Tag wird die Auftretenshäufigkeit um eins hochgesetzt. Bei Wörtern, für die alle im Lexikon angegebenen Tags auch im Korpus gefunden werden, ändert sich nichts an der Verteilung. Wenn aber ein nicht sehr häufig benutztes Tag für ein Wort im Korpus nicht gefunden wird, wird das Tag so behandelt, als ob es genau einmal gefunden worden wäre. Damit wird verhindert, daß ein Pfad durch die Multiplikation mit null (da das Wort mit diesem Tag im Korpus nicht vorhanden ist) die Wahrscheinlichkeit null erhält und damit aufgegeben wird, selbst wenn die Wahrscheinlichkeiten der restlichen Tags groß waren.

Kritik

Church (1988, 136) gibt die Zuverlässigkeit seines Systems mit 95-99% an, "depending on the definition of 'correct'", wobei die meisten Fehler auf falsche Angaben im Lexikon zurückzuführen seien. Worin die verschiedenen Definitionen von Korrektheit bestehen, wird nicht expliziert. Ein Versuch von de Marcken (1990) deutet jedoch darauf hin, daß die Rechtsorientierung der Disambiguierung nur geringfügigen Einfluß auf die Performanz des Systems hat. De Marcken (1990, 245) geht davon aus "that Church's strategy of using trigrams instead of digrams may be wasteful. ... If the rightward dependence of disambiguation is small, as the data suggest, then the extra effort may be for naught."

4.2.4. Das Hidden-Markov-Modell von Cutting et al.

Das System

Im Gegensatz zu den bisher vorgestellten probabilistischen Systemen verwenden Cutting et al. (1992) ein Hidden-Markov-Modell für ihren Tagger. Dem geht ein Tokenizer voraus, der Wortgrenzen erkennt und die einzelnen Wörter an das Lexikon weitergibt, wo sie lemmatisiert werden und die Ambiguitätsklasse erhalten. Bedingt durch das Hidden-Markov-Modell kann das System auf einem ungetaggten Korpus mittels der 'forward-backward'-Methode (Baum, Eagon 1967) trainiert werden. Im Hidden-Markov-Modell wird davon ausgegangen, daß nur einige Merkmale, in diesem Fall die Ambiguitätsklassen eines Worts (d.h. die Menge der Tags, die ein Wort haben kann), nicht aber die Wortklassen selbst, beobachtet werden können. Die Abhängigkeiten der Wortfolgen wird durch einen zugrundeliegenden Markov-Prozeß mit zustandsabhängigem Symbolgenerator simuliert. Die Anwendung des Hidden-Markov-Modells besteht aus zwei Phasen: zuerst müssen die Parameter für den Markov-Prozeß und den Symbolgenerator aus einem Trainingsset abgeschätzt werden. Das beste Modell ist dasjenige, mit dem die beste Vorhersage über die Wortfolgen im Trainingsset gemacht werden können. Danach wird die wahrscheinlichste Sequenz der zugrundeliegenden Übergangswahrscheinlichkeiten $S = \{S_1, S_2, ..., S_T\}$ mit $S_i \in W$ und $1 \leq i \leq T$ für neue Eingaben berechnet. Der Markov-Prozeß wird durch eine Menge von N Zuständen (= Anzahl der möglichen Tags), einer Übergangsmatrix $A = \{a_{ij}\}$ mit $1 \leq i, j \leq N$, wobei a_{ij} die Wahrscheinlichkeit des Übergangs von i nach j ist, und einen Vektor $\Pi = \{\pi_i\}$ mit $1 \leq i \leq N$ beschrieben, wobei π_i die Wahrscheinlichkeit ist, daß im Startzustand i begonnen wird. Der Symbolgenerator wird durch die Matrix der Symbolwahrscheinlichkeiten $B = \{b_{jk}\}$ mit $1 \leq j \leq N$ und $1 \leq k \leq M$ beschrieben, wobei M die Anzahl der Ambiguitätsklassen W ist und b_{jk} die Wahrscheinlichkeit, daß der Prozeß im Zustand j das Symbol s_k generiert. Die Einschätzung der Parameter, die die Wahrscheinlichkeit des Trainingssets maximieren, erfolgt über die rekursive Definition der zwei Wahrscheinlichkeitsmengen, der Vorwärtswahrscheinlichkeiten

$$\alpha_{t+1}(j) = \left[\sum_{i=1}^{N} \alpha_t(i)\, a_{ij} \right] b_j(S_{t+1})$$

mit $1 \leq t \leq T-1$ und $\alpha_t(i) = \pi_i b_i(S_1)$ und der Rückwärtswahrscheinlichkeiten

$$\beta_t(i) = \sum_{j=1}^{N} a_{ij} b_j(S_{t+1}) \beta_{t+1}(j)$$

mit $T-1 \leq t \leq 1$ und $\beta_T(j) = 1$ für alle j. Die Vorwärtswahrscheinlichkeit ist das Produkt der Wahrscheinlichkeit der Sequenz $\{S_1, S_2, ..., S_t\}$ bis zum Punkt t und der Wahrscheinlichkeit, daß sich der Markov-Prozeß zu diesem Punkt im Zustand i befindet. Die Rückwärtswahrscheinlichkeit ist die Wahrscheinlichkeit, daß die Sequenz $\{S_{t+1}, S_{t+2}, ... , S_T\}$ auftritt und sich der Prozeß zum Punkt t im Zustand i befindet. Die Klassifizierung einer neuen Eingabesequenz kann als Auswahl der Maximierung über alle Sequenzen betrachtet werden, die die neue Sequenz generieren. Die wird durch einen effizienten Algorithmus zur dynamischen Programmierung, dem Viterbi-Algorithmus (Forney 1973) geleistet.

Kritik

Cutting et al. (1992) geben den Korrektheitsgrad ihres Taggingsystem mit 96% an, womit es mit anderen probabilistischen Systemen vergleichbar ist. Im Gegensatz zu den anderen benötigt es jedoch kein bereits getaggtes Korpus, um die Zustandswahrscheinlichkeiten zu errechnen; für dieses Hidden-Markov-Modell wird nur ein ungetaggtes Korpus verwendet. Außerdem können die Berechnungen der forward-backward-Methode und des Viterbi-Algorithmus dahingehend angepaßt werden, daß ein Zeitverhalten von O(kTN) erreicht wird. Da der Wirkungsgrad dieses Systems vor allem von der Zuweisung der Ambiguitätsklassen abhängig ist, sollte der Behandlung von unbekannten Wörtern eine wichtige Rolle zufallen. Sie wird jedoch wie in den meisten Systemen mittels Suffixlisten angegangen. Ein allgemein bekanntes Problem von Hidden-Markov-Modellen besteht darin, daß nur bei guten Ausgangswerten für die Übergangswahrscheinlichkeiten die optimale Einstellung des Systems erreicht wird, sonst erreicht das System oft nur ein lokales Optimum. Chanod und Tapanainen (1995), die das System von Cutting et al. (1992) als Vergleichssystem verwendet haben, belegen dies: Sie berichten, daß nach dem Trainieren des Systems auf ihrem Korpus von französischsprachigen Texten lediglich eine Erfolgsquote von 87% erreicht werden konnte. Erst nach

einem Mann-Monat, in dem die Anfangseinstellungen des Systems von Hand angepaßt wurden, konnte ein Ergebnis von 96,8% erreicht werden (vgl. 5.3.3).

4.2.5. POST

Das System

Ebenso wie Cutting et al. (1992) verwenden auch Weischedel et al. (1993) ein Hidden-Markov-Modell in ihrem Tagginsystem POST. Sie benutzten das bereits getaggte TREEBANK Korpus, um ein bi-tag und ein tri-tag Modell mit einem Tagset von 47 Tags zu erstellen. Das überwachte Training ermittelt zuerst die Übergangswahrscheinlichkeiten von jedem Tagpaar zu den nachfolgenden Tags, dann die bedingte Wahrscheinlichkeit jedes Wortes bei einem gegebenen Tag. Ähnlich wie Church (1988) verwenden Weischedel et al. (1993) einen Glättungsmechanismus, um zu verhindern, daß eine Tagsequenz, die im Korpus nicht zu beobachten ist, als unmöglich eingestuft wird. Daher werden solche Sequenzen behandelt, als ob sie einmal im Korpus gefunden worden wären. Damit die Wahrscheinlichkeiten des Tagpaars sich zu eins aufaddieren lassen, muß bei allen anderen Tags, die dem Tagpaar folgen, ein dementsprechender Faktor abgezogen werden. Interessant an diesem System ist vor allem die Behandlung der unbekannten Wörter: Da das Tagset 22 verschiedene Tags für offene Klassen besitzt, wäre eine Zuweisung all dieser Kategorien als Default nicht sehr sinnvoll. Statt dessen wird ein probabilistisches Modell verwendet, das die Merkmale Flexionsendungen, Derivationsendungen, Schreibung mit Bindestrich und Großschreibung verwendet, um die Wahrscheinlichkeit zu ermitteln, daß ein bestimmtes Tag dem Wort zugeordnet wird. Mit dieser Methode läßt sich eine Erfolgsquote von 85% bei unbekannten Wörtern erzielen.

Kritik

Weischedel et al. (1993) geben den Korrektheitsgrad ihres Systems mit 96-97% an. Damit liegt es geringfügig über den meisten anderen Systemen. Weiterhin ergab sich aus dem Testen des Systems, daß die allgemein veranschlagte Größe des Lernkorpus für

POST nicht nötig ist. Erst bei weniger als 250.000 Wörtern ließ sich eine signifikante Vergrößerung der Fehlerquote feststellen, die allerdings immer noch unter 4% lag. Eine Anwendung des Systems auf ein anderes Korpus (MUC-3) ergab mit Training mit dem neuen Korpus eine Fehlerquote von 5,6 % (bi-tag) bzw. 5,7% (tri-tag). Ohne erneutes Training ergaben sich Fehlerquoten von 8,5% bzw. 8,3%. Damit wird deutlich, daß probabilistische Systeme selbst mit erneutem Training mit dem neuen Korpus nur mit Einbußen bezüglich der Performanz auf andere Korpora anwendbar sind.

4.3. Neuere Ansätze

Neben gemischt regelbasiert-probabilistischen System wie z.B. Marcus et al. (1993) gibt es auch neuere Ansätze, die ohne eine probabilistische Komponente auskommen. In diesem Abschnitt werden außer dem gemischten System von Lee et al. (1995), das eine Hidden-Markov-Komponente mit Brills (1992) Ansatz kombiniert, zwei syntaktische Taggingsysteme besprochen; bei Hindle (1989) werden die Regeln vom System gelernt, bei Voutilainen (1995) werden sie manuell erstellt. In dem constraint-basierten System bei Chanod und Tapanainen (1995) dagegen werden nur lexikalische Informationen verwendet. Zuerst werden jedoch zwei Systeme (Benello et al. 1989, Nakamura, Shikano 1989) vorgestellt, die mit neuronalen Netzen arbeiten.

4.3.1. Das konnektionistische Taggingsystem von Benello et al.

Das System

Benello et al. (1989) verwenden zur Disambiguierung von Wortklassen ein neuronales Netz mit einer verborgenen Schicht. Anders als die meisten probabilistischen Systeme verwendet das Netz nicht Bigramm-Übergangswahrscheinlichkeiten sondern benutzt ein Fenster, das sich über sechs Wörter spannt, wobei die ersten vier Wörter ebenso wie das letzte Wort eindeutig getaggt sein müssen und das fünfte ambig ist und disambiguiert werden soll. Das Netz besitzt insgesamt 560 Zellen, 236 in der Eingabeschicht, die das 6-Wort-Feld beschreiben, weitere 236 in der verborgenen Schicht und 88 in der Ausgabe-

schicht, um das disambiguierte Zielwortfeld darzustellen. Als Lernalgorithmus wird ein Backpropagation-Algorithmus mit einer symmetrischen Aktivierungsfunktion benutzt. Als Trainingsset wurden 900 Sätze laufender Text aus der Sparte 'Romanze' des Brown Korpus benutzt. Das Tagset des Brown Korpus wurde übernommen, allerdings erhielt jedes Wort eine Menge möglicher Tags zugeordnet. Das Training bestand aus einer aufeinanderfolgenden Reihe von 50, 100, 300 und 900 Sätzen. Das Lernen wurde für jede Satzmenge so lange fortgesetzt, bis nahezu asymptotische Werte erreicht wurden. Getestet wurde das Netz schließlich an 100 weiteren Texten aus derselben Sparte des Korpus.

Kritik

Benello et al. (1989) geben den Korrektheitsgrad ihres Systems mit 95% an, womit es in dieser Hinsicht ähnlich Ergebnisse wie die oben besprochenen probabilistischen Systeme erreicht. Es ist dabei jedoch anzumerken, daß der Test nur mit einer sehr kleinen Menge von 100 Sätzen durchgeführt wurde und daß diese Sätze ebenso wie die Trainingsmenge aus der Sparte 'Romanze' stammen, die sich durch eine größere syntaktische Homogenität als normale englische Prosa auszeichnen, was auch die Autoren einschränkend bemerken. Als weitere Verbesserungsmöglichkeiten schlagen Benello et al. (1989) vor, die Fehler ihres Netzes mit denen probabilistischer Systeme und auch mit menschlichem Verhalten zu vergleichen, weitere Ebenen einzuführen und die optimale Größe des Tagsets zu ermitteln, da sowohl zu feine als auch zu grobe Tagunterscheidungen dem System Probleme verursachen.

4.3.2. NETgrams

Das System

Ebenso wie Benello et al. (1989) verwenden auch Nakamura und Shikano (1989) ein neuronales Netz, jedoch nicht zur Wortklassendisambiguierung sondern direkt zur Wortklassifizierung. Das Netz mit zwei verborgenen Schichten basiert auf einem Bigramm-Modell, kann jedoch zu einem N-Gramm-Modell, also n-1 eindeutig getaggte Wörter gefolgt von einem zu taggenden Wort, erweitert werden, ohne daß die Zahl der freien Parameter sich exponentiell erhöht. Das Bigramm-Modell ist ohne Rückkoppelung (feedforward) verbunden; es besitzt einen Block von 89 Zellen, die den möglichen Tags des eindeutig getaggten Worts entsprechen. Die zwei verborgenen Schichten bestehen jeweils aus 16 Zellen, die Ausgabeschicht wiederum aus 89, die den möglichen Tags des zu taggenden Wortes entsprechen. Um das Netz zu einem Trigramm-Modell zu erweitern, wird ein weiterer Block von Eingabezellen hinzugefügt, der mit der Gewichtsmatrix w1' vollständig mit der ersten verborgenen Schicht des Bigramm-Netzes verbunden ist. Für ein 4-Gramm-Netz wird über den zweiten Eingabeblock eine verborgene Schicht gesetzt, die vollständig mit der zweiten verborgenen Schicht des Bigramm-Netzes verbunden ist. Der weitere Eingabeblock wird vollständig mit der neuen verborgenen Schicht verbunden. (Siehe Abb. 6)
Trainiert wird das System sukzessive, d.h. "first the basic Bigram network is trained, and next, the Trigram networks are trained with the link weight values trained by the Bigram network as initial values. 4-gram networks are trained in the same way." (Nakamura, Shikano 1989, 732). Als Lernalgorithmus führen Nakamura und Shikano eine neue Methode ein - die DCP (Dynamic Control training Parameters) -, die den Backpropagation-Algorithmus beschleunigen soll; beim Gradientenverfahren werden die Verbindungsgewichte durch folgende Regel angegeben:

$$\Delta wij(k) = \eta(-\partial Ep / \partial wij) + \alpha \Delta wij(k-1)$$

Ep ist hierbei die Fehlerfunktion, wij(k) das Gewicht zwischen der i-ten und der j-ten Zelle im k-ten Trainingsschritt. Im DCP-Verfahren werden die Trainingsparameter (η, α) alle N Trainingsdurchläufe dynamisch so verändert, daß Ep in der folgenden Gleichung minimiert wird:

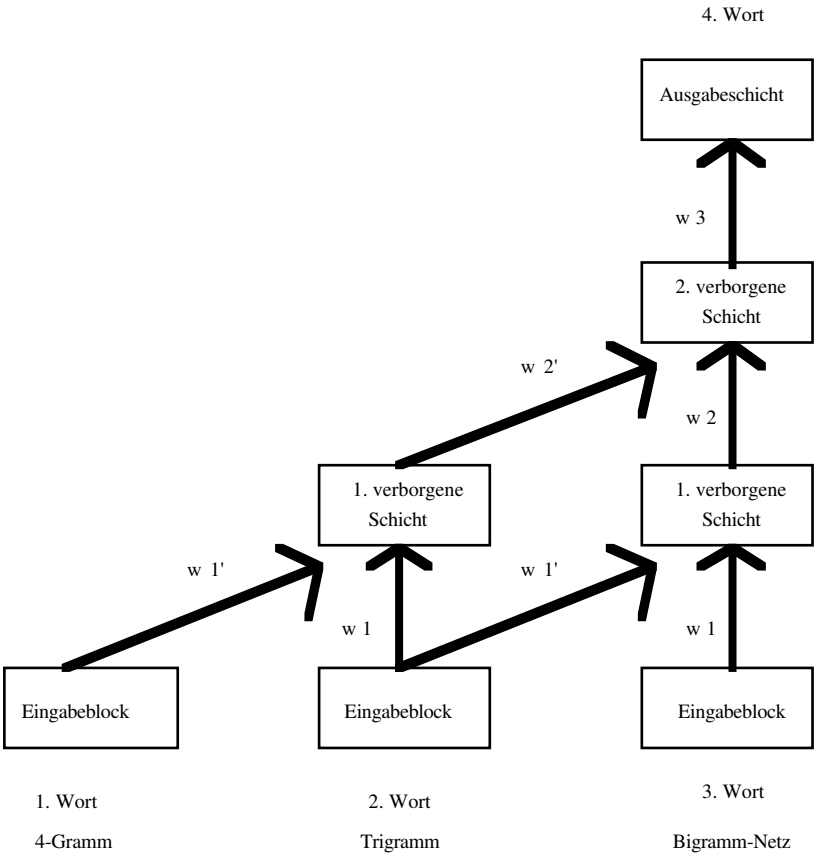

Abbildung 6 (nach Nakamura, Shikano 1989, 732)

$$Ep(wij(k)+\Delta wij(k)(\eta(k),\alpha(k)) = \underset{N,m}{\text{Min}} \quad Ep(wij(k)+\Delta wij(k)(\eta_j \alpha_m))$$

Nakamura und Shikano (1989, 734) geben an, daß die Trainingszeit 4,3 mal kürzer ist als mit dem Standard-Backpropagation-Algorithmus.

Kritik

Nakanura und Shikano (1989) machen keine genauen Angaben über den Korrektheitsgrad ihres Taggingsystems. Sie erklären lediglich, daß die Ergebnisse des Bigramm- und des Trigramm-Netzes mit denen probabilistischer Bigramm- bzw. Trigramm-Systeme vergleichbar sind, machen jedoch keine Angaben über das zum Vergleich verwendete probabilistische Taggingverfahren. Als positive Charakteristika ihres Systems geben die Autoren zum einen die Verkürzung der Lernphase an, schränken dies jedoch ein, indem sie als nächstes Ziel die weitere Beschleunigung eben dieser Phase nennen. Als weiteres positives Charakteristikum beschreiben die Autoren die Fähigkeit des Systems, linguistisch signifikante Strukturen natürlich zu lernen. Sie begründen dies damit, daß ein Ähnlichkeitsvergleich der Merkmale der unteren verborgenen Schicht fünf Klassen ergab: BE-Verben, HAVE-Verben, subjektivische Pronomen, Kategorien, die vor einem Nomen stehen sollten, und andere. Da eine weitere Aufteilung vor allem der vierten Klassen jedoch nicht möglich zu sein scheint, ist fraglich, inwieweit diese Klassifizierung sinnvoll ist.

4.3.3. Hindles Tagger

Das System

Hindle (1989) beschreibt einen regelbasierten Tagger, der als Teil des deterministischen Parsers Fidditch (Marcus 1980) zur Disambiguierung von ambigen Wörtern eingesetzt wird. Dieser Ansatz zur Wortklassifizierung ist nicht als eigenständiges System konzipiert, sondern nur als Taggingkomponente des Parsers, auf dessen Information es angewiesen ist. Im Gegensatz zu den ersten regelbasierten Ansätzen werden die Regeln dem Tagger jedoch nicht vorgegeben, sondern werden wie bei Brill (1992, 1993, 1994) gelernt. Der Lernprozeß findet während des Parsens statt, wobei als Lernkorpus das Brown Korpus benutzt wird: eine der Aktionen des Parsers ist die Auswahl eines Tags für ein Wort. Dazu wird ein Lexikon mit ca. 100.000 Einträgen verwendet. Wenn der Parser im Bearbeitungsfenster, das drei Wörter umfaßt, an der ersten oder zweiten Stelle auf ein ambiges Wort trifft, wird die Disambiguierungskomponente aufgerufen. Diese

besteht aus zwei Teilkomponenten: aus dem Erwerb neuer Regeln und der Deaktivierung von Regeln. Im Anfangszustand besteht die Regelmenge aus 136 Defaultregeln, die für jede mögliche Tagmenge das Defaulttag angeben.

Bsp: Die Defaultregel

[ADJ+N+V] = N [*] [*]

gibt an, daß ein Wort, das laut Lexikon ein Adjektiv, ein Nomen oder ein Verb sein kann, per Default als Nomen getaggt wird (vgl. Hindle 1989, 121-122).

In der Erwerbskomponente wird zuerst überprüft, ob bereits eine Disambiguierungsregel existiert, die in diesem Fall das richtige, d.h. das im Brown Korpus verwendete Tag liefert. Ist dies nicht der Fall, wird eine neue, spezifischere Regel aus der Information des Parsers zu der bereits existierenden Regelmenge hinzugefügt. Da die spezifischen Regeln vor den allgemeineren abgearbeitet werden, ist so sichergestellt, daß die neue Regel die allgemeinere unterdrückt.
Die zweite Teilkomponente des Taggers wird eingesetzt, wenn eine Regel die falsche Disambiguierung macht und keine spezifischere Regel möglich ist. Dann kann die Regel deaktiviert werden, falls der Anteil der falschen Disambiguierungen einen festgelegten Schwellenwert (z.B. 10%) übersteigt. Dies tritt ein, wenn eine spezifische Regel zwar den Spezialfall abdeckt, für den sie generiert wurde, aber auch bei allgemeineren Fällen fälschlicherweise angewandt werden kann.
Das Training besteht aus wiederholten Durchgängen des Parsers über den Text, wobei die Regelmenge weiter verfeinert wird. Dies ist nötig, da die Generierung neuer Regeln nicht nur von dem zu parsenden Satz abhängt, sondern auch von der Regelmenge. Eine weitere Verbesserung ergibt sich durch die Verwendung von lexikalischen Beziehungen zwischen Wortpaaren, die immer mit denselben Tags annotiert werden, und von grammatischen Abhängigkeiten. Unter den grammatischen Abhängigkeiten versteht Hindle (1989) die Beschränkung der verwendeten Regelkontexte auf Wörter, die in derselben syntaktischen Einheit stehen. Letzteres führte zu einer Verringerung der gelernten Regeln auf ein Drittel der Regeln, die ohne diese Einschränkung generiert wurden.

Kritik

Hindle (1989) gibt den Korrektheitsgrad seiner Disambiguierungskomponente mit 97% an, wobei die Verbesserungen durch die lexikalischen Beziehungen noch nicht berücksichtigt sind, womit bewiesen wurde, daß aus dem Text gelernte Information nicht unbedingt statistischer Natur sein muß. Die Anwendung dieses Taggers ist jedoch stark dadurch eingeschränkt, daß er auf die syntaktische Information des Parsers und damit auch auf das vom Parser verwendete Tagset angewiesen ist und deshalb nicht als eigenständiges System verwendet werden kann. Auch hier trifft zu, daß nicht sichergestellt werden kann, daß die auf dem Brown Korpus erworbenen Regeln auch andere Textsorten mit einer ebenso hohen Korrektheitsquote beschreiben. Das erneute Training auf eine neue Textsorte ist jedoch sehr aufwendig, zumal es sich hier nur um eine Komponente eines Parsers handelt. Hindle (1989, 121) bemerkt auch, daß sich eine weitere Verbesserung des Systems durch die Generalisierung der generierten Regel erreichen ließe. Damit könnte verhindert werden, daß zu viele sehr spezifische Regeln gebildet werden, während allgemeinere Zusammenhänge nicht erkannt werden. Soweit an Hindles (1989) Bericht zu erkennen ist, gibt es keine Möglichkeit, eine Rangfolge der Spezifität anzugeben, so daß nicht ausgeschlossen werden kann, daß die spezifischeren Regeln zuerst generiert werden.

4.3.4. ENGCG

Das System

Ähnlich wie die Taggingkomponente von Hindle (1989), beschränkt sich das Taggingsystem von Voutilainen (1995) bzw. Voutilainen und Tapanainen (1993), ENGCG (ENGlish Constraint Grammar), nicht darauf, nur den lexikalischen Kontext eines Wortes zu analysieren, vielmehr wird das System um eine syntaktische Analysekomponente erweitert, die einen Teil der Ambiguitäten auflösen soll. Voutilainens (1995) System besteht aus fünf Komponenten: aus dem regelbasierten Tokenizer, der einen Text in Wörter und Interpunktions- und Formatierungszeichen unterteilt, aus der ENGCG morphologischen Analysekomponente, der morphologischen Disambiguierungskomponente, der

Komponente, in der syntaktische Tags nachgeschlagen werden, und der syntaktischen Analysekomponente. In der morphologischen Analysekomponente wird das Wort zuerst in einem Lexikon nachgeschlagen, das aus 80.000 Einträgen mit sämtlichen flektierten Formen besteht und ein Tagset von 139 Tags verwendet. Wird ein Wort nicht im Lexikon gefunden, werden manuell entworfene heuristische Regeln angewandt, um die möglichen Tags für das Wort zu ermitteln. Die zweite Komponente, die ENGCG Disambiguierung, verwendet eine constraint-basierte Grammatik. Sie besteht aus einer Menge von constraint Regeln, die für jedes Tag einen oder mehrere Kontexte spezifizieren, in denen das Tag nicht verwendet werden kann. Wird also für ein Tag eines ambigen Worts eine solche Regel gefunden, kann dieses Tag ausgeschlossen werden, womit die Ambiguität des Worts aufgelöst oder zumindest verringert werden kann. Voutilainen (1995, 5) gibt an, daß "the constraints are partial and often negative paraphrases of 23 general, essentially syntactic generalizations about the form of the noun phrase, the prepositional phrase, the finite verb chain, etc." Da diese Komponente so konzipiert ist, daß nur unmögliche Tags eliminiert werden, bleiben 3-7% der Wörter auch nach dieser Phase noch ambig. Diese Ambiguitäten werden durch die syntaktische Analyse mittels einer endlichen Intersektionsgrammatik (vgl. Koskenniemi 1990) aufgelöst, d.h. die syntaktischen Regeln, ebenso wie die Sätze werden in endliche Automaten übersetzt und die Automaten dann parallel abgearbeitet. Nur Sätze, die von allen Automaten akzeptiert werden, werden von den Regelautomaten dann als Parses vorgeschlagen.

Kritik

Voutilainen (1995) gibt den Genauigkeitsgrad seines Systems mit über 99% an. Damit ist ENGCG das erste System in der Literatur, das die '97%-Barriere', die kein probabilistisches System erreichen konnte, überschritten hat. Der Test erfolgte an einem Korpus von 38.000 Wörtern, womit die Bewährungsprobe an einer großen Textmenge noch aussteht. Das System arbeitet allerdings mit Hilfe zweier Regelmengen, zum einen den morphologischen Disambiguierungsregeln, und zum anderen den syntaktischen Regeln. Es fragt sich jedoch, ob es wirklich notwendig ist, einen solchen Aufwand zu betreiben, ein vollständiges Syntaxmodell zu implementieren, nur um eine zweiprozentige Verbesserung zu erreichen.

4.3.5. Der constraint-basierte Ansatz von Chanod und Tapanainen

Das System

Das System von Chanod und Tapanainen (1995) ist ein constraint-basierter Ansatz zur automatischen Wortklassifizierung eines französischsprachigen Korpus. Es besteht aus drei Komponenten: dem 'Tokenizer', der den Eingabetext in einzelne Wörter zerlegt, der morphologischen Analysekomponente und der Disambiguierungskomponente. Die morphologische Analyse ist ebenso wie der Tokenizer in Form eines endlichen Transducers implementiert; hier werden die Wörter lemmatisiert und in einem Lexikon nachgeschlagen. Ist es dort nicht zu finden, wird ein anderer Transducer aktiviert, der 'Guesser', der mittels einer Suffixanalyse die möglichen Tags eines Worts ermittelt. Die letzte Komponente verwendet drei verschiedene Regeltypen zur vollständigen Disambiguierung der Wörter, die ebenfalls als endliche Transducer implementiert wurden. Zuerst werden 'sichere' Kontextregeln verwendet, die sicherstellen, daß seltene Tags für ein bestimmtes Wort nur in einem genau bestimmten Kontext auftreten dürfen, wo das häufigere Tag nicht möglich ist. "This means that the tagger errs only when a rare reading should be chosen in a context where the most common reading is still acceptable. This may never actually occur, depending on how accurate the contextual restrictions are." (Chanod, Tapanainen 1995, 3).
Da jedoch nicht alle Ambiguitäten diese Charakteristik besitzen, müssen sie in einem weiteren Schritt behoben werden. Dort werden ad-hoc-Heuristiken eingesetzt, die nicht immer das richtige Ergebnis erbringen, sondern eher Tendenzen in der Sprache beschreiben. Im Französischen ist es z.B. sehr schwierig, bei dem Wort 'des' zwischen Artikel ('Jean mange des pommes.') und Verschmelzung zwischen Präposition und Artikel ('Jean aime le bruit des vagues.') zu unterscheiden. Eine Heuristik besagt, daß bei einem satzinitialen 'des' die Lesart als Artikel bevorzugt wird.
Für Ambiguitäten, die auch in diesem Schritt nicht aufgelöst werden können, wird die Disambiguierung in einem letzten Schritt erzwungen: Durch nicht-kontext-abhängige Heuristiken wird jeweils aus den übrig gebliebenen Tags dasjenige ausgesucht, von dem die Autoren annahmen, es sei das wahrscheinlichste. So wird z.B. eine Präposition dem Adjektiv vorgezogen, oder ein Pronomen einem Partizip der Vergangenheit. Diese Regeln sind zwar nicht sehr zuverlässig, reichen aber nach Angaben der Autoren aus, da

sie nur angewendet werden, wenn durch die vorhergehenden Regeln keine vollständige Disambiguierung erreicht werden konnte.

Kritik

Chanod und Tapanainen (1995) testeten ihr System an zwei verschiedenen Korpora: zum einen an einem Korpus von Wirtschaftsberichten, der 255 Sätze umfaßte, zum anderen an einem als schwierig eingestuften Zeitungstext, der ca. 12.000 Wörter umfaßte und sowohl Namen enthielt, die auch als Lexikoneintrag vorhanden sind, als auch Tippfehler. Im ersten Fall erreichte das System eine Erfolgsquote von 98,7% im zweiten Fall wurden 97,5% des Textes richtig getaggt. Damit lag das System deutlich über dem zu Vergleichszwecken herangezogenen Hidden-Markov-Modell von Cutting et al. (1992), das selbst nach der manuellen Einstellung der Ausgangswerte für den ersten Text eine Erfolgsquote von 96,8% erreichte. Das Ergebnis dieses Vergleichs muß allerdings insoweit relativiert werden, daß das Trainingskorpus des probabilistischen Taggers wie auch das Testkorpus für beide Tagger sehr klein waren. Die mangelnde Größe des Trainingskorpus begründen die Autoren damit, daß das Trainieren sehr häufig wiederholt werden mußte. Es ist jedoch anzunehmen, daß die Performanz des probabilistischen Systems sich bei größeren Korpora verbessern wird, während bei dem constraint-basierten System eher mit einer Verschlechterung zu rechnen ist, da bei größeren Korpora die Möglichkeit gegeben ist, daß weitere Strukturen auftreten, die die Autoren in ihren Regeln nicht berücksichtigt haben. Allerdings ließen sich zumindest die nicht-kontext-abhängigen Regeln sicher noch weiter verbessern, z.B. durch die korpusbasierte Ermittlung des wahrscheinlichsten Tags für jede mögliche Ambiguitätsklasse (vgl. 5.2.4.).

4.3.6. TAKTAG

Das System

TAKTAG, das hybride regelbasiert-probabilistische System von Lee et al. (1995), ist als Taggingsystem für Koreanisch konzipiert. "Korean is classified as agglutinative languages [sic] in which the words (which is called Eojeol in Korean) consist of several morphemes that have clear-cut morpheme boundaries." (Lee et al. 1995, 3). Durch diese Charakteristika stellen sich für das Taggingsystem neuartige Probleme, die in europäischen Sprachen nicht auftreten: Im Koreanischen werden grammatische Rollen nicht durch die Eojeol-Stellung sondern durch Funktionsmorpheme ausgedrückt, daher muß die Kontextinformation selektiv auf Funktions- bzw. Inhaltsmorpheme angewandt werden. Eojeols können nicht nur als Wort, sondern auch als Phrase agieren, so daß die Wortklassifizierung auf Morphemebene ansetzen muß. Dies wird durch komplexe Allomorphbildungen, die durch die verschiedenen Morphemkombinationen ausgelöst werden, erschwert. Zusätzlich besitzt das Koreanische eine relativ freie Wortstellung, so daß Kontextregeln sehr schwierig zu formulieren sind.
Bedingt durch diese Faktoren haben bisher existierende Taggingsysteme nur deutlich schlechtere Ergebnisse als für indo-europäische Sprachen erbracht. Lee et al. (1995) kombinieren aus diesem Grund ein konventionelles Hidden-Markov-Modell und die kontext-sensitve Phase von Brill (1992, 1994). Dem geht eine morphologische Analysekomponente voraus, in der die Eojeols durch eine Suche im Morphemlexikon segmentiert und anhand einer Morphemübergangstabelle überprüft werden.
Das Hidden-Markov-Modell ist ein Bigramm-Modell mit Viterbi-Suche (Forney 1973):

$$T^* = \arg\max_T \prod_{i=1}^{n} prob(t_i / t_{i-1})\, prob(m_i / t_i)$$

wobei $prob(t_i/t_{i-1})$ die Übergangswahrscheinlichkeit darstellt und $prob(m_i/t_i)$ eine Morphem-Tag-Wahrscheinlichkeit. Das Hidden-Markov-Modell wurde mit einem ungetaggten Korpus trainiert. Allerdings verwendeten Lee et al. (1995) eine kleine Menge getaggter Morpheme, um die Anfangsbelegung der Wahrscheinlichkeiten zu optimieren. Der Hidden-Markov-Tagger nimmt als Eingabe die Morphemsequenzen aus der morphologischen Analysekomponente und sucht die optimale Tagsequenz, die dann an die regelbasierte Komponente weitergegeben wird.

In der zweiten Phase werden Verbesserungsregeln nach dem Verfahren von Brill (1992) gelernt. Dazu wurde ein Teilkorpus von Hand getaggt und dann an die morphologische Analyse und das Hidden-Markov-Modell weitergegeben. Gelernt wurde durch den Vergleich der Ausgabe des Hidden-Markov-Taggers und des manuell getaggten Textes.

Kritik

Lee et al. (1995) geben die Erfolgsquote ihres hybriden Systems mit 91,9% an. Damit liegt es deutlich unter den Ergebnissen, die bei Taggingsystemen für europäische Sprachen erreicht wurden. Sie liegt jedoch um über 10% höher als die Quote, die unter alleiniger Benutzung des Hidden-Markov-Modells zustande kam. Lee et al. (1995, 8) machen jedoch deutlich, daß dieses Ergebnis nur als Zwischenstand ihrer Bemühungen anzusehen ist: "The next step will be to analyze the learned rules carefully to extract the more desirable rule schema for Korean." Es wäre jedoch auch interessant zu sehen, wie die alleinige Verwendung des Taggers von Brill (1993), d.h. die Integrierung der kontext-freien Lernphase, sich auf die Ergebnisse auswirkt. Die kontext-freie Lernphase würde nicht nur die probabilistische Komponente ersetzen, sie könnte auch die Aufgabe der morphologischen Analysekomponente übernehmen. Diese Ansatzweise scheint vor allem im Lichte der Ergebnisse von Chanod und Tapanainen (1995) sinnvoll: Im Rahmen eines Vergleichs wurden die regelbasierte und die probabilistische Komponente auch hintereinandergeschaltet. Diese Kombination ergab jedoch mehr Fehler als der regelbasierte Tagger alleine.

4.4. Vergleich mit Brills System

Es gibt verschiedene Kriterien, nach denen ein Taggingsystem bewertet werden kann. Cutting et al. (1992) verwenden z.B. die Kriterien Robustheit, Effizienz, Genauigkeit, Anpassungsfähigkeit und Wiederverwendbarkeit. Von diesen Kriterien lassen sich jedoch nur die Effizienz und die Genauigkeit direkt aus der Systembeschreibung bestimmen, die anderen bleiben der Einschätzung der Autoren überlassen, die sich meist übermäßig optimistisch äußern. So waren auch Cutting et al. (1992) der Meinung, ihr System sei ohne

große Anpassungen für andere Sprachen verwendbar. Bei dem Test von Chanod und Tapanainen (1995) stellte sich jedoch heraus, daß ähnlich gute Ergebnisse für Französisch nur nach einer aufwendigen manuellen Anpassung der Wahrscheinlichkeiten zu erreichen waren.
Aus diesem Grund werden hier nur Kriterien verwendet, die sich direkt aus der Systembeschreibung ergeben.

4.4.1. Wie hoch ist die Erfolgsquote?

Die erste Frage, die sich bei der Bewertung eines Systems stellt, ist wohl immer die Frage nach der Effizienz. Wenn man einmal von den regelbasierten Systemen der ersten Generation absieht, haben die Tagger durchgehend eine Erfolgsquote zwischen 95% und 99%. Lediglich das hybride System für Koreanisch fällt mit 91, 9% aus diesem Bereich. Brills (1993) System liegt mit 95,3% am unteren Rand des Bereichs. Die Unterschiede sind jedoch minimal, da für eine vollständig korrekte Wortklassifizierung in jedem Fall eine Post-Editing-Phase nötig ist. Für Systeme, in denen die Wortklassifizierung nur eine Vorstufe zur syntaktischen Analyse darstellt, bietet es sich jedoch an, einen Tagger zu verwenden, der eine möglichst hohe Erfolgsquote hat, da ein manuelles Nacheditieren nicht möglich ist. Unter den hier verglichenen Systemen ist das von Voutilainen (1995) mit über 99% das erfolgreichste. Damit wird jedoch der syntaktischen Komponente eine Wortklassifizierungskomponente vorgestellt, die auf syntaktischem Wissen beruht.

4.4.2. Ist das System ein vollständiges Taggingsystem?

Eigentlich sollte man davon ausgehen, daß in der Literatur vorgestellte Systeme vollständig sind, d.h. ein Korpus mit ungetaggtem Text so bearbeiten können, daß jedes Wort genau ein Tag zugeordnet bekommt. Bei den hier vorgestellten Systemen ist es jedoch oft der Fall, daß sich die Autoren auf ein Gebiet konzentriert haben. Bei ASCOT (Akkerman et al. 1987) haben sich die Autoren aus oben genannten Gründen (vgl. 5.1.4.) auf die Ermittlung der möglichen Tags für ein Wort konzentriert, eine Disambiguierung wird nicht vorgenommen. THE TAGGER (Brodda 1982), alle probabilistischen Systeme

und die Systeme von Benello et al. (1989) sowie von Hindle (1989) beschränken sich auf die Disambiguierung der aus dem Lexikon und den Suffixlisten ermittelten Tags. Dies läßt sich damit begründen, daß die Lexikonkomponente und die Behandlung unbekannter Wörter überall nach demselben Schema erfolgt und deshalb als bekannt vorausgesetzt werden kann. Brill (1993) geht jedoch einen anderen Weg, er sucht nicht zuerst alle möglichen Tags, um dann das richtige Tag im Kontext zu ermitteln. Er sucht vielmehr das am häufigsten gebrauchte Tag für ein Wort und sucht dann Regeln, in welchem Kontext dieses Tag durch ein anderes ersetzt werden muß.

4.4.3. Können unbekannte Wörter systematisch behandelt werden?

Da bei der Erfolgsquote kaum Unterschiede vorhanden sind, ist die systematische Behandlung von unbekannten Wörtern ein weiteres wichtiges Kriterium für die Performanz des Systems. Ist diese Möglichkeit in einem System nicht gegeben, kann es nur bedingt wiederverwendet werden, da in einem neuen Korpus das Auftreten neuer Wörter nicht ausgeschlossen werden kann, und somit die Erfolgsquote sinkt. Von den hier betrachteten Systemen verwenden CLAWS (Garside 1987), ASCOT (Akkerman et al. 1987), das Hidden-Markov-Modell von Cutting et al. (1992) und das System von Chanod und Tapanainen den Ansatz, unbekannte Wörter über Suffixlisten zu bestimmen; Voutilainen (1995) verwendet manuell erstellte Heuristiken. Diese Ansätze sind dementsprechend nicht mehr anwendbar, wenn das System auf eine andere Sprache angewandt werden soll. Weischedel et al. (1993) bestimmen die möglichen Tags für ein unbekanntes Wort über eine probabilistische Methode, die die vier Merkmale Flexionsendungen, Derivationsendungen, Schreibung mit Bindestrich und Großschreibung verwendet. In Brills (1993) System ist die Behandlung unbekannter Wörter eine Nebenerscheinung der in der kontext-freien Phase gelernten Regeln. Die beiden letzten Ansätze sind damit die einzigen, die ohne sprachabhängiges Wissen auskommen.

4.4.4. Wird ein Lexikon verwendet?

Eine weitere wichtige Frage stellt sich in bezug auf das Lexikon: Benötigt das System ein Lexikon? Wenn ein Lexikon benötigt wird, kann das einen großen Arbeitsaufwand bedeuten, sämtliche Wörter, die flektierten Formen und ihre Tags einzutragen. Der Aufwand ist auch davon abhängig, ob ein vollständiges Lexikon benötigt wird oder ob nur z.B. Funktionswörter berücksichtigt werden.
Fast alle in diesem Kapitel betrachteten Taggingsysteme verwenden ein Lexikon, wobei nur CGC (Klein, Simmons 1963) und TAGGIT lediglich Funktionswörter und irreguläre Wörter im Lexikon eintragen, bei CLAWS (Garside 1987) sind außerdem noch die häufigsten Verben, Nomen und Adjektive eingetragen. VOLSUNGA (DeRose 1988) dagegen verwendet ein vollständiges Lexikon, das aus dem Brown Korpus ermittelt wurde. Alle Systeme verwenden das Lexikon, um alle möglichen Tags eines Wortes zu ermitteln. In einer weiteren Phase soll das Wort dann disambiguiert werden. Nur bei Nakamura und Shikano (1989) wird kein Lexikon erwähnt, wobei das Interesse allerdings mehr auf dem Training des neuronalen Netzes liegt. Brill (1993) verwendet zwar auch ein Lexikon, das ebenfalls vom Anwender zur Verfügung gestellt werden kann, aber i.a. während des Lernvorgangs in der kontext-freien Phase erstellt wird. Es enthält auch nicht alle möglichen Tags sondern nur dasjenige, mit dem im Lernkorpus das entsprechende Wort am häufigsten annotiert war. Das bedeutet, daß auch beim externen Erstellen automatische Verfahren angewandt werden können.

4.4.5. Welche Informationen müssen dem System bereitgestellt werden?

Der Arbeitsaufwand, der entsteht, wenn ein System auf ein neues Korpus angepaßt werden soll, entsteht nicht nur aus der Erstellung eines Lexikons, sondern auch aus der Bereitstellung anderer linguistischer Information. Dies fängt bei den Suffixlisten der Systeme an, die auf diese Art unbekannte Wörter klassifizieren. Bei den regelbasierten Systemen müssen die Regeln bereitgestellt werden, bei den meisten probabilistischen Systemen die Matrix der Übergangswahrscheinlichkeiten. Eine Ausnahme bei den probabilistischen Systemen bildet das Taggingsystem von Cutting et al. (1992), bei dem laut Aussage der Autoren nur die Ambiguitätsklassen bereitgestellt werden müssen. Nach

Erfahrungen von Chanod und Tapanainen (1995) müssen allerdings auch hier die Übergangswahrscheinlichkeiten initialisiert werden, damit ein optimales Ergebnis erreicht wird. Bei den beiden neuronalen Netzen von Benello et al. (1989) und von Nakamura und Shikano (1989) müssen die Gewichtungen initialisiert werden. Auch in dieser Kategorie fällt bei Brills (1993) System kein Arbeitsaufwand an, hier müssen lediglich die maximale Affixlänge und die Regelschemata bereitgestellt werden. Dabei stellt das System selbst fest, welche Regelschemata auf das Korpus anwendbar sind und welche nicht, so daß nicht für jedes Regelschema geprüft werden muß, ob es negative Effekte bewirkt. Sollte dies der Fall sein, wird dieses Schema nicht benutzt.

4.4.6. Ist ein bereits getaggtes Textstück nötig?

Weiterer Arbeitsaufwand entsteht dadurch, daß verschiedene Systeme ein bereits getaggtes Korpus brauchen, um daraus die nötigen linguistischen Informationen zu beziehen. Dies gilt vor allem für probabilistische Systeme, die den getaggten Text brauchen, um die Matrix der Übergangswahrscheinlichkeiten zu berechnen. Doch auch Brills (1993) System und TAKTAG (Lee et al. 1995) gehören zu dieser Gruppe. Da es sehr aufwendig ist, größere Textmengen manuell zu taggen, bleibt entweder die Möglichkeit, ein bereits getaggtes Korpus als Lernkorpus zu verwenden, oder die benötigte Menge an getaggtem Text möglichst klein zu halten. Die zweite Möglichkeit fällt bei den probabilistischen Systemen jedoch weg, da sie eine gewisse Menge Text brauchen, um sinnvolle Wahrscheinlichkeiten zu ermitteln. DeRose (1988) und Church (1988) ver-wenden die 1.000.000 Wörter des Brown Korpus zum Training, Garside (1987), Marshall (1987) und Atwell (1987) verwenden für CLAWS 200.000 Wörter daraus, Weischedel et al. (1993) verwenden Teile des TREEBANK Korpus, sie testeten verschiedene Textmengen zwischen 64.000 und 1.000.000 Wörtern, wobei allerdings bei 64.000 Wörtern eine deutliche Verschlechterung der Performanz zu bemerken war. In TAKTAG (Lee et al. 1995) sind nicht die Wörter, sondern die Morpheme ausschlaggebend, hier werden insgesamt 12.500 getaggte Morpheme verwendet, 2.000 zur Initialisierung des Hidden-Markov-Modells und 10.500 für die Fehlerkorrektur. Brill (1993) verwendet 47.000 Wörter, also deutlich weniger als die anderen Systeme.

4.4.7. Ist die im System gewonnene linguistische Information einsehbar?

Dies ist ein Bewertungskriterium, das nicht überall Anwendung findet, denn nicht in allen Systemen wird neue linguistische Information gewonnen. In den regelbasierten Systemen, bei denen die linguistischen Information bereitgestellt wird, ist sie selbstverständlich einsehbar. In den probabilistischen Systemen und den neuronalen Netzen jedoch wird die Information zum größten Teil erst durch das Training erworben, die manuell eingesetzten Anfangswerte sind nur Näherungen. In diesen Systemen liegt dann das neu gewonnene Wissen in Form von Übergangswahrscheinlichkeiten bzw. Verbindungsgewichten vor und kann somit anderweitig nicht genutzt werden. In Brills (1993) System dagegen wird die Information ebenfalls erworben, sie wird aber in Regeln abgelegt. Die Information ist zwar nicht in natürlicher Sprache sondern in formalen Regeln vorhanden, sie könnte durch ein entsprechendes Modul aber auch in natürlicher Sprache formuliert werden. Sie liegt jedoch zweifellos in einer Form vor, die auch anderweitig, z.B. in einer Grammatik verwendet werden.

4.4.8. Wird eine Lemmatisierung durchgeführt?

Ein weiteres Kriterium für die Bewertung eines Tagginsgsystems ergibt sich aus der Frage, ob eine Lemmatisierung durchgeführt wird. Die Lemmatisierung ist z.B. nötig, wenn als Teil der Analyse auch eine Semantikkomponente integriert ist. Natürlich kann die Lemmatisierung auch unabhängig von der Taggingkomponente erfolgen, das wäre jedoch doppelte Arbeit. Allerdings wird eine Lemmatisierung nicht in allen hier besprochenen Systemen durchgeführt. Dies ist damit zu erklären, daß die meisten Taggingsysteme die benötigte Information aus den flektierten Wortformen entnehmen. Lediglich ASCOT (Akkerman et al. 1987), das System von Cutting (1992), das von Voutilainen (1995) und das von Chanod und Tapanainen (1995) führen eine vollständige Lemmatisierung durch. In TAKTAG ist eine Lemmatisierung im eigentlichen Sinn nicht sinnvoll, hier werden die Eojeols auf Basismorpheme zurückgeführt. Auch bei Brill (1993) wird keine Lemmatisierung durchgeführt, da die Regeln bei flektierten Wortformen ansetzen und für eine Lemmatisierung zusätzliche linguistische Information, z.B. in Form eines Stammformenlexikons, zur Verfügung gestellt werden müßte.

4.5. Zusammenfassung

Die hier vorgestellten Systeme werden von den Autoren durchgehend als sehr erfolgreich beschrieben. Wie sich hier gezeigt hat, haben jedoch die meisten Systeme ihre Nachteile. Einige Schwachstellen eines Systems lassen sich jedoch nicht aus einer theoretischen Beschreibung erkennen, sondern werden erst bei der Implementierung und der Erprobung deutlich. Der transformationsbasierte Ansatz zur Wortklassifizierung von Brill (1992, 1993, 1994) zählt mit seiner Genauigkeitsquote von 95,3% nicht unter die besten der hier vorgestellten Taggingsysteme. Allerdings sind die beiden neueren Ansätze, die die '97%-Barriere' überschritten haben, nur an kleinen Textmengen getestet worden, die Bewährung an größeren Korpora muß erst noch erfolgen. Brills (1992, 1993, 1994) Ansatz besitzt jedoch den Vorteil, daß kaum linguistische Information zur Verfügung gestellt werden muß und daß bereits ein kleines Lernkorpus ausreicht, um diese Erfolgsquote zu erreichen. Es kann jedoch spekuliert werden, daß sich die Ergebnisse noch durch ein größeres Lernkorpus verbessern lassen, wozu es allerdings noch keine empirischen Daten gibt. Genaue Erkenntnisse können jedoch nur durch eine Erprobung erreicht werden.

5. Fazit

Das transformationsbasierte fehlergesteuerte Lernverfahren zur automatischen Wortklassifizierung ist ein System, das verschiedene Vorteile in sich vereint: Es ist zuverlässig, benötigt nur ein Minimum an linguistischer Information, es kann mit geringem Trainingsaufwand auf verschiedene Textsorten übertragen werden, es besitzt, theoretisch betrachtet, nur lineare Komplexität und die gelernten Regeln können auch anderweitig weiterverwendet werden. Oft stellt sich jedoch in der Praxis heraus, daß sich trotz der erfreulichen theoretischen Systemeigenschaften bei der Implementierung bzw. beim Anwenden des Systems Probleme ergeben können. Bei der hier vorliegenden Implementierung von Brills (1993) System ergaben sich die Probleme daraus, daß in der kontextfreien Phase eine unübersichtlich große Anzahl von Regelkandidaten gebildet werden muß, die dann in allen weiteren Schritten nochmals durchsucht werden, so daß keine Doppelteinträge entstehen. Dadurch wird das Verfahren sowohl speicher- als auch rechenzeitaufwendig.
Trotzdem sollte dieses System deswegen nicht verworfen werden, zumal kaum ein anderes System mit so vielen Vorteilen aufwarten kann. Vielmehr sollten Überlegungen angestellt werden, wie die Bildung der Regelkandidaten beschränkt werden könnte, so daß nicht eine Unzahl von Regelkandidaten entsteht, von denen viele keine Chance auf Erfolg haben. Eine andere Möglichkeit, die Effizienz des Systems zu steigern, liegt in der frühzeitigen Aussortierung nicht sehr erfolgversprechender Regelkandidaten. Dies würde die benötigte CPU-Zeit beträchtlich einschränken, da die möglichen Mehrfachbildungen im derzeitigen System auch erst durch den Vergleich mit den bestehenden Regelkandidaten aussortiert werden. Es ist jedoch fraglich, ob derartige Restriktionen formuliert werden können, ohne daß weitere linguistische Information eingebaut werden muß.

Anhang A: Das Tagset

Die hier verwendeten Tags entsprechen weitgehend den in Celex© gebrauchten Bezeichnungen.

nS:M - Substantiv, Nominativ, Singular, Maskulin
nS:F - Substantiv, Nominativ, Singular, Feminin
nS:N - Substantiv, Nominativ, Singular, Neutrum
gS:M - Substantiv, Genitiv, Singular, Maskulin
gS:F - Substantiv, Genitiv, Singular, Feminin
gS:N - Substantiv, Genitiv, Singular, Neutrum
dS:M - Substantiv, Dativ, Singular, Maskulin
dS:F - Substantiv, Dativ, Singular, Feminin
dS:N - Substantiv, Dativ, Singular, Neutrum
aS:M - Substantiv, Akkusativ, Singular, Maskulin
aS:F - Substantiv, Akkusativ, Singular, Feminin
aS:N - Substantiv, Akkusativ, Singular, Neutrum
nS:pM - Substantiv, Nominativ, Singular, Maskulin
nS:pF - Substantiv, Nominativ, Singular, Feminin
gS:pM - Substantiv, Genitiv, Singular, Personenname, Maskulin
gS:pF - Substantiv, Genitiv, Singular, Personenname, Feminin
dS:pM - Substantiv, Dativ, Singular, Personenname, Maskulin
dS:pF - Substantiv, Dativ, Singular, Personenname, Feminin
aS:pM - Substantiv, Akkusativ, Singular, Personenname, Maskulin
aS:pF - Substantiv, Akkusativ, Singular, Personenname, Feminin
nS:oM - Substantiv, Nominativ, Singular, Ortsname, Maskulin
nS:oF - Substantiv, Nominativ, Singular, Ortsname, Feminin
nS:oN - Substantiv, Nominativ, Singular, Ortsname, Neutrum
gS:oM - Substantiv, Genitiv, Singular, Ortsname, Maskulin
gS:oF - Substantiv, Genitiv, Singular, Ortsname, Feminin
gS:oN - Substantiv, Genitiv, Singular, Ortsname, Neutrum
dS:oM - Substantiv, Dativ, Singular, Ortsname, Maskulin
dS:oF - Substantiv, Dativ, Singular, Ortsname, Feminin
dS:oN - Substantiv, Dativ, Singular, Ortsname, Neutrum
aS:oM - Substantiv, Akkusativ, Singular, Ortsname, Maskulin
aS:oF - Substantiv, Akkusativ, Singular, Ortsname, Feminin
aS:oN - Substantiv, Akkusativ, Singular, Ortsname, Neutrum
nS:aM - Substantiv, Nominativ, Singular, sonstiger Name, Maskulin
nS:aF - Substantiv, Nominativ, Singular, sonstiger Name, Feminin
nS:aN - Substantiv, Nominativ, Singular, sonstiger Name, Neutrum
gS:aM - Substantiv, Genitiv, Singular, sonstiger Name, Maskulin
gS:aF - Substantiv, Genitiv, Singular, sonstiger Name, Feminin
gS:aN - Substantiv, Genitiv, Singular, sonstiger Name, Neutrum
dS:aM - Substantiv, Dativ, Singular, sonstiger Name, Maskulin
dS:aF - Substantiv, Dativ, Singular, sonstiger Name, Feminin
dS:aN - Substantiv, Dativ, Singular, sonstiger Name, Neutrum
aS:aM - Substantiv, Akkusativ, Singular, sonstiger Name, Maskulin
aS:aF - Substantiv, Akkusativ, Singular, sonstiger Name, Feminin
aS:aN - Substantiv, Akkusativ, Singular, sonstiger Name, Neutrum
nP:M - Substantiv, Nominativ, Plural, Maskulin
nP:F - Substantiv, Nominativ, Plural, Feminin
nP:N - Substantiv, Nominativ, Plural, Neutrum
gP:M - Substantiv, Genitiv, Plural, Maskulin
gP:F - Substantiv, Genitiv, Plural, Feminin
gP:N - Substantiv, Genitiv, Plural, Neutrum
dP:M - Substantiv, Dativ, Plural, Maskulin
dP:F - Substantiv, Dativ, Plural, Feminin

dP:N - Substantiv, Dativ, Plural, Neutrum
aP:M - Substantiv, Akkusativ, Plural, Maskulin
aP:F - Substantiv, Akkusativ, Plural, Feminin
aP:N - Substantiv, Akkusativ, Plural, Neutrum
nP:T - Substantiv, Nominativ, Plurale Tantum
gP:T - Substantiv, Genitiv, Plurale Tantum
dP:T - Substantiv, Dativ, Plurale Tantum
aP:T - Substantiv, Akkusativ, Plurale Tantum
nP:MF - Substantiv, Nominativ, Maskulin/Feminin z.B. MitarbeiterInnen
gP:MF - Substantiv, Genitiv, Maskulin/Feminin
dP:MF - Substantiv, Dativ, Maskulin/Feminin
aP:MF - Substantiv, Akkusativ, Maskulin/Feminin
nP:pM - Substantiv, Nominativ, Plural, Personenname, Maskulin
nP:pF - Substantiv, Nominativ, Plural, Personenname, Feminin
gP:pM - Substantiv, Genitiv, Plural, Personenname, Maskulin
gP:pF - Substantiv, Genitiv, Plural, Personenname, Feminin
dP:pM - Substantiv, Dativ, Plural, Personenname, Maskulin
dP:pF - Substantiv, Dativ, Plural, Personenname, Feminin
aP:pM - Substantiv, Akkusativ, Plural, Personenname, Maskulin
aP:pF - Substantiv, Akkusativ, Plural, Personenname, Feminin
nP:oF - Substantiv, Nominativ, Ortsname, Plural, Feminin
nP:oN - Substantiv, Nominativ, Ortsname, Plural, Neutrum
gP:oN - Substantiv, Genitiv, Ortsname, Plural, Neutrum
dP:oM - Substantiv, Dativ, Ortsname, Plural, Maskulin
dP:oN - Substantiv, Dativ, Ortsname, Plural, Neutrum
aP:oN - Substantiv, Akkusativ, Ortsname, Plural, Neutrum
nP:aM - Substantiv, Nominativ, sonstiger Name, Plural, Maskulin
nP:oT - Substantiv, Nominativ, Ortsname, Plurale Tantum
gP:oT - Substantiv, Genitiv, Ortsname, Plurale Tantum
dP:oT - Substantiv, Dativ, Ortsname, Plurale Tantum
aP:oT - Substantiv, Akkusativ, Ortsname, Plurale Tantum
nP:aT - Substantiv, Nominativ, sonstiger Name, Plurale Tantum

o0:A - Adjektiv, Positiv, nicht flektiert
o4:A - Adjektiv, Positiv, e-Endung
o5:A - Adjektiv, Positiv, en-Endung
o6:A - Adjektiv, Positiv, er-Endung
o7:A - Adjektiv, Positiv, em-Endung
o8:A - Adjektiv, Positiv, es-Endung
o9:A - Adjektiv, Positiv, s-Endung
c0:A - Adjektiv, Komparativ, nicht flektiert
c4:A - Adjektiv, Komparativ, e-Endung
c5:A - Adjektiv, Komparativ, en-Endung

c6:A - Adjektiv, Komparativ, er-Endung
c7:A - Adjektiv, Komparativ, em-Endung
u0:A - Adjektiv, Superlativ, nicht flektiert
u4:A - Adjektiv, Superlativ, e-Endung
u5:A - Adjektiv, Superlativ, en-Endung
u6:A - Adjektiv, Superlativ, er-Endung
u7:A - Adjektiv, Superlativ, em-Endung
u8:A - Adjektiv, Superlativ, es-Endung

1SIE:V - Verb, 1. Person, Singular, Indikativ, Präsens
2SIE:V - Verb, 2. Person, Singular, Indikativ, Präsens
3SIE:V - Verb, 3. Person, Singular, Indikativ, Präsens
13PIE:V - Verb, 1./3. Person, Plural, Indikativ, Präsens
2PIE:V - Verb, 2. Person, Plural, Indikativ, Präsens
13SIA:V - Verb, 1./3. Person, Singular, Indikativ, Vergangenheit
13PIA:V - Verb, 1./3. Person, Plural, Indikativ, Vergangenheit
13SKE:V - Verb, 1./3. Person, Singular, Konjunktiv, Präsens
13PKE:V - Verb, 1./3. Person, Plural, Konjunktiv, Präsens
13SKA:V - Verb, 1./3. Person, Singular, Konjunktiv, Vergangenheit
2SKA:V - Verb, 2. Person, Singular, Konjunktiv, Vergangenheit
13PKA:V - Verb, 1./3. Person, Plural, Konjunktiv, Vergangenheit
2PKA:V - Verb, 2. Person, Plural, Konjunktiv, Vergangenheit
i:V - Verb, Infinitiv
z:V - Verb, Infinitiv mit wortinternem zu
rS:V - Verb, Imperativ, Singular
rP:V - Verb, Imperativ, Plural
pE:V - Verb, Partizip, Präsens
pA:V - Verb, Partizip, Vergangenheit

Smn:R - Artikel, Singular, Maskulin, Nominativ
Smg:R - Artikel, Singular, Maskulin, Genitiv
Smd:R - Artikel, Singular, Maskulin, Dativ
Sma:R - Artikel, Singular, Maskulin, Akkusativ
Swna:R - Artikel, Singular, Feminin, Nominativ/Akkusativ
Swgd:R - Artikel, Singular, Feminin, Genitiv/Dativ
Ssna:R - Artikel, Singular, Neutrum, Nominativ/Akkusativ
Ssg:R - Artikel, Singular, Neutrum, Genitiv
Ssd:R - Artikel, Singular, Dativ
Pmwsna:R - Artikel, Plural, Maskulin/Feminin/Neutrum, Nominativ/Akkusativ
Pmwsg:R - Artikel, Plural, Maskulin/Feminin/Neutrum, Genitiv
Pmwsd:R - Artikel, Plural, Maskulin/Feminin/Neutrum, Dativ

0:O - Indefinitpronomen, Grundform
n:O - Personalpronomen, Nominativ
g:O - Personalpronomen, Genitiv
d:O - Personalpronomen, Dativ
a:O - Personalpronomen, Akkusativ
Smn:O - Demonstrativ-/Relativpronomen,Singular, Maskulin, Nominativ
Smg:O - Demonstrativ-/Relativpronomen, Singular, Maskulin, Genitiv
Smd:O - Demonstrativ-/Relativpronomen, Singular, Maskulin, Dativ

Sma:O - Demonstrativ-/Relativpronomen, Singular, Maskulin, Akkusativ
Swna:O - Demonstrativ-/Relativpronomen, Singular, Feminin, Nominativ/Akkusativ
Swgd:O - Demonstrativ-/Relativpronomen, Singular, Feminin, Genitiv/Dativ
Ssna:O - Demonstrativ-/Relativpronomen, Singular, Neutrum, Nominativ/Akkusativ
Ssg:O - Demonstrativ-/Relativpronomen, Singular, Neutrum, Genitiv
Ssd:O - Demonstrativ-/Relativpronomen, Singular, Neutrum, Dativ
Pmwsna:O - Demonstrativ-/Relativpronomen, Plural, Maskulin/Feminin/Neutrum, Nominativ/Akkusativ
Pmwsg:O - Demonstrativ-/Relativpronomen, Plural, Maskulin/Feminin/Neutrum, Genitiv
Pmwsd:O - Demonstrativ-/Relativpronomen, Plural, Maskulin/Feminin/Neutrum, Dativ
0:O - Possessiv-/Interrogativpronomen, Grundform
4:O - Possessiv-/Interrogativpronomen, e-Endung
5:O - Possessiv-/Interrogativpronomen, en-Endung
6:O - Possessiv-/Interrogativpronomen, er-Endung
7:O - Possessiv-/Interrogativpronomen, em-Endung
8:O - Possessiv-/Interrogativpronomen, es-Endung
X:O - Pronomen, Grundform

0:U - Numeral, Grundform
4:U - Numeral, e-Endung
5:U - Numeral, en-Endung
6:U - Numeral, er-Endung
7:U - Numeral, em-Endung
8:U - Numeral, es-Endung
9:U - Numeral, s-Endung

X:P - Präposition, Grundform
X:PR - Präposition + Artikel
X:C - Konjunktion, Grundform
X:D - Adverb, Grundform
X:J - Interjektion, Grundform
X:I - Interpunktion, Grundform

Anhang B - Das Programmlisting

```
/*************************************************************************/
/*                                                                       */
/*                        Datei: TAGGER.CC                               */
/*                                                                       */
/*************************************************************************/


#include "global.h"

/*
    Dieses Programm ist ein part-of-speech Tagger, der in drei Stufen
    arbeitet; in der ersten Stufe wird mittels kontext-freier Templates
    fuer jedes Wort das wahrscheinlichste Tag gesucht, in der zweiten
    Stufe werden kontex-sensitive Templates eingesetzt, um fuer jedes
    einzelne Wort das richtige Tag zu finden. In der dritten Stufe werden
    die Regeln eingesetzt, die in den ersten zwei Stufen gefunden wurden,
    um ein Testkorpus zu taggen.
*/

void main()
{
  CFrule cf_rules[500];
  CSrule cs_rules[500];
   char tagged1[15], tagged2[15], untagged[15], testtagged[15], dummy;
  int nrcfrules, nrcsrules;

  cout<<"Aus welcher Datei soll das kontext-freie getaggte Korpus eingelesen werden?\n";
  cin.get(tagged1, 15, '\n');
  cin>>dummy;
  cout<<"Aus welcher Datei soll das kontext-sensitive getaggte Korpus eingelesen werden?\n";
  cin.get(tagged2, 15, '\n');
  cin>>dummy;
  cout<<"Aus welcher Datei soll das ungetaggte Korpus gelesen werden?\n";
  cin.get(untagged, 15, '\n');
  cin>>dummy;
  cout<<"Aus welcher Datei soll das Testkorpus gelesen werden?\n";
  cin.get(testtagged, 15, '\n');
  cin>>dummy;
  cout<<"Soll statistische Information ausgegeben werden: Bitte geben Sie eine Zahl zwischen
0 und 3 ein.\n 0 = nein\n 1 = nur die Anzahl der falsch getaggten Woerter\n 2 = nur die Anzahl
der nicht im Lexikon enthaltenen Woerter\n 3 = beides\n";
  cin>>stat_flag;
  cin>>dummy;
  if (stat_flag > 3)
    error("Falsche Angabe fuer Statistik.\n");
   nrcfrules=cf_tagger(tagged1, untagged, cf_rules);
   nrcsrules=cs_tagger(tagged2, untagged, cf_rules, nrcfrules, cs_rules);
   apply_rules(testtagged, untagged, cf_rules, nrcfrules, cs_rules, nrcsrules);
}   //end main
```

```
/*************************************************************************/
/*                                                                       */
/*                          Datei: BRILL92.CC                            */
/*                                                                       */
/*************************************************************************/


#include "global.h"

void tag_corp();
int initialize_rules(CSrule cs_rules[]);


/*
   Dies ist die Version des Taggers, wie sie in Brill(1992) vorgestellt wurde.
   Hier wird in der ersten Phase fuer jedes Wort das am haeufigsten vorkommende
   Tag ermittelt. Fuer die zweite Phase wurden hier jedoch nicht die Templates
   von Brill(1992) verwendet, sondern diejenigen, die auch in Brill (1993)
   verwendet werden.
*/

void main()
{
  CSrule cs_rules[800];
  int nrtempl, nrrules;
  CStemplate templates[50];
  char infile[31], outfile[31], tmp[31];

  strcpy(infile, "temp1.dat");
  strcpy(outfile, "temp2.dat");
  cout<<"Lexikon einlesen ...\n";
  cout.flush();
  initialize_lexicon();
  fill_lexicon("data/lern2.dat");
  for (int wordcount=0; wordcount<maxword; wordcount++)
    if (!taglex[wordcount].is_empty() && !taglex[wordcount].no_texttag()
          && !taglex[wordcount].correct_tag())
      taglex[wordcount].liketag();
  safe_lexicon();
  cout<<"Korpus initialisieren ... \n";
  cout.flush();
  tag_corp();
  delete [] taglex;
  cout<<"bereits gelernte Regeln anwenden ... \n";
  cout.flush();
  nrtempl=read_cstemplates(templates);
  nrrules=initialize_rules(cs_rules);
  for (int i=0; i<nrrules; i++)
   {
      cout<<"Regel: "<<i<<'\n';
      cout.flush();
      cs_rules[i].apply_rule(infile, outfile);
      strcpy(tmp, infile);
      strcpy(infile, outfile);
      strcpy(outfile, tmp);
   }   //end for
  cout<<"neue Regeln finden ...\n";
  cout.flush();
```

```
  nrrules=find_tag_in_context(templates, nrtempl, cs_rules, infile, outfile, nrrules);
  safe_csrules(cs_rules, nrrules);
}     //end main


/*
   tag_corp traegt fuer jedes im Lexikon gefundene Wort das haeufigste Tag
   ein; nicht im Lexikon gefundene Woerter werden mit einem Default-Tag
   annotiert.
   Nebeneffekt: die annotierten Woerter werden in die Datei 'temp1.dat'
                        geschrieben
*/

void tag_corp()
{
  char word[wordlen], tag[taglen], mytag[taglen];
  int unknown=0;
  ifstream In;
  ofstream Out;

  all_words=0;
  In.open("data/lern2.dat");
  Out.open("temp1.dat");
  while (!In.eof())
   {
     read_word(In, word, tag, mytag);
     if (strcmp(word, ""))
           {
             all_words++;
             if (hash(word, mytag) < 0)
               unknown++;
             Out<<word<<'|'<<tag<<'|'<<mytag<<'\n';
           }   //end if
   }   //end while
  In.close();
  Out.close();
   cout<<"unbekannt: "<<unknown<<'\n';
}   //end tag_corp


/*
   initialize_rules liest die bereits gefundenen kontext-sensitiven Regeln
   in die Liste 'cs_rules' ein.
   rein+raus: cs_rules - die Liste der kontext-sensitiven Regeln
   raus: die Anzahl der Regeln
   Nebeneffekt: die Liste 'cs_rules' wird gefuellt
*/

int initialize_rules(CSrule cs_rules[])
{
  ifstream In;
  int i;

  In.open("csrules.dat");
  for (i=0; i<800 && !In.eof(); i++)
    cs_rules[i].read_content(In);
  In.close();
  return(i-1);
```

```
}  //end initialize_rules
```

```
/*************************************************************************/
/*                                                                       */
/*                          Datei: GLOBAL.H                              */
/*                                                                       */
/*************************************************************************/


#include <stdlib.h>
#include <iostream.h>
#include <fstream.h>
#include <string.h>
#include <ctype.h>
#include <math.h>

const maxword = 10007;
const wordlen = 41;
const taglen = 11;
const maxtag = 20;
const maxfreq = 200;
const affixlen = 4;
const maxpos = 100;
int maxpostag=0;
int stat_flag = 1;
int all_words;
char vowels[8][4];
char tagset[200][taglen];

struct cf_rulecand
{
  char action, where, from[taglen], to[taglen], affix[wordlen], word[wordlen];
  float score;
  int candnr;
  cf_rulecand *next;
};   //end struct

struct cs_rulecand
{
  char from[taglen], to[taglen], neighbor1[taglen], neighbor2[taglen], where, how;
  int read_nr, inv_nr;
  float score;
  cs_rulecand *next;
};    //end struct

struct freqlist
{
  char word[wordlen];
  int countnr;
};   //end struct

struct pos_word
{
 char word[wordlen], affix[affixlen];
};   //end struct

struct wordlist
{
  char word[wordlen], tag[taglen], texttag[taglen];
};   //end struct
```

```
struct notaglist
{
  char word[wordlen];
  notaglist *next;
};   //end struct


#include "utils.h"
#include "tag.h"
#include "tag.cc"

Tag taglex[maxword];

#include "bigram.cc"
#include "hash.cc"
#include "cftempl.h"
#include "cfrule.h"
#include "cfcand.h"
#include "cftempl.cc"
#include "cfrule.cc"
#include "cfcand.cc"
#include "cstempl.h"
#include "csrule.h"
#include "cscand.h"
#include "cstempl.cc"
#include "csrule.cc"
#include "cscand.cc"
#include "cftag.cc"
#include "cstag.cc"
#include "apply.cc"
```

```
/*************************************************************************/
/*                                                                       */
/*                        Datei: CFTAG.CC                                */
/*                                                                       */
/*************************************************************************/


void initialize_lexicon();
void initialize_frequents(freqlist frequents[]);
void initialize_vowels();
void initialize_tagset();
void read_lexicon(char* infile);
void fill_lexicon(char* tagged);
void find_frequents(freqlist frequents[]);
void insert_frequents(freqlist frequents[], Tag entry);
void sort_frequents(freqlist frequents[]);
void safe_cfrules(CFrule cf_rules[], int nrrules);
void safe_lexicon();
void tag_closed_classes();
int closed_class(char tag[]);
int find_most_likely_tag(char* untagged, CFrule cf_rules[], freqlist frequents[]);
int read_cftemplates(CFtemplate templates[]);
CFcandidate adjust_score(CFcandidate candidates, freqlist frequents[], char* untagged);
int find_bestrule(cf_rulecand candidates[], int nrcand);
int cf_tag_corpus(char* tagged1, char* tagged2, char* untagged, CFrule cf_rules[], int
nrcfrules);
void apply_cfrules(char word[], char tag[], CFrule cf_rules[], int nrrules, char* untagged);


/*
    cf_tagger erstellt ein Lexikon mit allen im kontext-freien Lernkorpus
    enthaltenen Woertern und den wahrscheinlichsten Tags. Das wahr-
    scheinlichste Tag fuer ein Wort wird durch das Erstellen von Regeln
     nach der transformationsbasierten fehlergesteuerten Methode gefunden.
    rein+raus: cf_rules - die Liste der kontext-freien Regeln
    rein: tagged - der Name des kontext-freien Lernkorpus
              untagged - der Name des ungetaggten Korpus
    raus: Anzahl der gefundenen Regeln
    Nebeneffekt: das Lexikon; die Liste der Bigramme und die Liste der
                       kontext-freien Regeln werden gefuellt
*/

int cf_tagger(char* tagged, char* untagged, CFrule cf_rules[])
{
  freqlist frequents[maxfreq];
  char prevlex[4], dummy;
  int nrrules;

  initialize_lexicon();
   initialize_frequents(frequents);
  initialize_vowels();
  initialize_tagset();
//  cout<<"Soll ein bereits vorhandenes Lexikon verwendet werden? [ja/nein]\n";
//  cin.get(prevlex, 5, '\n');
//  if (strcmp(prevlex, "ja") == 0)
//     read_lexicon("lexicon.dat");
   cout<<"\nDas Lernkorpus wird ins Lexikon eingelesen ...\n";
   cout.flush();
```

```
  fill_lexicon(tagged);
  cout<<"Die Liste der haeufigsten Woerter wird erstellt ...\n";
  cout.flush();
   find_frequents(frequents);
  tag_closed_classes();
  cout<<"Die bigrams werden erstellt ... \n";
  cout.flush();
  bigrams(untagged);
  untagged="bigrams.dat";
  cout<<"Die Hypothesen werden erstellt ...\n";
  cout.flush();
  nrrules=find_most_likely_tag(untagged, cf_rules, frequents);
  safe_cfrules(cf_rules, nrrules);
  safe_lexicon();
  if (stat_flag==1 || stat_flag==3)
   {
      cout<<"Die Statistik nach der ersten Komponente wird erstellt ...\n";
      cout.flush();
       cf_tag_corpus(tagged, "temp.dat", untagged, cf_rules, nrrules);
       cout<<"Nach der kontext-freien Komponente ...\n";
      cout.flush();
      statistics("temp.dat");
   }   //end if
  return(nrrules);
}   //end cf_tagger


/*
    initialize_lexicon initialisiert das Lexikon mit dem leeren String als
    Woerter.
    Nebeneffekt: das Lexikon wird mit leeren Strings initialisiert
*/


void initialize_lexicon()
{
  for (int i=0; i<maxword; i++)
   {
      taglex[i].put_word("");
      taglex[i].initialize_texttags();
    }   //end for
}   //end initialize_lexicon


/*
   initialize_frequents initialisiert die Liste der haeufigsten Woerter
   mit 0 als Anzahl der Woerter.
   rein+raus: frequents - die Liste der haeufigsten Woerter
   Nebeneffekt: jeder Eintrag in der Liste erhaelt die Anzahl 0 als
                        Initialisierung
*/

void initialize_frequents(freqlist frequents[])
{
  for (int i=0; i<maxfreq; i++)
     frequents[i].countnr=0;
}   //end initialize_frequents
```

```
/*
   initialize_vowels initialisiert die Liste der Vokale.
   Nebeneffekt: die Vokale werden in die Liste 'vowels' eingetragen
*/

void initialize_vowels()
{
  strcpy(vowels[0], "a");
  strcpy(vowels[1], "e");
  strcpy(vowels[2], "i");
  strcpy(vowels[3], "o");
  strcpy(vowels[4], "u");
  strcpy(vowels[5], "\\\"a");
  strcpy(vowels[6], "\\\"o");
  strcpy(vowels[7], "\\\"u");
}   //end initialize_vowels


/*
   initialize_tagset initialisiert die Liste der verwendeten Tags. Sie
   werden aus der Datei 'tagset.dat' eingelesen.
   Nebeneffekt: die Tags werden in die Liste 'tagset' eingetragen
*/

void initialize_tagset()
{
  ifstream In;
  char dummy;

   In.open("tagset.dat");
  while(!In.eof())
   {
      In.get(tagset[maxpostag], taglen, '\n');
     maxpostag++;
     In.get(dummy);
   }   //end for
  In.close();
}   //end initialize_tagset


/*
    read_lexicon liest bereits vorhandene Informationen fuer das Lexikon aus
    der Datei 'lexicon.dat' ein.
    rein: infile - die datei, aus der gelesen wird
    Nebeneffekt: das Lexikon wird gefuellt
*/

void read_lexicon(char* infile)
{
  Tag entry;
  char word[wordlen], tag[taglen], texttag[taglen], dummy;
  int lexcount=0, tagcount;
  ifstream In;

  In.open(infile);
  while(!In.eof())
   {
```

```
      if (lexcount >= maxword)
            error("Nicht genuegend Platz im Lexikon.\n");
      entry.initialize_texttags();
      In.get(word, wordlen, ' ');
      In.get(dummy);
      In.get(tag, taglen, ' ');
      In.get(dummy);
      entry.initialize(word, tag, "");
      while (!(dummy == '\n'))
            {
              In.get(texttag, taglen, ' ');
              In.get(dummy);
              In>>tagcount;
              In.get(dummy);
               entry.put_texttag_and_num(texttag, tagcount);
            }   //end while-2
      lexcount=hash_insert(entry, lexcount);
    }   //end while-1
  In.close();
}   //end read_lexicon


/*
    fill_lexicon erstellt das Lexikon mit den Woertern aus dem kontext-freien
    Lernkorpus.
    rein: tagged - der Name des kontext-freien Lernkorpus
    Nebeneffekt: das Lexikon wird mit den Woertern, ihren anfaenglichen Tags
                        und den im Text gefundenen Tags sowie ihren Auftretens-
                        haeufikeiten gefuellt
*/

void fill_lexicon(char* tagged)
{
  Tag entry;
  char word[wordlen], tag[taglen], texttag[taglen];
  int lexcount=0;
  ifstream In;

  In.open(tagged);
  while (!In.eof())
   {
      entry.initialize_texttags();
      read_word(In, word, texttag, tag);
      entry.initialize(word, tag, texttag);
      if (isprint(word[0]))
            lexcount=hash_insert(entry, lexcount);
    }   //end while
  In.close();
}   //end fill_lexicon


/*
    find_frequents sucht die 200 Woerter aus der kontext-freien Korpus heraus,
    die am haeufisten vorkommen. Diese sind Kandidaten fuer Nachbarwoerter fuer
    die Nachbarregel. Zur schnelleren Bearbeitung wird im Lexikon gesucht, das
    alle Woerter aus dem kontext-freien Korpus und ihre Haeufigkeit enthaelt.
    Die Liste ist so organisiert, dass immer das Wort mit der geringsten
    Haeufigkeit an erster Stelle steht. Nachdem die Liste feststeht, wird sie
```

```
   in sort_frequents alphabetisch aufsteigend sortiert.
   rein+raus: frequents - die Liste der 200 haeufigsten Woerter
   Nebeneffekt: die Liste 'frequents' wird gefuellt
*/

void find_frequents(freqlist frequents[])
{
  int found=0;

  for (int i=0; i<maxword; i++)
   {
      if (!taglex[i].is_empty() &&
             (taglex[i].wordcount() > frequents[0].countnr))
             insert_frequents(frequents, taglex[i]);
   }   //end for
  cout<<"sortieren ... \n";
  cout.flush();
  sort_frequents(frequents);
}   //end find_frequents


/*
   insert_frequents setzt ein Wort so in die Liste der haeufigsten Woerter
   ein, dass das Wort, das bisher die kleinste Haeufigkeit aufwies, rausfaellt,
   und dass das Wort, das danach die geringste Haeufigkeit besitzt, an erster
   Stelle steht.
   rein+raus: frequents die Liste der haeufigsten Woerter
   rein: entry - ein Lexikoneintrag
*/

void insert_frequents(freqlist frequents[], Tag entry)
{
  int count=10000, pos, wordcount;
  char word[wordlen];

   entry.output_word(word);
   wordcount=entry.wordcount();
  for (int i=1; i<maxfreq; i++)
    if (frequents[i].countnr < count)
    {
           pos=i;
           count=frequents[i].countnr;
     }   //end if
  if (wordcount > count)
   {
      strcpy(frequents[0].word, frequents[pos].word);
       frequents[0].countnr=frequents[pos].countnr;
      entry.output_word(frequents[pos].word);
      frequents[pos].countnr = wordcount;
   }   //end if
  else
   {
     strcpy(frequents[0].word, word);
     frequents[0].countnr = wordcount;
   }   //end else
}   //end insert_freq
```

```
/*
   sort_frequents sortiert die Liste der haeufigsten Woerter alphabetisch in
   aufsteigender Reihenfolge.
   rein+raus: frequents - die Liste der haeufigsten Woerter
*/

void sort_frequents(freqlist frequents[])
{
  int k, count;
  char word[wordlen];

  for (int i=0; i<maxfreq-1; i++)
    for (int j=i+1; j<maxfreq; j++)
    {
          for (k=0; frequents[i].word[k] == frequents[j].word[k]; k++);
          if (strcmp(frequents[j].word, "") &&
             ((int(frequents[i].word[k]) > int(frequents[j].word[k])) ||
              !frequents[i].countnr))
           {
             strcpy(word, frequents[i].word);
             count = frequents[i].countnr;
             strcpy(frequents[i].word, frequents[j].word);
             frequents[i].countnr = frequents[j].countnr;
             strcpy(frequents[j].word, word);
             frequents[j].countnr = count;
           }   //end if
     }   //end for
}   //end sort_frequents


/*
    safe_cfrules speichert die gefundenen kontext-freien Regeln in der Datei
    'cfrules.dat' ab.
    rein: cf_rules - die Liste der kontext-freien Regeln
             nrrules - Anzahl der Regeln
    Nebeneffekt: die Regeln werden in die Datei 'cfrules.dat' geschrieben
*/

void safe_cfrules(CFrule cf_rules[], int nrrules)
{
  int i;
  ofstream Out;

   Out.open("cfrules.dat");
  for (i=0; i<nrrules; i++)
     cf_rules[i].safe_content(Out);
  Out.close();
}   //end safe_cfrules


/*
    safe_lexicon speichert die Lexikoneintraege in die Datei 'lexicon.dat' ab.
    Nebeneffekt: die Datei 'lexicon.dat' wird mit den Lexikoneintraegen
                        gefuellt
*/

void safe_lexicon()
{
```

```
  int i;
  char word[wordlen], tag[taglen];
  ofstream Out;

   Out.open("lexicon.dat");
  for (i=0; i<maxword; i++)
   {
      taglex[i].output_word(word);
     if (strcmp(word, ""))
           taglex[i].safe_content(Out);
   }   //end for
  Out.close();
}   //end safe_lexicon


/*
   tag_closed_classed intialisiert jedes Wort, das im Text nur jeweils mit
   einem einzigen Tag einer geschlossenen Klasse getaggt ist, mit diesem Tag.
*/

void tag_closed_classes()
{
  char tag[taglen];

  for (int wordcount=0; wordcount<maxword; wordcount++)
   {
      if (!taglex[wordcount].is_empty() && !taglex[wordcount].no_texttag() &&
taglex[wordcount].unique_tag())
           {
             taglex[wordcount].get_texttag(tag, 0);
             if (closed_class(tag))
               taglex[wordcount].put_tag(tag);
           }   //end if
    }   //end for
}   //end tag_closed_classes


/*
   closed_class ueberprueft, ob ein Tag einer geschlossenen Klasse angehoert.
   rein: tag - das Tag
   raus: 0 / 1 - gehoert nicht zu einer geschlossenen Klasse / gehoert dazu
*/

int closed_class(char tag[])
{
  int len = strlen(tag);

  if (((tag[0] == 'X') && (tag[2] != 'D')) ||
     (tag[len-1] == 'R') ||
     (tag[len-1] == 'O'))
    return(1);
  else return(0);
}   //end closed_class


/*
    find_most_likely_tag sucht alle moeglichen Regeln, die bei der aktuellen
    Lage anwendbar sind, sucht sich die beste aus, die die meisten positiven
```

```
    Veraenderungen bewirkt und wendet diese an. Dies geschieht so lange, bis
    keine Regeln mehr gefunden werden, deren Verbesserungesquote ueber einem
    Schwellenwert liegt. Die gefundenen Regeln werden in safe_cfrules sofort
    abgespeichert.
    rein+raus: cf_rules - die bisher angewandten kontext-freien Regeln
    rein: untagged - der Name des ungetaggten Korpus
            frequents - die Liste der haeufigsten Woerter
    raus: die Anzahl der bisher angewandten Regeln
    Nebeneffekt: die Liste 'cf_rules' wird gefuellt und die gefundenen Regeln
                    in eine Datei abgespeichert
*/

int find_most_likely_tag(char* untagged, CFrule cf_rules[], freqlist frequents[])
{
  CFtemplate templates[50];
  CFcandidate candidates;
  cf_rulecand temp;
  int wordcount, templcount, nrtempl, rulecount=0, nrcand;
  float threshold=1.1;

  safe_lexicon();
  nrtempl=read_cftemplates(templates);
  while (threshold>1 && rulecount < 500)
  {
     threshold=0;
     candidates.reset();
     for (wordcount=0; wordcount<maxword; wordcount++)
          {
            if (!(wordcount % 1000))
            {
               cout<<"Lexikoneintrag: "<<wordcount<<'\n';
              cout.flush();
            }   //end if
            if (!taglex[wordcount].is_empty() && !taglex[wordcount].no_texttag()
               && !taglex[wordcount].correct_tag())
              for (templcount=0; templcount<nrtempl; templcount++)
                    candidates = templates[templcount].make_rule(taglex[wordcount], wordcount,
candidates, frequents, untagged);
          }   //end for
     nrcand=candidates.output_candnr();
     cout<<"Anzahl der Kandidaten: "<<nrcand<<'\n';
     cout.flush();
     if (!candidates.empty())
          {
            temp=candidates.find_bestrule();
            cf_rules[rulecount].initialize(temp);
            cf_rules[rulecount].apply_bestrule(untagged);
            rulecount++;
            cout<<"Regelnummer: "<<rulecount<<'\n';
            cout.flush();
            safe_cfrules(cf_rules, rulecount);
            threshold=temp.score;
          }   //end if
     else threshold=0;
   }   //end while
  return(rulecount);
}   //end find_most_likely_tag
```

```
/*
    read_cftemplates liest die kontext-freien Templates aus der Datei
    'cftempl.dat' ein.
    rein+raus: templates - die Liste der kontext-freien Templates
    raus: die Anzahl der kontext-freien Templates
    Nebeneffekt: die Liste 'templates' wird gefuellt
*/

int read_cftemplates(CFtemplate templates[])
{
  char dummy, tempa[wordlen], tempw[wordlen];
  int i;
  ifstream In;

   In.open("cftempl.dat");
  for (i=0; !In.eof(); i++)
   {
      In.get(tempa, 12, ' ');
      In.get(dummy);
      In.get(tempw, 10, '\n');
      In.get(dummy);
      if (strcmp(tempa, "") &&!isspace(tempa[0]))
            templates[i].initialize(tempa, tempw);
     else i--;
   }   //end for
  return(i);
}   //end read_templates


/*
    cf_tag_corpus wendet die in cf_tag gefundenen Informationen auf einen Text
    an, d.h. fuer jedes Wort wird zuerst ueberprueft, ob es im Lexikon
    enthalten ist. Ist das der Fall, wird das Wort mit dem im Lexikon
    angegebenen Tag in die Ausgabedatei eingetragen. Sonst werden die
    kontext-freien Regeln angewandt.
    rein: tagged1, tagged2 - die Namen der Ein- und Ausgabedatei
              untagged - der Name des ungetaggten Korpus
              cf_rules - die Liste der kontext-freien Regeln
              nrcfrules - die Anzahl der kontext-freien Regeln
    raus: die Anzahl der nicht im Lexikon gefundenen Woerter
    Nebeneffekt: der getaggte text wird in die Ausgabedatei geschrieben
*/

int cf_tag_corpus(char* tagged1, char* tagged2, char* untagged, CFrule cf_rules[], int nrcfrules)
{
  char word[wordlen], tag[taglen], mytag[taglen];
  int unknown=0;
  ifstream In;
  ofstream Out;

  all_words=0;
   In.open(tagged1);
   Out.open(tagged2);
  while (!In.eof())
   {
      read_word(In, word, tag, mytag);
      if (strcmp(word, ""))
```

```
            {
              all_words++;
              if (hash(word, mytag) < 0)
               {
                  apply_cfrules(word, mytag, cf_rules, nrcfrules, untagged);
                  unknown++;
               }   //end if
              Out<<word<<'|'<<tag<<'|'<<mytag<<'\n';
            }   //end if
    }   //end while
  In.close();
  Out.close();
  return(unknown);
}   //end cf_tag_corpus


/*
   apply_cfrules wendet die kontext-freien Regeln der Reihe nach auf das
   aktuelle Wort an.
   rein+raus: tag - das Tag des aktuellen Worts
   rein: word - das aktuelle Wort
              cf_rules - die Liste der kontext-freien Regeln
              nrrules - die Anzahl der Regeln
              untagged - der Name des ungetaggten Korpus
   Nebeneffekt: die Variable 'tag' bekommt einen Wert zugewiesen
*/

void apply_cfrules(char word[], char tag[], CFrule cf_rules[], int nrrules, char* untagged)
{
  for (int i=0; i<nrrules; i++)
     cf_rules[i].apply_rule(word, tag, untagged);
}   //end apply_cfrule
```

```
/*************************************************************************/
/*                                                                       */
/*                        Datei: CSTAG.CC                                */
/*                                                                       */
/*************************************************************************/


int read_cstemplates(CStemplate templates[]);
int find_tag_in_context(CStemplate templates[], int nrtempl, CSrule cs_rules[], char* infile, char*
outfile, int rulecount);
CScandidate adjust_cs_score(CScandidate candidates, char* infile);
void safe_csrules(CSrule cs_rules[], int nrrules);

/*
   cs_tagger erstellt mittels der transformationsbasierten fehlergesteuerten
   Methode eine Liste von kontext-sensitiven Regeln. Dies geschieht anhand
   eines bereits getaggten Korpus, der dann zuerst den kontext-freien Tagger
   durchlaeuft, bevor hier kontext-sensitive Regelkandidaten ermittelt werden.
   rein+raus: cs_rules - die Liste der gelernten kontext-sensitiven Regeln
   rein: tagged - der Name des bereits getaggten Lernkorpus
            untagged - der Name des ungetaggten Korpus
            cf_rules - die bereits gelernte kontext-freien Regeln
            nrcfrules - die Anzahl der kontext-freien Regeln
   Nebeneffekt: die Liste 'cs_rules' wird gefuellt und abgespeichert
*/

int cs_tagger(char* tagged, char* untagged, CFrule cf_rules[], int nrcfrules, CSrule cs_rules[])
{
  CStemplate templates[50];
  int nrtempl, nrrules, unknown;

   unknown=cf_tag_corpus(tagged, "temp.dat", untagged, cf_rules, nrcfrules);
  nrtempl=read_cstemplates(templates);
  nrrules=find_tag_in_context(templates, nrtempl, cs_rules, "temp.dat", tagged, 0);
  safe_csrules(cs_rules, nrrules);
  if (stat_flag > 1)
    stat_unknown(unknown);
  return(nrrules);
}   //end cs_tagger


/*
   read_cstemplates liest die fuer die Templates benoetigte Information aus
   der Datei 'cstempl.dat' ein.
   rein+raus: templates - die Liste der kontext-sensitiven Templates
   Nebeneffekt: die Liste 'templates' wird gefuellt
*/

int read_cstemplates(CStemplate templates[])
{
  int count;
  ifstream In;

   In.open("cstempl.dat");
  for (count=0; !In.eof(); count++)
    templates[count].read_template(In);
  In.close();
  return(count);
```

```
}   //end read_cstemplates


/*
   find_tag_in_context bildet fuer jedes template und fuer jedes Wort + Kontext
   Regelkandidaten, sucht sich den besten davon aus, und wendet ihn auf das
   Korpus an, sofern seine Wertung eine bestimmte Schwelle ueberschreitet. Dies
   wiederholt sich so lange, bis kein Regelkandidat mehr gefunden wird, dessen
   Wertung die Schwelle ueberschreitet.
   rein+raus: cs_rules - die Liste der gelernten kontext-sensitiven Regeln
   rein: templates - die Liste der Templates
            nrtempl - die Anzahl der Templates
            infile - der Name der Ausgangsdatei
            outfile - der Name der Zieldatei
   raus - die Anzahl der gelernten Regeln
   Nebeneffekt: die Liste 'cs_rules' wird gefuellt und abgespeichert
*/

int find_tag_in_context(CStemplate templates[], int nrtempl, CSrule cs_rules[], char* infile, char*
outfile, int rulecount)
{
  CScandidate candidates;
  wordlist words[7];
  float threshold=1.0;
  int nrtag, bestrule;
  ifstream In;
  char *tmp;
  cs_rulecand temp;

  while (threshold>0 && rulecount<800)
  {
     candidates.reset();
     In.open(infile);
     for (int k=4; k<7; k++)
          read_word(In, words[k].word, words[k].texttag, words[k].tag);
     while (!In.eof() || strcmp(words[4].tag, ""))
          {
            for (int i=1; i<7; i++)
             {
                strcpy(words[i-1].word, words[i].word);
                strcpy(words[i-1].tag, words[i].tag);
                strcpy(words[i-1].texttag, words[i].texttag);
             }   //end for
            if (In.eof())
             {
                strcpy(words[6].word, "");
                strcpy(words[6].tag, "");
                strcpy(words[6].texttag, "");
             }   //end if
            else read_word(In, words[6].word, words[6].texttag, words[6].tag);
            if (strcmp(words[3].tag, words[3].texttag))
              for (int j=0; j<nrtempl; j++)
                candidates=templates[j].make_rule(words, candidates);
          }   //end while-2
     In.close();
     if (!candidates.empty())
          {
            adjust_cs_score(candidates, infile);
```

```
            temp=candidates.find_bestrule();
            threshold=temp.score;
           if (threshold > 0)
            {
               cs_rules[rulecount].initialize(temp);
               cs_rules[rulecount].apply_rule(infile, outfile);
               rulecount++;
               cout<<"Regelnummer: "<<rulecount<<"     score: "<<threshold<<'\n';
               cout.flush();
               safe_csrules(cs_rules, rulecount);
               tmp=infile;
               infile=outfile;
               outfile=tmp;
             }   //end if-2
          }   //end if-1
     else threshold=0;
   }   //end while-1
  if (stat_flag==1 || stat_flag==3)
   {
      cout<<"Nach der kontext-sensitiven Komponente:\n";
     statistics(infile);
   }   //end if
  return(rulecount);
}    //end find_tag_in_context


/*
   adjust_cs_score errechnet die endgueltige Wertung der Regelkandidaten
   unter Einbeziehung der negativen Faelle, d.h. der Faelle, bei denen die
   Anwendung der Regel zu einer Verschlechterung des Ergebnisses fuehrt.
   rein: candidates - die Regelkandidaten
            infile - der Name der Ausgangsdatei
   raus: die aktualisierten Kandidaten
*/

CScandidate adjust_cs_score(CScandidate candidates, char* infile)
{
  ifstream In;
  wordlist words[7];

  In.open(infile);
  for (int k=4; k<7; k++)
    read_word(In, words[k].word, words[k].texttag, words[k].tag);
  while (!In.eof() || strcmp(words[4].tag, ""))
   {
     for (int i=1; i<7; i++)
           {
             strcpy(words[i-1].tag, words[i].tag);
             strcpy(words[i-1].texttag, words[i].texttag);
           }   //end for
     if (In.eof())
           strcpy(words[6].tag, "");
      else read_word(In, words[6].word, words[6].texttag, words[6].tag);
     candidates.change_score(words);
   }   //end while
  In.close();
  return(candidates);
}   //end adjust_cs_score
```

```
/*
   safe_csrules speichert die fuer die Regel wichtigen Informationen in der
   Datei 'csrules.dat' ab.
   rein: cs_rules - die Regeln
            nrrules - die Anzahl der Regeln
   Nebeneffekt: die Regeln werden in der Datei 'csrules.dat' abgespeichert
*/

void safe_csrules(CSrule cs_rules[], int nrrules)
{
  ofstream Out;

   Out.open("csrules.dat");
  for (int i=0; i<nrrules; i++)
    cs_rules[i].safe_content(Out);
  Out.close();
}   //end safe_csrules
```

```
/*************************************************************************/
/*                                                                       */
/*                          Datei: APPLY.CC                              */
/*                                                                       */
/*************************************************************************/


/*
   apply_rules wendet die in der kontext-freien und in der kontext-
   sensitiven Phase gelernten Regeln auf das Testkorpus an.
   rein: tagged - der Name des Testkorpus
         untagged - der Name des ungetaggten Korpus
         cf_rules - die Liste der kontext-freien Regeln
         nrcfrules - die Anzahl der kontext-freien Regeln
         cs_rules - die Liste der kontext-sensitiven Regeln
         nrcsrules - die Anzahl der kontext-sensitiven Regeln
   Nebeneffekt: der getaggte Text wird in die Ausgabedatei geschrieben
*/

void apply_rules(char* tagged, char* untagged, CFrule cf_rules[], int nrcfrules, CSrule
cs_rules[], int nrcsrules)
{
  char infile[wordlen], outfile[wordlen], tmp[wordlen];
  int unknown;

  strcpy(infile, tagged);
  strcpy(outfile, "temp1.dat");
  cout<<"Testlauf: \n\n";
  unknown=cf_tag_corpus(infile, outfile, untagged, cf_rules, nrcfrules);
 for (int i=0; i<nrcsrules; i++)
  {
     strcpy(tmp, infile);
     strcpy(infile, outfile);
     strcpy(outfile, tmp);
     cs_rules[i].apply_rule(infile, outfile);
   }   //end for
 if (stat_flag==1 || stat_flag==3)
   statistics(outfile);
 if (stat_flag > 2)
    stat_unknown(unknown);
}   //end apply_rules
```

```
/*************************************************************************/
/*                                                                       */
/*                          Datei: TAG.H                                 */
/*                                                                       */
/*************************************************************************/


/*
   Tag ist die Klasse der Lexikoneintraege; ein Objekt dieser Klasse
   enthaelt ein Wort, das aktuelle Tag, sowie eine Liste von Tags, die
   dem Wort im kontext-freien Lernkorpus zugewiesen bekommen hat, und
   die Anzahl dieses Auftretens.
*/

class Tag
{
  struct entry
   {
     char tag[taglen];
     int count;
   };   //end struct
  entry texttags[maxtag];
  char word[wordlen];
  char tag[taglen];

public:
  void liketag();
  void initialize(char nword[], char ntag[], char ttag[]);
  void initialize_texttags();
  void safe_content(ofstream& Out);
  int correct_tag();
  int unique_tag();
  void put_word(char nword[]) {strcpy(word, nword);}
  void put_tag(char ntag[]) {strcpy(tag, ntag);}
  void put_texttag(char texttag[]);
  void put_texttag_and_num(char texttag[], int tagcount);
  void output_word(char oword[]) {strcpy(oword, word);}
  void output_tag(char otag[]) {strcpy(otag, tag);}
  void get_texttag(char texttag[], int num) {strcpy(texttag, texttags[num].tag);}
  int is_empty() {return(strcmp(word, "") == 0);}
  int no_texttag() {if (strcmp(texttags[0].tag, "") == 0) return(1); else return(0);}
  int is_equal(char nword[]) {return(strcmp(word, nword) == 0);}
  int is_smaller_than(char nword[]);
  int get_nr(char intag[]);
  int wordcount();
};   //end class
```

```
/*************************************************************************/
/*                                                                       */
/*                          Datei: TAG.CC                                */
/*                                                                       */
/*************************************************************************/


/*
   initialize initialisiert einen Lexikoneintrag mit dem Wort, dem Tag vom
   System, dem bereits im Lernkorpus vergebenen Tag und setzt die Haeufigkeit
   des im Lernkorpus gefundenen Tags auf 1.
   rein: nword - das Wort
            ntag - das vom System vergebene Tag
            ttag - das im Lernkorpus vergebene Tag
*/

void Tag::initialize(char nword[], char ntag[], char ttag[])
{
  strcpy(word, nword);
  strcpy(tag, ntag);
  strcpy(texttags[0].tag, ttag);
   texttags[0].count=1;
}   //end initialize


/*
   initialize_texttags initialisiert die Liste eines Lexikoneintrags, in
   der die im Lernkorpus beobachteten Tags eingetragen werden, mit dem
   leeren String.
*/

void Tag::initialize_texttags()
{
  for (int i=0; i<maxtag; i++)
   {
      strcpy(texttags[i].tag, "");
      texttags[i].count=0;
    }   //end for
}   //end initialize_texttags


/*
   safe_content schreibt den Inhalt eines Lexikoneintrags in den
   Ausgabestream.
   rein: Out - der Ausgabestream
*/

void Tag::safe_content(ofstream& Out)
{
  Out<<word<<' '<<tag;
  for (int i=0; i<maxtag && strcmp(texttags[i].tag, ""); i++)
     Out<<' '<<texttags[i].tag<<' '<<texttags[i].count;
  Out<<'\n';
}   //end safe_content


/*
   correct_tag ueberprueft, ob das vom System vergebene Tag mit einem im
```

```
   Lernkorpus stehenden identisch ist.
   raus: 0 / 1 - nicht identisch / identisch
*/

int Tag::correct_tag()
{
  int found=0;

  for (int i=0; i<maxtag && strcmp(texttags[i].tag, "") && !found; i++)
    if (strcmp(texttags[i].tag, tag) == 0)
      found=1;
  return(found);
}   //end correct_tag


/*
   unique_tag ueberprueft, ob das Wort genau ein moegliches Tag in der Liste
   'texttags' besitzt.
   raus: 0 / 1 - mehr als ein Tag / genau ein Tag
*/

int Tag::unique_tag()
{
  if (strcmp(texttags[1].tag, "") == 0)
    return(1);
  else return(0);
}   //end unique_tag


/*
    is_smaller_than entscheidet fuer ein Wort, ob es in der alphabetischen
    Reihenfolge vor oder nach dem Wort des aktuellen Lexikoneintrags steht.
    rein: nword -  das zu untersuchende Wort
    raus: 0 / 1 - nword steht vor / nach dem aktuellen Lexikoneintrag
*/

int Tag::is_smaller_than(char nword[])
{
  int i;

  for (i=0; (i<=strlen(word)) && (word[i] == nword[i]); i++);
  if (int(word[i]) < int(nword[i]))
    return(1);
  else return(0);
}   //end is_smaller_than


/*
    put_texttag traegt ein Tag in die Liste der im kontext-freien Lern-
    korpus beobachteten Tags ein, oder erhoeht die Anzahl, falls das Tag
    schon in der Liste eingetragen ist.
    rein: texttag - das einzutragende Tag
*/

void Tag::put_texttag(char texttag[])
{
  for (int i=0; i<maxtag; i++)
   {
```

```
      if (strcmp(texttags[i].tag, "") == 0)
            {
              strcpy(texttags[i].tag, texttag);
              texttags[i].count = 1;
              break;
            }   //end if
      else if (strcmp(texttags[i].tag, texttag) == 0)
            {
              texttags[i].count++;
              break;
            }   //end if
    }   //end for
}   //end put_texttag


/*
   put_texttag_and_num traegt ein Tag und seine Auftretenshaeufigkeit im
   Lernkorpus in die Liste der beobachteten Tags ein.
   rein: texttag - das beobachtete Tag
            tagcount - die Auftretenshaeufigkeit des Tags
*/

void Tag::put_texttag_and_num(char texttag[], int tagcount)
{
  int found=0;

  for (int i=0; i<maxtag && !found; i++)
    if (strcmp(texttags[i].tag, "") == 0)
    {
            strcpy(texttags[i].tag, texttag);
            texttags[i].count=tagcount;
            found=1;
     }   //end if
}   //end put_texttag_and_num


/*
    get_nr sucht die Auftretenshaeufigkeit eines Tags in der Liste der im
    kontext-freien Lernkorpus gefundenen Tags heraus.
    rein: intag - das Tag, dessen Auftretenshaeufigkeit festgestellt
                        werden soll
    raus: die Haeufigkeit des Auftretens des Tags
*/

int Tag::get_nr(char intag[])
{
  int found=0;

  for (int i=0; i<maxtag && !found; i++)
    if (strcmp(intag, texttags[i].tag) == 0)
    {
            found=texttags[i].count;
            break;
     }   //end if
  return(found);
}   //end get_nr
```

```
/*
    wordcount stellt die Haeufigkeit des Auftretens eines Worts im
    kontext-freien Lernkorpus fest.
    raus: die Anzahl des Auftretens
*/

int Tag::wordcount()
{
  int count=0;

  for (int i=0; i<maxtag; i++)
   {
      if (strcmp(texttags[i].tag, "") == 0)
            break;
      else count += texttags[i].count;
   }   //end for
  return(count);
}   //end wordcount


/*
   liketag sucht das Tag aus den Texttags heraus, das am haeufigsten
   vorgekommen ist und traegt die als Arbeitstag ein.
   Nebeneffekt: der Variable 'tag' wird das am haeufigsten auftretende
                      Tag zugewiesen
*/

void Tag::liketag()
{
  int score=0;

  for (int i=0; i<maxtag; i++)
    if (strcmp(texttags[i].tag, "") && score<texttags[i].count)
    {
           strcpy(tag, texttags[i].tag);
           score=texttags[i].count;
     }   //end if
}   //end liketag
```

```
/*************************************************************************/
/*                                                                       */
/*                        Datei: CFTEMPL.H                               */
/*                                                                       */
/*************************************************************************/


class CFcandidate;


/*
    CFtemplate ist die Klasse der kontext-freien Templates, sie enthaelt
    die Handlung (Loeschen, Anfuegen, oder Vergleichen), die beim Test auf
    die Anwendbarkeit des Templates ausgefuehrt werden muss, und die Stelle
    (Praefix, Suffix, Buchstabe, oder Nachbarwort).
*/

class CFtemplate
{
 protected:
  char action, where;

  CFcandidate delete_affix(Tag lexentry, int lexnr, CFcandidate candidates, char* untagged);
  void initialize_poswords(pos_word poswords[]);
  CFcandidate search_words(pos_word poswords[], int num, Tag lexentry, int lexnr,
CFcandidate candidates, char* untagged);
  int compare(char char1, char char2);
  CFcandidate add_affix(Tag lexentry, int lexnr, CFcandidate candidates, char* untagged);
  CFcandidate compare_part(Tag lexentry, int lexnr, CFcandidate candidates, char* untagged);
  CFcandidate find_neighbor(Tag lexentry, int lexnr, CFcandidate candidates, char* untagged,
freqlist frequents[]);
   int frequents_member(char neighbor[], freqlist frequents[]);
  CFcandidate find_ablaut(Tag entry, int lexnr, CFcandidate candidates, char* untagged);
  int is_vowel(char str[]);
  void put_ablaut(char from[], char vowel[], int num, pos_word to[], int nrpos);
 public:
  void initialize(char tempa[], char tempw[]);
  void set_action(char naction) {action=naction;}
  void set_where(char nwhere) {where=nwhere;}
  CFcandidate make_rule(Tag lexentry, int lexnr, CFcandidate candidates, freqlist frequents[],
char* untagged);
};
```

```
/*************************************************************************/
/*                                                                       */
/*                          Datei: CFTEMPL.CC                            */
/*                                                                       */
/*************************************************************************/


/*
  initialize initialisiert das aktuelle Template mit der Aktion, wie
  getestet wird, und dem Ort, an dem der Test angesetzt wird.
  rein: tempa - die Aktion
        tempw - der Ort
*/

void CFtemplate::initialize(char tempa[], char tempw[])
{
 if (strcmp(tempa, "loeschen") == 0)
   action='d';
 else
  {
    if (strcmp(tempa, "anhaengen") == 0)
          action='a';
    else
          {
           if (strcmp(tempa, "vergleichen") == 0)
             action='c';
           else
            {
              if (strcmp(tempa, "nachbar") == 0)
                    action='n';
              else
                    {
                     if (strcmp(tempa, "ablaut") == 0)
                       action='b';
                     else error("Komische Template-Aktion");
                    }  //end else-4
            }   //end else-3
          }   //end else-2
  }   //end else-1
 if (strcmp(tempw, "vorne") == 0)
   where='p';
 else
  {
    if (strcmp(tempw, "hinten") == 0)
          where='s';
    else
          {
           if (strcmp(tempw, "buchstabe") == 0)
             where='c';
           else
            {
              if (strcmp(tempw, "innen") == 0)
                    where='i';
              else error("Komischer Template-Ort");
            }   //end else-3
          }   //end else-2
  }   //end else-1
}   //end initialize
```

```
/*
   make_rule stellt sicher, dass die Aktion des aktuellen Templates auf
   das Wort angewendet wird.
   rein: lexentry - der Lexikoneintrag des Worts, das untersucht wird
            lexnr - die Position des Lexikoneintrags im Lexikon
            candidates - die Liste der moeglichen Regeln, unter denen
                     diejenige ausgesucht wird, die im naechsten
                     Schritt angewendet wird
            frequents - die Liste der haeufigsten Woerter im Lexikon
            untagged - der Name des ungetaggten Korpus
   raus: die aktualisierten Kandidaten
*/

CFcandidate CFtemplate::make_rule(Tag lexentry, int lexnr, CFcandidate candidates, freqlist
frequents[], char* untagged)
{
  switch (action)
   {
    case 'd' : candidates = delete_affix(lexentry, lexnr, candidates, untagged); break;
    case 'a' : candidates = add_affix(lexentry, lexnr, candidates, untagged); break;
    case 'c' : candidates = compare_part(lexentry, lexnr, candidates, untagged); break;
    case 'b' : candidates = find_ablaut(lexentry, lexnr, candidates, untagged); break;
    case 'n' : candidates = find_neighbor(lexentry, lexnr, candidates, untagged, frequents); break;
   }   //end switch
  return(candidates);
}   //end make_rule


/*
   delete_affix loescht ein Affix von bestimmter Laenge vom Wort und ueber-
   prueft im ungetaggten Korpus, ob es sich bei dem Restwort um ein
   existierendes Wort handelt. Ist dies der Fall, wird in der Klasse
   CFcandidate create_rule aufgerufen, wo die neue Regel erstellt wird.
   rein: lexentry - der Lexikoneintrag des Worts, das untersucht wird
            lexnr - die Position des Lexikoneintrags im Lexikon
            candidates - Liste der bisher gefundenen moeglichen Regeln
            untagged - der Name des ungetaggten Korpus
   raus: die aktualisierten Kandidaten
*/

CFcandidate CFtemplate::delete_affix(Tag lexentry, int lexnr, CFcandidate candidates, char*
untagged)
{
  char word[wordlen];
  pos_word poswords[maxpos];
  int len, wordsnum, nrwords=0;

  initialize_poswords(poswords);
  lexentry.output_word(word);
  len=strlen(word);
  if (len-1 < affixlen)
     wordsnum=len-1;
  else wordsnum=affixlen;
  for (int num=1; num<=wordsnum; num++)
  {
     if (where=='p')
```

```
          {
            strncpy(poswords[nrwords].affix, word, num);
            poswords[nrwords].affix[num]=0;
            strcpy(poswords[nrwords].word, &word[num]);
            nrwords++;
          }   //end if
    if (where=='s')
          {
            strcpy(poswords[nrwords].affix, &word[len-num]);
            strncpy(poswords[nrwords].word, word, len-num);
            poswords[nrwords].word[len-num]=0;
            nrwords++;
          }   //end if
    if (where=='i')
          for (int i=1; i<len-num; i++)
           {
             strncpy(poswords[nrwords].word, word, i);
             strncpy(poswords[nrwords].affix, &word[i], num);
             poswords[nrwords].affix[num]=0;
             strncpy(&poswords[nrwords].word[i], &word[i+num], len-num-i);
             poswords[nrwords].word[len-num]=0;
             nrwords++;
           }   //end for-2
  }   //end for-1
 if (wordsnum)
   candidates = search_words(poswords, nrwords, lexentry, lexnr, candidates, untagged);
 return(candidates);
}  //end delete_affix


/*
   initialize_poswords initialisiert die Liste der moeglichen Woerter mit
   dem leeren String.
   rein+raus: poswords - die Liste de moeglichen Woerter
   Nebeneffekt: die Liste 'poswords' wird initialisiert
*/

void CFtemplate::initialize_poswords(pos_word poswords[])
{
 for (int i=0; i<maxpos; i++)
   strcpy(poswords[i].word, "");
}   //end initialize


/*
    search_words untersucht, ob bestimmte Woerter in einem ungetaggten
    Korpus enthalten sind.
    rein: poswords - die Liste der zu suchenden Woerter
              num - die Anzahl der Woerter
              lexentry - der aktuelle Lexikoneintrag
              lexnr - die Position des Lexikoneintrags im Lexikon
              candidates - die Regelkandidaten
              untagged - der Name des ungetaggten Korpus
    raus: die aktualisierten Kandidaten
*/

CFcandidate CFtemplate::search_words(pos_word poswords[], int num, Tag lexentry, int lexnr,
CFcandidate candidates, char* untagged)
```

```
{
  ifstream ln;
  char word[wordlen];
  int nrwords=num;

   ln.open(untagged);
  while (!ln.eof() && nrwords)
   {
     bigr_word(ln, word);
     for (int i=0; i<num; i++)
          {
            if (strcmp(word, poswords[i].word) == 0 ||
                (strcmp(&word[1], &poswords[i].word[1]) == 0 &&
                 int(word[0]) == int(poswords[i].word[0])-32))
             {
                candidates.create_rule(poswords[i].affix, lexentry, lexnr, action, where, untagged);
                strcpy(poswords[i].word, "");
                nrwords--;
             }   //end if
          }   //end for
   }   //end while
  ln.close();
  return(candidates);
}   //end search_words


/*
    add_affix untersucht, mit welchen Affixen zusammen aus dem zu unter-
    suchenden Wort ein im ungetaggten Korpus vorhandenens Wort entsteht.
    Fuer diese Affixe wird dann in create_rule eine Regel erstellt.
    rein: lexentry - Lexikoneintrag des zu untersuchenden Worts
              lexnr - die Position des Lexikoneintrags im Lexikon
              candidates - Liste der bisher gefundenen moeglichen Regeln
              untagged - Name des ungetaggten Korpus
    raus: die aktualisierten Kandidaten
*/

CFcandidate CFtemplate::add_affix(Tag lexentry, int lexnr, CFcandidate candidates, char*
untagged)
{
  char affix[affixlen], word[wordlen], posword[wordlen];
  int len;
  ifstream ln;

   ln.open(untagged);
   lexentry.output_word(word);
   len=strlen(word);
   if (where == 'i' && len < 2);
   else while(!ln.eof())
   {
     bigr_word(ln, posword);
     if (where == 's' && strlen(posword)>len && strlen(posword)<=len+affixlen
            && (strncmp(posword, word, len) == 0))
          {
            strcpy(affix, &posword[len]);
            candidates.create_rule(affix, lexentry, lexnr, action, where, untagged);
          }   //end if
     else for (int num=1; num<len && num<=affixlen; num++)
```

```
          {
            if (where == 'p' && (strcmp(&posword[num], word) == 0))
            {
               strncpy(affix, posword, num);
               affix[num]=0;
               candidates.create_rule(affix, lexentry, lexnr, action, where, untagged);
            }  //end if
            if (where == 'i')
              for (int i=1; i<len-1; i++)
              {
                    if (strncmp(posword, word, i) == 0 &&
                       strcmp(&posword[i+num], &word[i]) == 0)
                    {
                       strncpy(affix, &posword[i], num);
                       affix[num]=0;
                       candidates.create_rule(affix, lexentry, lexnr, action, where, untagged);
                    }  //end if
              }  //end for-2
          }  //end for-1
   }  //end while
  ln.close();
  return(candidates);
}  //end add_affix


/*
    compare_part untersucht, welche Affixe oder bestimmten Buchstaben in
    einem Wort vorhanden sind. Fuer diese Faelle wird create-rule aufgerufen.
    rein+raus: candidates - Liste der bisher gefundenen moeglichen Regeln
    rein: lexentry - Lexikoneintrag des zu untersuchenden Worts
             lexnr - die Position des Lexikoneintrags im Lexikon
             candidates - die Regelkandidaten
             untagged - der Name des ungetaggten Korpus
    raus: die aktualisierten Kandidaten
*/

CFcandidate CFtemplate::compare_part(Tag lexentry, int lexnr, CFcandidate candidates, char*
untagged)
{
  char word[wordlen], affix[affixlen];
  int len;

   lexentry.output_word(word);
  len=strlen(word);
  if (where == 'c')
    for (int num=0; num<len; num++)
    {
          affix[0]=word[num];
          affix[1]=0;
          candidates.create_rule(affix, lexentry, lexnr, action, where, untagged);
     }  //end for
  if (where == 'p')
    for (int num1=1; num1<=affixlen && num1<len; num1++)
    {
          strncpy(affix, word, num1);
          affix[num1]=0;
          candidates.create_rule(affix, lexentry, lexnr, action, where, untagged);
     }  //end for
```

```
  if (where == 's')
    for (int num2=1; num2<=affixlen && num2<len; num2++)
    {
          strcpy(affix, &word[len-num2]);
          candidates.create_rule(affix, lexentry, lexnr, action, where, untagged);
     }   //end for
  if (where == 'i')
    for (int num3=2; num3<=affixlen && num3<len-2; num3++)
      for (int i=1; i<len-num3; i++)
          {
            strncpy(affix, &word[i], num3);
            affix[num3]=0;
            candidates.create_rule(affix, lexentry, lexnr, action, where, untagged);
          }   //end for
  return(candidates);
}   //end compare_part


/*
   find_neighbor untersucht, welche Nachbarn das aktuelle Wort hat. Um den
   Suchaufwand zu begrenzen werden nur Nachbarn in Betracht gezogen, die zu
   den haeufigsten Woertern im getaggten Korpus gehoeren. Mit diesen
   Nachbarn wird dann create_rule aufgerufen, wo die Regel erstellt wird.
   rein: lexentry - der Lexikoneintrag des aktuellen Worts
          lexnr - die Position des Lexikoneintrags im Lexikon
          candidates - Liste der bisher gefundenen moeglichen Regeln
          untagged - der Name des ungetaggten Korpus
          frequents - die Liste der haeufigsten Woerter
   raus: die aktualisierten Kandidaten
*/

CFcandidate CFtemplate::find_neighbor(Tag lexentry, int lexnr, CFcandidate candidates, char*
untagged, freqlist frequents[])
{
  char word[wordlen], w1[wordlen], w2[wordlen];
  ifstream ln;

   ln.open(untagged);
   lexentry.output_word(word);
   while (!ln.eof())
   {
     bigr_pair(ln, w1, w2);
     if (where == 'p' && strcmp(word, w2) == 0 &&
          frequents_member(w1, frequents))
        candidates.create_rule(w1, lexentry, lexnr, action, where, untagged);
     if (where == 's' && strcmp(word, w1) == 0 &&
          frequents_member(w2, frequents))
        candidates.create_rule(w1, lexentry, lexnr, action, where, untagged);
   }   //end while
  ln.close();
  return(candidates);
}   //end find_neighbor


/*
   frequents_member untersucht, ob ein Wort in der Liste der haeufigsten
   Woerter enthalten ist.
   rein: neighbor - das zu suchende Wort
```

```
            frequents: die Liste der haeufigsten Woerter
   raus: 0 / 1 - nicht gefunden / gefunden
*/

int CFtemplate::frequents_member(char neighbor[], freqlist frequents[])
{
  int i;
  for (i=0; i<maxfreq && is_smaller(frequents[i].word, neighbor); i++);
  if (strcmp(neighbor, frequents[i].word) == 0)
    return(1);
  else return(0);
}   //end frequents_member


/*
   find_ablaut ueberprueft, ob sich durch Ersetzen der Vokale eines Wortes
   durch andere Vokale Woerter ergeben, die im ungetagten Korpus gefunden
   werden koennen.
   rein: entry - der Lexikoneintrag des aktuellen Worts
            lexnr - die Position des Lexikoneintrags im Lexikon
            candidates - die Liste der bisherigen Regelkandidaten
            untagged - der Name des ungetaggten Korpus
   raus: die aktualisierten Kandidaten
*/

CFcandidate CFtemplate::find_ablaut(Tag entry, int lexnr, CFcandidate candidates, char*
untagged)
{
  pos_word poswords[maxpos];
  char word[wordlen], vowel[3];
  int len, nrpos=0;

  initialize_poswords(poswords);
  entry.output_word(word);
  len=strlen(word);
  for (int i=0; i<len; i++)
   {
     if (int(word[i]) == 92 && i<len-2)
           {
             strncpy(vowel, &word[i], 3);
             vowel[3]=0;
           }   //end if
     else
           {
             vowel[0]=word[i];
             vowel[1]=0;
           }   //end else
     if (is_vowel(vowel))
           for (int k=0; k<8; k++)
            {
              if (strncmp(vowels[k], &word[i], strlen(vowels[k])))
              {
                    put_ablaut(word, vowels[k], i, poswords, nrpos);
                    nrpos++;
               }   //end if
            }   //end for
   }   //end for
  if (nrpos)
```

```
    candidates = search_words(poswords, nrpos, entry, lexnr, candidates, untagged);
  return(candidates);
}   //end find_ablaut


/*
   is_vowel ueberprueft ob es sich bei einer Buchstabensequenz um einen
   Vokal handelt. Die Buchstabensequenz ist wegen der Darstellung der Umlaute
   im Korpus (z.B. '\"o') noetig.
   rein: str - die Buchstabensequenz
   raus: 0 / 1 - kein Vokal / Vokal
*/

int CFtemplate::is_vowel(char str[])
{
  int found=0;

  for (int i=0; i<8 && !found; i++)
    if(strcmp(str, vowels[i]) == 0)
      found=1;
  return(found);
}


/*
   put_ablaut bildet eine Liste mit moeglichen Ablautformen eines Wortes,
   d.h. immer ein Vokal im Wort wird durch einen anderen Vokal ersetzt.
   rein+raus: to - die Liste der moeglichen Ablautformen
   rein: from - das aktuelle Wort
              vowel - der Vokal, der in das Wort eingesetzt werden soll
              num - die Stelle, an der Der Ablaut gebildet werden soll
              nrpos - die Anzahl der bereits eingetragenen Ablautformen
   Nebeneffekt: die Liste 'to' wird gefuellt
*/

void CFtemplate::put_ablaut(char from[], char vowel[], int num, pos_word to[], int nrpos)
{
  if (strlen(vowel) == 1)
   {
      strcpy(to[nrpos].word, from);
      to[nrpos].word[num]=vowel[0];
   }   //end if-2
  else
   {
      strncpy(to[nrpos].word, from, num);
      strcpy(&to[nrpos].word[num], vowel);
      strcpy(&to[nrpos].word[num+3], &from[num+1]);
   }   //end else
  strcpy(to[nrpos].affix, vowel);
}   //end put_ablaut
```

```
/*****************************************************************************/
/*                                                                           */
/*                        Datei: CFTEMPL.DAT                                 */
/*                                                                           */
/*****************************************************************************/


loeschen vorne
vergleichen vorne
anhaengen vorne
loeschen hinten
vergleichen hinten
anhaengen hinten
vergleichen buchstabe
loeschen innen
anhaengen innen
vergleichen innen
nachbar vorne
nachbar hinten
ablaut buchstabe
```

```
/*************************************************************************/
/*                                                                       */
/*                          Datei: CFCAND.H                              */
/*                                                                       */
/*************************************************************************/


/*
   CFcandidate ist die Klasse der kontext-freien Regelkandidaten. Sie
   besteht aus einer einfach verketteten Liste der einzelnen Kandidaten.
*/

class CFcandidate
{
    cf_rulecand  *candidate;

 public:
  CFcandidate() {candidate=0;}
  void reset();
  int empty();
  int output_candnr();
  cf_rulecand find_bestrule();
  void create_rule(char affix[], Tag lexentry, int lexnr, char action, char where, char* untagged);
 private:
  int candsearch(char action, char where, char affix[], char to[], char from[], char word[], float
score);
  void build_cand(cf_rulecand *cand, char action, char where, char to[], char from[], char affix[],
char word[], float score, int candnr, int lexnr, char* untagged);
float adjust_score(cf_rulecand *cand, int lexnr, char* untagged);
};   //end class
```

```
/*************************************************************************/
/*                                                                       */
/*                          Datei: CFCAND.CC                             */
/*                                                                       */
/*************************************************************************/


/*
   reset setzt die Liste der Kandidaten zurueck auf null.
   Nebeneffekt: die Liste der ehemaligen Kandidaten wird freigegeben
*/

void CFcandidate::reset()
{
  cf_rulecand *ptr=candidate, *next;

  candidate=0;
  while (ptr != 0)
   {
     next = ptr->next;
     delete ptr;
     ptr=next;
   }   //end while
}   //end reset


/*
   empty ueberprueft, ob die Liste der Kandidaten leer ist.
   raus: 1 / 0 - leer / nicht leer
*/

int CFcandidate::empty()
{
   if (!candidate)
     return(1);
   else return(0);
}   //end empty


/*
   output_candnr gibt die laufende Nummer des Regelkandidaten aus.
   raus: die laufende Nummer des Regelkandidaten
*/

int CFcandidate::output_candnr()
{
  return(candidate->candnr);
}


/*
    find_bestrule sucht aus der Liste der moeglichen Regeln diejenige
    heraus, die die groesste positive Veraenderung bewirkt.
    raus: der Inhalt der besten Regel in der Liste
*/

cf_rulecand CFcandidate::find_bestrule()
{
```

```
  cf_rulecand *ptr, *best;
  float bestscore=0.0;

  for (ptr=candidate; ptr != 0; ptr=ptr->next)
   {
     if (bestscore < ptr->score)
           {
             best=ptr;
             bestscore=ptr->score;
           }   //end if
   }   //end for
  return(*best);
}   //end find_bestrule


/*
    create_rule erstellt einen Regelkandidaten mit Informationen ueber das
    Ausgangstag, das neue Tag, das Affix und das Template.
    rein: affix - der betroffene Wortteil
              lexentry - Lexikoneintrag des untersuchten Worts
              lexnr - die Position des Lexikoneintrags im Lexikon
              action- die Handlung
              where - der Ort der Handlung
*/

void CFcandidate::create_rule(char affix[], Tag lexentry, int lexnr, char action, char where, char*
untagged)
{
  char from[taglen], to[taglen], word[wordlen];
  cf_rulecand *cand;
  int found, wordnr, candnr;
  float posnr, score;

   lexentry.output_word(word);
   lexentry.output_tag(from);
   wordnr=lexentry.wordcount();
   for (int tagnr=0; tagnr<maxpostag; tagnr++)
   {
     strcpy(to, tagset[tagnr]);
     posnr=float(lexentry.get_nr(to));
     score=(posnr-(wordnr-posnr))/wordnr;
     found=candsearch(action, where, affix, to, from, word, score);
     if (!found && posnr>0)
           {
             cand = new cf_rulecand;
             if (!candidate)
               candnr=0;
             else candnr = candidate->candnr + 1;
             build_cand(cand, action, where, to, from, affix, word, score, candnr, lexnr, untagged);
             cand->next=candidate;
             candidate=cand;
           }   //end if
    }   //end for
}   //end create_rule


/*
    candsearch untersucht, ob eine bestimmte Regel schon vorhanden ist. Wenn
```

```
   ja, wird die Wertung hochgesetzt.
   rein: action - die Regelaktion
            where - der Ort, an dem die Regel ansetzt
            affix - der betroffene Wortteil
            to - das neue Tag
            from - das im Moment eingetragene Tag
            word - das aktuelle Wort
            score - die aktuelle Wertung
   raus: 0 / 1 - nicht gefunden / gefunden
*/

int CFcandidate::candsearch(char action, char where, char affix[], char to[], char from[], char
word[], float score)
{
  int found=0;
  cf_rulecand *ptr=candidate;

  while (!found && ptr != 0)
  {
     if (ptr->action == action &&
            ptr->where == where &&
            strcmp(ptr->to, to) == 0 &&
            strcmp(ptr->from, from) == 0 &&
            strcmp(ptr->affix, affix) == 0)
           {
            found=1;
            if ((action != 'n') || strcmp(word, ptr->word))
            {
               ptr->score+=score;
               if (action == 'n')
                     strcpy(ptr->word, word);
            }   //end if-2
           }   //end if-1
     else ptr=ptr->next;
   }   //end while
  return(found);
}   //end candsearch


/*
   build_cand initialisiert den neu geschaffenen Regelkandidaten mit
   der Handlung, dem Ort, dem Ziel- und dem Ausgangstag und dem Affix.
   rein+raus: der Pointer auf den neuen Regelkandidaten
   rein: cand - der Regelkandidat
            action - die Aktion
            where - der Handlungsort
            to - das Zieltag
            from - das Ausgangstag
            affix - das Affix
            word - das Wort
            score - die Wertung
            candnr - die laufende Nummer des Kandidaten
            lexnr - die Position des Lexikoneintrags im Lexikon
            untagged - der Name des ungetaggten Korpus
*/

void CFcandidate::build_cand(cf_rulecand *cand, char action, char where, char to[], char from[],
char affix[], char word[], float score, int candnr, int lexnr, char* untagged)
```

```
{
  cand->action = action;
  cand->where = where;
  cand->candnr = candnr;
  strcpy(cand->to, to);
  strcpy(cand->from, from);
  strcpy(cand->affix, affix);
  strcpy(cand->word, word);
  cand->score = adjust_score(cand, lexnr, untagged);
}   //end build_cand


/*
   adjust_score ueberprueft fuer einen Regelkandidaten, welche negativen
   Auswirkungen er auf die bisher schon abgearbeiteten Lexikoneintraege
   besitzt, und gleicht die Wertung dementsprechend an.
   rein: cand - der Regelkandidat
            lexnr - die Position des in create_rule aktuellen Lexikoneintrag
            untagged - der Name des ungetaggten Korpus
   raus: die aktualisierte Wertung des Kandidaten
*/

float CFcandidate::adjust_score(cf_rulecand *cand, int lexnr, char* untagged)
{
  CFrule testrule;
  char word[wordlen], tag[taglen];

  for (int i=0; i<lexnr; i++)
    if (!taglex[i].is_empty() && !taglex[i].no_texttag())
    {
           taglex[i].output_word(word);
           taglex[i].output_tag(tag);
           if (strlen(word) > 1 &&
              taglex[i].get_nr(cand->to) == 0 &&
              (strcmp(cand->from, "") == 0 ||
               strcmp(tag, cand->from) == 0))
            {
              testrule.initialize(*cand);
              if (testrule.test(word, untagged))
                cand->score--;
            }   //end if-2
     }   //end if-1
  return(cand->score);
}   //end adjust_score
```

```
/*************************************************************************/
/*                                                                       */
/*                        Datei: CFRULE.H                                */
/*                                                                       */
/*************************************************************************/


/*
   CFrule ist die Klasse der kontext-freien Regeln. Kontext-freie Regeln
   enthalten Information ueber das Ausgangstag, dessen Vorkommen die
   Regel ausloest, das Zieltag, und das Affix bzw. Nachbarwort.
*/

class CFrule : public CFtemplate
{
  char from[taglen], to[taglen], affix[wordlen];

  int delete_a(char word[], char* untagged);
  int add_a(char word[], char* untagged);
  int aneighbor(char word[], char* untagged);
  int amember(char word[]);
  int ablaut(char word[], char* untagged);
 public:
  void initialize(cf_rulecand cand);
  void safe_content(ofstream& Out);
  int test(char word[], char* untagged);
  int find_words(pos_word words[], int num, char* untagged);
  int compare(char char1, char char2);
  void apply_bestrule(char* untagged);
  void apply_rule(char word[], char tag[], char* untagged);
};
```

```
/*************************************************************************/
/*                                                                       */
/*                          Datei: CFRULE.CC                             */
/*                                                                       */
/*************************************************************************/


/*
   initialize initialisiert eine kontext-freie Regel mit den entsprechenden
   Angaben, also Aktion, Ort, Ausgangstag, Zieltag und Affix bzw. Nachbarwort
   aus einem Regelkandidaten.
   rein: cand - der Regelkandidat
*/

void CFrule::initialize(cf_rulecand cand)
{
  action=cand.action;
  where=cand.where;
  strcpy(from, cand.from);
  strcpy(to, cand.to);
  strcpy(affix, cand.affix);
}


/*
   safe_content schreibt die noetige Information einer Regel, die das aktuelle
   Template verwendet, in den output-stream.
   rein: Out - der Ausgabestream
   Nebeneffekt: der Regelinhalt wird in den Ausgabestream geschrieben
*/

void CFrule::safe_content(ofstream& Out)
{
  Out<<from<<' '<<to<<' '<<affix<<' ';
  switch(action)
   {
      case 'd' : Out<<"loeschen "; break;
      case 'a' : Out<<"anhaengen "; break;
      case 'c' : Out<<"vergleichen "; break;
      case 'n' : Out<<"nachbar"; break;
      default : error("Komische Regel");
   }   //end switch
  switch(where)
   {
      case 'p' : Out<<"vorne"; break;
      case 's' : Out<<"hinten"; break;
      case 'c' : Out<<"buchstabe"; break;
      case 'i' : Out<<"innen"; break;
      default : error("Komische Regel");
   }   //end switch
  Out<<'\n';
}   //end safe-content


/*
    apply_bestrule wendet die beste Regel auf das Lexikon an, das heisst,
    bei den Woerten, bei denen der Test der Regel gelingt, wird das Tag
    geaendert.
```

```
    rein: untagged - der Name des ungetaggtes Korpus
*/

void CFrule::apply_bestrule(char* untagged)
{
  char word[wordlen], tag[taglen];

  for (int i=0; i<maxword; i++)
   {
     if(!taglex[i].is_empty())
          {
            taglex[i].output_tag(tag);
            if (strcmp(from, "") == 0  ||
               strcmp (from, tag) == 0)
             {
               taglex[i].output_word(word);
               if (test(word, untagged))
                    taglex[i].put_tag(to);
             }   //end if-2
          }   //end if-1
   }   //end for
}   //end apply_bestrule


/*
   apply_rule ueberprueft, ob der eine Regel fuer ein Wort anwendbar ist,
   und ist dies der Fall, wird das Tag geaendert.
   rein+raus: tag - das Tag
   rein: word - das betreffende Wort
           untagged - der Name des ungetaggten Korpus
*/

void CFrule::apply_rule(char word[], char tag[], char* untagged)
{
  if (strcmp(from, word) == 0 ||
      strcmp(from, "") == 0)
    if (test(word, untagged))
      strcpy(tag, to);
}


 /*
     test untersucht, ob der Test des aktuellen Templates auf das zu unter-
     suchende Wort anwendbar ist. Je nachdem, welche Aktion in der Regel
     abgelegt ist, wird delete_a, add_a, aneighbor oder amember aufgerufen.
     rein: word - das zu untersuchende Wort
              untagged - der name des ungetaggten Korpus
     raus: 0 / 1 - nicht gefunden / gefunden
 */

int CFrule::test(char word[], char* untagged)
{
  int found=0;

  switch (action)
   {
      case 'd' : found=delete_a(word, untagged); break;
      case 'a' : found=add_a(word, untagged); break;
```

```
      case 'n' : found=aneighbor(word, untagged); break;
      case 'c' : found=amember(word); break;
      case 'b' : found=ablaut(word, untagged); break;
    }   //end switch
  return(found);
}   //end test


 /*
    delete_a untersucht, ob durch die Abtrennung des in der aktuellen
    Regel enthaltenen Affixes vom zu untersuchenden Wort ein im ungetaggten
    Korpus existierendes Wort entsteht.
    rein: word - das zu untersuchende Wort
              untagged - der Name des ungetaggten Korpus
    raus: 0 / 1 - Wort nicht gefunden / Wort gefunden
*/

int CFrule::delete_a(char word[], char* untagged)
{
  pos_word poswords[maxpos];
  int num=strlen(affix), len=strlen(word);

  strcpy(poswords[0].word, "");
  if (where == 'p' && strncmp(word, affix, num) == 0)
    strcpy(poswords[0].word, &word[num]);
  if (where == 's' && strcmp(&word[len-num], affix) == 0)
   {
     strncpy(poswords[0].word, word, len-num);
     poswords[0].word[len-num]=0;
   }   //end if
  if (where == 'i')
    for (int k=1; k<len-num-1; k++)
    {
           if (strncmp(affix, &word[k], num) == 0)
            {
              strncpy(poswords[0].word, word, k);
              strcpy(&poswords[0].word[k], &word[k+num]);
            }   //end if
     }   //end for
  if (strcmp(poswords[0].word, ""))
    return(find_words(poswords, 1, untagged));
  else return(0);
}   //end delete_a


/*
    find_words untersucht, ob ein Wort in einem ungetaggten Korpus enthalten
   ist.
    rein: words - die Liste der zu suchenden Woerter
              num - die Anzahl der zu suchenden Woerter
              untagged - der Name des ungetaggten Korpus
    raus: 0 / 1 - nicht gefunden / gefunden
*/

int CFrule::find_words(pos_word poswords[], int num, char* untagged)
{
  ifstream In;
  int found=0;
```

```
  char tchar, word[wordlen];

  In.open(untagged);
  while (!In.eof() && !found)
   {
     bigr_word(In, word);
     for (int i=0; i<num; i++)
           if (strcmp(word, poswords[i].word) == 0 ||
              (strcmp(&word[1], &poswords[i].word[1]) == 0 &&
               int(word[0]) == int(poswords[i].word[0])-32))
             found=1;
   }   //end while
  In.close();
  return(found);
}   //end find_words


/*
    compare untersucht, ob es sich bei zwei Zeichen um denselben Buchstaben
    handelt, Gross- und Kleinschreibung nicht beachtet.
    rein: char 1, char2 - die zwei Zeichen
    raus: 0 / 1 - nicht gleich / gleich
*/

int CFrule::compare(char char1, char char2)
{
  if (char1 == char2 ||
      char1 == char2 - 32 ||
      char1 - 32 == char2)
    return(1);
  else return(0);
}   //end compare


/*
    add_a untersucht, ob durch das Hinzufuegen des Affixes der aktuellen
    Regel zum zu untersuchenden Wort ein im ungetaggten Korpus existierendes
    Wort entsteht.
    rein: word - das zu untersuchende Wort
             untagged - der Name des ungetaggten Korpus
    raus: 0 / 1 - Wort nicht gefunden / Wort gefunden
*/

int CFrule::add_a(char word[], char* untagged)
{
  pos_word poswords[maxpos];
  int num=strlen(affix), len=strlen(word), found=0;

  if (where == 'p')
   {
     strcpy(poswords[0].word, affix);
     strcpy(&poswords[0].word[num], word);
     found=find_words(poswords, 1, untagged);
   }   //end if
  if (where == 's')
   {
     strcpy(poswords[0].word, word);
     strcpy(&poswords[0].word[len], affix);
```

```
      found=find_words(poswords, 1, untagged);
    }   //end if
  if (where == 'i')
    for (int i=1; i<len-num-1 && !found; i++)
    {
            strncpy(poswords[0].word, word, i);
            strcpy(&poswords[0].word[i], affix);
            strcpy(&poswords[0].word[i+num], &word[i]);
            found=find_words(poswords, 1, untagged);
     }   //end for
  return(found);
}   //end add_a


/*
   aneighbor untersucht, ob das in der aktuellen Regel angegebene
   Nachbarwort im ungetaggten Korpus als Nachbar des zu untersuchenden
   Worts auftaucht.
   rein: word - das zu untersuchende Wort
            untagged - der Name des ungetaggten Korpus
   raus: 0 / 1 Kombination Wort+Nachbar nicht gefunden / gefunden
*/

int CFrule::aneighbor(char word[], char* untagged)
{
  char word1[wordlen], word2[wordlen];
  int found=0;
  ifstream In;

  In.open(untagged);
  while(!In.eof() && !found)
  {
     bigr_pair(In, word1, word2);
     if ((where == 'p' &&
              (strcmp(word1, affix) == 0)  &&
              (strcmp(word2, word) == 0)) ||
             (where == 's' &&
              (strcmp(word1, word) == 0) &&
              (strcmp(word2, affix) == 0)))
            found=1;
     else strcpy(word1, word2);
  }   //end while
  In.close();
  return(found);
}   //end aneighbor


/*
    amember untersucht, ob ein Wort den in der Regel spezifizierten
    Wortteil enthaelt.
    rein: word - das zu untersuchende Wort
    raus: 0 / 1 - Wortteil nicht enthalten / Wortteil enthalten
*/

int CFrule::amember(char word[])
{
  int i, found=0, num=strlen(affix);
```

```
  switch (where)
   {
      case 'p' : if (strncmp(word, affix, num) == 0) found=1; break;
      case 's' : for (i=0, found=1; i<num; i++)
                          if (!(affix[i] == word[strlen(word)-num+i]))
                            { found=0; break;} break;
      case 'c' : for (i=0; i<strlen(word) && !found; i++)
                          if (affix[0] == word[i])
                            found=1;
                        break;
      case 'i' : for (i=1; i<strlen(word)-num-1 && !found; i++)
                          if (strncmp(&word[i], affix, num) == 0)
                            found=1;
                        break;
    }   //end switch
  return(found);
}   //end amember


/*
   ablaut untersucht, ob ein Wort, das durch Ersetzen eines Vokals im aktuellen
   Wort mit dem in der Regel angebebenen entsteht, im ungetaggten Korpus
   enthalten ist.
   rein: word - das aktuelle Wort
            untagged - der Name des ungetaggten Korpus
   raus: 0 / 1 - Wort nicht gefunden / Wort gefunden
*/

int CFrule::ablaut(char word[], char* untagged)
{
  pos_word poswords[maxpos];
  char vowel[3];
  int len, nrpos=0, found=0;

  initialize_poswords(poswords);
  len=strlen(word);
  for (int i=0; i<len; i++)
   {
      if (int(word[i]) == 92 && i<len-2)
            {
              strncpy(vowel, &word[i], 3);
              vowel[3]=0;
            }   //end if
     else
            {
              vowel[0]=word[i];
              vowel[1]=0;
            }   //end else
      if (is_vowel(vowel))
            {
              put_ablaut(word, affix, i, poswords, nrpos);
              nrpos++;
            }   //end if
    }   //end for
  if (nrpos)
     found = find_words(poswords, nrpos, untagged);
  return(found);
}   //end ablaut
```

```
/*************************************************************************/
/*                                                                       */
/*                        Datei: CSTEMPL.H                               */
/*                                                                       */
/*************************************************************************/


class CScandidate;


/*
   CStemplate ist die Klasse der kontext-sensitiven Templates. Ein kontext-
   sensitives Template enthaelt Information ueber die Anzahl der Worter
   die als Kontext dienen, (= read_nr), die Anzahl der zu betrachtenden Woerter
   im Kontext (= inv_nr), ob der Kontext vor oder hinter dem Wort betrachtet
   wird (= where), und die Stelle des zu betrachtenden Wortes im Kontext, falls
   nicht der ganze Kontext betrachtet wird (= how).
*/

class CStemplate
{
// protected:
 public:
  int read_nr, inv_nr;
  char where, how;

 public:
  void read_template(ifstream& In);
  CScandidate make_rule(wordlist words[], CScandidate candidates);
};   //end class
```

```
/*************************************************************************/
/*                                                                       */
/*                          Datei:CSTEMPL.CC                             */
/*                                                                       */
/*************************************************************************/


/*
   read_template liest die Template-Daten aus einer Datei ein und
   initialisiert das Template mit den entsprechenden Informationen.
   rein: In - der stream auf die Template-Datei
*/

void CStemplate::read_template(ifstream& In)
{
  char w[6], h[5], dummy;

  In>>read_nr>>inv_nr;
  In.get(dummy);
  In.get(w, 7, ' ');
  In.get(dummy);
  if (strcmp(w, "vorne") == 0)
    where='f';
  if (strcmp(w, "hinten") == 0)
    where='e';
  if (strcmp(w, "beides") == 0)
    where='b';
  In.get(h, 6, '\n');
  if (strcmp(h, "oder") == 0)
    how='o';
  if (strcmp(h, "genau") == 0)
    how='e';
  for (; !In.eof() && isspace(dummy);)
    In.get(dummy);
  if(!In.eof())
    In.putback(dummy);
}   //end read_template


/*
   make_rule stellt sicher, dass die das aktuelle Template auf das Wort und
   den im Template spezifizierten Kontext angewendet wird.
   rein: words - das aktuelle Wort und dessen Kontext
            candidates - die Liste der Regelkandidaten, aus denen dann die
                                    beste ausgesucht wird
   raus: die aktualisierten Regelkandidaten
*/

CScandidate CStemplate::make_rule(wordlist words[], CScandidate candidates)
{
  if (read_nr == 1)
   {
     if (where == 'f')
          candidates.create_rule(words[3].tag, words[3].texttag, words[2].tag, "", where, how,
read_nr, inv_nr);
     if (where == 'e')
          candidates.create_rule(words[3].tag, words[3].texttag, words[4].tag, "", where, how,
read_nr, inv_nr);
```

```
   }   //end if
  else
  {
     if ((inv_nr == 1) && (how == 'o'))
           for (int i=1; i<=read_nr; i++)
            {
              if (where == 'f')
                candidates.create_rule(words[3].tag, words[3].texttag, words[3-i].tag, "", where,
how, read_nr, inv_nr);
              if (where == 'e')
                candidates.create_rule(words[3].tag, words[3].texttag, words[3+i].tag, "", where,
how, read_nr, inv_nr);
            }    //end for
     if ((inv_nr == 1) && (how == 'e'))
           {
            if (where == 'f')
              candidates.create_rule(words[3].tag, words[3].texttag, words[3-read_nr].tag, "",
where, how, read_nr, inv_nr);
            if (where == 'e')
              candidates.create_rule(words[3].tag, words[3].texttag, words[3+read_nr].tag, "",
where, how, read_nr, inv_nr);
           }   //end if
     if (inv_nr == 2)
           {
            if (where == 'f')
              candidates.create_rule(words[3].tag, words[3].texttag, words[1].tag, words[2].tag,
where, how, read_nr, inv_nr);
            if (where == 'e')
              candidates.create_rule(words[3].tag, words[3].texttag, words[4].tag, words[5].tag,
where, how, read_nr, inv_nr);
            if (where == 'b')
              candidates.create_rule(words[3].tag, words[3].texttag, words[2].tag, words[4].tag,
where, how, read_nr, inv_nr);
           }   //end if
   }   //end else
  return(candidates);
}   //end make_rule
```

```
/*************************************************************************/
/*                                                                       */
/*                        Datei: CSTEMPL.DAT                             */
/*                                                                       */
/*************************************************************************/


1 1 vorne genau
1 1 hinten genau
2 2 beides genau
2 2 vorne genau
2 2 hinten genau
2 1 vorne oder
2 1 hinten oder
3 1 vorne oder
3 1 hinten oder
2 1 vorne genau
2 1 hinten genau
```

```
/*************************************************************************/
/*                                                                       */
/*                          Datei: CSCAND.H                              */
/*                                                                       */
/*************************************************************************/


/*
   CScandidate ist die Klasse der kontext-sensitiven Regelkandidaten. Sie
   enthaelt die Liste der Kandidaten.
*/

class CScandidate
{
  cs_rulecand *candidate;

  void build_cand(cs_rulecand *cand, char from[], char to[], char neigh1[], char neigh2[], char
where, char how, int read_nr, int inv_nr);
 public:
  CScandidate() {candidate=0;}
  void reset();
  int empty();
  void create_rule(char tag[], char texttag[], char neigh1[], char neigh2[], char where, char how,
int read_nr, int inv_nr);
  void change_score(wordlist words[]);
  cs_rulecand find_bestrule();
};   //end class
```

```
/*************************************************************************/
/*                                                                       */
/*                          Datei: CSCAND.CC                             */
/*                                                                       */
/*************************************************************************/


/*
   reset setzt die Liste der Kandidaten auf null.
*/

void CScandidate::reset()
{
  cs_rulecand *ptr=candidate, *next;

  candidate=0;
  while (ptr != 0)
   {
     next = ptr->next;
     delete ptr;
     ptr=next;
   }   //end while
}   //end reset


/*
   empty ueberprueft, ob die Liste der Regelkandidaten leer ist.
   raus: 0 / 1 - nicht leer / leer
*/

int CScandidate::empty()
{
  if (!candidate)
    return(1);
  else return(0);
}   //end empty


/*
   create_rule ueberprueft, ob fuer das aktuelle Wort und seinen Nachbar bzw.
   und seine Nachbarn bereits ein Regelkandidat besteht. Ist dies der Fall,
   wird die Wertung aktualisiert, sonst wird ein neuer Kandidat geschaffen.
   rein: tag - das Tag des aktuellen Worts
            texttag - das vorgegebene Tag fuer das aktuelle Wort
            neigh1 - das erste Nachbarwort
            neigh2 - das zweite Nachbarwort, wird nur ein Nachbar untersucht,
                     ist es der leere String ""
            where - Ort, an dem die Regel ansetzt
            how - Art der Ueberpruefung
            read_nr - Anzahl der zu lesenden Woerter
            inv_nr - Anzahl der zu betrachtenden Woerter
*/

void CScandidate::create_rule(char tag[], char texttag[], char neigh1[], char neigh2[], char
where, char how, int read_nr, int inv_nr)
{
  int found=0;
  cs_rulecand *cand;
```

```
  if (strcmp(neigh1, ""))
   {
      for (cs_rulecand *ptr=candidate; !found && ptr != 0; ptr=ptr->next)
            {
             if ((strcmp(tag, ptr->from) == 0) &&
                 (strcmp(texttag, ptr->to) == 0) &&
                 (strcmp(neigh1, ptr->neighbor1) == 0) &&
                 ((strcmp(neigh2, "") == 0) ||
                  (strcmp(neigh2, ptr->neighbor2) == 0)) &&
                 (where == ptr->where) &&
                 (how == ptr->how) &&
                 (read_nr == ptr->read_nr) &&
                 (inv_nr == ptr->inv_nr))
              {
                 ptr->score++;
                 found=1;
               }   //end if
            }   //end for
     if (!found)
            {
             cand = new cs_rulecand;
             build_cand(cand, tag, texttag, neigh1, neigh2, where, how, read_nr, inv_nr);
             cand->next=candidate;
             candidate=cand;
            }   //end if-2
   }   //end if-1
}    //end create_rule


/*
   build_cand initialisiert den neu gebildenten Regelkandidaten.
   rein+raus: cand - der Pointer auf den Kandidaten
   rein: from - das Ausgangstag
            to - das Zieltag
            neigh1, neigh2 - die beiden Nachbartags
            where - Ort, an dem die Regel ansetzt
            how - Art der Ueberpruefung
            read_nr - Anzahl der zu lesenden Woerter
            inv_nr - Anzahl der zu betrachtenden Woerter
*/

void CScandidate::build_cand(cs_rulecand *cand, char from[], char to[], char neigh1[], char
neigh2[], char where, char how, int read_nr, int inv_nr)
{
  strcpy(cand->from, from);
  strcpy(cand->to, to);
  strcpy(cand->neighbor1, neigh1);
  strcpy(cand->neighbor2, neigh2);
  cand->where=where;
  cand->how=how;
  cand->read_nr=read_nr;
  cand->inv_nr=inv_nr;
  cand->score=1.0;
}   //end build_cand


/*
   change_score ueberprueft, ob der aktuelle Regelkandidat auf den Kontext
```

```
   eines Wortes anwendbar ist und eine negative Veraenderung bewirkt. Ist
   dies der Fall, wird der Wert dementsprechend veraendert.
   rein: words - das Wort mit seinem Kontext
*/

void CScandidate::change_score(wordlist words[])
{
  CSrule testrule;

  for (cs_rulecand *ptr=candidate; ptr != 0; ptr=ptr->next)
   {
     testrule.initialize(*ptr);
     if ((strcmp(ptr->from, words[3].tag) == 0) &&
           strcmp(ptr->to, words[3].texttag) &&
           testrule.test(words))
         ptr->score--;
    }   //end for
}   //end change_score


/*
   find_bestrule sucht unter den Regelkandidaten denjenigen aus, der die
   beste Wertung bekommen hat.
   raus: der Inhalt des besten Regelkandidaten in der Liste
*/

cs_rulecand CScandidate::find_bestrule()
{
  cs_rulecand *ptr, *best;
  float bestscore=0.0;

  for (ptr=candidate; ptr != 0; ptr=ptr->next)
   {
     if (bestscore < ptr->score)
         {
           best=ptr;
           bestscore=ptr->score;
         }   //end if
    }   //end for
  return(*best);
}   //end find_bestrule
```

```
/*************************************************************************/
/*                                                                       */
/*                        Datei: CSRULE.H                                */
/*                                                                       */
/*************************************************************************/


/*
   CSrule ist die Klasse der kontext-sensitiven Regeln. Eine Regel enthaelt
   Information ueber das Ausgangstag, bei dessen Auffinden die Regel aktiviert
   wird, das Zieltag, in das das gefundene tag geaendert werden soll, und den
   oder die Nachbarn, durch die die Regel aktiviert wird.
*/

class CSrule : public CStemplate
{
  char from[taglen], to[taglen], neighbor1[taglen], neighbor2[taglen];

 public:
  int test(wordlist words[]);
  void initialize(cs_rulecand cand);
  void safe_content(ofstream& Out);
  void apply_rule(char* infile, char* outfile);
  void read_content(ifstream& In);
};   //end class
```

```
/*************************************************************************/
/*                                                                       */
/*                        Datei: CSRULE.CC                               */
/*                                                                       */
/*************************************************************************/


/*
   initialize initialisiert die kontext-sensitiven Regeln mit den Angaben aus
   dem Regelkandidaten, also mit Ausgangs- und Zieltag, mit dem oder den
   Nachbarwoertern, mit dem Ort und der Art, wo und wie gelesen wird, mit der
   Anzahl der zu lesenden und der daraus zu untersuchenden Nachbarwoerter.
   rein: cand - der Regelkandidat
*/

void CSrule::initialize(cs_rulecand cand)
{
  strcpy(from, cand.from);
  strcpy(to, cand.to);
   strcpy(neighbor1, cand.neighbor1);
   strcpy(neighbor2, cand.neighbor2);
  where=cand.where;
  how=cand.how;
  read_nr=cand.read_nr;
  inv_nr=cand.inv_nr;
}   //end initialize


/*
   safe_content schreibt den Inhalt der Regel, also das Ausgangstag, das
   Zieltag, den bzw. die Nachbarn, den Ort, die Art und Weise, die Anzahl der
   zu lesenden und der zu untersuchenden Woerter in die Datei 'csrules.dat'.
   rein: Out - der Stream auf die Ausgabedatei
   Nebeneffekt: die Regel wird in den Ausgabestream geschrieben
*/

void CSrule::safe_content(ofstream& Out)
{
  Out<<from<<' '<<to<<' '<<neighbor1<<' ';
  Out<<neighbor2<<' ';
  switch (where)
   {
      case 'f' : Out<<"vorne "; break;
      case 'e' : Out<<"hinten "; break;
      case 'b' : Out<<"beides "; break;
      default : error("komische Regel");
   }   //end switch
  switch (how)
   {
      case 'o' : Out<<"oder "; break;
      case 'e' : Out<<"genau "; break;
      default : error("komische Regel");
   }   //end switch
  Out<<read_nr<<' '<<inv_nr<<'\n';
}   //end safe_content


/*
```

```
   apply_rule wendet die aktuelle Regel auf eine Datei an und schreibt das
   Ergebnis in eine zweite Datei.
   rein: infile - der Name der Ausgangsdatei
            outfile - der Name der Zieldatei
            Nebeneffekt: das Korpus, das durch die Regelanwendung entstanden
                         ist, wird in die Ausgabedatei geschrieben
*/

void CSrule::apply_rule(char* infile, char* outfile)
{
  ifstream In;
  ofstream Out;
  wordlist words[7];

  In.open(infile);
  Out.open(outfile);
  if (!In)
    error("Infile nicht gefunden.");
  else
   {
     for (int j=4; j<7; j++)
          read_word(In, words[j].word, words[j].texttag, words[j].tag);
     while (!In.eof() || strcmp(words[4].tag, ""))
          {
            for (int i=1; i<7; i++)
             {
               strcpy(words[i-1].word, words[i].word);
               strcpy(words[i-1].tag, words[i].tag);
               strcpy(words[i-1].texttag, words[i].texttag);
             }   //end for
            if (In.eof())
            {
               strcpy(words[6].word, "");
               strcpy(words[6].tag, "");
             }   //end if
            else read_word(In, words[6].word, words[6].texttag, words[6].tag);
            if ((strcmp(words[3].tag, from) == 0) &&
                (strcmp(words[3].texttag, to) == 0) &&
                (test(words)))
               Out<<words[3].word<<'|'<<words[3].texttag<<'|'<<to<<'\n';
             else Out<<words[3].word<<'|'<<words[3].texttag<<'|'<<words[3].tag<<'\n';
          }   //end while
   }   //end else
  In.close();
  Out.close();
}   //end apply_rule


/*
   test ueberprueft, ob der Test der Regel auf ein Wort anwendbar ist, d.h.
   es wird getestet, ob die durch die Angaben in der Regel spezifizierten
   Nachbarn die von der Regel geforderten Tags haben.
   rein: words - das aktuelle Wort und seine Nachbarn
   raus: 0 / 1 - trifft nicht zu / trifft zu
*/

int CSrule::test(wordlist words[])
{
```

```
char neigh1[taglen], neigh2[taglen];
int found=0;

if (read_nr == 1)
 {
   if (where == 'f')
         strcpy(neigh1, words[2].tag);
   if (where == 'e')
         strcpy(neigh1, words[4].tag);
    if (strcmp(neigh1, neighbor1) == 0)
         found=1;
 }  //end if
if ((inv_nr == 1) && (how == 'o'))
  for (int i=1; i<=read_nr && !found; i++)
  {
         if (where == 'f')
           strcpy(neigh1, words[3-i].tag);
         if (where == 'e')
           strcpy(neigh1, words[3+i].tag);
         if (strcmp(neigh1, neighbor1) == 0)
           found=1;
   }   //end for
else
 {
   if ((inv_nr == 1) && (how == 'e'))
         {
          if (where == 'f')
           {
             strcpy(neigh1, words[3-read_nr].tag);
             strcpy(neigh2, "");
           }  //end if-2
          if (where == 'e')
           {
             strcpy(neigh1, words[3+read_nr].tag);
             strcpy(neigh2, "");
           }  //end if-2
         }  //end if-1
   if (inv_nr == 2)
         {
          if (where == 'f')
           {
             strcpy(neigh1,  words[1].tag);
             strcpy(neigh2, words[2].tag);
           }  //end if-2
          if (where == 'e')
           {
             strcpy(neigh1,  words[4].tag);
             strcpy(neigh2, words[5].tag);
           }  //end if-2
          if (where == 'b')
           {
             strcpy(neigh1,  words[2].tag);
             strcpy(neigh2, words[4].tag);
           }  //end if-2
         }  //end if-1
   if ((strcmp(neigh1, neighbor1) == 0) &&
          ((strcmp(neigh2, "") == 0) ||
           (strcmp(neigh2, neighbor2) == 0)))
```

```
      found=1;
    }   //end else
  return(found);
}   //end test


/*
   read_content liest den Inhalt einer Regel aus der Eingabedatei ein.
   rein: In - der Stream auf die Eingabedatei
   Nebeneffekt: die Regel wird initialisiert
*/

void CSrule::read_content(ifstream& In)
{
  char dummy, w[16], h[16];

  In.get(from, taglen, ' ');
  In.get(dummy);
  In.get(to, taglen, ' ');
  In.get(dummy);
  In.get(neighbor1, wordlen, ' ');
  In.get(dummy);
  In.get(neighbor2, wordlen, ' ');
  In.get(dummy);
  In.get(w, 7, ' ');
  In.get(dummy);
  if (strcmp(w, "vorne") == 0)
    where='f';
  if (strcmp(w, "hinten") == 0)
    where='e';
  if (strcmp(w, "beides") == 0)
    where='b';
  In.get(h, 6, ' ');
  In.get(dummy);
  if (strcmp(h, "oder") == 0)
    how='o';
  if (strcmp(h, "genau") == 0)
    how='e';
  In>>read_nr>>inv_nr;
  In.get(dummy);
}   //end read_content
```

```
/*************************************************************************/
/*                                                                       */
/*                          Datei: UTILS.H                               */
/*                                                                       */
/*************************************************************************/


/*
   error bricht das Programm mit einer Fehlermeldung ab, wenn an einer
   Stelle im Programm ein Fehler aufgetreten ist.
   rein: str - die Fehlermeldung
*/

void error(char str[])
{
  cerr<<str<<'\n';
  exit(1);
}   //end error


/*
   read_word liest ein Wort aus einer getaggten Datei zusammen mit dem vor-
   gegebenen und dem vom Programm gewaehlten Tag. Ist noch kein vom Programm
   gewaehltes Tag eingetragen, wird diese Variable mit nS:M initialisiert.
   rein+raus: word - das gelesene Wort
                texttag - das vorgegebene Tag
                tag - das vom Programm gewaehlte Tag
   rein: In - der Stream auf die Eingabedatei
*/

void read_word(ifstream& In, char word[], char texttag[], char tag[])
{
  char dummy=' ';

  while (isspace(dummy) && !In.eof())
    In.get(dummy);
  if (In.eof())
   {
     strcpy(word, "");
     strcpy(tag, "");
     strcpy(texttag, "");
    }  //end if
  else
   {
     In.putback(dummy);
     In.get(word, wordlen, '|');
     In.get(dummy);
     In.get(texttag, taglen, '|');
     In.get(dummy);
     In.get(tag, taglen, '\n');
     In.get(dummy);
     if (strcmp(tag, "") == 0)
           strcpy(tag, "nS:M");
    } //end else
}   //end read_word


/*
```

```
   get_word liest ein Wort aus aus einem ungetagten Korpus ein.
   rein+raus: word - das einzulesende Wort
   rein: In - der Stream auf die Korpusdatei
*/

void get_word(ifstream& In, char word[])
{
  char dummy=' ';
  int i, len, found=0;

  strcpy(word, "");
  while (!In.eof() && isspace(dummy) ||
            (ispunct(dummy) && !(int(dummy) == 92)))
    In.get(dummy);
  In.putback(dummy);
  for (i=0; i<wordlen && !found && !In.eof(); i++)
   {
      In.get(word[i]);
      if (isspace(word[i]))
     {
            word[i]=0;
            found=1;
      }   //end if
   }   //end for
  word[wordlen]=0;
  len=strlen(word);
  while (ispunct(word[len-1]) && !(int(word[len-1]) == 45))
   {
      word[len-1]=0;
      len--;
    }   //end while
}   //end get_word


/*
   is_smaller untersucht, ob ein Wort in einer alphabetischen Reihenfolge vor
   dem anderen steht, oder nicht.
   rein: word1, word2 - die zu vergleichenden Woerter
   raus: 0 / 1 - steht dahinter / davor
*/

int is_smaller(char word1[], char word2[])
{
  int i;

  if (strcmp(word1, word2) == 0)
    return(0);
  else
   {
      for (i=0; word1[i] == word2[i]; i++);
      return(word1[i] < word2[i]);
   }
}   //end is_smaller


/*
   statisitics berechnet die Erfolgsquote des Programms, d.h. die Anzahl
   und den Prozentsatz der falsch getaggten Woerter und gibt sie aus.
```

```
   rein infile - der Name des Korpus, aus dem die Statistik erstellt werden
                 soll
*/

void statistics(char* infile)
{
  ifstream In;
  char word[wordlen], texttag[taglen], tag[taglen];
  float wordcount=0.0, errcount=0.0;

  In.open(infile);
  while (!In.eof())
   {
     read_word(In, word, texttag, tag);
     wordcount++;
     if (strcmp(texttag, tag))
          errcount++;
   }   //end while
    In.close();
  cout<<"Von "<<wordcount<<" Woertern wurden "<<errcount<<" falsch getaggt.";
  cout<<"\nFehlerprozentsatz: "<< errcount / wordcount<<"\n\n\n";
}   //end statistics


/*
   stat_unknown berechnet den Prozentsatz der unbekannten Woerter, also
   der Woerter, die nicht im Lexikon enthalten sind, und gibt ihn dann aus.
   rein: unknown - die Anzahl der unbekannten Woerter
*/

void stat_unknown(int unknown)
{
  float percents=unknown * 100 / all_words;

  cout<<"Von "<<all_words<<" Woertern wurden "<<unknown<<" nicht im Lexikon\n";
  cout<<" gefunden. Das ist ein Prozentsatz von "<<percents<<"\n\n\n";
}   //end stat_unknown
```

```
/*************************************************************************/
/*                                                                       */
/*                          Datei: HASH.CC                               */
/*                                                                       */
/*************************************************************************/


int primary_hash(char word[]);
int power(int base, int expo);
int rehash(Tag entry, int index, int key, int lexcount);

/*
    hash_insert fuegt das aktuelle Wort an der richtigen Stelle in das
    Lexikon ein. Die hierbei verwendete Methode ist das ordered double
    hashing. primary_hash errechnet dabei die erste Position, an der
    das aktuelle Wort in das Lexikon eingefuegt werden kann. Ist diese
    schon besetzt, wird rehash aufgerufen.
    rein: entry - der Lexikoneintrag des aktuellen Worts
            lexcount - die Anzahl der bereits ins Lexikon eingetragenen
                       Woerter
    raus: die aktualisierte Anzahl der Woerter
*/

int hash_insert(Tag entry, int lexcount)
{
  Tag temp;
  char word[wordlen], texttag[taglen];
  int index, key;

   entry.output_word(word);
  index=primary_hash(word);
  if (taglex[index].is_empty())
   {
     taglex[index] = entry;
     lexcount++;
   }   //end if
  else
   {
     if (taglex[index].is_equal(word))
           {
             entry.get_texttag(texttag, 0);
             taglex[index].put_texttag(texttag);
           }   //end if
     else
           {
             if (taglex[index].is_smaller_than(word))
               lexcount=rehash(entry, index, index, lexcount);
             else
              {
                 temp=taglex[index];
                 taglex[index] = entry;
                 temp.output_word(word);
                 key=primary_hash(word);
                 lexcount=rehash(temp, index, key, lexcount);
               }   //end else-3
           }   //end else-2
   }   //end else-1
  return(lexcount);
```

```
}   //end hash_insert


/*
   hash stellt fest, ob ein Wort im Lexikon vorhanden ist und weist der
   Variable 'tag' das im Lexikon vergebene Tag zu.
   rein+raus: tag - das Tag, das dem zu suchenden Wort zugeordnet ist
   rein: word - das zu suchende Wort
   raus: -1, wenn das Wort nicht vorhanden ist, sonst die Position, an der
            das Wort eingetragen ist
*/

int hash(char word[], char tag[])
{
  int index, step, found;

  index=primary_hash(word);
  if (!taglex[index].is_empty())
   {
      if (taglex[index].is_equal(word))
            {
              taglex[index].output_tag(tag);
             return(index);
            }   //end if-2
     else
            {
             step = 1+index % maxword;
             index = (index + step) % maxword;
             for (found=-2; found<-1; index = (index + step) % maxword)
              {
                 if (taglex[index].is_equal(word))
                       {
                         taglex[index].output_tag(tag);
                         found=index;
                       }   //end if-2
                 else if (!taglex[index].is_smaller_than(word) ||
                             taglex[index].is_empty())
                       found=-1;
              }   //end for
             return(found);
            }   //end else
    }   //end if-1
  else return(-1);
}   //end hash


/*
   primary_hash errechnet die erste Position, an der das aktuelle Wort in
   das Lexikon eingefuegt werden kann.
   rein: word - das aktuelle Wort
   raus: die moegliche Position
*/

int primary_hash(char word[])
{
  int key=0;
  int i;
```

```
  for (i=0; i<strlen(word) && i<6; i++)
    key += int((int(word[i]) % 10) * power(10, i));
  return(key % maxword);
}   //end primary_hash


/*
   power berechnet exp(x, y).
   rein: base - die Basis
             expo - der Exponent
   raus: das Ergebnis
*/

int power(int base, int expo)
{
  int x=1;

  for (int i=0; i<expo; i++)
    x = x*base;
  return(x);
}   //end power

/*
    rehash durchsucht mittels der zweiten Hash-Funktion das Lexikon so
    lange, bis eine leere Position gefunden ist, an der das Wort eingetragen
    werden kann.
    rein: entry - der Lexikoneintrag fuer das aktuelle Wort
              index - die letzte bereits untersuchte, belegte Position
              key - die erste von primary_hash errechnete Position fuer das Wort
              lexcount - die Anzahl der bereits in das Lexikon eingetragenen
                         Woerter
    raus: die aktualisierte Anzahl der im Lexikon eingetragenen Woerter
*/

int rehash(Tag entry, int index, int key, int lexcount)
{
  int step, flag=0;
  char word[wordlen], texttag[taglen];
  Tag temp;

  step = 1+key % maxword;
  index = (index + step) % maxword;
   entry.output_word(word);
  while (!flag)
   {
      if (taglex[index].is_equal(word))
            {
              flag = 1;
               entry.get_texttag(texttag, 0);
               taglex[index].put_texttag(texttag);
            }   //end if
     else
            {
              if (taglex[index].is_empty())
               {
                  taglex[index]=entry;
                  lexcount++;
                  flag=1;
```

```
            }   //end if
           else
            {
              if (taglex[index].is_smaller_than(word))
                    index = (index + step) % maxword;
              else
                    {
                      temp=taglex[index];
                      taglex[index]=entry;
                      entry=temp;
                      entry.output_word(word);
                     step = 1 + primary_hash(word) % maxword;
                    }  //end else-3
            }   //end else-2
          }   //end else-1
   }   //end while
  return(lexcount);
}   //end rehash
```

```
/*************************************************************************/
/*                                                                       */
/*                          Datei: BIGRAM.CC                             */
/*                                                                       */
/*************************************************************************/


void bigrams(char* untagged);
int find_bigram(char word1[], char word2[]);


/*
   bigrams durchsucht das ungetaggte Korpus und ueberprueft fuer jedes
   Bigramm, ob es schon in der Liste der Bigramme in der Datei 'bigrams.dat'
   steht. Wenn nicht, wird es eingetragen.
   rein: der Name des ungetaggten Korpus
   Nebeneffekt: die Liste der Bigramme wird in die Datei 'bigrams.dat'
                geschrieben
*/

void bigrams(char* untagged)
{
  ifstream In;
  ofstream Out;
  char word1[wordlen], word2[wordlen];

   In.open(untagged);
  get_word(In, word1);
  while (!In.eof())
   {
     get_word(In, word2);
     if (!find_bigram(word1, word2))
          {
            Out.open("bigrams.dat", ios::app);
            Out<<word1<<' '<<word2<<'\n';
            Out.flush();
            Out.close();
          }   //end if
     strcpy(word1, word2);
   }   //end while
  In.close();
}   //end bigrams


/*
   find_bigrams untersucht, ob ein Bigramm schon in der Datei 'bigrams.dat'
   vorhanden ist.
   rein: word1, word2 - die zwei Woerter des Bigramms
   raus: 0 / 1 - nicht gefunden / gefunden
*/

int find_bigram(char word1[], char word2[])
{
  ifstream In("bigrams.dat");
  char w1[wordlen], w2[wordlen], dummy;
  int found=0;

  if (In)
```

```
    while (!In.eof() && !found)
    {
            In.get(w1, wordlen, ' ');
            In.get(dummy);
            In.get(w2, wordlen, '\n');
            In.get(dummy);
            if (strcmp(w1, word1) == 0 &&
                strcmp(w2, word2) == 0)
              found=1;
     }   //end while
  In.close();
  return(found);
}   //end find_bigram


/*
   bigr_word liest das erste Wort eines Bigramms aus der Datei 'bigrams.dat'
   ein.
   rein+raus: word1 - das zu lesende Wort
   rein: In - der Stream auf die Datei 'bigrams.dat'
*/

void bigr_word(ifstream& In, char word1[])
{
  char word2[wordlen], dummy;

  In.get(word1, wordlen, ' ');
  In.get(dummy);
  In.get(word2, wordlen, '\n');
  In.get(dummy);
}   //end bigr_word


/*
   bigr_pair liest die beiden Woerter eines Bigramms aus der Datei
   'bigrams.dat' ein.
   rein+raus: word1, word 2 - die zwei Woerter des zu lesenden Bigramms
   rein: In - der Stream auf die Datei 'bigrams.dat'
*/

void bigr_pair(ifstream& In, char word1[], char word2[])
{
  char dummy;

  In.get(word1, wordlen, ' ');
  In.get(dummy);
  In.get(word2, wordlen, '\n');
  In.get(dummy);
}   //end bigr_pair
```

Anhang C - Die aus 1.000 Sätzen gelernten kontext-sensitiven Regeln

Ausgangs-tag	Zieltag	Nachbar 1	Nachbar 2	Wo?	Wie?	Anzahl 1	Anzahl 2
Swgd:R	Smn:R	nS:M		hinten	oder	2	1
Pmwsd:R	Sma:R	aS:M		hinten	oder	3	1
Swna:R	Pmwsna:R	nP:M		hinten	oder	3	1
X:P	X:D	X:I		hinten	genau	1	1
Ssna:R	Smn:R	nS:M		hinten	oder	2	1
Swna:R	Pmwsna:R	nP:F		hinten	oder	3	1
nS:M	dS:M	X:PR		vorne	oder	2	1
Smg:R	Ssg:R	gS:N		hinten	oder	3	1
Ssna:R	Ssna:O	3SIE:V		hinten	genau	1	1
Swgd:R	Pmwsg:R	gP:M		hinten	oder	2	1
Swna:R	Pmwsna:R	nP:N		hinten	oder	3	1
Swna:R	Pmwsna:R	aP:N		hinten	oder	2	1
nS:M	aS:M	Pmwsd:R		vorne	oder	3	1
Ssd:R	Smd:R	dS:M		hinten	oder	3	1
Pmwsd:R	Sma:R	aS:M		hinten	oder	2	1
Smd:R	Ssd:R	dS:N		hinten	oder	2	1
Swgd:R	Smn:O	X:I		vorne	genau	1	1
Swna:R	Pmwsna:R	o5:A		hinten	oder	2	1
Swgd:R	Pmwsg:R	gP:F		hinten	oder	2	1
Swna:R	Pmwsna:R	aP:M		hinten	oder	3	1
nS:M	dS:M	Smd:R		vorne	oder	2	1
Swna:R	Pmwsna:R	nP:T		hinten	oder	3	1
Swna:R	Pmwsna:R	aP:F		hinten	oder	2	1
nS:N	dS:N	X:PR		vorne	oder	3	1
i:V	13PIE:V	X:D		hinten	genau	1	1
d:O	0:O	nS:M		hinten	genau	1	1
Swgd:O	Smn:O	nS:M		hinten	oder	3	1
Swgd:R	Pmwsg:R	gP:N		hinten	oder	2	1
i:V	13PIE:V	X:P		hinten	genau	1	1
aS:M	dS:M	X:PR		vorne	oder	2	1
i:V	13PIE:V	Pmwsna:R		hinten	genau	1	1
nS:F	dS:F	X:P	Swgd:R	vorne	genau	2	2
nS:F	gS:F	Swgd:R		vorne	genau	1	1
Swna:R	Swna:O	nS:F	X:I	vorne	genau	2	2
gS:F	nS:F	o4:A		vorne	oder	2	1
Swna:R	Pmwsna:O	nP:F		vorne	genau	2	1
dS:F	aS:F	Swna:R		vorne	genau	2	1
Ssna:R	Ssna:O	n:O		hinten	oder	2	1
Ssna:R	Ssna:O	X:C		hinten	genau	1	1
nS:M	dS:M	7:O		vorne	oder	3	1
nP:M	dP:M	Pmwsd:R		vorne	oder	2	1
nS:M	aS:M	Sma:R		vorne	oder	3	1
dS:F	aS:F	X:P	Swna:R	vorne	genau	2	2
dS:F	nS:F	Swna:R		vorne	oder	2	1
Swna:R	Pmwsna:R	gP:oT		hinten	oder	2	1
Swgd:R	Pmwsg:R	gP:oT		hinten	oder	3	1
nP:N	gP:N	Swgd:R		vorne	oder	2	1
Swgd:R	Pmwsg:R	gP:N		hinten	oder	2	1

dS:F	aS:F	Swna:R		vorne	genau	1	1
nS:pM	dS:pM	X:P		vorne	genau	1	1
nS:pM	dS:pM	dS:pM		vorne	oder	3	1
nS:M	aS:M	5:O		vorne	oder	3	1
Swgd:R	Smn:R	o4:A		hinten	oder	2	1
nS:F	dS:F	Swgd:R		vorne	oder	2	1
13PIE:V	i:V	X:D	X:I	beides	genau	2	2
X:P	X:D	X:C		hinten	genau	1	1
i:V	0:O	aS:N		hinten	genau	1	1
Swgd:R	Smn:O	X:I	X:C	vorne	genau	2	2
Ssna:R	Ssna:O	X:D		hinten	genau	1	1
aS:M	dS:M	Smd:R		vorne	oder	2	1
aS:F	dS:F	X:P	Swgd:R	vorne	genau	2	2
i:V	13PIE:V	i:V	X:I	beides	genau	2	2
gP:oT	dP:oT	Pmwsd:R		vorne	oder	3	1
aS:pM	dS:pM	dS:pM		vorne	oder	2	1
X:C	X:P	X:D	0:U	hinten	genau	2	2
nP:T	aP:T	X:P		vorne	oder	3	1
nS:N	dS:N	Ssd:R		vorne	oder	3	1
dS:F	aS:F	X:P	4:O	vorne	genau	2	2
dS:oN	nS:oN	nS:aM		vorne	genau	1	1
gP:M	nP:M	Swna:R		vorne	oder	2	1
Swna:R	Pmwsna:R	nP:M		hinten	genau	1	1
X:D	X:P	dP:N		hinten	genau	1	1
nP:M	aP:M	o8:A		hinten	oder	3	1
Swna:O	Pmwsna:O	13PIE:V		hinten	genau	2	1
dS:N	nS:N	X:I	Ssna:R	vorne	genau	2	2
dS:N	aS:N	Ssna:R	X:D	beides	genau	2	2
a:O	d:O	aS:N		hinten	oder	3	1
dS:N	nS:N	o0:A		hinten	genau	2	1
n:O	a:O	n:O	X:D	beides	genau	2	2
Ssna:R	Ssna:O	13SKA:V		hinten	genau	1	1
Ssna:R	Ssna:O	13SIA:V		hinten	genau	1	1
13PIE:V	i:V	X:I	Swna:R	hinten	genau	2	2
Swna:R	Pmwsna:O	13PIE:V		hinten	oder	2	1
Swna:R	Pmwsna:R	0:U		hinten	oder	2	1
Pmwsd:R	Sma:R	dS:M		hinten	oder	2	1
Swna:R	Pmwsna:O	gP:M		vorne	genau	2	1
X:O	X:D	X:I		vorne	genau	1	1
13PIE:V	i:V	0:O		vorne	oder	3	1
nS:N	dS:oN	X:P		vorne	genau	1	1
Ssna:R	Ssna:O	Ssna:O		hinten	oder	3	1
Ssna:R	X:D	X:I		hinten	genau	1	1
X:D	X:P	Swgd:R		hinten	genau	1	1
Ssna:R	Ssna:O	gS:N	X:I	vorne	genau	2	2
Ssna:R	Ssna:O	nS:F		hinten	oder	3	1
i:V	0:O	nS:M		hinten	oder	2	1
dS:oF	nS:oF	X:I		vorne	oder	3	1
nS:F	aS:F	X:P	Swna:R	vorne	genau	2	2
nP:M	aP:M	0:U	pA:V	beides	genau	2	2
nS:pF	dS:pF	gS:M		hinten	oder	3	1
nS:pF	aS:pF	X:P		vorne	oder	2	1
Swna:R	Swna:O	nS:pF		vorne	oder	3	1

a:O	d:O	nS:pF		hinten	oder	3	1
Swna:R	Pmwsna:O	X:I	Ssna:R	beides	genau	2	2
dS:pF	aS:pF	aS:pM		vorne	oder	3	1
dS:pM	aS:pM	aS:pM		hinten	oder	3	1
nS:aF	dS:aF	X:P	Swgd:R	vorne	genau	2	2
gP:M	nP:M	o4:A		vorne	oder	2	1
pA:V	3SIE:V	Swna:R		hinten	genau	1	1
gP:oT	nP:oT	X:I		hinten	genau	1	1
gP:oT	aP:oT	z:V		hinten	oder	3	1
aS:F	nS:F	a:O		vorne	genau	2	1
Swna:R	Pmwsna:O	Pmwsd:R		hinten	oder	2	1
X:D	o0:A	aS:oN		hinten	oder	3	1
aS:M	dS:M	dS:pM		hinten	oder	2	1
aS:pM	nS:pM	i:V		vorne	genau	2	1
1SIE:V	13SKE:V	pA:V		vorne	genau	2	1
1SIE:V	13SKE:V	dS:M		hinten	oder	3	1
1SIE:V	13SKE:V	o5:A		vorne	genau	2	1
i:V	13PIE:V	nP:aT		hinten	oder	3	1
i:V	13PIE:V	0:U		hinten	genau	1	1
aP:F	dP:F	dP:F		hinten	genau	2	1
aP:F	nP:F	nP:F		vorne	oder	3	1
nS:M	dS:M	Ssd:R		vorne	oder	3	1
Ssd:R	Smd:R	dS:M		hinten	oder	2	1
1SIE:V	13SKE:V	X:P		hinten	genau	1	1
nS:pF	dS:pF	dS:M		hinten	oder	3	1
dS:M	aS:M	aS:N		vorne	oder	2	1
o0:A	X:D	Smd:R		vorne	oder	3	1
Ssna:O	Ssg:O	gS:N		hinten	oder	3	1
aS:F	gS:F	Swgd:R		vorne	genau	1	1
Swna:R	Swna:O	X:D	o0:A	hinten	genau	2	2
pA:V	3SIE:V	n:O		hinten	genau	1	1
Swna:R	Swna:O	a:O		hinten	genau	2	1
13PIE:V	i:V	13SIA:V		hinten	oder	2	1
Swna:R	Pmwsna:O	Pmwsna:O		vorne	oder	3	1
nS:M	dS:M	dS:pM		vorne	oder	3	1
dS:pM	nS:pM	13SIA:V		vorne	oder	3	1
nS:N	aS:N	X:P	Ssna:R	vorne	genau	2	2
dS:M	aS:M	Sma:R	X:C	beides	genau	2	2
Swna:R	Pmwsna:R	aP:MF		hinten	oder	3	1
Swna:R	Pmwsna:O	X:I	X:C	beides	genau	2	2
nS:M	gP:M	Swgd:R	0:U	vorne	genau	2	2
Swna:O	Pmwsna:O	nP:F		hinten	oder	3	1
nP:pM	gP:pM	X:D		hinten	oder	3	1
Swgd:R	Swgd:O	Swna:R	nS:F	hinten	genau	2	2
nS:pF	gS:pF	gS:pF		hinten	oder	3	1
Swna:R	Pmwsna:O	dP:N		vorne	genau	2	1
1SIE:V	13SKE:V	Pmwsg:R		vorne	oder	3	1
nS:M	nS:N	X:C	Ssg:R	beides	genau	2	2
Swgd:R	Smn:O	Pmwsd:R		hinten	genau	2	1
Swgd:R	Pmwsg:R	5:O	o5:A	hinten	genau	2	2
dS:oF	aS:oF	X:P	Swna:R	vorne	genau	2	2
dS:aF	gS:aF	nS:pM		hinten	oder	3	1
nS:F	dS:F	X:P	o6:A	vorne	genau	2	2

Ssd:O	Smd:O	dS:M		hinten	oder	3	1
X:P	X:C	X:I	n:O	beides	genau	2	2
X:D	X:P	Pmwsd:R	dP:N	hinten	genau	2	2
dS:N	aS:N	Smn:O		vorne	genau	2	1
X:D	X:P	dS:N		hinten	genau	1	1
nP:N	aP:N	X:P		vorne	oder	2	1
Swna:O	Pmwsna:O	aP:F		hinten	oder	3	1
aS:N	dS:N	5:U		vorne	oder	3	1
nS:M	dS:M	13SKE:V	nS:M	hinten	genau	2	2
dP:F	aP:F	o4:A		vorne	oder	2	1
nS:F	aS:F	1SIE:V		vorne	oder	3	1
dS:N	aS:N	o8:A		vorne	oder	3	1
aP:N	dS:N	Smd:R		vorne	oder	3	1
aP:N	nP:N	Pmwsna:R	o6:A	vorne	genau	2	2
aS:N	dS:N	o5:A		vorne	oder	2	1
gS:F	dS:F	aS:F		hinten	oder	3	1
aP:N	nP:N	0:U	X:I	beides	genau	2	2
dS:N	nS:N	Ssna:R		vorne	genau	1	1
Swna:O	Pmwsna:O	X:C		hinten	oder	2	1
Sma:R	5:U	13SIA:V		hinten	genau	1	1
a:O	d:O	X:I	3SIE:V	vorne	genau	2	2
Swgd:R	Smn:O	X:D	3SIE:V	vorne	genau	2	2
Pmwsd:R	Sma:O	n:O		hinten	oder	3	1
o0:A	X:D	nS:pF		vorne	oder	2	1
Pmwsd:R	Sma:O	nS:pF		hinten	oder	3	1
a:O	d:O	X:P	13PIE:V	beides	genau	2	2
Swgd:R	Pmwsd:R	dP:M		hinten	genau	1	1
a:O	d:O	3SIE:V	X:P	vorne	genau	2	2
d:O	n:O	2PIE:V		vorne	oder	3	1
nP:M	aP:M	Smg:R		hinten	oder	3	1
Swna:R	Pmwsna:O	aP:M	X:I	vorne	genau	2	2
Smd:R	Ssd:R	X:P	dS:oN	beides	genau	2	2
n:O	a:O	n:O	3SIE:V	vorne	genau	2	2
X:C	X:D	X:I	X:I	hinten	genau	2	2
dS:F	aS:F	n:O		vorne	genau	2	1
dS:F	nS:F	4:O		vorne	oder	2	1
n:O	a:O	X:P		vorne	genau	1	1
X:D	X:P	Pmwsd:R	dP:M	hinten	genau	2	2
Smd:R	Smd:O	nS:F		hinten	oder	3	1
Swna:R	Swna:O	aS:M		hinten	genau	2	1
nS:M	dS:M	Ssd:O		vorne	oder	3	1
Ssd:O	Smd:O	3SIE:V		hinten	genau	2	1
Swgd:R	Pmwsg:R	nP:M		hinten	genau	1	1
nS:F	dS:F	6:O		vorne	oder	3	1
X:P	X:D	3SIE:V		hinten	genau	1	1
Swna:R	Pmwsna:O	Swna:R	X:I	vorne	genau	2	2
Swna:R	Pmwsna:O	Pmwsna:O		hinten	genau	2	1
dS:oN	aS:oN	Pmwsna:O		vorne	oder	3	1
dS:F	gS:F	X:I	X:PR	hinten	genau	2	2
d:O	0:O	o8:A		hinten	oder	2	1
dS:N	aS:N	Ssna:R		vorne	genau	2	1
X:D	X:P	dP:F	X:C	hinten	genau	2	2
gS:F	aS:F	X:D	Swna:R	vorne	genau	2	2

nS:F	aS:F	dS:oF	X:P	vorne	genau	2	2
nS:F	dS:F	X:P		vorne	genau	1	1
dS:oF	nS:F	3SIE:V		vorne	genau	2	1
aS:N	nS:N	8:O	o5:A	hinten	genau	2	2
Swna:R	Pmwsna:O	a:O	X:D	hinten	genau	2	2
pA:V	o0:A	Swna:O		vorne	genau	2	1
6:U	5:U	X:I		vorne	genau	2	1
nP:M	aP:M	13PIA:V		hinten	genau	2	1
nP:M	dP:M	aP:N		hinten	genau	2	1
3SIE:V	1SIE:V	n:O	5:O	beides	genau	2	2
nS:M	nP:M	dS:pF		hinten	oder	3	1
Ssna:R	Smn:R	Smn:R		vorne	oder	3	1
X:C	X:P	n:O	Smg:R	beides	genau	2	2
Swna:R	Swna:O	X:D	3SIE:V	hinten	genau	2	2
3SIE:V	1SIE:V	n:O	i:V	hinten	genau	2	2
nS:F	aS:F	X:C	X:C	hinten	genau	2	2
dS:N	nS:N	Ssna:R		vorne	genau	2	1
aP:M	nP:M	nS:F		hinten	oder	2	1
n:O	a:O	o0:A	i:V	hinten	genau	2	2
nP:M	aP:M	nS:M		vorne	genau	2	1
13SKE:V	1SIE:V	aS:N	n:O	beides	genau	2	2
Pmwsd:R	Sma:R	5:U		hinten	oder	3	1
gS:M	dS:M	X:PR		vorne	oder	2	1
X:C	X:P	X:D	dS:N	beides	genau	2	2
13SKA:V	13SIA:V	n:O	i:V	hinten	genau	2	2
aP:N	nS:N	Swgd:R	gS:F	hinten	genau	2	2
gS:F	nS:F	13SKE:V		vorne	genau	2	1
c5:A	5:O	X:C		vorne	genau	2	1
nS:N	aS:N	gS:pM		vorne	genau	2	1
gS:pM	nS:pM	aS:N		hinten	genau	2	1
nS:F	aS:F	gS:pM	13SIA:V	hinten	genau	2	2
X:D	X:C	13SKE:V	X:C	hinten	genau	2	2
X:I	3SIE:V	gS:M	3SIE:V	vorne	genau	2	2
o0:A	X:D	o0:A	X:I	hinten	genau	2	2
aS:F	aP:F	X:P	aS:F	hinten	genau	2	2
nP:F	gP:F	gS:aN		hinten	genau	2	1
dS:M	aS:M	i:V	X:D	hinten	genau	2	2
aP:N	nP:N	c4:A		vorne	genau	2	1
Ssna:R	Swna:R	nS:N	6:O	hinten	genau	2	2
dP:F	nP:F	nP:M		hinten	oder	2	1
Swna:R	Swna:O	Swna:O		hinten	oder	2	1
i:V	pA:V	13SKE:V		vorne	genau	2	1
nS:M	nS:F	dS:oN		vorne	genau	2	1
dP:F	nP:F	13PKA:V	X:I	hinten	genau	2	2
dS:M	aS:M	g:O		vorne	oder	2	1
aS:M	dS:M	n:O	X:P	vorne	genau	2	2
aP:F	dP:F	3SIE:V	Swna:R	hinten	genau	2	2
13PIE:V	13SKE:V	c4:A		hinten	oder	2	1
dS:N	aS:N	13SKE:V		hinten	oder	3	1
nS:pM	dS:pM	nS:aF		vorne	genau	2	1
13SKA:V	13SIA:V	0:U		hinten	genau	2	1
X:C	X:D	aS:M	aP:M	beides	genau	2	2
gS:F	dS:F	X:D	X:I	hinten	genau	2	2

nS:F	aS:F	gS:M		vorne	genau	2	1
gS:aF	dS:aF	Swgd:R		vorne	genau	2	1
1SIE:V	13SKE:V	d:O	0:O	hinten	genau	2	2
1SIE:V	13SKE:V	X:I	nS:pM	vorne	genau	2	2
i:V	13PIE:V	X:I	nP:M	vorne	genau	2	2
aP:F	nP:F	nP:M		hinten	genau	2	1
nS:aF	dS:aF	o4:A		vorne	oder	3	1
Pmwsd:R	Sma:R	Pmwsna:R		hinten	oder	3	1
aS:pM	nS:pM	0:O		vorne	genau	2	1
nS:pM	aS:pM	aS:pM		vorne	oder	3	1
gS:oN	dS:oN	aS:pM		hinten	oder	3	1
n:O	a:O	X:C	0:O	vorne	genau	2	2
3SIE:V	pA:V	X:I	Sma:R	hinten	genau	2	2
aS:F	dS:F	Swgd:O		vorne	oder	2	1
aP:F	dP:F	X:P	Sma:R	hinten	genau	2	2
aS:F	dS:F	dP:N		vorne	genau	2	1
13PIA:V	13PIE:V	aS:pF		vorne	genau	2	1
dS:M	aS:pF	aS:pM		vorne	oder	3	1
nS:pM	aS:pM	aS:pM		hinten	oder	3	1
nS:aF	aS:aF	Pmwsna:R		hinten	oder	3	1
1SIE:V	13SKE:V	nS:N		vorne	oder	3	1
nS:N	dS:N	Smd:R	13SIA:V	beides	genau	2	2
Smd:R	Ssd:R	dS:N	13SIA:V	hinten	genau	2	2
aP:N	nP:N	dS:N		vorne	genau	2	1
Ssna:R	Smn:R	nS:pM		hinten	oder	2	1
X:D	X:P	aP:MF		vorne	genau	2	1
aP:MF	nP:MF	X:P		hinten	genau	2	1
Ssna:R	Ssna:O	7:O		hinten	genau	2	1
nS:M	gS:aM	Ssg:R		vorne	genau	2	1
aP:M	gP:M	nP:F		vorne	oder	3	1
5:U	o5:A	nP:F		vorne	genau	2	1
nS:F	aS:F	X:D	i:V	hinten	genau	2	2
nS:F	aS:F	X:I	o4:A	hinten	genau	2	2
i:V	13PIE:V	n:O	a:O	hinten	genau	2	2
i:V	13PIE:V	13PIE:V		hinten	genau	2	1
nP:M	aP:M	Pmwsd:R		vorne	oder	3	1
dS:oN	aS:oN	n:O	3SIE:V	vorne	genau	2	2
Swna:R	Pmwsna:O	a:O	X:P	hinten	genau	2	2
Swna:R	Swna:O	a:O		hinten	oder	2	1
aP:M	nP:M	i:V		vorne	genau	2	1
i:V	13PIE:V	13PIA:V		vorne	genau	2	1
X:D	o4:A	Swna:R	aS:F	beides	genau	2	2
Swna:R	Pmwsna:O	nS:N	X:I	vorne	genau	2	2
nS:F	aS:F	Smg:R	gS:N	hinten	genau	2	2
nP:F	aP:F	c4:A	X:I	beides	genau	2	2
13SIA:V	o4:A	nS:M	Swgd:R	hinten	genau	2	2
X:C	X:P	Smn:O	Smg:R	beides	genau	2	2
Ssna:R	Ssna:O	o4:A		hinten	genau	2	1
X:D	c0:A	13SKA:V	pA:V	beides	genau	2	2
aS:pM	nS:pM	Swgd:R		hinten	genau	2	1
X:P	X:D	aS:pM	X:P	hinten	genau	2	2
dS:oN	aS:oN	dS:N		hinten	genau	2	1
gP:oT	nP:oT	X:P		hinten	genau	2	1

X:D	o0:A	aS:N	pA:V	beides	genau	2	2
0:O	X:D	Swna:R		vorne	genau	2	1
dP:F	nP:F	Pmwsna:O		vorne	oder	2	1
aS:pM	dS:pM	Swna:O		hinten	genau	2	1
dS:oN	nS:oN	13SIA:V	X:D	hinten	genau	2	2
dS:oN	aS:oN	Sma:R		vorne	genau	2	1
dS:F	nS:F	u4:A		vorne	oder	3	1
i:V	13PIE:V	a:O	X:D	hinten	genau	2	2
aS:M	gS:M	gS:pM		hinten	oder	3	1
dS:F	gS:F	gS:N	Swgd:R	vorne	genau	2	2
aP:F	nP:F	Sma:R		hinten	genau	2	1
nS:M	aP:M	o5:A	gS:F	hinten	genau	2	2
Pmwsd:R	Smn:R	X:P	Swna:R	hinten	genau	2	2
nS:M	dS:M	Swna:O		vorne	genau	2	1
o0:A	pA:V	nS:M	3SIE:V	beides	genau	2	2
dS:oN	nS:oN	Pmwsna:R		hinten	genau	2	1
Swna:R	Pmwsna:O	d:O		hinten	genau	2	1
aS:pM	dS:pM	Smn:O		hinten	genau	2	1
aS:F	aS:N	Ssna:R	13PIA:V	beides	genau	2	2
Swgd:R	Swgd:O	o4:A	nP:M	hinten	genau	2	2
gS:F	dS:F	13SIA:V		hinten	oder	2	1
X:P	X:D	dP:M	X:P	beides	genau	2	2
aS:F	gS:F	X:P	g:O	vorne	genau	2	2
X:C	X:P	dS:oN	13PIE:V	vorne	genau	2	2
dS:F	aS:F	X:P	o4:A	vorne	genau	2	2
Smd:R	Ssd:O	6:O		hinten	oder	2	1
gS:F	aS:F	o8:A		hinten	oder	2	1
dP:M	gP:M	z:V		hinten	oder	3	1
Swgd:R	Pmwsg:R	Pmwsna:R	aP:N	vorne	genau	2	2
aS:pM	nS:pM	d:O		hinten	genau	2	1
dS:oN	aS:oN	o4:A		vorne	genau	2	1
nS:N	aS:N	aS:oN		hinten	oder	3	1
dS:N	aS:N	X:C	Ssna:R	vorne	genau	2	2
gP:F	aP:F	gS:N		hinten	genau	2	1
Swna:R	Pmwsna:R	8:O		hinten	genau	2	1
aS:F	nS:F	Smg:R	o5:A	hinten	genau	2	2
dP:F	gP:F	X:PR		vorne	oder	3	1
Swgd:R	Pmwsg:R	gP:F		hinten	oder	2	1
gS:F	gP:F	13PIA:V		hinten	genau	2	1
Swgd:R	Pmwsg:R	gP:F		hinten	oder	2	1
gS:F	aS:F	13SIA:V		vorne	genau	2	1
gS:F	nS:F	Swna:R		vorne	oder	2	1
Swgd:R	Smn:R	X:PR	dS:M	hinten	genau	2	2
Smd:R	Smd:O	n:O		hinten	oder	2	1
dS:F	gS:F	Smd:O		hinten	oder	3	1
o4:A	13SKE:V	aP:N		vorne	oder	2	1
Smd:R	Smd:O	nS:M		hinten	genau	2	1
aP:N	dP:N	X:O		vorne	oder	3	1
X:O	X:D	Pmwsd:R		hinten	genau	2	1
aS:F	dP:F	aS:M	o5:A	vorne	genau	2	2
d:O	0:O	o5:A		vorne	genau	2	1
Swna:R	Pmwsna:O	0:U	aP:N	hinten	genau	2	2
aS:F	gS:F	o5:A	pA:V	beides	genau	2	2

X:D	X:P	o7:A		hinten	oder	2	1
0:O	X:D	nP:M		hinten	genau	2	1
aS:F	nS:F	aS:pF		vorne	oder	3	1
i:V	13PIE:V	Pmwsg:R		vorne	genau	2	1
Sma:O	Pmwsd:O	Swna:R		hinten	genau	2	1
X:PR	X:P	u5:A	X:D	hinten	genau	2	2
nS:F	aS:F	X:C	aS:F	hinten	genau	2	2
X:P	X:D	dS:N	13PIE:V	vorne	genau	2	2
nS:N	dS:N	o7:A		vorne	oder	3	1
dS:F	aS:F	pA:V	X:I	vorne	genau	2	2
dP:F	gP:F	o0:A		hinten	genau	2	1
nS:N	dS:N	4:O	nP:F	hinten	genau	2	2
Smd:R	Ssd:R	4:O		hinten	genau	2	1
Swna:R	Swna:O	X:D	Swna:R	hinten	genau	2	2
Swgd:R	Smd:R	dP:oM		vorne	oder	3	1
dP:oM	dP:M	Pmwsd:R		vorne	genau	2	1
Ssna:R	Ssna:O	nP:M		hinten	genau	2	1
nS:aF	gS:aF	X:D		hinten	genau	2	1
aP:N	dS:N	gS:oN		vorne	genau	2	1
13PIA:V	pA:V	3SIE:V	X:I	hinten	genau	2	2
nS:F	gS:F	X:I	o5:A	vorne	genau	2	2
dS:M	aS:M	13PIA:V	aS:M	vorne	genau	2	2
Smd:R	Smn:R	13PKE:V		vorne	genau	2	1
gP:M	nP:M	X:D	X:I	hinten	genau	2	2
nS:aN	dS:aN	dS:F		vorne	genau	2	1
Swna:R	Pmwsna:O	Swna:R	nS:F	hinten	genau	2	2
Swgd:R	6:U	Pmwsg:R	o5:A	hinten	genau	2	2
dS:oN	gS:oN	gS:oN		vorne	genau	2	1
i:V	13PIE:V	aP:F		hinten	oder	3	1
13PIE:V	13SKE:V	Pmwsna:R	o5:A	vorne	genau	2	2
X:P	X:D	0:U	aP:T	hinten	genau	2	2
i:V	13PIE:V	o6:A		hinten	genau	2	1
aP:F	dP:F	X:I	pA:V	hinten	genau	2	2
nP:N	aP:N	13SIA:V		vorne	genau	2	1
dS:M	aS:M	Smn:O	aS:M	vorne	genau	2	2
Pmwsd:R	Pmwsd:O	Smg:R		vorne	oder	3	1
Smg:R	Ssg:R	nS:pM		hinten	oder	3	1
aS:M	dS:M	dS:M	i:V	hinten	genau	2	2
Ssna:R	Smn:R	X:P	Swgd:R	hinten	genau	2	2
gP:F	dP:F	0:U		hinten	genau	2	1
gS:F	dS:F	dS:M	6:O	vorne	genau	2	2
aS:F	dS:F	X:PR		vorne	oder	2	1
dP:M	nP:M	4:O		vorne	genau	2	1
aP:F	nP:F	13PKE:V		hinten	oder	3	1
13PIE:V	i:V	nS:M		vorne	genau	2	1
nS:M	dP:M	X:P	0:U	vorne	genau	2	2
X:D	X:P	gS:F	dP:M	beides	genau	2	2
3SIE:V	pA:V	13SKE:V	X:P	hinten	genau	2	2
dS:oF	gS:oF	aS:M		vorne	genau	2	1
dS:M	aS:M	X:C	aS:F	hinten	genau	2	2
aS:F	dS:F	o5:A	Ssg:R	beides	genau	2	2
nS:F	gS:F	13SIA:V	X:PR	hinten	genau	2	2
13SKA:V	13PKA:V	aP:M	i:V	vorne	genau	2	2

aS:F	gS:F	gP:F		vorne	oder	3	1
dS:N	nS:N	dP:M	o4:A	vorne	genau	2	2
dS:aN	gS:aN	Ssna:O		hinten	genau	2	1
Smg:R	Ssg:R	Ssna:O		hinten	oder	3	1
X:D	X:P	dS:oN	pA:V	hinten	genau	2	2
o4:A	X:P	dS:F		hinten	genau	2	1
X:D	pA:V	o8:A		vorne	genau	2	1
dP:N	nS:N	dP:M		vorne	genau	2	1
dP:N	aS:N	dS:oN		hinten	genau	2	1
dS:oF	gS:F	aS:F		hinten	genau	2	1
dS:N	aS:N	aP:T		hinten	genau	2	1
X:D	0:O	X:P	aS:N	hinten	genau	2	2
nS:pM	aS:pM	X:I	X:P	beides	genau	2	2
13PIE:V	i:V	aS:pM		hinten	genau	2	1
4:U	o4:A	X:I		hinten	genau	2	1
nP:F	dP:F	dP:F		hinten	oder	3	1
nP:F	aP:F	aP:F		hinten	oder	3	1
Swna:R	Swna:O	dS:N		hinten	genau	2	1
X:C	X:P	aS:F	0:U	beides	genau	2	2
Ssna:R	Smn:R	4:O		hinten	genau	2	1
n:O	a:O	0:O	13SKE:V	vorne	genau	2	2
a:O	d:O	13SKE:V	a:O	vorne	genau	2	2
dS:F	gS:F	dS:oF		hinten	genau	2	1
nS:N	aS:N	nS:pM		vorne	genau	2	1
X:D	13PIA:V	X:D	nS:pM	beides	genau	2	2
aS:oF	dS:F	X:P		hinten	genau	2	1
Swna:R	Swna:O	o5:A		hinten	genau	2	1
nS:M	dS:M	13PIE:V	n:O	hinten	genau	2	2
Ssd:R	Smd:R	X:C		hinten	genau	2	1
o0:A	X:D	g:O		hinten	oder	2	1
dP:M	nP:M	Sma:R		hinten	oder	2	1
Swgd:R	6:U	gS:oN		vorne	oder	3	1
dP:F	nP:F	gS:oN		vorne	genau	2	1
gS:oN	gS:N	nP:F		hinten	genau	2	1
6:O	X:D	o5:A		hinten	oder	2	1
Swna:R	Pmwsna:R	aP:oN		hinten	genau	2	1
dS:N	dP:N	X:P	dP:M	hinten	genau	2	2
nS:M	aS:M	aS:pM	X:P	hinten	genau	2	2
nS:M	aS:pM	aS:pM		vorne	oder	3	1
Swgd:R	Ssna:R	X:D	13SIA:V	vorne	genau	2	2
Pmwsd:R	Sma:R	X:D	X:D	hinten	genau	2	2
dS:F	gS:F	7:O		vorne	oder	3	1
gS:pM	dS:pM	nS:M		vorne	oder	3	1
nS:M	dS:M	dS:pM		hinten	oder	3	1
nS:pM	dS:pM	dS:pM		hinten	oder	3	1
nS:N	nS:oN	X:I	X:PR	hinten	genau	2	2
Swgd:R	Smn:R	X:I	13SIA:V	vorne	genau	2	2
X:D	o0:A	13SKE:V	n:O	hinten	genau	2	2
6:O	X:D	aS:N		hinten	genau	2	1
X:D	X:P	dS:N	dP:M	beides	genau	2	2
aS:F	nS:F	X:D	c4:A	vorne	genau	2	2
nS:M	gS:N	o5:A		vorne	genau	2	1
aP:N	aS:N	gS:F		hinten	genau	2	1

gS:F	dS:F	aP:N		vorne	oder	3	1
aP:F	nP:F	13PKA:V	Pmwsna:R	vorne	genau	2	2
X:D	X:P	aS:M	Smd:R	beides	genau	2	2
1SIE:V	13SKE:V	nS:pM		vorne	genau	2	1
dS:F	aS:F	13SKE:V	X:C	hinten	genau	2	2
aP:F	gP:F	Swna:R		vorne	genau	2	1
Swgd:R	Pmwsg:R	gP:F		hinten	oder	2	1
i:V	13PIE:V	aS:F	13PIE:V	vorne	genau	2	2
13PIE:V	i:V	13PIE:V	X:I	hinten	genau	2	2
Swna:R	Swna:O	X:D	pA:V	hinten	genau	2	2
nS:M	dS:M	gS:pM		vorne	oder	3	1
nS:M	nS:N	g:O		vorne	oder	2	1
Smd:R	Ssd:O	nP:F		hinten	oder	3	1
gP:F	dP:F	o6:A		hinten	genau	2	1
dS:F	gS:F	nS:M	Swgd:R	vorne	genau	2	2
dS:N	aS:N	Swna:R	o4:A	hinten	genau	2	2
gP:oT	gP:T	gS:oN		hinten	genau	2	1
Swgd:R	Smn:R	X:P	Swgd:R	hinten	genau	2	2
Swna:R	Pmwsna:O	Swna:R	o4:A	hinten	genau	2	2
Swna:R	Swna:O	X:I	Swna:R	beides	genau	2	2
nS:M	dS:N	3SIE:V	X:P	vorne	genau	2	2
X:C	X:P	nS:M	Swgd:R	beides	genau	2	2
aP:F	nP:F	pA:V		hinten	genau	2	1
nS:N	aS:N	aP:F		vorne	oder	3	1
nS:M	aS:M	aS:N	X:P	vorne	genau	2	2
dS:oN	nS:oN	13PIA:V	X:D	hinten	genau	2	2
dS:oN	nS:oN	nS:oN		hinten	genau	2	1
dS:aM	nS:aM	nS:oN		hinten	oder	3	1
dS:oN	dS:aN	X:P	dS:aM	vorne	genau	2	2
dS:F	aS:F	dS:oN		vorne	genau	2	1
X:D	o4:A	aS:F	Pmwsd:R	hinten	genau	2	2
dS:oN	nS:oN	X:PR	0:U	hinten	genau	2	2
aS:aM	nS:aM	X:PR		hinten	genau	2	1
aS:aM	dS:aM	X:C		hinten	genau	2	1
aS:M	dS:M	X:P	Swgd:R	beides	genau	2	2
aS:F	dS:F	aS:oN		hinten	genau	2	1
aP:F	dP:F	Sma:O		hinten	genau	2	1
5:U	4:U	Ssg:R		hinten	genau	2	1
dS:oN	aS:oN	aS:aM		vorne	oder	3	1
gS:pM	aS:pM	aS:F		vorne	genau	2	1
gP:M	gS:M	aS:F		hinten	genau	2	1
X:P	X:D	Ssd:R	5:U	hinten	genau	2	2
Swgd:R	Pmwsg:R	X:P	X:D	hinten	genau	2	2
X:P	X:D	8:U		hinten	genau	2	1
Swna:R	Swna:O	gS:pM		vorne	genau	2	1
nS:pM	gS:pM	gS:pM		hinten	oder	2	1
gP:M	nP:M	Pmwsna:R		vorne	genau	2	1
dS:N	aS:N	X:PR	o4:A	vorne	genau	2	2
dP:F	aP:F	o4:A		hinten	genau	2	1
nS:N	aS:N	nP:M		hinten	genau	2	1
Swna:R	Pmwsna:R	X:P	dS:M	hinten	genau	2	2
nS:N	dS:oN	nP:M		hinten	oder	2	1
nS:M	dS:M	13PIA:V		hinten	genau	2	1

5:U	o5:A	nP:N		hinten	oder	3	1
o0:A	X:D	dP:F		hinten	genau	2	1
dP:F	nP:F	Swna:R		vorne	oder	2	1
Swna:R	Pmwsna:R	nP:F		hinten	oder	3	1
aP:F	dP:F	Pmwsd:R	o5:A	vorne	genau	2	2
c4:A	o4:A	X:C		hinten	oder	3	1
Smd:R	Swgd:R	13SIA:V		hinten	genau	2	1
X:P	X:C	c4:A		hinten	genau	2	1
g:O	Ssg:O	X:P		hinten	genau	2	1
Swgd:R	Pmwsg:R	X:I	0:U	vorne	genau	2	2
aS:F	dP:F	13SIA:V	Swna:R	hinten	genau	2	2
nS:N	dS:N	dS:N		vorne	genau	2	1
8:O	Ssna:O	Smd:R		hinten	genau	2	1
13SIA:V	3SIE:V	Ssg:O		vorne	genau	2	1
X:P	X:D	nP:N		hinten	genau	2	1
Smd:R	Smn:O	nP:N		hinten	genau	2	1
a:O	d:O	X:I	nP:F	hinten	genau	2	2
aS:M	dS:M	o7:A		vorne	oder	3	1
X:D	X:P	nS:pM	X:D	hinten	genau	2	2
aS:F	nS:F	u4:A		hinten	oder	3	1
3SIE:V	pA:V	X:I	o0:A	hinten	genau	2	2
3SIE:V	1SIE:V	X:D	d:O	hinten	genau	2	2
Swna:R	Pmwsna:O	dP:F	X:I	vorne	genau	2	2
5:U	o5:A	Smd:R		vorne	oder	2	1
gS:F	aS:F	X:D	i:V	hinten	genau	2	2
X:C	X:P	nS:N	13SIA:V	vorne	genau	2	2
Swna:R	Pmwsna:R	nS:aF		hinten	genau	2	1
X:I	X:D	0:O	Swna:R	beides	genau	2	2
0:O	o0:A	Swna:R		hinten	genau	2	1
d:O	n:O	2PIE:V		hinten	oder	3	1
dS:F	nS:F	X:C	nS:M	vorne	genau	2	2
a:O	d:O	3SIE:V	X:I	hinten	genau	2	2
13PIA:V	13SIA:V	d:O		hinten	genau	2	1
Swna:O	Pmwsna:O	n:O		hinten	genau	2	1
a:O	d:O	Swna:O	n:O	vorne	genau	2	2
nP:M	aS:M	X:C	o5:A	vorne	genau	2	2
X:C	X:D	Pmwsd:R	o5:A	vorne	genau	2	2
Pmwsd:R	Sma:R	o5:A	X:D	hinten	genau	2	2
Swgd:R	Smn:R	nS:M	X:I	vorne	genau	2	2
nS:M	dS:M	X:I	o5:A	vorne	genau	2	2
o0:A	X:D	i:V	d:O	vorne	genau	2	2
i:V	13PIE:V	d:O	X:D	hinten	genau	2	2
X:D	X:P	dS:M	0:U	hinten	genau	2	2
Swna:R	Pmwsna:R	X:P	dS:F	hinten	genau	2	2
X:D	0:O	aS:M	X:P	hinten	genau	2	2
dS:oN	nS:oN	aS:M		hinten	genau	2	1
dS:M	gS:M	Ssna:O		vorne	oder	2	1
Ssna:O	Smg:O	gS:M		hinten	oder	3	1
Smd:R	Ssd:R	dP:N		hinten	genau	2	1
Smd:R	Smd:O	X:P	X:P	beides	genau	2	2
i:V	13PIE:V	pA:V	X:I	hinten	genau	2	2
aP:F	nP:F	Smn:R		hinten	genau	2	1
gP:M	aP:M	aP:M		hinten	genau	2	1

nS:F	dS:F	X:PR		vorne	genau	2	1
dS:oN	aS:oN	i:V	X:C	hinten	genau	2	2
dS:oN	dS:N	7:O		vorne	genau	2	1
n:O	a:O	Pmwsna:R	nP:N	hinten	genau	2	2
aS:M	dS:F	13PIA:V		hinten	genau	2	1
nP:F	gP:F	Swgd:R		vorne	genau	2	1
Swgd:R	Pmwsg:R	gP:F		hinten	genau	2	1
Pmwsd:R	Sma:R	pE:V		vorne	genau	2	1
Swgd:R	Swgd:O	gS:pM		vorne	oder	3	1
dS:F	gS:F	c5:A	3SIE:V	beides	genau	2	2
Ssna:R	SSna:O	Ssd:R		hinten	oder	2	1
aS:N	dS:N	Ssd:R		hinten	oder	3	1
Swgd:R	Pmwsg:R	n:O	Swgd:R	vorne	genau	2	2
Swgd:R	Smn:O	Pmwsg:R	0:U	hinten	genau	2	2
Ssna:R	Ssna:O	o0:A	X:D	hinten	genau	2	2
Swna:R	Swna:O	Pmwsna:R		hinten	genau	2	1
Swgd:R	Pmwsg:R	gP:pM		hinten	oder	3	1
Smd:R	Ssd:O	c5:A		hinten	genau	2	1
Ssna:R	0:U	0:U		vorne	genau	2	1
nS:pF	nS:pM	5:O		hinten	oder	3	1
dS:N	nS:N	o8:A		vorne	oder	3	1
nP:F	dP:F	X:I	n:O	hinten	genau	2	2
aS:M	13SIA:V	8:O		vorne	oder	3	1
Ssna:R	Smn:R	Ssna:R		vorne	genau	2	1
Ssna:R	Ssna:O	Smn:R		hinten	genau	2	1
dS:F	nS:F	Ssna:O		vorne	oder	3	1
X:D	X:P	5:O	dP:F	hinten	genau	2	2
X:C	X:D	gS:pF		hinten	genau	2	1
X:D	X:P	o5:A	dP:N	hinten	genau	2	2
dP:M	:gP:M	Pmwsg:R		vorne	oder	3	1
Swgd:R	Pmwsd:R	dP:F		hinten	genau	2	1
dS:aN	dS:aM	Smn:O		hinten	genau	2	1
pA:V	3SIE:V	a:O	Ssna:O	vorne	genau	2	2
nS:M	as:M	Smg:O		hinten	oder	3	1
Pmwsd:R	Pmwsna:R	X:C	3SIE:V	vorne	genau	2	2
aP:N	gP:N	Smg:R		vorne	genau	2	1
Swgd:R	Pmwsg:R	gP:N		hinten	oder	2	1
Smg:R	SSna:R	gP:N		hinten	genau	2	1
X:D	X:P	dS:F	X:C	hinten	genau	2	2
nS:M	gS:M	nS:F		vorne	genau	2	1
1SIE:V	13SKE:V	X:D	i:V	hinten	genau	2	2
13PIE:V	i:V	13SKE:V		vorne	oder	3	1
1SIE:V	13SKE:V	dS:F		hinten	genau	2	1
1SIE:V	13SKE:V	Pmwsna:R		vorne	genau	2	1
dS:oN	gS:oN	3SIE:V		hinten	genau	2	1
Swna:R	Smn:O	X:I	Smd:R	hinten	genau	2	2
Swgd:R	Smn:R	Smn:O		hinten	oder	3	1
aP:M	nP:M	o6:A		hinten	oder	3	1
Ssd:O	Smd:O	X:D		hinten	oder	3	1
nP:N	aP:N	X:P	X:D	hinten	genau	2	2
nS:pF	aS:pF	4:O		hinten	oder	3	1
nS:F	aS:F	Smn:O		vorne	genau	2	1
X:P	X:D	Swna:R	4:U	hinten	genau	2	2

1SIE:V	13SKE:V	o4:A		hinten	oder	3	1
nS:M	dS:M	X:P	Ssna:R	hinten	genau	2	2
nS:F	aS:F	dP:F		vorne	genau	2	1
nS:M	gS:M	X:P	a:O	hinten	genau	2	2
nS:M	gS:aM	Smg:R		vorne	oder	3	1
nP:F	dP:F	X:P	0:U	vorne	genau	2	2
nS:M	dS:M	X:D	i:V	hinten	genau	2	2
X:D	X:P	aP:M	dS:M	beides	genau	2	2
aP:F	dP:F	aS:M	X:D	hinten	genau	2	2
X:P	X:D	aP:N		vorne	genau	2	1
o0:A	X:D	Pmwsna:R	aP:N	hinten	genau	2	2
aP:M	dS:M	Smd:R		hinten	genau	2	1
nS:M	o6:A	nS:M	3SIE:V	hinten	genau	2	2
dS:oN	aS:oN	aS:oF		vorne	genau	2	1
nS:M	gP:M	Pmwsna:R		vorne	oder	3	1
nS:M	nP:M	13PIE:V		vorne	genau	2	1
Swna:R	Pmwsna:R	nP:M		hinten	oder	3	1
X:D	X:C	n:O	o4:A	hinten	genau	2	2
nP:M	dP:M	dS:oN		vorne	genau	2	1
dS:oN	nS:oN	Pmwsna:R		vorne	oder	3	1
3SIE:V	1SIE:V	o4:A	aS:F	vorne	genau	2	2
nS:pF	dS:pF	o4:A	aS:F	hinten	genau	2	2
nP:F	aP:F	13PIE:V		hinten	genau	2	1
Swna:R	Pmwsd:R	dP:N		hinten	oder	3	1
dS:F	nS:F	X:P	Pmwsd:R	hinten	genau	2	2
aS:N	nS:N	pA:V	3SIE:V	hinten	genau	2	2
Smd:R	Smd:O	0:O		hinten	oder	3	1
Swna:R	Pmwsna:O	13PIA:V	X:D	hinten	genau	2	2
X:D	X:C	gS:M		hinten	genau	2	1
X:D	o0:A	X:C	Ssna:R	hinten	genau	2	2
X:D	X:P	Smn:O	i:V	beides	genau	2	2
aP:N	aS:N	13SIA:V	n:O	hinten	genau	2	2
i:V	13PIE:V	dS:oM		vorne	oder	3	1
dS:M	dP:M	Pmwsd:R		vorne	oder	2	1
o0:A	pA:V	X:D	nS:pM	hinten	genau	2	2
X:C	X:D	pA:V	X:D	hinten	genau	2	2
13PIE:V	i:V	13PKA:V		vorne	genau	2	1
Swna:O	Swgd:O	nP:N		hinten	oder	3	1
o0:A	X:D	a:O		hinten	genau	2	1
dS:F	gS:F	X:D	o0:A	hinten	genau	2	2
X:P	X:D	X:D	Smn:R	beides	genau	2	2
13PIE:V	i:V	d:O	X:I	beides	genau	2	2
o0:A	X:D	1SIE:V		hinten	genau	2	1
3SIE:V	1SIE:V	X:I	aS:N	vorne	genau	2	2
nS:M	aS:M	i:V		vorne	genau	2	1
X:D	X:P	7:O	dS:M	hinten	genau	2	2
nS:pM	dS:pM	1SIE:V		hinten	oder	3	1
X:D	X:P	dS:pM	1SIE:V	hinten	genau	2	2
5:U	o5:A	dS:N	13PIE:V	hinten	genau	2	2
13PIE:V	i:V	Ssna:R	aS:N	vorne	genau	2	2
Ssna:R	Smn:R	o6:A	X:I	hinten	genau	2	2
c0:A	o6:A	X:D		vorne	genau	2	1
13SKE:V	1SIE:V	3SIE:V		hinten	genau	2	1

13PIE:V	13PIA:V	a:O	i:V	hinten	genau	2	2
dS:N	nS:N	u8:A		vorne	oder	3	1
3SIE:V	1SIE:V	8:O	X:D	hinten	genau	2	2
13SKE:V	1SIE:V	o4:A	aS:F	vorne	genau	2	2
0:O	X:D	0:O		vorne	genau	2	1
0:O	X:D	X:C	3SIE:V	vorne	genau	2	2
X:C	X:D	1SIE:V	n:O	hinten	genau	2	2
dS:N	nP:N	Ssna:R		vorne	oder	2	1
1SIE:V	3SIE:V	aS:F		vorne	genau	2	1
X:D	0:O	aS:M	X:C	hinten	genau	2	2
0:O	X:D	n:O	1SIE:V	vorne	genau	2	2
dP:F	nP:F	nS:M		vorne	genau	2	1
X:P	X:D	aS:F	X:D	beides	genau	2	2
dP:N	nP:N	nS:pM		hinten	genau	2	1
nS:N	dS:N	nP:N		hinten	genau	2	1
Swgd:R	Pmwsg:R	dP:F	X:I	hinten	genau	2	2

Anhang D - Die aus 2.000 Sätzen gelernten kontext-sensitiven Regeln

Ausgangs-tag	Zieltag	Nachbar 1	Nachbar 2	Wo?	Wie?	Anzahl 1	Anzahl 2
Swgd:R	Smn:R	nS:M		hinten	oder	2	1
Pmwsd:R	Sma:R	aS:M		hinten	oder	3	1
nS:M	dS:M	X:PR		vorne	oder	2	1
X:P	X:D	X:I		hinten	genau	1	1
Ssna:R	Smn:R	nS:M		hinten	oder	2	1
Swna:R	Pmwsna:R	nP:M		hinten	oder	3	1
Swna:R	Pmwsna:R	nP:F		hinten	oder	3	1
Pmwsd:R	Sma:R	nS:M		hinten	oder	3	1
nS:M	aS:M	Sma:R		vorne	oder	3	1
Ssna:R	Ssna:O	3SIE:V		hinten	genau	1	1
Smd:R	Ssd:R	dS:N		hinten	oder	2	1
Smg:R	Ssg:R	gS:N		hinten	oder	3	1
Swgd:R	Pmwsg:R	gP:M		hinten	oder	3	1
Swna:R	Pmwsna:R	o5:A		hinten	oder	2	1
Swgd:R	Smn:O	X:I		vorne	genau	1	1
nS:N	dS:N	X:PR		vorne	oder	3	1
nS:M	dS:M	Smd:R		vorne	oder	2	1
Swna:R	Pmwsna:R	nP:N		hinten	oder	2	1
Swna:R	Pmwsna:R	aP:N		hinten	oder	2	1
nS:F	dS:F	X:P	Swgd:R	vorne	genau	2	2
Swgd:R	Pmwsg:R	gP:F		hinten	oder	2	1
Swna:R	Pmwsna:R	aP:M		hinten	oder	3	1
13SKE:V	1SIE:V	n:O		hinten	oder	3	1
aS:F	dS:F	X:P	Swgd:R	vorne	genau	2	2
i:V	13PIE:V	X:D		hinten	genau	1	1
nS:F	gS:F	Swgd:R		vorne	genau	1	1
Swna:R	Pmwsna:R	aP:F		hinten	genau	1	1
Swna:R	Pmwsna:R	nP:T		hinten	oder	3	1
13PIE:V	i:V	X:D	X:I	beides	genau	2	2
dS:N	aS:N	Ssna:R		vorne	genau	2	1
Ssg:R	Smg:R	gS:M		hinten	oder	3	1
d:O	0:O	nS:M		hinten	genau	1	1
Swgd:O	Smn:O	nS:M		hinten	oder	3	1
Swgd:R	Pmwsg:R	gP:N		hinten	oder	2	1
nS:M	dS:M	7:O		vorne	oder	3	1
i:V	13PIE:V	X:P		hinten	genau	1	1
nP:M	dP:M	Pmwsd:R		vorne	oder	2	1
Swna:R	Swna:O	aS:F	X:I	vorne	genau	2	2
dS:N	nS:N	Ssna:R		vorne	oder	2	1
dS:F	aS:F	X:P	Swna:R	vorne	genau	2	2
nS:pM	dS:pM	X:P		vorne	oder	3	1
aS:M	dS:M	X:PR		vorne	oder	2	1
Ssna:R	Ssna:O	n:O		hinten	oder	2	1
Ssna:R	Ssna:O	X:C		hinten	genau	1	1
Ssna:R	Ssna:O	X:D		hinten	genau	1	1
i:V	13PIE:V	Pmwsna:R		hinten	genau	1	1
i:V	13PIE:V	i:V	X:I	beides	genau	2	2

nP:F	dP:F	X:P		vorne	oder	2	1
Pmwsd:R	Sma:R	dS:M		hinten	oder	2	1
dS:M	aS:M	Sma:R		vorne	oder	2	1
Smd:O	Ssd:O	dS:N		hinten	oder	3	1
Swna:R	Swna:O	nS:F	X:I	vorne	genau	2	2
Swna:R	Pmwsna:R	gP:oT		hinten	oder	2	1
Swgd:R	Pmwsg:R	gP:oT		hinten	oder	3	1
dS:F	aS:F	4:O		vorne	genau	1	1
Swna:R	Pmwsna:R	dP:F		hinten	oder	2	1
dS:N	aS:N	Ssna:R		vorne	oder	2	1
nS:M	aS:M	5:O		vorne	oder	3	1
Swgd:R	Smn:R	o4:A		hinten	oder	2	1
X:P	X:C	X:I	n:O	beides	genau	2	2
nS:F	dS:F	Swgd:R		vorne	oder	2	1
i:V	0:O	aS:N		hinten	genau	1	1
Swgd:R	Smn:O	X:I	X:C	vorne	genau	2	2
Swna:R	Pmwsna:R	dP:N		hinten	oder	3	1
nS:N	dS:N	Smd:R		vorne	oder	2	1
aS:M	dS:M	Smd:R		vorne	oder	2	1
nS:F	aS:F	X:P	Swna:R	vorne	genau	2	2
gP:oT	dP:oT	Pmwsd:R		vorne	oder	3	1
nP:T	aP:T	X:P		vorne	oder	3	1
dS:oN	nS:oN	nS:aM		vorne	genau	1	1
gP:M	nP:M	Swna:R		vorne	oder	2	1
aP:F	dP:F	Pmwsd:R		vorne	oder	3	1
aS:F	gS:F	Swgd:R		vorne	genau	1	1
Swna:R	Pmwsna:O	dP:F	X:I	vorne	genau	2	2
nP:N	gP:N	Swgd:R		vorne	oder	2	1
Swgd:R	Pmwsg:R	gP:N		hinten	oder	2	1
nS:M	dS:M	dS:pM		vorne	oder	3	1
nS:M	dS:M	dS:pM		hinten	oder	3	1
X:P	X:D	X:C		hinten	genau	1	1
aP:M	dS:M	Smd:R		vorne	oder	2	1
nP:M	aP:M	o8:A		hinten	oder	3	1
Swna:O	Pmwsna:O	13PIE:V		hinten	genau	2	1
X:P	X:C	X:PR		hinten	genau	1	1
13SKE:V	1SIE:V	3SIE:V		hinten	oder	3	1
dP:F	nP:F	Pmwsna:R		vorne	genau	1	1
n:O	a:O	n:O	X:D	beides	genau	2	2
13PIE:V	i:V	3SIE:V	X:I	hinten	genau	2	2
Ssna:R	Ssna:O	13SKA:V		hinten	genau	1	1
Ssna:R	Ssna:O	13SIA:V		hinten	genau	1	1
aS:F	dS:F	X:PR		vorne	oder	3	1
13PIE:V	i:V	X:I	Swna:R	hinten	genau	2	2
Swna:R	Pmwsna:O	gP:M	X:I	vorne	genau	2	2
X:O	X:D	X:I		vorne	genau	1	1
Ssna:R	Ssna:O	Ssna:O		hinten	oder	3	1
Ssna:R	X:D	X:I		hinten	genau	1	1
Ssna:R	Ssna:O	gS:N	X:I	vorne	genau	2	2
Ssna:R	Ssna:O	nS:F		hinten	oder	3	1
Smd:R	Ssd:R	o5:A	dS:N	hinten	genau	2	2
i:V	0:O	nS:M		hinten	oder	2	1
13PIE:V	i:V	aS:F	X:I	beides	genau	2	2

Swna:R	Pmwsna:O	13PIE:V		hinten	oder	2	1
Swna:R	Pmwsna:R	13PIE:V		hinten	genau	2	1
gS:F	nS:F	Swna:R		vorne	genau	1	1
gS:F	aS:F	Swna:R		vorne	oder	2	1
Swgd:R	Pmwsg:R	nP:M		hinten	genau	1	1
dS:oF	nS:oF	X:I		vorne	oder	3	1
aS:N	dS:N	Smd:R		vorne	oder	3	1
Smd:R	Ssd:R	dS:N	X:P	hinten	genau	2	2
dS:F	nS:F	3SIE:V		vorne	genau	2	1
dS:F	aS:F	o4:A		vorne	oder	2	1
dS:F	nS:F	o4:A		vorne	oder	2	1
nS:F	dS:F	6:O		vorne	oder	3	1
X:D	X:P	0:U	dS:F	hinten	genau	2	2
nS:pF	dS:pF	gS:M		hinten	oder	3	1
nS:pF	aS:pF	X:P		vorne	oder	2	1
a:O	d:O	nS:pF		hinten	oder	3	1
Swna:R	Pmwsna:O	X:I	Ssna:R	beides	genau	2	2
dS:oN	gS:oN	gS:aF		vorne	oder	2	1
dP:F	nP:F	nP:M		hinten	oder	2	1
nS:pM	aS:pM	dS:oN		vorne	oder	3	1
dS:pF	aS:pF	aS:pM		vorne	oder	3	1
dS:N	nS:N	13SKE:V		hinten	oder	3	1
nS:aF	dS:aF	X:P	Swgd:R	vorne	genau	2	2
nS:M	gS:aM	aS:pM		hinten	oder	3	1
13PKE:V	13PIE:V	Swna:R		vorne	oder	3	1
o5:A	5:U	0:U	Swgd:R	hinten	genau	2	2
gP:oT	nP:oT	X:I		hinten	genau	1	1
gP:oT	aP:oT	z:V		hinten	oder	3	1
X:D	o0:A	aS:oN		hinten	oder	3	1
Pmwsd:R	Sma:R	dS:pM		vorne	oder	3	1
aS:pM	dS:pM	o5:A		hinten	genau	2	1
dS:pM	aS:pM	aS:pM		hinten	oder	3	1
nS:pM	dS:pM	dS:pM		hinten	oder	3	1
Swna:R	Swna:O	nS:pF		vorne	oder	3	1
pA:V	3SIE:V	nS:pF		hinten	oder	3	1
aS:pM	nS:pM	i:V		vorne	genau	2	1
dS:F	aS:F	13PIA:V		hinten	genau	2	1
nP:F	aP:F	o5:A	i:V	beides	genau	2	2
gS:F	nS:F	Swna:R		vorne	genau	2	1
gP:M	nP:M	o4:A		vorne	oder	2	1
i:V	13PIE:V	nP:aT		hinten	oder	3	1
i:V	13PIE:V	0:U		hinten	genau	1	1
nS:N	aS:N	nS:pM		vorne	oder	3	1
nS:N	dS:oN	X:P		vorne	genau	1	1
nS:pF	dS:pF	dS:M		hinten	oder	3	1
aP:F	gP:F	Swgd:R		vorne	genau	1	1
Swgd:R	Pmwsg:R	gP:F		hinten	oder	2	1
Swna:R	Pmwsna:O	gP:N	X:I	vorne	genau	2	2
dS:oN	nS:oN	13PIA:V		hinten	oder	3	1
dS:aM	nS:aM	nS:oN		hinten	oder	3	1
aS:M	dS:M	X:P	3SIE:V	beides	genau	2	2
nS:pM	gS:pM	gS:pM		hinten	oder	2	1
o0:A	X:D	Smd:R		vorne	oder	3	1

dP:F	gP:F	o6:A		vorne	oder	2	1
Ssna:O	Ssg:O	gS:N		hinten	oder	3	1
nS:F	gS:F	o5:A		vorne	oder	2	1
gS:F	dS:F	X:D		vorne	oder	3	1
pA:V	3SIE:V	n:O		hinten	genau	1	1
Swna:R	Pmwsna:O	a:O		hinten	genau	1	1
Swna:R	Swna:O	a:O		hinten	oder	2	1
dS:pM	nS:pM	13SIA:V		vorne	oder	3	1
Swna:R	Pmwsna:R	aP:MF		hinten	oder	3	1
i:V	13PIE:V	X:I	n:O	vorne	genau	2	2
aP:N	nP:N	n:O		vorne	genau	2	1
o0:A	X:D	13PIE:V	d:O	vorne	genau	2	2
Swna:R	Pmwsna:O	X:I	X:C	beides	genau	2	2
aS:F	nS:F	a:O		vorne	genau	2	1
nS:M	gP:M	Swgd:R	0:U	vorne	genau	2	2
Swna:O	Pmwsna:O	nP:F		hinten	oder	3	1
nP:pM	gP:pM	X:D		hinten	oder	3	1
Swgd:R	Swgd:O	Swna:R	nS:F	hinten	genau	2	2
i:V	13PIE:V	a:O		hinten	genau	1	1
nS:pF	gS:pF	gS:pF		hinten	oder	3	1
Swna:R	Pmwsna:O	dP:N		vorne	genau	2	1
X:D	X:P	dP:N		hinten	genau	1	1
aP:N	gP:N	Swgd:R		vorne	oder	2	1
Swgd:R	Pmwsg:R	gP:N		hinten	oder	2	1
13PIE:V	i:V	13SKE:V		vorne	oder	3	1
nS:M	nS:N	X:C	Ssg:R	beides	genau	2	2
Swgd:R	Smn:O	Pmwsd:R		hinten	genau	2	1
dP:F	aP:F	3SIE:V		vorne	oder	2	1
dP:F	aP:F	c4:A		vorne	oder	3	1
dS:aF	gS:aF	nS:pM		hinten	oder	3	1
nS:M	nP:M	13PIE:V		vorne	genau	2	1
Swna:R	Pmwsna:R	X:P	0:U	beides	genau	2	2
aS:F	dS:F	o6:A	X:I	beides	genau	2	2
nP:F	aP:F	4:O	X:D	beides	genau	2	2
X:D	X:P	Pmwsd:R	dP:N	hinten	genau	2	2
o0:A	X:D	dP:N		hinten	oder	3	1
dS:N	nS:N	o4:A	X:P	beides	genau	2	2
nS:F	dS:F	nS:N		vorne	genau	2	1
nS:N	aS:N	d:O		vorne	oder	3	1
nP:N	aP:N	X:P		vorne	oder	2	1
Swna:O	Pmwsna:O	aP:F		hinten	oder	3	1
X:D	X:P	dS:N	13SIA:V	hinten	genau	2	2
aP:M	nP:M	Pmwsna:O		hinten	oder	3	1
nS:M	dS:M	13SKE:V	nS:M	hinten	genau	2	2
dP:F	aP:F	X:D	o4:A	vorne	genau	2	2
dP:F	nP:F	o4:A		vorne	oder	3	1
dS:N	aS:N	o8:A		vorne	oder	3	1
5:O	o5:A	X:PR	dS:M	beides	genau	2	2
aP:N	dS:N	Smd:R		vorne	oder	3	1
13PIE:V	i:V	o0:A	i:V	vorne	genau	2	2
aP:N	nP:N	Pmwsna:R	o6:A	vorne	genau	2	2
o0:A	X:D	X:D	X:C	vorne	genau	2	2
X:D	X:P	Swgd:R		hinten	genau	1	1

dP:M	nP:M	X:I	n:O	hinten	genau	2	2
Swna:R	Pmwsna:R	nP:M		hinten	genau	1	1
X:D	X:P	Smd:R	o5:A	hinten	genau	2	2
dS:F	nS:F	X:C	Swna:R	vorne	genau	2	2
Swna:O	Pmwsna:O	X:C		hinten	oder	2	1
Sma:R	5:U	13SIA:V		hinten	genau	1	1
a:O	d:O	X:I	3SIE:V	vorne	genau	2	2
Swgd:R	Smn:O	X:D	3SIE:V	vorne	genau	2	2
Pmwsd:R	Sma:O	n:O		hinten	oder	3	1
o0:A	X:D	nS:pF		vorne	oder	3	1
Pmwsd:R	Sma:O	nS:pF		hinten	oder	3	1
Swgd:R	Pmwsd:R	dP:M		hinten	oder	2	1
d:O	a:O	n:O	13PIE:V	beides	genau	2	2
d:O	n:O	2PIE:V		vorne	oder	3	1
13PKE:V	13PIE:V	n:O	X:D	hinten	genau	2	2
nP:M	aP:M	Smg:R		hinten	oder	3	1
Swna:R	Pmwsna:O	aP:M	X:I	vorne	genau	2	2
Smd:R	Ssd:R	X:P	dS:oN	beides	genau	2	2
n:O	a:O	n:O	3SIE:V	vorne	genau	2	2
X:C	X:D	X:I	X:I	hinten	genau	2	2
dS:F	aS:F	n:O		vorne	genau	2	1
n:O	a:O	X:P		vorne	genau	1	1
aP:M	nP:M	a:O		hinten	oder	3	1
X:D	X:P	Pmwsd:R	dP:M	hinten	genau	2	2
aS:F	dS:F	Swgd:R	o5:A	vorne	genau	2	2
aS:F	gS:F	Swgd:R	o5:A	vorne	genau	2	2
Swna:R	Swna:O	gS:F	X:I	vorne	genau	2	2
nS:M	dS:M	Smd:O		vorne	oder	3	1
dP:N	aP:N	3SIE:V		vorne	oder	3	1
dP:N	nP:N	X:D	Pmwsna:R	vorne	genau	2	2
dP:N	aP:N	Pmwsna:R		vorne	oder	2	1
Swna:R	Pmwsna:O	Swna:R	X:I	vorne	genau	2	2
Swna:R	Pmwsna:O	Pmwsna:O		hinten	genau	2	1
dS:oN	aS:oN	Pmwsna:O		vorne	oder	3	1
dS:F	gS:F	X:I	X:PR	hinten	genau	2	2
nS:N	aS:N	X:P	Ssna:R	vorne	genau	2	2
nS:F	dS:F	X:PR		vorne	oder	2	1
13PIE:V	i:V	X:D	X:C	beides	genau	2	2
d:O	0:O	aS:N		hinten	oder	2	1
nP:F	aP:F	aP:F		vorne	oder	3	1
Swna:R	Pmwsna:O	nP:F		vorne	genau	2	1
nP:M	aP:M	0:U	pA:V	beides	genau	2	2
aP:M	nP:M	dS:M		hinten	oder	3	1
aP:F	dP:F	dP:F		hinten	genau	2	1
X:D	X:P	dP:F	X:C	hinten	genau	2	2
i:V	13PIE:V	o4:A	nP:F	vorne	genau	2	2
aP:F	nP:F	13PIA:V		vorne	genau	2	1
nS:F	aS:F	dS:aF		vorne	genau	2	1
dS:aF	nS:F	3SIE:V		vorne	genau	2	1
pA:V	o0:A	Swna:O	nS:F	vorne	genau	2	2
nS:F	aS:F	o0:A	pA:V	hinten	genau	2	2
6:U	5:U	X:I		vorne	genau	2	1
nP:M	aP:M	13PIA:V		hinten	genau	2	1

nP:M	dP:M	aP:N		hinten	genau	2	1
3SIE:V	1SIE:V	5:O	aS:M	hinten	genau	2	2
nS:M	nP:M	dS:pF		hinten	oder	3	1
aP:N	nS:N	3SIE:V	Ssna:R	vorne	genau	2	2
Ssna:R	Smn:R	Smn:R		vorne	oder	3	1
X:C	X:P	n:O	Smg:R	beides	genau	2	2
13SKE:V	1SIE:V	4:O		hinten	genau	2	1
dS:F	nS:F	13SIA:V	Swna:R	vorne	genau	2	2
3SIE:V	1SIE:V	n:O	i:V	hinten	genau	2	2
dP:F	aP:F	Pmwsna:R		vorne	genau	2	1
nS:F	aS:F	X:C	X:C	hinten	genau	2	2
nP:M	aP:M	nS:M		vorne	genau	2	1
Pmwsd:R	Sma:R	5:U		hinten	oder	3	1
dP:F	aP:F	Ssna:O		vorne	oder	3	1
dP:F	aP:F	nS:M		vorne	genau	2	1
gS:M	dS:M	X:PR		vorne	oder	2	1
aP:F	nP:F	o0:A		hinten	genau	2	1
Swna:R	Swna:O	X:D	o0:A	hinten	genau	2	2
o5:A	5:U	dS:M	13SKA:V	hinten	genau	2	2
13SKA:V	13SIA:V	n:O	i:V	hinten	genau	2	2
aP:N	nS:N	Swgd:R	gS:F	hinten	genau	2	2
c5:A	5:O	X:C		vorne	genau	2	1
nS:N	aS:N	gS:pM		vorne	genau	2	1
gS:pM	nS:pM	aS:N		hinten	genau	2	1
nS:F	aS:F	gS:pM	13SIA:V	hinten	genau	2	2
X:D	X:C	13SKE:V	X:C	hinten	genau	2	2
X:I	3SIE:V	gS:M	3SIE:V	vorne	genau	2	2
o0:A	X:D	o0:A	X:I	hinten	genau	2	2
nP:F	gP:F	gS:aN		hinten	genau	2	1
dS:F	nS:F	Swna:R	Ssg:R	beides	genau	2	2
dS:M	aS:M	i:V	X:D	hinten	genau	2	2
aP:N	nP:N	c4:A		vorne	genau	2	1
Ssna:R	Swna:R	nS:N	6:O	hinten	genau	2	2
dS:aF	dS:oF	13PIE:V		hinten	oder	3	1
aP:F	nP:F	gP:F		hinten	genau	2	1
nS:F	aS:F	2PKA:V		hinten	oder	3	1
dP:F	dS:F	gS:M		hinten	genau	2	1
nS:M	nS:F	dS:oN		vorne	genau	2	1
13PKE:V	13PIE:V	Swgd:R		vorne	genau	2	1
dP:F	nP:F	13PKA:V	X:I	hinten	genau	2	2
dS:M	aS:M	g:O		vorne	oder	2	1
nS:N	aS:N	aP:N		hinten	genau	2	1
aS:M	dS:M	n:O	X:P	vorne	genau	2	2
13PKE:V	13SKE:V	c4:A		hinten	oder	3	1
nS:pM	dS:pM	nS:aF		vorne	genau	2	1
13SKA:V	13SIA:V	0:U		hinten	genau	2	1
nS:F	dS:F	X:P	o6:A	vorne	genau	2	2

Literaturverzeichnis

Akkerman, Eric; Willem Meijs; Hetty Voogt-van Zutphen (1987) Grammatical Tagging in ASCOT. IN: Willem Meijs (Ed.) *Corpus Linguistics and Beyond*. Amsterdam: Rodopi, 181-193.

Atwell, Eric (1987) Constituent-Likelyhood Grammar. IN: Roger Garside; Geoffrey Leech; Geoffrey Sampson (Eds.) *The Computational Analysis of English*. London, New York: Longman, 57-66.

Atwell, Eric; Geoffrey Leech; Roger Garside (1984) Analysis of the LOB Corpus: Progress and Prospects. IN: Jan Aarts; Willem Meijs (Eds.) *Corpus Linguistics: Recent Advances in the Use of Computer Corpora in English Language Research*. Amsterdam: Rodopi, 41-52.

Baum, Leonard; J.A. Eagon (1967) An Inequality with Application to Statistical Estimation for Probabilistic Functions of Markov Processes and to a Model for Ecology. *Bulletin of the American Mathematicians Society* 73, 360-363.

Benello, Julian; Andrew W. Mackie; James A. Anderson (1989) Syntactic Category Disambiguation with Neural Networks. *Computer Speech and Language* 3, 203-217.

Brill, Eric (1992) A Simple Rule-Based Part of Speech Tagger. *Proceedings of the DARPA Speech and Natural Language Workshop*. New York: Harriman, 112-116.

Brill, Eric (1993) *A Corpus-Based Approach to Language Learning*. Ph. D. Dissertation. Department of Computer and Information Science, University of Pennsylvania.

Brill, Eric (1994) A Report of Recent Progress in Transformation-Based Error-Driven Learning. *Proceedings of the AAAI 94,* 722-727.

Brodda, Benny (1982) Problems with Tagging - and a Solution. *Nordic Journal of Linguistics* 5, 93-116.

Chanod, Jean-Pierre; Pasi Tapanainen (1995*) Tagging French - Comparing a Statistical and a Constraint-Based Method*. The Computation and Language E-Print Archive. CMP-LG@XXX.LANL.GOV.

Chomsky, Noam (1965) *Aspects of the Theory of Syntax.* Cambridge, Mass.: MIT Press.

Church, Kenneth W. (1988) A Stochastic Parts Program and Noun Phrase Parser for Unrestricted Text. *Proceedings of the Second Conference on Applied Natural Language Processing*, Association for Computational Linguistics, 136-143.

Cohen, Paul; Edward Feigenbaum (1982) *The Handbook of Artifical Intelligence, Vol. III*. Los Altos, Cal.: William Kaufmann, 323-511.

Cutting, Doug; Julian Kupiec; Jan Pedersen; Penelope Sibun (1992) A Practical Part-of-Speech Tagger. *Proceedings of the Third Conference on Applied Natural Language Processing*, Association for Computational Linguistics, 133-140.

Danø, Sven (1975) *Nonlinear and Dynamic Programming*. Wien, New York: Springer.

DeRose, Steven J. (1988) Grammatical Category Disambiguation by Statistical Optimization. *Computational Linguistics* 14:1, 31-39.

Eeg-Olofsson, Mats (1985) A Probabilistic Model for Computer-Aided Word-Class Determination. *ALLC Journal* 5, 25-30.

Eeg-Olofsson, Mats (1987) Assigning New Tags to Old Texts: An Experiment in Word Class Tagging. IN: Willem Meijs (Ed.) *Corpus Linguistics and Beyond*. Amsterdam: Rodopi, 45-47.

Forney, G. David (1973) The Viterby Algorithm. *Proceedings of the IEEE* 61, 268-278.

Garside, Roger (1987) The CLAWS Word-Tagging System. IN: Roger Garside; Geoffrey Leech; Geoffrey Sampson (Eds.) *The Computational Analysis of English*. London, New York: Longman, 30-41.

Greene, Barbara; Gerald Rubin (1971) *Automated Grammatical Tagging of English*. Department of Linguistics, Brown University, Providence, Rhode Island.

Hindle, Donald (1989) Acquiring Disambiguation Rules from Text**.** *Proceedings of the 27th Annual Meeting of the Association for Computational Linguistics*, 118-125.

Ihalainen, Ossi (1990) A Source of Data for the Study of English Dialectal Syntax: the Helsinki Corpus. IN: Jan Aarts; Willem Meijs (Eds.) *Theory and Practice in Corpus Linguistics*. Amsterdam, Atlanta: Rodopi, 83-103.

Klein, Sheldon; Robert F. Simmons (1963) A Computational Approach to Grammatical Coding of English Words. *Journal of the Association for Computing Machinery* 10, 334-347.

Koskenniemi, Kimmo (1990) Finite-State Parsing and Disambiguation. *Proceedings of COLING 90*, Helsinki, 229-232.

Lee, Geunbae; Jong-Hyeok Lee; Sanghyun Shin (1995) *TAKTAG: Two-Phase Learning Method for Statistical/Rule-Based Part-Of-Speech Disambiguation*. The Computation and Language E-Print Archive. CMP-LG@XXX.LANL.GOV.

de Marcken, Carl G. (1990) Parsing the LOB Corpus. *Proceedings of the 28th Annual Meeting of the Association for Computational Linguistics*, 243-251.

Marcus, Mitchell (1980) *A Theoy of Syntactic Recognition for Natural Language*. Cambridge, Mass.: MIT Press.

Marcus, Mitchell; Beatrice Santorini; Mary Ann Marcinskiewicz (1993) Building a Large Annotated Corpus of English: The Penn Treebank. *Computational Linguistics* 19:2, 313-330.

Marshall, Ian (1987) Tag Selection Using Probabilistic Methods. IN: Roger Garside; Geoffrey Leech; Geoffrey Sampson (Eds.) *The Computational Analysis of English*. London, New York: Longman, 42-56.

Merialdo, Bernard (1994) Tagging English Text with a Probabilistic Model. *Computational Linguistics* 20, 155-168.

Morik, Katharina (1993) Maschinelles Lernen. IN: Günther Görz (Ed.) *Einführung in die künstliche Intelligenz*. Bonn, Paris, Reading, Mass.[et al.]: Addison-Wesley, 247-301.

Nakamura, Masami; Kiyohiro Shikano (1989) A Study of English Category Prediction Based on Neural Networks. *IEEE Proceedings of the ICASSP*, Glasgow, 731-734.

Roche, Emmanuel; Yves Schabes (1995) Deterministic Part-Of-Speech Tagging with Finite-State Transducers. *Computational Linguistics* 21, 227-253.

Simmons, Robert; Sheldon Klein; Keren McConlogue (1962) Toward the Synthesis of Human Language Behavior. *Behavioral Science* 7, 402-407.

Simon, Herbert (1984) Why Should Machines Learn? IN: Ryszard Michalski; Jaime Carbonell; Tom Mitchell (Eds.) *Machine Learning: An Artificial Intelligence Approach*. Berlin, Heidelberg, New York, Tokyo: Springer, 25-38.

Stroustrup, Bjarne (1992) *The C++ Programming Language*. Reading, Mass., Menlo Park, Cal., New York [et al.]: Addison-Wesley.

Su, Keh-Yih; Ming-Wen Wu; Jing-Shin Chang (1992) A New Quantitative Quality Measure for Machine Translation Systems. *Proceedings of COLING 92*, Nantes, 433-439.

Tenebaum, Aaron; Moshe J.Augenstein (1986) *Data Structures Using Pascal*. Englewood Cliffs, N.J.: Prentice-Hall.

Voutilainen, Atro (1995) *A Syntax-Based Part-Of-Speech Analyser*. The Computation and Language E-Print Archive. CMP-LG@XXX.LANL.GOV.

Voutilainen, Atro; Pasi Tapanainen (1993) Ambiguity Resolution in a Reductionistic Parser. *6th Conference of the European Chapter of the ACL*, Utrecht, 394-403.

Weischedel, Ralph; Marie Meteer; Richard Schwartz; Lance Ramshaw; Jeff Palmucci (1993) Coping with Ambiguity and Unknown Words through Probablilistic Models. *Computational Linguistics* 19:2, 359-382.

Zell, Andreas (1994) *Simulation Neuronaler Netze*. Bonn, Paris, Reading, Mass. [et al.]: Addison Wesley.