Ludwig-Maximilians-Universität München
Centrum für Informations- und Sprachverarbeitung

Magisterarbeit im Studiengang Computerlinguistik

Entwurf und Implementierung eines natürlichsprachigen Anfragesystems für Vorlesungsverzeichnisse im Internet

Stefan Partusch

März 2009
(überarbeitet September 2011)

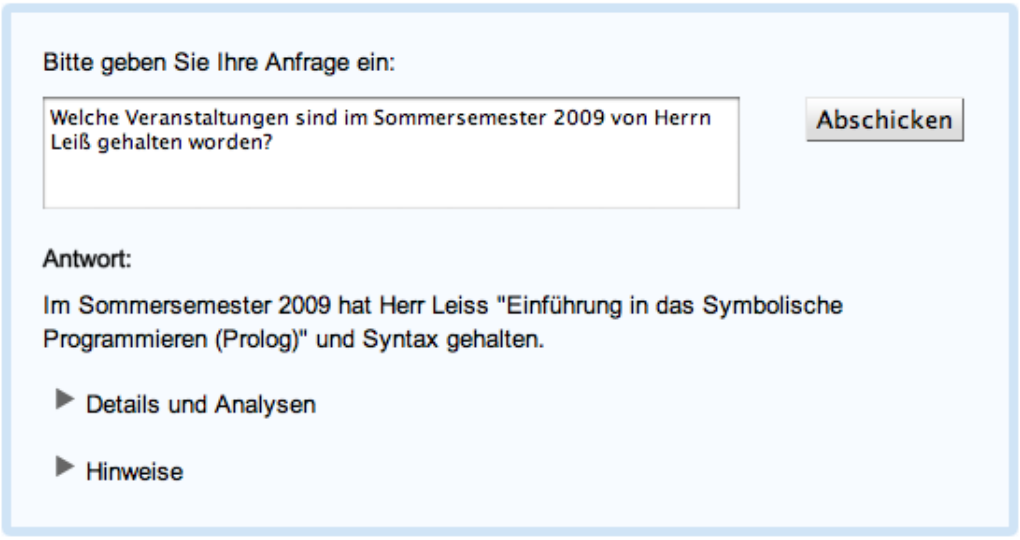
Ein natürlichsprachiges Anfragesystem für Vorlesungsverzeichnisse im Internet

Magisterarbeit im Studiengang Computerlinguistik

Stefan Partusch

März 2009

(überarbeitet September 2011)



Vorwort zur Überarbeitung

Im März 2009 habe ich meine Magisterarbeit bei der Ludwig-Maximilians-Universität München eingereicht. Die hier vorliegende Fassung der Magisterarbeit ist eine grundlegende Aktualisierung und Überarbeitung der abgegebenen Magisterarbeit. So wurde das Google Web Toolkit[1] von Version 1.5.3 auf Version 2.4.0 aktualisiert. Auch dessen neu eingeführter UiBinder wird nun beim Client eingesetzt. Und statt Apache Lucene[2] Version 2.4.0 wird jetzt die Version 3.3.0 genutzt. Da die bisher verwendete Java-Prolog-Schnittstelle - Declarativa InterProlog[3] - seit mehreren Jahren nicht mehr weiterentwickelt wurde, wird nun stattdessen jpl für die Kommunikation zwischen Java und Prolog verwendet. jpl ist in aktuellen Versionen von SWI-Prolog[4] integriert und wird in diesem Rahmen weiterentwickelt und gepflegt. Aufgrund der Verwendung von jpl ist es allerdings nicht mehr möglich, den in Java entwickelten Server zusammen mit YAP-Prolog[5] zu verwenden. Die in Prolog geschriebenen Teile der Magisterarbeit entsprechen aber nach wie vor dem ISO-Standard.
Das von dem natürlichsprachigen Anfragesystem verwendete Fragment des Deutschen wurde bei der Überarbeitung um das Verb *ähneln* ergänzt. Die in Prolog entwickelte Grammatik, die dieses Sprachfragment implementiert, wurde entsprechend erweitert. Diese Erweiterung und die grundlegenden Aktualisierungen der Softwarekomponenten haben es notwendig gemacht, den Server neu zu schreiben.
Der vorliegende Text wurde an mehreren Stellen an diese Veränderungen angepasst und manche Formulierung überarbeitet. Das dritte Kapitel wurde erweitert und die Kapitel vier, fünf und sechs wurden hinzugefügt bzw. neu geschrieben. Der Schluss berücksichtigt jetzt auch das erst nach Abgabe der Magisterarbeit veröffentlichte SIREn.[6]
Die verwendeten Vorlesungsdaten umfassen nun die Veranstaltungen des Centrums für Informations- und Sprachverarbeitung der Ludwig-Maximilians-Universität München bis zum Wintersemester 2011/2012.

Stefan Partusch
München, 13. September 2011

[1] `http://code.google.com/webtoolkit/`
[2] `http://lucene.apache.org/`
[3] `http://www.declarativa.com/interprolog/`
[4] `http://www.swi-prolog.org/`
[5] `http://www.dcc.fc.up.pt/~vsc/Yap/`
[6] `http://siren.sindice.com/`

Inhaltsverzeichnis

Teil I.

Entwurf

1. Einleitung

Sprachliche Kommunikation mit Maschinen ist ein in der Informatik und der Computerlinguistik schon lange verfolgter Traum. Natürliche Sprachen, wie zum Beispiel Deutsch, sind einerseits ein mächtiges Werkzeug zur Informationsübermittelung und werden andererseits zumindest von erwachsenen Muttersprachlern als mühelos verwendbar empfunden. Aus diesem Grund sind Computerprogramme und Systeme, die sich durch sprachliche Äußerungen steuern lassen und auch selbst natürlichsprachliche Antworten erzeugen, sehr attraktiv, da sie eine einfache und „natürliche“ Bedienung ermöglichen können. Insbesondere das mitunter aufwendige Erlernen der Bedienung eines bestimmten Programmes könnte durch sprachliche Schnittstellen entfallen oder wenigstens stark reduziert werden. Doch in der Praxis verwenden selbst Systeme, die auf den Umgang mit sprachlichen Daten spezialisiert sind, kaum natürlichsprachliche Eingaben der Benutzer. So sind die Internetsuchmaschinen mit den größten Marktanteilen in den USA [SCO09] Google[1], Yahoo![2] und Microsoft Live Search[3] auf die Suche mit und nach Stichwörtern spezialisiert. Enthalten Suchanfragen bei diesen Anbietern komplexe sprachliche Strukturen, wie es bei natürlichsprachigen Sätzen und Fragen der Fall ist, wird dies nicht erkennbar anders als verglichen mit einfachen Stichwörtern behandelt. Dennoch ist anzunehmen, dass Menschen auch bei diesen Suchmaschinen immer wieder natürliche Sprache - damit sind hier linguistisch strukturierte Fragmente oder ganze Sätze gemeint - verwenden, um ihre Anfragen zu formulieren. So entfallen bei Live Search ungefähr fünf Prozent aller Anfragen auf diese Art natürlichsprachiger Suchanfragen. Erfahrungen mit Suchmaschinen, die neben Stichwörtern auch natürlichsprachige Anfragen erlauben, zeigen zudem, dass Benutzer von einer derartigen Möglichkeit noch größeren Gebrauch machen, wenn natürlichsprachliche Anfragen erkennbar unterstützt werden [LIV08].
Der wichtigste Grund, warum trotz der vorhandenen Nachfrage natürlichsprachliche Anfragen in Deutsch oder einer anderen natürlichen Sprache bei den meisten Suchmaschinen nicht möglich sind, dürfte in dem Umfang der natürlichen Sprachen liegen. Durch ihre Komplexität und Vielfältigkeit lassen sie sich ausgesprochen schwer algorithmisch modellieren und in Computersystemen implementieren.
Schränkt man den Sprachumfang jedoch ein, der modelliert und implementiert werden soll, dann vereinfacht dies zwar einerseits Modellierung und Implementierung. Andererseits „entdecken“ Menschen, wenn sie mit eingeschränkten sprachlichen Systemen arbeiten, in der Regel jedoch, dass die Systeme eingeschränkt sind und nicht beliebige natürlichsprachige Anfragen verarbeiten können. In der Praxis führt dies schnell zur Frustration der Benutzer und zu geringer Akzeptanz des sprachverarbeitenden Systems.

[1] `http://www.google.com/`
[2] `http://www.yahoo.com/`
[3] `http://www.live.com/`

Denn die Benutzer müssen nun oft durch ausprobieren den Sprachumfang des Systems ermitteln oder zumindest, in welcher Form ihre jeweilige Anfrage verarbeitbar ist. Dass die Bedienung nicht extra erlernt werden muss - einer der wichtigsten Punkte, der für natürlichsprachige Schnittstellen spricht - scheint damit in Frage gestellt zu sein. Die Akzeptanz natürlichsprachlicher Systeme wird dadurch weiter erschwert, dass Benutzer im Fehlerfall dazu neigen, nicht eigene Eingaben als die Ursache für misslungene Interaktionen zu sehen, sondern die Verantwortung dem System zuzuschreiben. Bei traditionellen nicht-sprachlichen Bedienkonzepten ist dies weniger ausgeprägt [RON01].
Um von Benutzern akzeptiert zu werden, muss ein natürlichsprachige Anfragen verarbeitendes System eine Lösung für die dargestellten Herausforderungen finden. Ein Lösungsansatz soll im Rahmen dieser Magisterarbeit vorgestellt werden. Entwickelt wurde ein Anfragesystem, das über einen eingeschränkten Sprachumfang des Deutschen verfügt. Innerhalb dieses Fragments werden auf Deutsch formulierte Anfragen verarbeitet und auf Deutsch von dem System beantwortet. Zur Vermeidung des diskutierten Problems bei der Verwendung eines eingeschränkten Sprachumfangs wird eine automatische „Suggest-Funktion“ implementiert. Darunter ist eine Funktion zu verstehen, die während die schriftlichen Anfragen formuliert und eingegeben werden, bereits Vorschläge macht, was für Fragen mit der bisherigen Eingabe gestellt werden können. Diese Funktion soll Benutzer des Systems unbemerkt zu Formulierungen führen, die das System auch tatsächlich verarbeiten kann. Auf diese Weise soll die Notwendigkeit, dass der Benutzer das durch das System verarbeitbare Sprachfragment herausfinden muss, stark abgeschwächt werden.
Das System, welches in dieser Magisterarbeit entwickelt wurde, implementiert ein Fragment des Deutschen für zahlreiche Anfragen im Zusammenhang mit Vorlesungsverzeichnissen von Universitäten (Kapitel 2.2.2). Wann immer Beispielanfragen und -antworten gezeigt werden, basieren diese auf den Vorlesungsdaten des Centrums für Informations- und Sprachverarbeitung der Ludwig-Maximilians-Universität München (LMU) vom Sommersemester 2004 bis einschließlich dem Wintersemester 2011/2012. Darüber hinaus soll das System auch Vorteile natürlichsprachlicher Systeme verdeutlichen indem es zum Beispiel auf gezielte Fragen ebenso gezielt antwortet. Fragen wie „Welche Hauptseminare finden statt?“ oder „In welchen Räumen fand das Proseminar Syntax statt?“ erlauben exakte Antworten in einem einzigen Satz. Mit einer natürlichsprachigen Antwort kann dem Benutzer so die gesuchte Information zielsicher und prägnant präsentiert werden. Besonders in diesen Fällen ist ein natürlichsprachiges System traditionellen Suchmaschinen überlegen, die nur eine Auswahl an Dokumenten präsentieren können, in welchen im Idealfall die gesuchten Informationen vorkommen.
Um auch Themenbeschreibungen und Veranstaltungstitel durchsuchbar zu machen, baut das hier entwickelte System zusätzlich auf der Funktionalität einer Volltextsuche auf (Kapitel 2.3.1). Volltextsuchen sind auf das effiziente Durchsuchen von Textdokumenten nach Stichwörtern und das Ranking der Dokumente der Ergebnismenge nach Relevanz zur Anfrage spezialisiert. Durch die Architektur des Gesamtsystems sollen die Vorteile von Volltextsuche und einem natürlichsprachigen Anfragesystem kombiniert werden.

2. Architektur des Systems

2.1. Allgemein

Dem im Rahmen dieser Magisterarbeit entwickelten natürlichsprachigen Anfragesystem für Vorlesungsverzeichnisse im Internet (NASfVI) liegt eine Client-Server-Architektur zu Grunde. Der Server ist dabei die zentrale Komponente und das Herzstück des Systems. Er verfügt über eine Schnittstelle zur Kommunikation mit Clients, empfängt von diesen Anfragen und bearbeitet sie. Die Aufgabe des Clients ist es, die Anfragen eines Benutzers aufzunehmen und an den Server weiterzuleiten und schließlich die Antwort des Servers für den Benutzer aufzubereiten und anzuzeigen. Client und Server sind technisch unabhängig voneinander lauffähig und müssen sich daher nicht auf demselben Computer befinden. Eine sprachverarbeitende Komponente, welche das Sprachfragment des Deutschen implementiert, ist Bestandteil des Servers, wird jedoch aus organisatorischen und technischen Gründen in dieser Magisterarbeit als eigenständige Komponente behandelt.[1]

Die Grundkomponenten von NASfVI sind also:

- eine sprachverarbeitende Komponente, die das Sprachfragment des Deutschen bereitstellt;
- ein Client in Form einer Web-Oberfläche;
- ein Server, der die Suchkomponente mit Volltextsuche und in dieser Form auch die Veranstaltungsdaten verwaltet, der die sprachverarbeitende Komponente kapselt und der nach Außen eine Web-Schnittstelle zum Gesamtsystem, sowie den Client für Browser bereitstellt;

Die Implementierung des Servers wird ferner durch eine Funktion ergänzt, um Veranstaltungsinformationen aus dem Online-Vorlesungsverzeichnis der LMU[2] zu extrahieren (siehe Anhang B.2). Die damit extrahierten Daten können dem Index des Servers hinzugefügt werden.
NASfVI ist auf den serverbasierten Einsatz im Internet ausgelegt und benötigt einen Webserver in dessen Kontext die Serverkomponente von NASfVI gestartet wird. Benutzer sehen beim Aufrufen des Webservers in Browsern den NASfVI-Client und können so

[1] Der Server umfasst letztlich die Volltextsuche, sowie die sprachverarbeitende Komponente und implementiert eine Schnittstelle nach Außen. Alle drei Aspekte lassen sich funktional unterscheiden. Technisch ist die sprachverarbeitende Komponente mit der logischen Programmiersprache Prolog realisiert, der restliche Server dagegen mit der objektorientierten Programmiersprache Java.

[2] `https://lsf.verwaltung.uni-muenchen.de`

Anfragen an NASfVI stellen. Es muss daher keinerlei spezielle Software für das Stellen von Anfragen an NASfVI installiert werden.
In diesem Kapitel werden nachfolgend die Komponenten in ihren Grundzügen vorgestellt.

2.2. Die Sprachverarbeitungskomponente

Die sprachverarbeitende Komponente ist der umfassendste Teil von NASfVI und ist in ISO-Prolog [ISO13211-1] geschrieben. Prolog bietet sich für die Implementierung des Sprachfragments an, da es eine deklarative Programmierung erlaubt und mit dem *Definite Clause Grammar*-Formalismus [ISO13211-3] über einen Formalismus verfügt, welcher automatisch von Prolog zum Parsen und Generieren von Ausdrücken einer Grammatik genutzt werden kann. Die sprachverarbeitende Komponente von NASfVI lässt sich dabei grundsätzlich in die Bestandteile Syntax, Morphologie, Semantik und Lexikon untergliedern. Die Syntax-Regeln, die für das Erkennen und Generieren von natürlichsprachigen, deutschen Eingaben verwendet werden, liegen als „Definite Clause Grammar"-Regeln vor. Obwohl in einem Standard definiert [ISO13211-2], werden in der sprachverarbeitenden Komponente keine Module verwendet, da das Standard-Modulsystem von den meisten Prolog-Implementierungen nicht unterstützt wird. Durch die Konzentrierung auf Standard-Prolog, wie es durch die Internationale Organisation für Normung (ISO) spezifiziert wurde [ISO13211-1], ist die Sprachverarbeitung grundsätzlich an keine bestimmte Implementierung von Prolog gebunden. So wurde die sprachverarbeitende Komponente erfolgreich mit SWI-Prolog[3] und YAP[4] getestet. Damit ist einerseits die Nutzung der zahlreichen Werkzeuge und Bibliotheken, die SWI-Prolog für die Entwicklung von Prolog-Programmen zur Verfügung stellt, möglich (siehe Anhang A.1). Andererseits kann die Sprachverarbeitungskomponente aber auch mit dem hoch-performanten YAP oder anderen den ISO-Standard unterstützdende Prolog-Implementierungen eingesetzt werden.
Die Idee für die Sprachverarbeitung in NASfVI ist durch die Proseminarvorlesung „Computerlinguistik II" von Dr. Hans Leiß aus dem Wintersemester 2005/06 am Centrum für Informations- und Sprachverarbeitung der LMU [LES05] motiviert. NASfVI ist jedoch eine vollständige Neuentwicklung und arbeitet mit einem anderen theoretischen Ansatz und verfolgt daraus resultierend ein anderes Vorgehen bei der Implementierung. Denn die Komponente muss im Vergleich zu „Computerlinguistik II" andere Ziele erfüllen: Um die Generierung von Vorschlägen zu unterstützen, müssen zu gegebenen Satzanfängen möglichst viele Sätze erzeugt werden können, die mit diesen beginnen. Dabei darf es keine Rolle spielen, ob ab einem Phrasenende, einem Wortende oder ab einem unvollständig getippten Wort ergänzt werden muss. Damit der Benutzer nicht die Übersicht verliert und einen tatsächlichen Nutzen von dieser Vorschlagsfunktion hat, muss die Anzahl der Vorschläge allerdings auch übersichtlich bleiben. Deswegen muss auch ein Mechanismus vorhanden sein, der geeignet ist, die Vorschlagszahl zu reduzieren, ohne die nicht angezeigten Vorschläge zu unterdrücken. Bei einem Einsatz im Internet und bei Satzvorschlägen, die während der Eingabe von Anfragen eingeblendet werden, muss das System

[3] `http://www.swi-prolog.org/`
[4] `http://www.dcc.fc.up.pt/~vsc/Yap/`

auch möglichst effizient arbeiten und schnell auf Eingaben reagieren. Dies lässt sich vor allem dadurch erreichen, dass zu kurze Satzanfänge, die ganz besonders viele Vorschläge ermöglichen, noch nicht vollständig verarbeitet werden. Die Eingabe muss also eine gewisse Mindestlänge haben, bevor Vorschläge erzeugt werden. Da der Benutzer dadurch anfangs jedoch ohne Vorschläge einen Satzanfang eingeben muss, sollte die Syntax sehr flexibel sein - und zum Beispiel auch freie Phrasenstellungen erlauben.
Da das Anfragesystem mit universitären Veranstaltungsdaten arbeitet, muss die sprachverarbeitende Komponente außerdem mit einer Vielzahl von Eigennamen umgehen können. Im Besonderen Veranstaltungstitel sind von Fachterminologie und verschiedener syntaktischer Komplexität geprägt. Dies wird bereits an zwei Beispielen für Veranstaltungstitel an der LMU deutlich: sowohl „Syntax“[5] als auch „Kognitionspsychologische und emotionspsychologische Aspekte von Hilflosigkeit und Depression“[6] sind Veranstaltungstitel. NASfVI löst dieses Problem indem Anfragen zwar syntaktisch analysiert werden, für Eigennamen jedoch nicht analysierte Leerstellen vorgesehen sind. Mit Hilfe der gewonnenen syntaktischen Informationen werden Eigennamen dadurch ohne umfangreiche Lexika und spezialisierte Regeln bestimmbar.

2.2.1. Theoretische Grundlagen

Die Sprachverarbeitungskomponente von NASfVI modelliert ein sprachliches Fragment des Deutschen auf lexikalischer, morphologischer, syntaktischer und semantischer Ebene. Ein solches Modell basiert zwingend auf theoretischen Annahmen über den modellierten Gegenstand - in diesem Fall auf Annahmen über das Deutsche. Die Modellierung und Verarbeitung des Deutschen in NASfVI geht auf die Konstituentenanalyse, das Feldermodell des Deutschen, die Optimalitätstheorie, die Valenz-Theorie und die Montague-Grammatik zurück. Diese theoretischen Grundlagen werden nachfolgend erläutert.

Die Konstituentenanalyse

Der Syntax in NASfVI liegt die Annahme der Existenz von Konstituenten zugrunde. Der Begriff der Konstituente stammt aus dem amerikanischen Strukturalismus. Dürscheid ([DUE07], Seite 29) definiert Konstituenten als „sprachliche Einheiten, die Teile einer größeren Einheit sind“, die in der Syntax auch als Phrasen bezeichnet werden. Diese syntaktischen Einheiten werden durch sogenannte syntaktische Tests wie dem Permutationstest, dem Substitutionstest, dem Eliminierungstest oder dem Koordinationstest bestimmt, welche Dürscheid [DUE07] erläutert. Diese Tests setzen jedoch einen kompetenten Sprecher der modellierten Sprache voraus, der die Grammatikalität von Äußerungen intuitiv beurteilen kann. Aus diesem Grund kann die Objektivität der syntaktischen Tests angezweifelt werden und die Tests müssen im Allgemeinen kritisch beurteilt werden. Andererseits gibt es psychologische Untersuchungen, die darauf hindeuten, dass die

[5] Eine im Sommersemester 2008 am Centrum für Informations- und Sprachverarbeitung (LMU) von Dr. Jörg Schuster gehaltene Proseminarvorlesung.
[6] Ein im Sommersemester 2006 an der Fakultät für Psychologie und Pädagogik (LMU) von Dipl.-Psych. Silvia Queri gehaltenes Seminar.

mit diesen Verfahren gewonnen, traditionellen Einheiten von Nominal-, Verbal- und Präpositionalphrasen eine perzeptive Entsprechung bei Menschen haben könnten [FB65]. Dieser Fragen ungeachtet haben Konstituentenanalysen eine lange Tradition und erweisen sich als ausgesprochen nützlich. In Prolog steht mit der Unterstützung von Definite Clause Grammars (DCG) ein Formalismus zur Verfügung, der eine direkte Modellierung von Phrasenstrukturgrammatiken und damit von Konstituenten erlaubt. Dies wird in der Sprachverarbeitungskomponente von NASfVI ausgenutzt: Die syntaktische Analyse basiert auf Konstituenten oder Phrasen, welche mit DCG-Regeln modelliert werden.

Das Feldermodell des Deutschen

Eine weitere syntaktische Annahme ist, dass deutschen Sätzen das Feldermodell des Deutschen als Topologie zugrunde gelegt werden kann. Dieses Modell postuliert das allgemeine Satzmuster [Vorfeld] [linke Satzklammer] [Mittelfeld] [rechte Satzklammer] [Nachfeld]. Linke und rechte Satzklammer werden dabei von den Verbformen des Prädikats besetzt, während das Vorfeld, das Mittelfeld und das Nachfeld von nicht verbalen Phrasen besetzt werden. Besetzung bedeutet, dass sich eine Phrase im allgemeinen Satzmuster im Bereich eines Feldes befindet, oder eine Verbform im Bereich einer Satzklammer. Das strukturierende Element, also das Element, dass die Felder begrenzt, sind die Satzklammern. Innerhalb des Mittelfelds und des Nachfelds können theoretisch unbeschränkt viele Phrasen auftreten und innerhalb der rechten Satzklammer unbeschränkt viele Verbformen. Das Vorfeld kann dagegen nur ein Satzglied beinhalten, das bedeutet innerhalb des Modells von NASfVI eine Phrase beinhalten, oder insgesamt nicht vorhanden sein. Die linke Satzklammer kann exakt eine Verbform aufnehmen. Das Modell macht keine Annahmen über die lineare Reihenfolge der Phrasen innerhalb eines Feldes. Es werden grundsätzlich drei Satzformen anhand der Stellung des finiten Verbs innerhalb der Satzklammern unterschieden. Ist das Vorfeld mit einer Phrase besetzt und befindet sich die finite Verbform in der linken Satzklammer, spricht man von einem Verbzweitsatz. Fehlt das Vorfeld und befindet sich die finite Verbform in der linken Satzklammer, dann spricht man von einem Verberstsatz. Befindet sich die finite Verbform dagegen in der rechten Satzklammer, spricht man von einem Verbletztsatz [DUG06].

Vorfeld	linke Sk.	Mittelfeld	rechte Sk.
Wer	hält	eine Vorlesung über Syntax	
Ein Dozent	hat	über Syntax eine Vorlesung	gehalten
Von wem	ist	eine Vorlesung über Syntax	gehalten worden
	Hält	ein Dozent eine Vorlesung	
	War	eine Vorlesung	gehalten worden

Tabelle 2.1.: Auf dem Feldermodell basierende Satzanalysen mit jeweils leerem Nachfeld

NASfVI implementiert das Feldermodell des Deutschen explizit (siehe Kapitel 3.2.8). Da das Modell keine Annahme über die Reihenfolge einzelner Phrasen innerhalb der Felder

macht, ist die Reihenfolge der Phrasen innerhalb der Felder auch in NASfVI zunächst völlig frei.

Die Optimalitätstheorie

Zwar ist im Deutschen im Allgemeinen eine vergleichsweise freie Phrasenstellung möglich. Doch bei tatsächlichen sprachlichen Äußerungen zeigt die Abfolge der Phrasen innerhalb eines Satzes Regelmäßigkeiten. So kann zum Beispiel eine Nominalphrase mit einem expletiven Es nur im Vorfeld auftreten oder als erste Phrase im Mittelfeld. Ein Satz wie **Gibt eine Vorlesung es* ist sehr stark markiert und gilt in der Regel als ungrammatisch. Doch auch bei zweifelsfrei grammatischen Sätzen folgt die Phrasenstellung gewissen Tendenzen. Eine derartige Tendenz ist zum Beispiel, dass Nominalphrasen, die etwas Belebtes bezeichnen, im Mittelfeld vor Nominalphrasen stehen, die etwas Unbelebtes bezeichnen ([DUG06], 1362). Die Tendenzen können aber auch rein syntaktisch sein, wie die, dass Dativobjekte im Mittelfeld vor Akkustivobjekten stehen ([DUG06], 1353). Eine weitere Tendenz ist die, dass definite Phrasen im Mittelfeld vor indefiniten stehen ([DUG06], 1363). Diese Tendenzen sind jedoch nicht absolut, sondern sind lediglich Indizien zur Markiertheit eines Satzes. Eine Tendenz kann ohne Weiteres „verletzt“ werden, ohne dass ein Satz dadurch ungrammatisch würde. Dies sei an folgenden Beispielen verdeutlicht:

- Franz gibt dem Verkäufer fünf Euro (Dativobjekt vor Akkusativobjekt)
- Franz gibt fünf Euro dem Verkäufer (Akkusativobjekt vor Dativobjekt)
- Hält ein Professor die Vorlesung (belebt/indefinit vor unbelebt/definit)
- Hält die Vorlesung ein Professor (unbelebt/definit vor belebt/indefinit)
- Hält ein Professor eine Vorlesung (belebt/indefinit vor unbelebt/indefinit)
- ?Hält eine Vorlesung ein Professor (unbelebt/indefinit vor belebt/indefinit)

Derartige bedingte Einschränkungen der Stellungsvarianz im Deutschen werden bei der Sprachverarbeitung in NASfVI direkt modelliert. Geeignete theoretische Grundlagen stellt dafür die Optimalitätstheorie bereit. Bei der Optimalitätstheorie handelt es sich um keine konkrete syntaktische Theorie, sondern um ein allgemeines Modell, das Entscheidungsprozesse in Systemen beschreibt, die nach Wichtigkeit geordnete Anforderungen berücksichtigen (vgl. [DUE07], Seite 157). Eine zentrale Annahme in der Optimalitätstheorie ist, dass diese Anforderungen nicht alle gleichzeitig erfüllt sein müssen, sondern einzelne Anforderungen auch verletzt sein können. Bei mehreren Realisierungsmöglichkeiten für einen sprachlichen Ausdruck - zum Beispiel bei verschiedenen Phrasenstellungen eines Satzes - entscheidet eine hierarchische Ordnung der Anforderungen nach Wichtigkeit, welcher dieser Kandidaten für den Ausdruck „optimal“ ist. Die verschiedenen Kandidaten stehen dabei im Wettstreit miteinander. Verletzt ein Kandidat für den Ausdruck mehr gleichwertige Anforderungen als ein anderer oder eine höherwertige Anforderung als ein anderer Kandidat, scheidet der fragliche Kandidat aus. Der Kandidat,

der am Ende dieses Prozesses übrig bleibt, ist der „optimale“ Kandidat für den Ausdruck. Die Anforderungen selbst können beliebig modelliert werden, da die Optimalitätstheorie lediglich das Verfahren beschreibt.
Das optimalitätstheoretische Verfahren lässt sich gut auf die syntaktische Stellungsvarianz der Phrasen innerhalb der Felder übertragen. Die Tendenzen, welche die Stellungen der Phrasen beeinflussen, werden dabei als die optimalitätstheoretische Anforderungen modelliert. Dadurch ist die sprachverarbeitende Komponente nun in der Lage, die Markiertheit einer Eingabe zu bewerten und selbst nur möglichst gering markierte Sätze als Vorschläge im Rahmen der Suggest-Funktion zu generieren. Statt der strengen Hierarchie bei der Wichtigkeit der Anforderungen, wird in NASfVI jedoch für jede Anforderung ein numerischer Markiertheitswert definiert. Die Markiertheit eines Satzes ergibt sich dabei aus der Summe der Markiertheiten der durch den Satz verletzten Anforderungen. Dadurch ist es theoretisch möglich, dass in NASfVI ein Kandidat unter Umständen nicht ausgeschlossen wird, obwohl er höherwertige Anforderungen verletzt, andere Kandidaten aber an sich niederwertigere Anforderungen mehrfach verletzen und dadurch in der Summe eine höhere Markiertheit erzielen. In der Optimalitätstheorie ist dies nicht möglich, da eine höherwertige Anforderung immer schwerer wiegt als niederwertigere Anforderungen, egal wie häufig diese auch verletzt sein mögen. Die Umsetzung in NASfVI ist jedoch ein guter Kompromis, der die Implementierung vereinfacht.
Kritisch angemerkt werden muss, dass die numerischen Markiertheitswerte der Anfoderungen anhand des Sprachfragments und des erwünschten Verhaltens des Systems ermittelt und nicht anderweitig theoretisch hergeleitet sind. Dies ist jedoch auch im allgemeinen Fall bei optimalitätstheoretischen Ansätzen eine schwer umzusetzende Forderung.

Die Valenztheorie

Welche Phrasen können in einem Satz auftreten? In NASfVI wird das Auftreten von Phrasen durch die Verbvalenz, die Nomenvalenz und allgemeinen Angaben auf Satzebene gesteuert. Der Begriff der Valenz wurde von dem französischen Sprachwissenschaftler Lucien Tesnière geprägt, der mit dem aus der Chemie stammenden Valenzbegriff den Umstand beschrieb, dass Verben mit bestimmten Argumenten auftreten. Verben öffnen gemäß der Valenztheorie gleichsam Leerstellen in Sätzen, welche von bestimmten Phrasen gefüllt werden müssen oder können ([DUE07], Seite 109f). So öffnen transitive Verben im Aktiv zwei syntaktische Leerstellen: eine für das Subjekt und eine für ein Akkusativobjekt. Beiden Leerstellen werden semantische Rollen zugewiesen, weswegen auch von syntaktisch-semantischer Valenz gesprochen wird. Die Phrasen, die ein Verb auf dieser Weise fordert, bilden dessen Valenzrahmen. Das transitive Verb *lieben* verlangt zum Beispiel ein Subjekt und ein Akkusativobjekt: *Maria liebt den Linguisten.* Dabei entspricht das Subjekt der semantischen Rolle des Agens, der handelnden Entität, und das Akkusativobjekt dem Patiens, der betroffenen Sache oder Person ([DUG06], 521ff). Das Auftreten der Argumente hängt funktional von dem Vorhandensein des Verbs ab, das diese Argumente fordert. Die Argumente des Verbs werden auch als Ergänzungen bezeichnet und können fakultativ oder obligatorisch sein. Im Gegensatz zu fakultativen Ergänzungen müssen obligatorische Ergänzungen in jedem Fall realisiert werden. Das

heißt, die entsprechenden Leerstellen müssen ausgefüllt werden, da der Satz sonst ungrammatisch würde: **Den Linguisten liebt.*
Im Zusammenhang mit der Verbvalenz müssen weitere Faktoren berücksichtigt werden. Denn die Art der Ergänzungen hängt auch von weiteren Faktoren wie zum Beispiel dem Genus Verbi ab. So unterscheidet sich zum Beispiel der Valenzrahmen eines transitiven Verbs im Passiv systematisch von dem im Aktiv. Denn bei transitiven Verben wird das Agens im Aktiv als Subjekt im Nominativ realisiert, im Passiv dagegen durch eine fakultative Präpositionalphrase häufig mit der Präposition *von*. Das Akkusativobjekt im Aktiv wird im Passiv dagegen zum Subjekt im Nominativ ([DUG06], 798). Da die Agensphrase dabei fakultativ ist, kann zum Beispiel mit dem transitiven Verb *lieben* im Passiv der Satz *Der Linguist wird geliebt* gebildet werden, bei dem das Agens nicht realisiert ist.

Einige Linguisten sprechen nicht nur Verben, sondern auch Nomen eine Valenz zu. Für einen Überblick über die verschiedenen Klassifikationen bei der Nomenvalenz siehe Iroaie [IRO07]. Auch in NASfVI sind Nomen grundsätzlich zur Valenz fähig und verfügen über einen Valenzrahmen. Die einzigen Nomen, die in dem hier modellierten Sprachfragment Ergänzungen fordern, sind allerdings Veranstaltungsbezeichnungen wie Vorlesung, Proseminar und Hauptseminar. Diese verfügen über eine Themaergänzung nach Teubert [TEU79] zitiert in Iroaie [IRO07]. Die Themaergänzung dieser Nomen wird in dem modellierten Sprachfragment immer durch eine fakultative Präpositionalphrase mit *über* gebildet. Eine Besonderheit bei der Realisierung der Nomenvalenz ist in NASfVI, dass die nominalen Ergänzungen nicht der Nominalphrase syntaktisch untergeordnet werden, sondern auf der obersten syntaktischen Ebene des Satzes zusammen mit den Ergänzungen des Verbes positioniert sind. Dadurch unterliegen die Ergänzungen der nominalen Valenzrahmen derselben Stellungsvarianz wie die Ergänzungen der Verbvalenz. Es ist also ein Satz wie *Über Syntax gibt es eine Vorlesung* möglich, bei dem das expletive Es und das Akkusativobjekt aus dem Valenzrahmen des Verbs *geben* stammen, während die *über*-Präpositionalphrase als fakultative Ergänzung der Nomenvalenz der *Vorlesung* entstammt.

Die Montague-Grammatik

Die Semantik der einzelnen Wörter ist in dem hier entwickelten Modell vollständig im Lexikon definiert. Die Lexikon-Einträge beinhalten sowohl eine einfache semantische Typisierung als auch logische Terme, die den Lambda-Kalkül verwenden, um die Bedeutung von Phrasen und Sätzen kompositional beschreiben zu können. Semantische Kompositionalität bedeutet, dass sich die Semantik von zusammengesetzten Ausdrücken vollständig aus der Semantik der Unterausdrücke ergibt. Um dies zu ermöglichen, wird die Implementierung des ungetypten Lambda-Kalküls aus „Computerlinguistik 2“ [LES05] verwendet. Die semantischen Formeln folgen in NASfVI den Kategoriendefinitionen von Richard Montague [MON73]. Eine Ausnahme davon bilden jedoch die an der Nomenvalenz beteiligten Wörter und Phrasen. Diese Abweichung wird in Kapitel 3.1.1 erläutert. Obgleich eine Implementierung des ungetypten Lambda-Kalküls verwendet wird, ergibt sich in NASfVI implizit ein typisierte Verwendung der Variabeln aufgrund der Berücksichtigung

der aus den Lexikoneinträgen stammenden semantischen Typen.
Die syntaktischen Kategorien, die Montague definiert, bauen auf den beiden Basisausdrücken e für Entitäten und t für Wahrheitswerte auf. Intransitive Verbalphrasen werden von Montague als t/e definiert. Dies bedeutet, dass intransitive Verbalphrasen (IV) ein Argument vom Typ e erwarten und beim Anwenden einer Variabel des Typs e auf die Phrase den Typ t erzeugen. Anders ausgedrückt ist die Kategorie t/e eine Funktion von e zu t. Weitere für NASfVI relevante Kategorien sind $t/(t/e)$ oder abgekürzt t/IV für Nominalphrasen und Eigennamen (T), die Kategorie $t//e$, welche ebenfalls eine Funktion von e zu t ist, für Gattungsnamen (CN), die Kategorie IV/T für transitive Verben (TV), die Kategorie IV/IV für „intransitive Adverbien“ (IAV) und die Kategorie IAV/T für Präpositionen [MON73]. Dowty et al ergänzen außerdem eine Kategorie für Artikel (DET), welche der Kategoriendefinition T/CN entspricht ([DWP81], Seite 183).
Bei Montage entsprechen Sätze dem Basisausdruck t. Und so ist zum Beispiel die folgende syntaktische Struktur aufgrund der Kategoriedefinitionen für einen Satz möglich:

$$(DET \cdot (CN \cdot e)) \cdot (IAV \cdot (TV \cdot (DET \cdot (CN \cdot e))))$$

Durch die Kategorien wird definiert, welcher Ausdruck mit welchem anderem Ausdruck kombiniert werden kann, um einen zusammengesetzten Ausdruck zu erzeugen. Da bei Montague die Syntax immer parallel zur Semantik aufgebaut wird und es eine Entsprechung zwischen syntaktischen und semantischen Regeln gibt, übertragen sich die syntaktischen Kategorien auch auf die Semantik. Zwar wird die *Intensional Logic* von Montague in NASfVI nicht weiter berücksichtigt. Doch die logischen $\lambda-$Terme der Semantik in NASfVI folgen bei der Verbvalenz den von Montague definierten Kategorien. Es ergibt sich bei der Semantik also eine Kompositionalität und Form der zusammengesetzten Terme, wie sie durch die syntaktischen Kategorien hier skizziert wurde.
An dieser Stelle muss betont werden, dass sich der syntaktische Aufbau in NASfVI aus dem Feldermodell des Deutschen ergibt. Das bedeutet insbesondere, dass es auf syntaktischer Ebene keine Verbalphrasen, sondern nur Satzklammern gibt. Die Semantik folgt aber der hier skizzierten Form und kann dadurch die von Montague verwendeten Terme für Nomen, Verben, Präpositionen, Adverbien, Nominalphrasen und Präpositionalphrasen nutzen.

2.2.2. Ein Fragment des Deutschen

Das Fragment des Deutschen, welches in NASfVI implementiert ist, behandelt ausschließlich den universitären Kontext von Vorlesungsverzeichnissen und in diesem Zusammenhang Aussagen und Fragen. Das Lexikon umfasst daher Nomen wie Kurs, Veranstaltung, Vorlesung, Seminar, Proseminar und Hauptseminar, um verschiedene Veranstaltungen zu bezeichnen. Aber auch Bezeichnungen für Personen wie Dozent, Dozentin, Professor, Professorin, Herr und Frau sind Teil des Fragements. Diese können entweder als Appositionen in Anreden[7] oder als Gattungsnamen verwendet werden. Weitere Nomen

[7] Zum Appositionsbegriff in NASfVI siehe Kapitel 3.1.1

im Fragment sind Raum für Raumangaben, sowie Semester, Sommersemester und Wintersemester für Semesterangaben und die Wochentage.[8] Die Nomen können mit einem definitem, indefinitem oder interrogativem Artikel zusammen Nominalphrasen bilden. Im Fall von Veranstaltungstiteln, Personennamen oder Raumangaben schließt sich auch eine Blackbox für die jeweilige Angabe an. Eine Blackbox ist in NASfVI ein Bereich, der in seinen Bestandteilen nicht morphosyntaktisch oder lexikalisch analysiert wird. Aus diesem Grund unterliegt ein als Blackbox analysierter Bereich der Beschränkung, dass er nur aus exakt einem Token bestehen darf oder alternativ mit Anführungszeichen der Anfang und das Ende der Blackbox markiert sein muss (siehe Kapitel 3.2.2 und Anhang A.2). Veranstaltungstitel können zudem in Form einer Blackbox auch direkt Nominalphrasen bilden. Beispiele für mögliche Nominalphrasen sind also *Syntax, der Dozent, ein Seminar, welcher Raum, Professor Schulz, der Raum 1.14*.
Die Verben im modellierten Fragement sind stattfinden, halten, handeln, geben und ähneln, sowie die Hilfsverben haben, werden und sein. Das Verb halten ist im behandelten Fragment des Deutschen nur im Sinn von *eine Veranstaltung halten* modelliert und das Verb handeln nur in der Bedeutung *von etwas handeln*, wie in *„Welche Vorlesung handelt von Syntax“*. Mehrere Lesarten stehen dagegen von dem Verb geben zur Verfügung. Es kann einerseits die Existenz eines Akkusativobjekts erfragen wie in *„Gibt es eine Vorlesung über Syntax“*, es kann aber auch als Synonym für halten wie in *„Welcher Dozent gibt eine Vorlesung über Syntax“* verwendet werden. Eine besondere Verarbeitung geht mit dem Verb ähneln einher, welche es ermöglicht, Veranstaltungen zu erfragen, die einander ähneln. Die dafür notwendige Verarbeitung wird in Kapitel 4.2.5 dargestellt. Im modellierten Sprachfragment kann ähneln nur transitiv mit zwei Veranstaltungen verwendet werden, wie in *„Welche Vorlesung ähnelt dem Proseminar Syntax“*.
Die Hilfsverben haben, werden und sein ermöglichen die Bildung der zusammengesetzten Zeitformen im Plusquamperfekt, Perfekt und Futur 1, welche neben den einfachen Zeiten Präsens und Präteritum im Fragment zur Verfügung stehen, sowie die Bildung des Passivs. Die Zeitformen werden in der Auswertung von Anfragen berücksichtigt und auf Semester abgeblidet. Wenn keine explizite Semesterangabe vorhanden ist, führt eine Anfrage im Präsens so zur impliziten Annahme des aktuellen Semesters, während Anfragen in einer Zeit der Vergangenheit verschieden abgestuft vorangegangene Semester miteinbeziehen (Kapitel 4.2.4). In allen Zeitformen können sowohl die Aktiv- als auch die Passiv-Formen des Verbs gebildet werden. Alle Verbformen sind stets im Indikativ, da Anfragen der Form *„Würde jemand eine Vorlesung halten“* aufgrund der Daten in Vorlesungsverzeichnissen nicht angemessen beantwortet werden können.
Zur Formulierung von Fragen stehen die interrogativen Pronomen wer und was, sowie die Adverbien wo und wann zur Vefügung. Die Indefinitpronomen etwas und jemand werden innerhalb des Sprachfragments ebenfalls interrogativ modelliert. Dadurch sind zum Beispiel Fragen wie *„Hält jemand eine Vorlesung“* und *„Wer hält eine Vorlesung“*, sowie die Fragen *„Wer hält etwas über Syntax“* und *„Wer hält was über Syntax“* gleichbedeutend. Dies stellt eine einfache Realisierung der mit diesen Fragen verbundenen Pragmatik dar. NASfVI setzt in Antworten nur Daten ein, die interrogativ erfragt wurden. So erfragt

[8] Zur Problematik von Uhrzeitangaben siehe Kapitel 2.3.1

Wer in der Frage *„Wer hält eine Vorlesung über Syntax“* den Namen des Dozenten. Durch die interrogative Interpretation von *jemand* fragt *„Hält jemand eine Vorlesung über Syntax“* nach demselben Namen und erfüllt dadurch die Erwartungshaltung des Benutzers, dass das System mit einem Namen antwortet. Dass als Nebeneffekt eine Aussage wie *„Jemand hält eine Vorlesung über Syntax“* ebenfalls als Frage nach dem Namen des Dozenten interpretiert wird, sollte kein Problem darstellen. Denn auch von einem menschlichen Gesprächspartner würde man keine einfache Ja/Nein-Antwort auf diese Aussage erwarten, sondern ebenfalls die Information, wer die Vorlesung hält.
Mit dem Adverb wann ist eine Besonderheit verbunden: Ist in der Anfrage eine Phrase vorhanden, die eine Semesterangabe enthält, oder ist das Tempus der finiten Verbform Präsens, erfragt wann den Wochentag. In allen anderen Fällen bezieht sich wann immer auf eine Frage nach dem Semester. Das Fragment umfasst zudem die Adverbien montags, dienstags, mittwochs, donnerstags, freitags, samstags und sonntags. Alle Adverben des modellierten Fragments können eigenständig Adverbialphrasen ausmachen.
Präpositionalphrasen werden im Fragment mit den Präpositionen von, über, in, im und am gebildet und beinhalten neben einer Präposition immer auch ein Nomen, eine Blackbox oder eine Nominalphrase. Mit den Präpositionalphrasen werden u. a. Themenangaben, Raumangaben, Wochtentags- und Semesterangaben gebildet. Themenangaben werden mit der Präposition über und einer Blackbox gebildet. Beispiele für Themenangaben sind demnach *über Syntax* und *über “Lokale Grammatiken”*. Raumangaben werden mit der definiten Präposition im, dem Nomen Raum und einer Blackbox gebildet oder mit der Präposition in und einer Nominalphrase: *im Raum 1.14* oder *in dem Raum 1.14*. Wochentagsangaben werden durch die definite Präposition am und dem Nomen für den Wochentag gebildet: *am Montag*. Semesterangaben werden mit der definiten Präpostion im oder der Präposition in und einem definiten Artikel, einem Nomen für eine Semesterangabe und einer zwei oder vierstelligen Jahreszahl für Sommersemester und zwei Jahreszahlen mit zwei oder vier Stellen, die durch / getrennt sind, für Wintersemester gebildet. Allgemeinere Präpositionalphrasen sind mit einer Präposition und einer Nominalphrase möglich. Die Nominalphrase kann dabei sowohl definit, als auch indefinit oder interrogativ sein. Dadurch sind Präpositionalphrasen wie *von dem Dozenten, von Herrn Mueller* und *in welchem Semester* möglich.
Das in NASfVI modellierte Fragment des Deutschen umfasst auch die Koordination von Nominal- und Präpositionalphrasen. Implementiert sind die Junktoren *und* und *oder*. Die koordinierten Phrasen können jedoch nur denselben semantischen Typ beinhalten (siehe Kapitel 3.2.7). Eine Phrase wie *eine Vorlesung und ein Dozent* ist also nicht möglich, da die Vorlesung einen anderen Typ als der Dozent hat. Eine mögliche Nominalphrase im Plural ist dagegen zum Beispiel *ein Hauptseminar und eine Vorlesung* oder im Singular *Herr Schulz oder welcher Dozent*. Analog werden Präpositionalphrasen koordiniert: *in dem Sommersemester 2007 und im Wintersemester 2007/08* und *im Raum 1.14 oder im Raum 0.1*. Doch auch die Definitheit der Phrasen muss identisch sein, damit sie koordiniert werden können. Als Ausnahme ist es allerdings möglich, dass nur eine der koordinierten Phrasen interrogativ ist. In diesem Fall wird die koordinierte Gesamtphrase ebenfalls interrogativ. Die koordinierte Nominalphrase *Herr Schulz und welcher Dozent* ist also interrogativ. Bei der Koordination von Nominalphrasen ist es zudem möglich mit

einem Artikel und einem Nomen im Plural zu beginnen und dann mit der Junktion *und* Blackbox-Elemente zu koordinieren: *Die Dozenten Schulz und Leiß*. Bei allen Koordinationstypen kann die Grammatik grundsätzlich unendlich viele Phrasen anhängen. Die Koordinationen sind also nicht auf eine bestimmte Anzahl von Phrasen beschränkt.
Als Satzmuster beherrscht die sprachverarbeitende Komponente Verberst- und Verbzweitsätze.[9] Mögliche vollständige Sätze sind daher zum Beispiel:

- Wer hat im Sommersemester 2008 oder im Sommersemester 2007 über Syntax ein Seminar gehalten?
- Fand ein Seminar über Syntax im Raum 1.14 statt?

Von transitiven Verben kann in NASfVI das Passiv gebildet werden. Dabei ist die Agensphrase stets fakultativ. Sie kann also, muss aber nicht realisiert werden. Dies ermöglicht Sätze der Form *„Eine Vorlesung wird von einem Professor gehalten“* genau so wie Sätze der Form *„Eine Vorlesung wird gehalten“*.

2.2.3. Aufbau der Komponente

Die sprachverarbeitende Komponente verfügt über drei Anfragemöglichkeiten. Bei *Parse*-Anfragen wird ein Satz analysiert, bei *Suggest*-Anfragen werden alle generierbaren Sätze erzeugt, die eine Anfrage als Präfix beinhalten, und bei *Beantworte*-Anfragen wird eine natürlichsprachige Antwort auf eine Anfrage generiert. Diese Anfragemöglichkeiten stellen eine Schnittstelle der Sprachverarbeitungskomponente nach Außen dar. Sie werden in Kapitel 3.5 näher behandelt. Intern kann die Sprachverarbeitungskomponente dagegen anhand der funktionalen Aspekte Lexikon, Syntax-Verarbeitung, optimalitätstheoretische Evaluation und Semantik-Verarbeitung strukturiert werden. Diese Teile der internen Struktur werden im Folgenden skizziert.

Das Lexikon

Der zentrale Bereich der Sprachverarbeitung, welcher die Wortformen für die restlichen Bereiche bereitstellt, ist das Lexikon. Es umfasst alle Wortformen des modellierten Sprachfragments und ermöglicht den Zugriff auf die Vollformen der einzelnen Lexeme. Für jede flektierte Form eines Lexems existiert somit ein eigener Eintrag im Lexikon. Um die Suggest-Funktion in NASfVI zu ermöglichen, muss das Lexikon nicht nur Anfragen mit vollständigen Wortformen verarbeiten, sondern zu gegebenen Präfixen auch Vollformen ermitteln können. Soll zum Beispiel das Präfix „sem“ ergänzt werden, muss das Lexikon alle passenden Vollformen von „Semester“ und „Seminar“ verfügbar machen. Gleichzeitig muss es jedoch auch erkennen, ob ein Präfix nicht ergänzt werden soll. Das ist immer dann der Fall, wenn das Präfix bereits einer vollständigen Vollform im Lexikon entspricht. Durch diese Bedingung sollen nur dann Ergänzungen vom Lexikon gesucht werden, wenn dies notwendig ist. Damit das Lexikon möglichst wenig Zeit für den Zugriff

[9] Verberst- und Verbzweitsätze sind im Abschnitt über das Feldermodell des Deutschen in Kapitel 2.2.1 erläutert.

auf die Vollformen benötigt, ist es unabdingbar, dass die Vollformen bereits in vorausberechneter Form vorliegen und nur ausgelesen werden müssen. Ein solches Lexikon ist für Menschen jedoch unhandlich und schwer zu pflegen. Deswegen ist der Ausgangspunkt des NASfVI-Lexikons die Grundform der Wörter - die Zitierform. Bei flektierbaren Wortarten ist das im Deutschen für Nomen und Artikel der Nominativ Singular und für Verben der Infinitiv. Bei nicht flektierbaren Wortarten wie Präpositionen und Adverbien existiert keine gesonderte Zitierform, da es nur eine Wortform gibt. Aus diesen Grundformen kann das Lexikon die Vollformen erzeugen und abspeichern, um schneller auf die Vollformen zugreifen zu können. Das Lexikon ist damit drei-schichtig und besteht aus:

- der obersten Schicht, welche Präfixerkennungen durchführen kann und einen Zugriff auf die Vollformen anbietet;
- den Vollformen, welche auf den Grundformen aufbauen;
- und den Grundformen mit den Grunddaten als Ausgangspunkt;

Jeder Grundformeneintrag enthält alle modellierten Informationen über das entsprechende Lexem: dessen Wortart, Grundform, Artmerkmale wie Genus oder Definitheit, Angaben zur Flexion, um aus der Grundform die Vollformen zu erzeugen, den semantischen Typ und die logischen Terme der Semantik, sowie im Falle von Nomen und Verben die syntaktisch-semantische Valenz. Mit Hilfe der Flexionsangaben werden im Lexikon die Vollformen erzeugt und statt dieser Flexionsangaben in den Einträgen der Vollformen die morphosyntaktischen Merkmale der Vollform vermerkt. Die morphosyntaktischen Merkmale der Vollform (die Formmerkmale) sind zum Beispiel im Fall von Nomen Numerus und Kasus und im Fall von Verben Verbstellung,[10] Person, Numerus und Tempus. Mit Ausnahme der Flexionseigenschaften werden alle anderen Informationen des Grundformeintrags auf die Vollformen übertragen und stehen damit in der Syntax und Semantik zur Verfügung.

Syntax- und Evaluationskomponente

Bei der Verarbeitung der Syntax lassen sich die Ebenen der Phrasen, die Ebene der Felder und die Satzebene unterscheiden. Auf der Phrasenebene werden die in Kapitel 2.2.2 dargestellten Phrasen mit Definite Clause Grammars implementiert. Auf der Felderebene werden mit demselben Formalismus die in Kapitel 2.2.1 dargelegten Felder des Feldermodells umgesetzt und auf Satzebene die Analyse von Verberst- und Verbzweitsätzen, sowie das Generieren von Verbzweitsätzen im Rahmen von *Beantworte*-Anfragen. Bei den einzelnen Verarbeitungsschritten greift die Syntax-Komponente auf das Lexikon zu.
Der Syntax nachgeschaltet ist eine Evaluationskomponente, welche das im Kapitel 2.2.1 skizzierte optimalitätstheoretische Verfahren anwendet, um die Markiertheit der in der Syntax verarbeiteten Sätze zu berechnen.

[10] Die Verbstellung muss bei der Vollform vermerkt sein, da sie bei Verben mit Partikeln eine Rolle spielt. So lautet die Vollform von „stattfinden“ für dritte Person, Singular, Präsens bei Verberst- und Verbzweitstellung „findet statt“, bei Verbeletztstellung dagegen „stattfindet“. Zwar ist die Verbletztstellung nicht im modellierten Sprachfragment vorhanden (Kapitel 2.2.2). Doch da das Lexikon sie berücksichtigt, kann die Syntax jederzeit entsprechend erweitert werden.

Semantik-Formalismen

Zur Berechnung der Bedeutung von Anfragen verwendet NASfVI zwei verschiedene Formalismen. Zum einen werden die in Kapitel 2.2.1 erwähnten Terme nach Montague verwendet und zum anderen die Anfragesprache der Volltextsuche. Die Terme nach Montague entstammen dabei den Grundformeinträgen des Lexikons, welche natürlichsprachige Bedeutungen formal-logisch modellieren. Die Kompositionalität dieser semantischen Formeln wird mit Hilfe des Lambda-Kalküls umgesetzt. Zur Beantwortung von Anfragen in NASfVI müssen jedoch Daten in einem separaten Suchsystem angefragt werden. Die bisher diskutierten semantischen Formeln können durch dieses Suchsystem allerdings nicht verarbeitet werden. Aus diesem Grund müssen die logischen Formeln der Semantik in die Anfragesprache des Suchsystems übersetzt werden. Nur so lässt sich das Suchsystem und damit dessen Fähigkeiten Texte zu durchsuchen und gefundene Dokumente nach Relevanz für die Anfrage zu ordnen nutzen. Aus diesem Grund verfügt die Semantik-Komponente sowohl über die Fähigkeit, die logischen λ-Terme zu normalisieren, als auch über die Fähigkeit, diese normalisierten Formeln in die Anfragesprache des Suchsystems zu übersetzen. Grundsätzlich wäre es möglich, den Lambda-Kalkül zur direkten Berechnung der Anfragen für die Suche zu benutzen, und damit den Zwischenschritt über die Prädikatenlogik einzusparen. Doch wäre dadurch das Lexikon, in welchem die Semantik der Lexeme festgehalten ist, direkt von der Wahl des Suchsystems beeinflusst. Durch den Zwischenschritt über die Prädikatenlogik ist eine zusätzliche Abstraktionsstufe vorhanden, die die Verwendung des Lexikons und der Syntax unverändert auch mit anderen Suchsystemen erlaubt. Durch diese Abstraktion wird die Wiederverwendbarkeit der Komponenten also erhöht. Dies führt zu einer zwei-stufigen Semantik-Komponente: die erste Stufe normalisiert λ-Terme und die zweite Stufe übersetzt die normalisierten Terme in die Anfragesprache des verwendeten Suchsystems.

2.3. Der Server

Die Aufgabe des Server ist es, die Suchkomponente und damit die Vorlesungsdaten zu verwalten, eine Schnittstelle zum Gesamtsystem nach Außen anzubieten und mit Hilfe der sprachverarbeitenden Komponente Anfragen zu analysieren, zu bearbeiten und zu beantworten.

2.3.1. Die Volltextsuche

Die Volltextsuche ist eine zentrale Komponente des Servers. Ihre Aufgabe ist es, Textdaten nach Stichwörtern durchsuchbar zu machen, und Veranstaltungsdaten in Form von Dokumenten zu verwalten. Bei Anfragen ermittelt die Volltextsuche zudem eine Liste der Veranstaltungen, welche nach Relevanz für die Anfrage geordnet sind. Die Dokumente haben eine feste Struktur mit Feldern, die bestimmte Daten aufnehmen. So nimmt ein Feld zum Beispiel den Veranstaltungstitel auf, ein anderes Feld die Namen der Dozenten, das nächste Feld das Semester in dem die Veranstaltung gehalten wird und so weiter. Eine derartige Felderstruktur ist notwendig, damit die verschiedenen Informa-

tionen zuverlässig voneinander unterschieden werden können. Das Suchsystem soll nicht einen zusammenhängenden Text durchsuchen, sondern Dokumente mit bekannter Struktur und innerhalb dieser Struktur Felder mit textuellem Inhalt. Ein Suchsystem, das diese Unterscheidung nach Feldern nicht treffen kann, wäre für die natürlichsprachige Anwendung in NASfVI von geringem Nutzen, da zum Beispiel Namen von Dozenten, die eine Veranstaltung halten, nicht zuverlässig von Veranstaltungstiteln oder Veranstaltungsbeschreibungen unterschieden werden könnten. In Abbildung 2.1 wird ein Dokument des Suchsystems in XML-Notation [XML10] dargestellt. Zu beachten ist, dass Felder mehrfach belegt sein können, was hier am Beispiel des Felds <dozent> gezeigt ist. Dies ist eine notwendige Anforderung, damit eindeutig unterscheidbar ist, ob mehrere Dozenten eine Veranstaltung halten oder eine Veranstaltung zu mehreren Terminen stattfindet.

```
<veranstaltung>
<semester>2009</semester>
<titel>Spezialsuchmaschinen</titel>
<dozent>F. Guenthner</dozent>
<dozent>G. Rolletschek</dozent>
<typ>Hauptseminar</typ>
<termin>Mi 16-18 Uhr Raum 1.13</termin>
<beschreibung>Im Gegensatz zu sog. Universalsuchmaschinen, die ganz
    unterschiedliche Dokumente indexieren, haben Spezialsuchmaschinen
    den Vorteil, dass sie die Semantik der jeweiligen Anwendungen
    besser ausnuetzen koennen. Beispiele fuer Spezialsuchmaschinen
    sind Jobsuchmaschinen, Reisesuchmaschinen und Buchsuchmaschinen.
    In diesem Seminar werden wir die Architektur von
    Spezialsuchmaschinen anhand einiger sich in der Entstehung
    befindender Suchmaschinen analysieren. Neben semantischen Fragen
    werden auch andere linguistische Aspekte sowie diverse Ansaetze
    zur Suche in strukturierten und semi-strukturierten Daten
    vorgestellt und verglichen.</beschreibung>
</veranstaltung>
```

Abbildung 2.1.: Ein Dokument des Suchsystems als XML-Datensatz dargestellt

Ein freies Suchsystem, das diese Anforderungen erfüllt, ist Lucene.[11] Bei Lucene handelt es sich um eine in Java[12] geschriebene Bibliothek zur Volltextsuche, die sich in andere Java-Programme einbinden lässt. Entwickelt wird Lucene von der Apache Software Foundation.[13] Lucene ist eine sehr leistungsfähige Bibliothek, die nicht nur einfache Stichwortsuchen ermöglicht, sondern neben Tokenisierung auch das Filtern von Stoppwörtern, Stemming und Kompositazerlegung unterstützt.
Die Anfragesprache, die Lucene verwendet, erlaubt eine explizite Suche in Feldern, die Kombination von Termen mit booleschen Operatoren und Gruppierungen mit Klammern [LUCQ]. Eine Anfrage in dieser sogenannten „Query Syntax“ von Lucene sieht zum

[11] http://lucene.apache.org/
[12] http://java.sun.com/
[13] http://www.apache.org/

Beispiel wie folgt aus:

```
typ:"seminar" AND (titel:"syntax" OR beschreibung:"syntax")
```

Diese Anfrage ermittelt alle Dokumente, bei denen im Feld „typ“ das Token „seminar“ vorkommt und die im Feld „titel“ oder im Feld „beschreibung“ das Token „syntax“ beinhalten.
Der Ansatz mit einer Volltextsuche hat jedoch nicht nur Vorteile, sondern auch einen Nachteil. Volltextsuchen sind auf die lexikographische, zeichenbasierte Verarbeitung von Token spezialisiert und die Felder, über die gesucht wird, lassen sich nicht ohne Weiteres verschachteln. Das heißt, einem Terminfeld können keine Felder „Tag“ oder „Uhrzeit“ untergeordnet werden. Aus diesem Grund sind Tages-, Raum- und Uhrzeitangaben in einem Feld <termin> zusammengefasst. So lässt sich die Zusammengehörigkeit dieser Angaben auch dann noch gewährleisten, wenn mehrere Zeitangaben vorhanden sind. Bei Tages- und Raumangaben ist dies keine zu große Einschränkung, da diese Angaben auch als Token direkt gesucht werden können. Bei Uhrzeitangaben dagegen sollte die Uhrzeit nicht nur als eigenes Feld vorhanden sein, sondern auch numerisch ausgewertet werden. Eine numerische Auswertung liegt jedoch außerhalb des Leistungsspektrums von Lucene. In NASfVI wird dennoch der Ansatz der Volltextsuche verfolgt. Denn für die Recherche nach Veranstaltungen wird die Bedeutung der Durchsuchbarkeit von Veranstaltungstiteln und -beschreibungen hier als wichtiger eingeschätzt.

2.3.2. Aufbau des Servers

Für die Funktionalität des Servers ist eine Kommunikation mit der Volltextsuche und der Sprachverarbeitungskomponente zentral. Die Volltextsuche liegt dabei als Java-Bibliothek vor. Die sprachverarbeitende Komponente dagegen ist in Prolog geschrieben. Es gibt verschiedene Bibliotheken, die eine Kommunikation zwischen Java und Prolog ermöglichen. NASfVI verwendet SWI-Prolog,[14] welches mit „jpl“ eine solche Bibliothek bereitstellt. Sie ermöglicht es von Java aus Prolog-Ziele aufzurufen und umgekehrt. Prologs Terme, Atome, Prädikate und Variabeln, können über die Sprachgrenze hinweg genutzt werden. Der in ISO-Prolog definierte throw-catch-Mechanismus, mit dem in Prolog Exceptions definiert sind (Abschnitt 7.8.9f in [ISO13211-1]), wird ebenfalls unterstützt, so dass Prolog-Exceptions in Java-Exceptions übersetzt werden. Damit lassen sich Prolog-Aufrufe bequem in Java integrieren.
Da die Volltextsuche in Java geschrieben ist und Prolog in Java eingebunden werden kann, ist es nur konsequent, den Server ebenfalls in Java zu schreiben. Denn in Java gibt es zudem mit den sogenannten Servlets [JST] einen Standard, der es ermöglicht, „Web Applications“ zu schreiben. Dies sind Anwendungen, die innerhalb eines „Servlet-Containers“ ausgeführt werden. Servlet-Container können entweder als eigenständige Web-Server agieren oder in andere Web-Server eingebettet werden. Eine Servlet-Klasse hat definierte Schnittstellen für die Bearbeitung von z. B. Anfragen über HTTP. Servlets werden vom Servlet-Container gestartet und warten anschließend auf Anfragen. Wird ein

[14] `http://www.swi-prolog.org/`

Servlet mit einer Anfrage aufgerufen, bearbeitet es die Anfrage, schickt das Ergebnis an den aufrufenden Client zurück und wartet anschließend auf neue Anfragen. Der Server von NASfVI beinhaltet zwei Servlets als grundlegende Schnittstelle nach Außen:

- ein Servlet, das Suggest-Anfragen bearbeitet (siehe Kapitel 4.3.2), und
- ein Servlet, das Anfragen analysiert, die Volltextsuche aufruft und eine Antwort zurückliefert (siehe Kapitel 4.3.3)

Diese Servlets sind zu einer „Web Application“ zusammengefasst. Dadurch können sie sich dieselbe sprachverarbeitende Komponente und dieselbe Instanz der Volltextsuche teilen und so die Geschwindigkeit erhöhen, da diese Ressourcen nun nicht bei jedem Servletaufruf neu erzeugt werden müssen. Anders als zum Beispiel bei CGI[15] wird ein Servlet nur einmal initialisiert und erzeugt bei neuen Anfragen lediglich Threads und keine neuen Prozesse.

2.3.3. Schnittstellen

Die Servlets des Servers stellen die grundlegende Schnittstelle für die Kommunikation nach Außen bereit. Anfragen werden über HTTP empfangen und beantwortet. Das Format der Antworten ist in allen Fällen JSON.[16] Bei JSON, kurz für JavaScript Object Notation, handelt es sich um ein Datenaustauschsformat bei dem die Daten in JavaScript-Syntax beschrieben werden. Es hat weniger Redundanz als XML und lässt sich als Untermenge von JavaScript direkt in Browsern interpretieren. JSON ist trotz des Namens jedoch nicht auf die Verwendung mit JavaScript beschränkt, sondern kann als einfaches Textformat mit allen Programmiersprachen genutzt werden. Weite Verbreitung findet JSON im sogenannten „Web 2.0“ als Alternative zu XML. Mit dem Begriff des „Web 2.0“ sind Techniken und Technologien verbunden, die es ermöglichen, Webseiten vergleichbar mit lokalen Programmen zu bedienen. Eine zentrale Technik dafür ist AJAX.[17] AJAX ermöglicht es, dass Webseiten Daten bei Webservern anfragen können, ohne dass die betreffende Webseite neu geladen werden müsste. Es ist also eine Kommunikation mit dem Webserver im Hintergrund möglich, während der Benutzer mit der Webseite arbeitet. Diese Technik wird auch in NASfVI verwendet. Die Schnittstellen des Servers kommunizieren die Ergebnisse von Anfragen im JSON-Format, welches auf Seite des Clients erst ausgewertet und dargestellt werden muss. Eine solche Kommunikation mit dem Server im Hintergrund ist eine zentrale Anforderung in NASfVI, um die Suggest-Funktionalität umsetzen zu können.

OpenSearch-Schnittstelle

OpenSearch ist ein von der Amazon.com-Tochter A9.com entwickeltes Format, um die Dienste von Suchmaschinen in maschinenlesbarer Form zu publizieren und damit für Anwendungen von Drittanbietern direkt nutzbar zu machen. Zu diesem Zweck spezifiziert

[15] Siehe das RFC 3875, „The Common Gateway Interface (CGI) Version 1.1“
[16] Siehe `http://www.json.org/`
[17] AJAX steht für **A**synchronous **J**avaScript **a**nd **X**ML.

OpenSearch ein XML-Format [OPN11]. Diese XML-Datei enthält zahlreiche Angaben, die den jeweiligen Suchdienst beschreiben. Darunter fallen insbesondere Angaben zu dem Namen des Dienstes, verwendeten Zeichensätzen und Internetadressen (URLs) mit denen der Dienst genutzt werden kann. Damit diese URLs dynamisch zusammengesetzt werden können, definiert OpenSearch Platzhalter für Suchbegriffe. Ein Computerprogramm, dass diese Informationen ausliest, ist damit in der Lage, Anfragen an die beschriebene Suchmaschine automatisch zu stellen. NASfVI verfügt über eine solche Selbstbeschreibung nach der OpenSearch-Spezifikation.

```
<?xml version="1.0" encoding="UTF-8"?>
<OpenSearchDescription xmlns="http://a9.com/-/spec/opensearch/1.1/">
        <ShortName>NASfVI</ShortName>
        <LongName>Natürlichsprachiges Anfragesystem für
            Vorlesungsverzeichnisse im Internet</LongName>
        <Description>Mit NASfVI können verschiedene Anfragen in natü
            rlichem Deutsch an ein Vorlesungsverzeichnis gestellt
            werden.</Description>
        <Developer>Stefan Partusch</Developer>
        <Language>de-DE</Language>
        <OutputEncoding>UTF-8</OutputEncoding>
        <InputEncoding>UTF-8</InputEncoding>
        <Image height="16" width="16" type="image/x-icon">http://
            localhost:8080/favicon.ico</Image>
        <Url type="text/html" template="http://localhost:8080/#{
            searchTerms}&amp;0"/>
        <Url type="application/x-suggestions+json" template="http://
            localhost:8080/suggest?q={searchTerms}"/>
        <SyndicationRight>limited</SyndicationRight>
        <AdultContent>false</AdultContent>
</OpenSearchDescription>
```

Abbildung 2.2.: Eine mögliche OpenSearch-Beschreibungsdatei für NASfVI

OpenSearch wird von verschiedenen Anwendungen direkt unterstützt. So implementieren zum Beispiel die Browser Windows Internet Explorer,[18] Mozilla Firefox[19] und Google Chrome[20] die OpenSearch-Spezifikation. Wird mit einem dieser Browser die Webseite einer Suchmaschine aufgerufen, die eine OpenSearch-Beschreibungsdatei bereitstellt, kann der Benutzer diese Suchmaschine fest im Browser installieren und von nun an direkt mit dem Suchfeld des jeweiligen Browsers Anfragen an die Suchmaschine schicken.
In aktuellen Versionen der Browser wird außerdem die OpenSearch-Erweiterung „Suggestions“ unterstützt. Diese Erweiterung ermöglicht die zusätzliche Einbindung einer Suggest-Funktionalität. Dies erfolgt über die Angabe einer URL mit dem Typ „application/x-suggestions+json“ (siehe Abbildung 2.2). Der Typ deutet bereits an, dass die

[18] http://www.microsoft.com/windows/internet-explorer/
[19] http://www.mozilla-europe.org/de/
[20] http://www.google.com/chrome/

OpenSearch-Erweiterung ein JSON-Format für diesen Zweck definiert. Eine Suchmaschine, die es implementiert, muss auf Suggest-Anfragen in JSON-Notation mit einem Array antworten, dessen erstes Element die Suchanfrage wiederholt und dessen zweites Element ein Array mit Suchvorschlägen ist [OPS11]:

> ["wer hält eine vorles", ["Wer hält eine Vorlesung", "Wer hält eine Vorlesung (Titel)", "Wer hält eine Vorlesung (Titel) und (Titel)", "Wer hält eine Vorlesung (Titel) über (Thema)", "Wer hält eine Vorlesung im (Semesterangabe)", "Wer hält eine Vorlesung im (Semesterangabe) über (Thema)", "Wer hält eine Vorlesung und (Titel)", "Wer hält eine Vorlesung und (Titel) im (Semesterangabe)", "Wer hält eine Vorlesung und (Titel) über (Thema)", "Wer hält eine Vorlesung über (Thema)", "Wer hält eine Vorlesung über (Thema) im (Semesterangabe)"]]

Es bietet sich an, dieses Format im „Suggestlet“ von NASfVI zu verwenden und damit eine OpenSearch-Schnittstelle zu ermöglichen. Programme, die die OpenSearch-Spezifikationen unterstützten, können dadurch mit der entsprechenden Beschreibungsdatei teilweise als ein Client für NASfVI verwendet werden. Die Abbildungen 2.3 auf Seite 28 und 2.4 auf Seite 29 zeigen die Suggest-Funktionalität von NASfVI mit verschiedenen Browsern. OpenSearch-Clients von NASfVI sind jedoch dahingehend eingeschränkt, dass sie nur die Anfragevorschläge anzeigen und zum im folgenden Abschnitt erläuterten NASfVI-Client weiterleiten können. Wird also im Suchfeld eines unterstützenden Browsers eine natürlichsprachige Anfrage an NASfVI geschickt, wird der NASfVI-Client geladen und die Anfrage dort in natürlicher Sprache beantwortet.

Schnittstelle zum Client

Der NASfVI-Client ist eine Webseite, die mit dem NASfVI-Server kommuniziert, wie in Kapitel 2.4 gezeigt wird. Der Client nutzt dabei nicht nur die im vorangegangenen Abschnitt beschriebene OpenSearch-Schnittstelle für die Suggest-Funktion („Suggestlet“), sondern auch eine für diesen Client spezifische Schnittstelle (das „Parselet“), über die der Server dem Client die natürlichsprachigen Antworten zur Verfügung stellt. Auch diese Schnittstelle verwendet wie die OpenSearch-Schnittstelle JSON. Durch die einheitliche Verwendung von JSON wird die Verarbeitung beider Schnittstellen im Client vereinfacht. Im Vergleich zur OpenSearch-Schnittstelle für die Suggest-Funktion transportiert diese Client-spezifische Schnittstelle deutlich mehr Informationen. Dieses umfangreichere Antwortformat des Servers findet sich in Kapitel 4.3.3 dargestellt.

2.4. Der Client

2.4.1. Die graphische Oberfläche

Der Client stellt die graphische Oberfläche von NASfVI dar und erlaubt den Zugriff auf die gesamte Funktionalität und alle Informationen, die der Server über seine Schnittstellen dem Client zur Verfügung stellt. Die zentralen Elemente sind ein Eingabebereich für

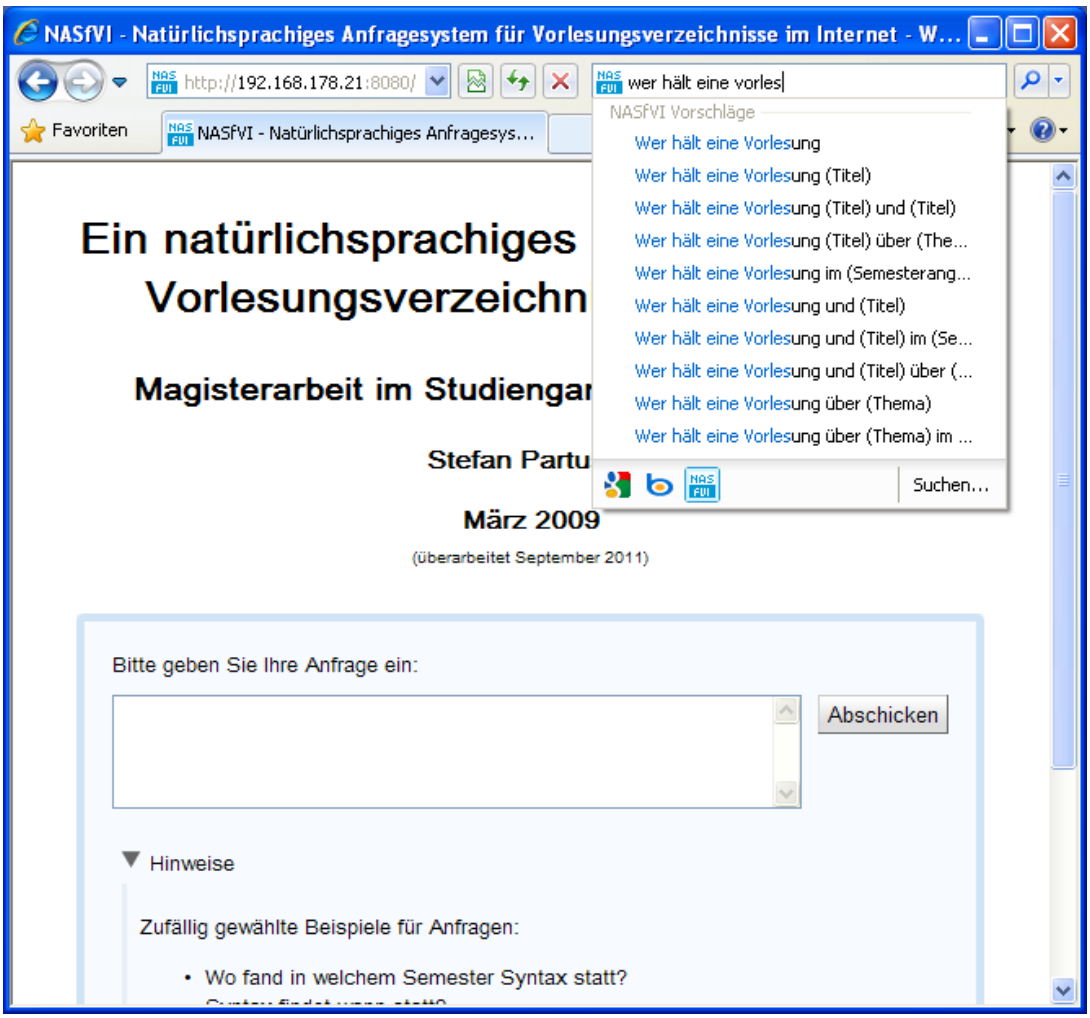


Abbildung 2.3.: Durch die Implementierung der Suggestions-Erweiterung von OpenSearch kann auch die Vorschlagsfunktion in vielen Browsern direkt verwendet werden. Diese Abbildung zeigt die Vorschlagsfunktion, wie sie in dem Browser Windows Internet Explorer 8 (unter Windows XP) verwendet wird. Im Hintergrund ist als geöffnete Webseite der normale NASfVI-Client zu sehen.

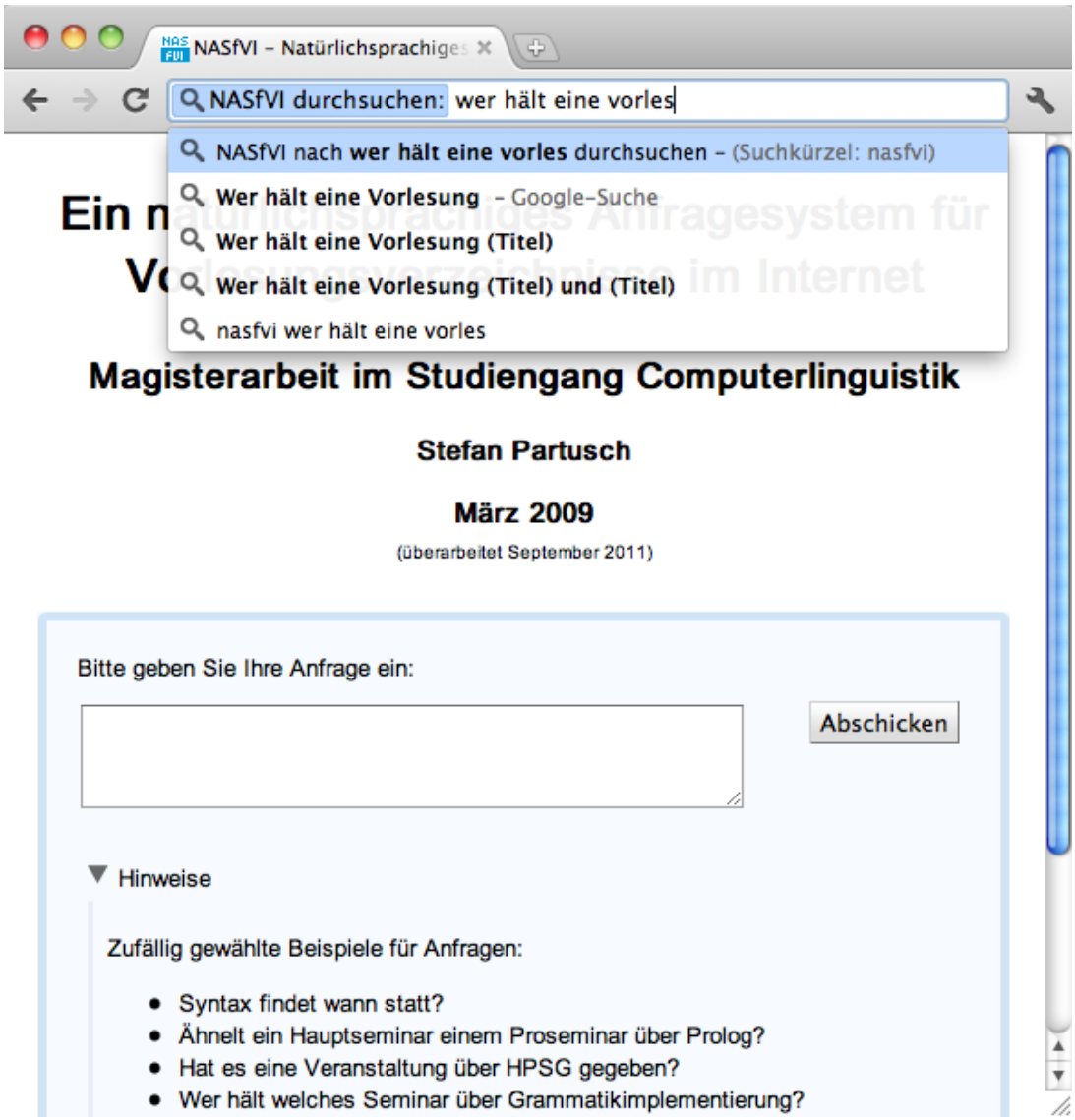
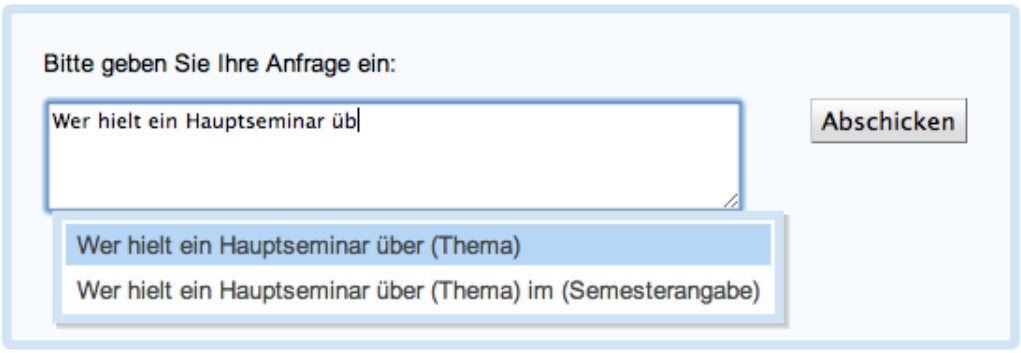


Abbildung 2.4.: Diese Abbildung zeigt die Vorschlagsfunktion, wie sie in dem Browser Google Chrome 13 (unter Mac OS X 10.6) verwendet wird. Google Chrome zeigt nur die ersten drei Vorschläge unterhalb des Eingabefelds an. Im Hintergrund ist auch hier der normale NASfVI-Client als geöffnete Webseite zu sehen.

die Anfrage und ein Ausgabebereich für die natürlichsprachige Antwort des Servers. Bei der Eingabe einer Anfrage muss der Client im Hintergrund mit dem Server kommunizieren und die so erhaltenen Vorschläge in der Oberfläche einblenden, ohne die Eingabe der Anfrage zu behindern. Die Vorschläge sind kein Zwang, sondern sollen den Benutzer unauffällig an das in Kapitel 2.2.2 beschriebene Sprachfragment heranführen. Trotz der Informationsfülle, die der Server mitteilt, soll die Oberfläche übersichtlich und elegant gestaltet sein. Das Verhalten des Clients bei der Eingabe sei an folgendem Bild demonstriert:



Der Benutzer gibt gerade die Anfrage „Wer hielt ein Hauptseminar üb“ ein. Sobald der Benutzer kurz bei der Eingabe pausiert,[21] übermittelt der Client im Hintergrund die bisherige Eingabe an den Server und blendet daraufhin dessen Vorschläge von passenden vollständigen Anfragen unterhalb des Eingabgefeldes ein. Der Benutzer kann nun einen der Vorschläge auswählen, die durch Klammern markierten Platzhalter ersetzen und die Abfrage abschicken. Er kann aber auch mit der Eingabe fortfahren und so eine andere Anfrage formulieren. Denn durch den in Kapitel 2.2.1 beschriebenen optimalitätstheoretischen Ansatz werden nicht immer alle möglichen Anfragen angezeigt, die mit der bisherigen Eingabe möglich wären. Stattdessen werden nur die am wenigsten markierten Vorschläge vom Server übermittelt. Dies führt einerseits zu größerer Übersicht und andererseits zu geringerem Rechenaufwand auf der Seite des Servers. Da der Client jedoch im Hintergrund mit dem Server in Kontakt steht, werden fortlaufend Vorschläge angefragt und damit gleichzeitig überprüft, ob die bisherige Anfrage verarbeitbar ist. Ist dies nicht der Fall, erscheinen keine Ergänzungsvorschläge mehr.
Sobald der Benutzer im Eingabefeld die Eingabetaste (Enter) drückt, oder auf die Schaltfläche „Abschicken“ klickt, wird die Anfrage vom Client an den Server übermittelt, damit dieser sie vollständig analysieren kann und eine Antwort auf die Anfrage berechnet. Hat der Server die Anfrage verarbeitet, schickt er die berechnete Antwort an den Client, welcher daraufhin die natürlichsprachige Antwort des Servers anzeigt.
Die vollständige graphische Oberfläche des Clients ist in Abbildung 2.5 auf Seite 32

[21] Der Client wartet 400 Millisekunden Inaktivität ab, bevor er eine Suggest-Anfrage an den Server schickt. Dies soll die Serverlast reduzieren und ist notwendig, damit der Server nicht durch zuviele Anfragen blockiert wird.

dargestellt. Unter dem ausklappbaren Element „Details und Analysen“ macht der Client die vom Server übermittelten und in Kapitel 4.3.3 dargestellten Informationen zugänglich. Dabei handelt es sich um die berechnete Suchanfrage für die Volltextsuche, die berechnete Suchanfrage für die Ähnlichkeitssuche (siehe Kapitel 4.2.5), die Namen der Felder des Ergebnisdokuments, nach deren Inhalt in der Anfrage gefragt wurde, die Anzahl der gefundenen möglichen Antwort-Dokumente, sowie die jeweilige linguistische Satzanalyse der Anfrage und der Antwort. Die Anzeige dieser Details wird in Abbildung 2.6 auf Seite 33 beispielhaft gezeigt.

2.4.2. Die technische Umsetzung

Die dynamischen Elemente der graphischen Oberfläche - die Anzeige der Antwort des Servers, das Ein- und Ausblenden der Details und Analysen, sowie das Einblenden der Vorschläge für Anfragen - und die Kommunikation mit dem Server im Hintergrund lassen sich nicht mit HTML alleine umsetzen. Für diese Form der Interaktivität ist eine Skriptsprache notwendig. Die einzige Skriptsprache, die sich in Webseiten einbetten lässt und die über Betriebssystem- und Browsergrenzen hinweg unterstützt wird, ist JavaScript. Andere Skriptsprachen sind entweder nur unter bestimmten Betriebssystemen oder in bestimmten Browsern verfügbar. Doch obwohl JavaScript grundsätzlich in dieser Form plattformunabhängig ist, ist es schwer und umständlich eine nicht triviale Funktionalität in JavaScript plattformübergreifend umzusetzen. Denn gerade was moderne Funktionen wie die asynchrone Kommunikation mit einem Webserver im Hintergrund betrifft,[22] gibt es starke Unterschiede zwischen den Browsern. Aus diesem Grund existieren Programmbibliotheken für JavaScript, die die individuellen Unterschiede der Browser berücksichtigen, diese aber vor dem Programmierer verbergen und ihm so die Nutzung einer einheitlichen Schnittstelle erlauben, welche mit den meisten Browsern kompatibel ist.
Eine solche Bibliothek ist das „Google Web Toolkit“[23] von Google. Das „Google Web Toolkit“ (GWT) ermöglicht nicht nur die Plattformunabhängigkeit der geschriebenen Programme, es stellt auch Lösungen für häufig wiederkehrende Aufgaben bereit. Dies erhöht einerseits die Qualität der Programme, da die Komponenten des Toolkits vielfach getestet sind, und beschleunigt andererseits auch die Entwicklung eigener Programme. Ein großer Unterschied zu anderen Bibliotheken für die Programmierung dynamischer Webseiten ist jedoch, dass das „Google Web Toolkit“ eine Java-Bibliothek ist. Das bedeutet, dass Programme, die unter Verwendung des Toolkits entwickelt werden, nicht in JavaScript, sondern in Java geschrieben werden. Das Toolkit erzeugt aus diesem Java-Code dann auf Geschwindigkeit optimierten und in allen gängigen Browsern (Mozilla Firefox, Windows Internet Explorer, Google Chrome, Apple Safari[24] und Opera[25]) lauf-

[22] Asynchrone Kommunikation bedeutet, dass die Kommunikation im Hintergrund stattfindet und der Client nicht lange so lange blockiert bis der Server antwortet. Der Client kann so also zu jedem Zeitpunkt neue Eingaben verarbeiten. Bei synchroner Kommunikation ist die Anfrage des Clients und die Antwort des Servers dagegen ein Ablauf, der nicht durch andere Aktivitäten unterbrochen werden kann.
[23] `http://code.google.com/webtoolkit/`
[24] `http://www.apple.com/de/safari/`
[25] `http://www.opera.com/`

Abbildung 2.5.: Vollständige Oberfläche des NASfVI-Clients mit einer natürlichsprachigen Anfrage und der natürlichsprachigen Antwort des Servers.

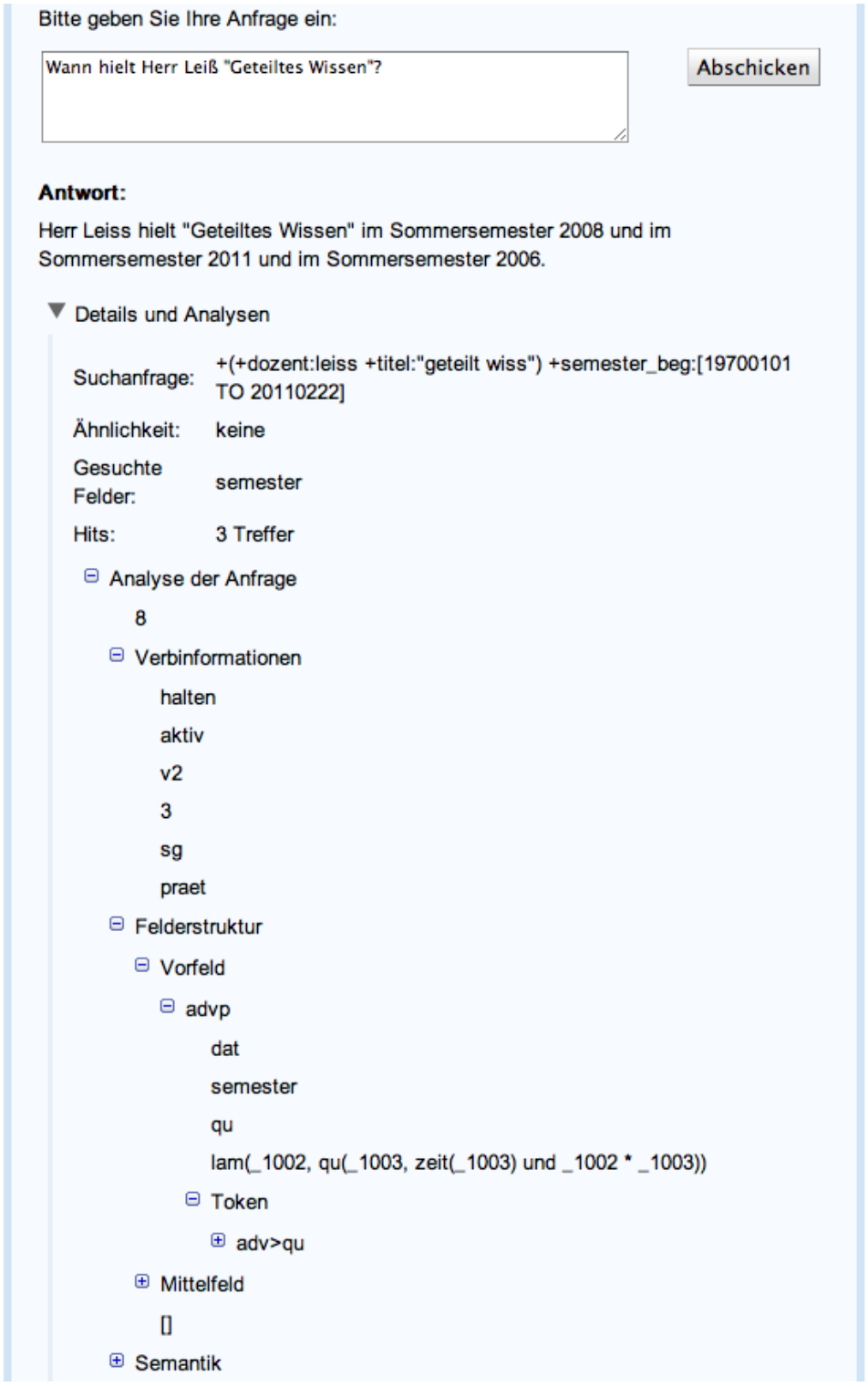


Abbildung 2.6.: Ausschnitt der Oberfläche des Clients mit eingeblendeten und teilweise ausgeklappten Details und Analysen.

fähigen JavaScript-Code [GWT23]. Die Übersetzung von Java in JavaScript stellt eine nicht triviale Herausforderung dar. Denn trotz der Namensähnlichkeit sind JavaScript und Java von Grund auf verschiedene Sprachen, die kaum Gemeinsamkeiten haben.
Durch die Verwendung des „Google Web Toolkits“ in NASfVI kann der Client, genau wie der Server, in Java entwickelt werden. Allerdings ist bei der Programmierung des Clients die Einschränkung auf Java-Bestandteile notwendig, die der Compiler des GWT auch nach JavaScript übersetzen kann. Die Implementierung des Clients mit dem Google Web Toolkit wird in Kapitel 5 diskutiert.

Teil II.

Diskussion der Implementierung

3. Die Sprachverarbeitungskomponente

Die sprachverarbeitende Komponente von NASfVI ist vollständig in Prolog geschrieben. In diesem Kapitel werden ausgewählte Stellen des Prolog-Quellcodes der Sprachverarbeitungskomponente diskutiert, um die Implementierung der Komponente zu veranschaulichen. Anzumerken ist, dass die Sprachverarbeitung mit normalisierten Eingaben arbeitet. Das bedeutet, dass Eingaben, die sprachlich analysiert werden sollen, durchgehend klein geschrieben sein müssen und nur Ziffern, Leerzeichen und die anderen in Anhang A.2 dargestellten, erlaubten Zeichen enthalten dürfen.
Der Quellcode der Komponente ist eingebettet im Server als auch als eigenständige Prolog-Anwendung lauffähig.

3.1. Lexikon und Flexion

3.1.1. Das Grundformenlexikon

Das Grundformenlexikon ist der Ausgangspunkt des gesamten Lexikons. In ihm sind sämtliche Lexeme des modellierten Sprachfragments verzeichnet. Die Einträge umfassen je nach Wortart die Grundform des Lexems, Angaben zur Flexion, die syntaktisch-semantische Valenz des Lexems, die Definitheit, den semantischen Typ, sowie den semantischen λ-Term. Die Einträge von flektierbaren Wortarten sind Fakten der grundform-Prädikate: Verben sind Fakten des Prädikats `grundform/6` und Nomen des Prädikats `grundform/7`. Flektierbare Wortarten, die in NASfVI jedoch nicht aktiv flektiert werden, sind Fakten mit dem Funktor form: Pronomen sind Fakten des Prädikats `form/7` und Artikel des Prädikats `form/6`. Nicht flektierbare Wortarten sind ebenfalls Fakten mit dem Funktor form: Junktoren sind Fakten des Prädikats `form/3`, Adverbien des Prädikats `form/4` und Präpositionen des Prädikats `form/5`. Da somit im Grundformenlexikon nicht nur die aktiv flektierten Wortarten vorhanden sind, sondern alle Wortarten des Sprachfragments, ist es der zentrale Speicherort des gesamten Wortschatzes. Zu diesem „Hort des Wortschatzes“ zählt auch, dass sämtliche Informationen über die einzelnen Lexeme zentral im Lexikon abgespeichert sind. Insbesondere sollen im Lexikon auch Valenz, Syntax und Semantik bereits auf Wortebene zusammenarbeiten und nicht als streng getrennte Module vorliegen (vgl. auch [SCH08]).
Wortarten werden in NASfVI immer in der Form a>b notiert, wobei es sich bei a und b um Prolog-Atome handelt. Diese Notation soll ausdrücken, dass die Wortart zur übergeordneten Wortart der Kategorie a gehört und vom Untertyp b ist. Handelt es sich um keinen besonderen Untertyp, entsprechen sich a und b. Beispiele für Wortarten in NASfVI sind v>v (Vollverben), v>aux (Hilfsverben), n>n (einfache Nomen), n>app (Appositionen), art>def (definite Artikel), art>indef (indefinite Artikel) und art>qu (interrogative

Artikel).

grundform/6: Verben

Die Einträge der Verben haben sechs Argumentstellen:

1. die Verbart. Dies kann entweder ein Vollverb (v>v) oder ein Hilfsverb (v>aux) kodieren,
2. der Infinitiv des Verbs ohne Verbpartikel,
3. falls vorhanden die Verbpartikel,
4. die Flexionsklasse,
5. die syntaktisch-semantische Valenz des Verbs,
6. der semantische λ-Term des Verbs;

Die Verben in NASfVI können entweder regelmäßig oder unregelmäßig konjugiert werden. Die regelmäßige Konjugation wird durch die Flexionsklasse *rg* angezeigt. Eine unregelmäßige, starke Konjugation wird durch die Flexionsklasse *urg*(*Ablautreihe* : *Position*, *Besonderheit*) angegeben. Die *Ablautreihe* ist eine Liste aus drei Elementen und gibt den veränderlichen Vokal oder Diphtong des Stammes im Präsens, Präteritum und beim Partizip II an. *Position* markiert die Stelle im Infinitiv, an der sich der veränderliche Stammvokal befindet, wenn beginnend mit dem ersten Buchstaben des Infinitivs bei eins mit dem Zählen begonnen wird. Die *Besonderheit*-Angabe verfeinert die Angaben zur Flexion zusätzlich. Derzeit sind in NASfVI zwei Besonderheiten bei den Verben möglich: *uml* und *eiw*. Verben mit der Besonderheit *uml* verlangen eine Umlautung des Stammvokals in der zweiten und dritten Person Singular Präsens. Einen e-i-Wechsel im Stammvokal führen dagegen Verben mit der Besonderheit *eiw* in der dritten und zweiten Person Singular Präsens aus.
Examplarisch seien drei Vollverben aus dem Grundformenlexikon herausgegriffen.

Beispiel: stattfinden

```
grundform(v>v, finden, statt, urg([i,a,u]:2, -),
        [patiens(event, SemPa), ?loc(SemLoc), ?temp_d(SemDies)],
        SemPa * lam(Y, SemDies * lam(D, SemLoc * lam(L,
                halten('_', Y, L, D))))
).
```

Die Angabe der Partikel *statt* zusammen mit *finden* und der Flexionsklasse *urg([i,a,u]:2, -)* gibt an, dass das Verb *stattfinden* unregelmäßig konjugiert wird und in der dritten Person Singular die Formen ***fi**ndet statt*, ***fa**nd statt* bzw. *st**a**ttgefunden* im Präsens, Präteritum und beim Partizip II in Verberst- und Verbzweitsätzen bildet. Der syntaktisch-semantische Valenzrahmen des Verbs besteht aus einem Patiens, einer fakultativen Ortsangabe und einer fakultativen Wochentagsangabe. Der semantische Typ des Patiens ist

zudem auf den Wert *event* festgelegt. Das bedeutet, dass *stattfinden* nur im Zusammenhang mit Veranstaltungen verwendet werden kann. Die verschiedenen semantischen Typen können der Tabelle 3.1 auf Seite 40 entnommen werden. Der ?-Präfixoperator ist in NASfVI definiert, um Optionalität innerhalb der Valenz anzugeben.
Die λ-Terme der Verben sind die zentralen Bausteine der Semantik in NASfVI, denn sie liefern das Gerüst der Semantik auf Ebene der Sätze. Über die Variabeln des Valenzrahmens gelangt die Semantik der einzelnen Phrasen bei der Satzanalyse durch Unifikation in den λ-Term des Vollverbs. Da somit die Semantik-Formel der Sätze in dem Lexikoneintrag des Vollverbs angelegt ist, sind die Positionen, in denen die Semantik der valenzgebunden Ergänzungen ausgewertet werden, stets dieselben. Die syntaktische Stellung der Phrasen hat also keinen Einfluss auf diese Position. Das hat zur Folge, dass der Skopus aller Quantoren in der Semantik stets gleich ist und sich daher keine Ambiguitäten durch Skopusvariationen ergeben können. Dieses Verhalten ist erwünscht. Denn der universitäre Kontext eines Vorlesungsverzeichnisses ist in NASfVI bekannt und somit eine eindeutige Modellierung und eine eindeutige Beantwortung der Anfragen möglich.
Die Verben des Sprachfragments bilden in der Regel Stützverbkonstruktionen. Aus diesem Grund finden sich die Verben nicht explizit in der Semantik wider. Stattdessen wird bei den meisten Verbeinträgen das vierstellige Prädikat *halten* in der Semantik verwendet, um *eine Veranstaltung halten* zu modellieren. Die vier Argumente dieses Prädikats nehmen Variabeln für den Dozenten, die Veranstaltung, den Ort und die Wochentagsangabe auf. Ist eine dieser Variabeln nicht gebunden oder in der Valenz des Verbes nicht vorgesehen, so bleibt die entsprechende Stelle im Prädikat ungebunden oder frei. Da bei *stattfinden* kein Agens und damit kein Dozent in dem Valenzrahmen vorgesehen ist, bleibt das erste Argument von *halten* im Falle von *stattfinden* immer unbesetzt.

Beispiel: halten

```
grundform(v>v, halten, '', urg([a,ie,a]:2, uml),
        [agens(hum, SemAg), patiens(event, SemPa), ?loc(SemLoc),
                ?temp_d(SemDies)],
        SemAg * lam(X, SemDies * lam(D, SemLoc * lam(L, SemPa
                * lam(Y, halten(X, Y, L, D))))))
).
```

Das Verb *halten* wird ebenfalls unregelmäßig konjugiert und bildet dabei gemäß der angegebenen Flexionsklasse in der dritten Person Singular die Formen *hält* und *hielt* im Präsens und Präteritum, sowie *gehalten* als Partizpi II. Im Gegensatz zu *stattfinden* enthält der Valenzrahmen von *halten* ein Agens. Dieses Agens muss vom Typ *hum* und damit menschlich sein. Das Patiens dagegen ist wie bei *stattfinden* auch eine Veranstaltung. Das Verb *halten* ist in NASfVI also nur in der Bedeutung *jemand hält eine Veranstaltung* verwendbar, was dem in Kapitel 2.2.2 erläuterten Sprachfragment entspricht.

Beispiel: geben

```
grundform(v>v, geben, '', urg([e,a,e]:2, eiw),
        [expletiv, patiens(_, Sem)],
        Sem * lam(X, X)
).
```

Das Verb *geben* hat keine Verbpartikel, wird ebenfalls unregelmäßig konjugiert und weist einen e-i-Wechsel in der Flexion auf. Diese Besonderheit bedeutet, dass in der zweiten und dritten Person Singular Präsens statt dem Vokal *e* aus der Ablautreihe der Vokal *i* verwendet wird: *du gibst, er gibt*. Valenzrahmen und Semantik unterscheiden sich bei *geben* stark von den Verben *stattfinden* und *halten*. Das Verb *geben* fordert in der hier gezeigten Lesart eine expletive Nominalphrase mit einem expletivem Es und ein Patiens mit einem beliebigen semantischen Typ. Die Semantik des Verbeintrags - und damit die Semantik des gesamten Satzes, der mit dem Verb *geben* in dieser Lesart gebildet wird - enthält kein Prädikat sondern lediglich die Semantik des Patiens und den Term $\lambda x.x$. Durch diese Semantik wird nichts über das Patiens ausgesagt außer dessen Existenz. Auf diese Weise modelliert der Lexikoneintrag von *geben* die Lesart wie in *Gibt es eine Vorlesung über Semantik*.
Verglichen mit den Vollverben sind die Lexikoneinträge der Hilfsverben dagegen deutlich inhaltsärmer:

```
grundform(v>aux, haben, '', rg, [], '').
grundform(v>aux, werden, '', urg([e,u,o]:2, eiw), [], '').
grundform(v>aux, sein, '', -, [], '').
```

Die Hilfsverben haben keine Verbpartikel, keine syntaktisch-semantische Valenz und keinen λ-Term. Sie treten in der Semantik nicht auf und werden in der Syntax u. a. als finite Verbformen bei der Bildung zusammengesetzter Zeiten und des Passivs benötigt.

grundform/7: Nomen

Die Lexikoneinträge der Nomen haben insgesamt sieben Argumente:

1. die Nomenart. Dies kann entweder ein einfaches Nomen (n>n) oder eine Apposition (n>app) kodieren,
2. der Nominativ Singular des Nomens als Grundform,
3. der semantische Typ des Nomens,
4. das Genus des Nomens,
5. die Flexionsklasse,
6. die syntaktisch-semantische Nomenvalenz,
7. der λ-Term des Nomens;

In NASfVI werden zwei Nomenarten unterschieden: einfache Nomen und Appositionen. Im Gegensatz zu den einfachen Nomen können Appositionen mit einer Blackbox ergänzt werden. Eine Blackbox ist in dem hier entwickelten sprachverarbeitenden System ein sprachlich nicht analysiertes Token. Bei den Nomen entspricht eine solche Blackbox immer einer Form von nicht lexikalischer, individueller Bezeichnung. Das kann der Name einer Person sein, aber auch eine Raumnummer oder Vergleichbares. Die Bezeichnung

Semantischer Typ	Bedeutung
hum	ein Mensch - in NASfVI immer ein Dozent
event	eine Veranstaltung
loc	eine Ortsangabe
temp_d	ein Wochentag
semester	ein Winter- oder Sommersemester
semester>sose	ein Sommersemester
semester>wise	ein Wintersemester

Tabelle 3.1.: Die semantischen Typen der Nomen in NASfVI

Apposition bezieht sich in dieser Arbeit also ausschließlich auf appositive Nebenkerne ([DUG06], 1562ff).
Wie bereits in Kapitel 2.2.1 über die theoretischen Grundlagen erwähnt, sind die Phrasen in NASfVI mit semantischen Typen ausgezeichnet. Als Ursprung dieser Typen können die Nomen betrachtet werden. Alle Nomen tragen in NASfVI einen semantischen Typ. Die verschiedenen semantischen Typen der Nomen werden mit ihrer Bedeutung in Tabelle 3.1 angegeben.

Beispiel: Montag

```
grundform(n>n, 'Montag', temp_d, mask, [es, e], [],
        lam(X, tag(X, 'mo'))).
```

Dieser Lexikoneintrag beschreibt den Wochentag Montag innerhalb des Sprachfragments. Wochentage sind einfache Nomen, da sie mit keinem zusätzlichen Eigennamen verbunden werden, und haben das grammatische Geschlecht maskulin. Die Flexionsklasse besteht bei Nomen aus mindestens zwei Schemata für die Deklination. Das erste beschreibt das Deklinationsmuster im Singular und das zweite das Deklinationsmuster im Plural. Diese Schemata entsprechen den in Anhang A.3 enthaltenen Endungstabellen. *Montag* wird also im Singular mit den Endungen der Tabelle *es* dekliniert und im Plural mit den Endungen der Tabelle *e*. Der Eintrag für *Montag* definiert eine leere Nomenvalenz. Denn wie in Kapitel 2.2.1 im Abschnitt über die Valenztheorie dargestellt, verfügen in NASfVI nur Lexme mit dem semantischen Typ *event* - also nur jene, die Veranstaltungen bezeichnen - über eine Nomenvalenz.

Beispiel: Raum

```
grundform(n>app, 'Raum', loc, mask, [es, e, umlaut:2], [],
        lam(Y, lam(X, raum(X,Y)))).
```

Der Lexikoneintrag der Apposition *Raum* zeigt, dass auch Nomen bei der Flexion Besonderheiten haben können. Die Flexionsbesonderheit *umlaut* : 2 bewirkt, dass der zweite Buchstaben der Grundform *Raum* im Plural in einen Umlaut umgewandelt wird. Aus *Raum* wird so durch Umlautung und dem Plural-Deklinationsschema *e* die Wortform *Raeume*.

Beispiel: Semester

```
grundform(n>n, 'Semester', semester>sem, neut, [s, -], [],
        lam(Y, lam(X, semester(X, Y)))).
```

Bei dem Lexikoneintrag zu *Semester* ist der im Vergleich mit dem Eintrag zu *Montag* abweichende λ-Term auffällig. Denn der Term im Eintrag für *Montag* bindet nur eine Variabel. Da durch den semantischen Typ in der Syntax sichergestellt ist, dass dies nur eine Variabel vom Typ einer Entität, gewissermaßen eine Wochentagskonstante, sein kann, entspricht dies der Montague-Semantik. *Semester* dagegen bindet zwei Variabeln. Der Grund dafür ist, dass *Semester* nicht nur eine Variabel als Semesterkonstante erwartet, sondern zusätzlich eine Semesterangabe in Form von Jahreszahlen. Grundsätzlich wäre dies auch als Apposition zusammen mit einer Blackbox modellierbar. Da Semesterangaben im Sprachfragment aber von zentraler Bedeutung sind, ihnen eine eindeutige Form zugeordnet werden kann[1] und für die Auswertung in NASfVI auch muss, wird hier keine Blackbox mit beliebigem Inhalt zugelassen, sondern nur Angaben in einem definierten, festen Format (siehe Kapitel 3.2.2).

Beispiel: Frau

```
grundform(n>app, 'Frau', hum, fem, [-, en], [],
        lam(Y, lam(X, dozent(X, Y)))).
```

Appositionen basieren in NASfVI semantisch auf zwei-stelligen Prädikaten ähnlich wie bei *Semester*. Bei den Appositionen ist Y aber der Wert einer Blackbox. Zwar können Appositionen auch mit einem leeren Wert für Y und somit ohne Blackbox syntaktisch als einfache Nomen auftreten (Kapitel 3.1.5), doch oft bilden sie zusammen mit einer Blackbox den nominalen Kern.
Außer dadurch, dass Appositionen in der Semantik zwei-stellig sind, unterscheiden sich deren Lexikoneinträge nicht grundlegend von denen der einfachen Nomen. Die valenzfähigen Nomen des semantischen Typs *event* unterscheiden sich allerdings durch die Valenz sowohl von den einfachen Nomen als auch von den anderen Appositionen:

Beispiele: Kurs, Vorlesung und Hauptseminar

```
grundform(n>app, 'Kurs', event, mask, [es, e],
        [?thema(akk, SemThema)],
        lam(Y, lam(X, veranstaltung(X, Y) und SemThema*X))
).

grundform(n>app, 'Vorlesung', event, fem, [-, en],
        [?thema(akk, SemThema)],
        lam(Y, lam(X, veranstaltung(X, Y) und typ(X, vorlesung) und
            SemThema*X))
).

grundform(n>app, 'Hauptseminar', event, neut, [s, e],
        [?thema(akk, SemThema)],
```

[1] Sommersemester werden in der Regel mit einfachen Jahreszahlen bezeichnet. Wintersemester werden dagegen häufig als Jahreszahl/Jahreszahl angegeben. Die Jahrszahlen können dabei zwei- oder vierstellig sein.

```
        lam(Y, lam(X, veranstaltung(X, Y) und typ(X, hauptseminar)
            und SemThema*X))
).
```

Nomen diesen Typs öffnen eine syntaktische und semantische Leerstelle für eine fakultative Themaergänzung. Die Semantik der Themaergänzung wird durch eine Variabel mittels Unifikation in den λ-Term der jeweiligen Apposition übertragen. Diese Themaergänzung wird im modellierten Sprachfragment nur in Form einer Präpositionalphrase mit der Präposition *über* realisiert (Kapitel 2.2.1). Nach den von Montague definierten Kategorien haben Präpositionen den Typ IAV/T, müssen also mit einer Nominalphrase kombiniert werden, um dann einen Ausdruck des Typs IV/IV zu ergeben [MON73]. Die Präposition müsste also mit einer Nominalphrase und die Präpositionalphrase mit einem intransitiven Verb zusammengebracht werden. Dies geschieht bei der Nomenvalenz in NASfVI aus zwei Gründen nicht. Erstens ist eine verbale Komponente bei der Themaergänzung nie geben. Und zweitens wird innerhalb der Präpositionalphrase der Themaergänzung die eigentliche Thema-Angabe als Blackbox verarbeitet. Sie ist also nicht in ihren Bestandteilen syntaktisch analysiert und kann daher nicht einfach als Nominalphrase angenommen werden. Daher wird in der Nomenvalenz nur die die Veranstaltung identifizierende Variabel an die Themaergänzungsphrase übergeben.

form/7: Pronomen

Pronomen werden in NASfVI nicht aktiv flektiert. Bei vielen Pronomen des Sprachfragments ist die Flexion zu speziell, als dass sich die Implementierung einer Pronomenflexion lohnen würde. Pronomen, deren Formen sich sehr stark unterscheiden, wie *wer, wessen, wem, wen*, oder deren Erscheinungsformen sich nicht oder kaum unterscheiden, wie *etwas* oder *was, wessen, was, was*, lassen sich besser durch einfaches Aufzählen implementieren.

Expletives Es:

```
form(pro>expl, 'es', expl, [sg, nom], es, [], '').
```

Das expletive Es tritt nur im Singular Nominativ auf. Es verwendet *expl* als Pronomen-Untertyp und Definitheit, wie auch als semantischen Typ. Es ist das einzige Lexem mit diesem semantischen Typ im Sprachfragment.

Interrogativpronomen:

```
form(pro>qu, Grundform, hum, Form, Atom, [], Sem) :-
        Sem = lam(P, qu(X, dozent(X, '') und P*X)),
        form_(pro>qu, Grundform, hum, Form, Atom).

form(pro>qu, Grundform, event, Form, Atom, Val, Sem) :-
        Val = [?thema(akk, SemThema)],
        Sem = lam(P, qu(X, veranstaltung(X, '')
                und SemThema*X und P*X)),
        form_(pro>qu, Grundform, event, Form, Atom).
```

Um bei den Interrogativpronomen die Semantik und Valenz nur einmal angeben zu müssen, ergänzt `form/7` die Formdaten aus `form_/5` mit der Semantik und gegebenfalls der

Valenz. Die Interrogativpronomen des Typs hum entsprechen semantisch einer interrogativen Nominalphrase mit einem Nomen desselben Typs als Kern. Auch die Interrogativpronomen des Typs event entsprechen in Valenz und Semantik exakt einer interrogativen Nominalphrasen des Typs event.

`form_/5`-Formdaten der Interrogativpronomen im Singular Nominativ:

```
form_(pro>qu, 'wer', hum, [sg, nom], wer).
form_(pro>qu, 'jemand', hum, [sg, nom], jemand).
form_(pro>qu, 'was', event, [sg, nom], was).
form_(pro>qu, 'etwas', event, [sg, nom], etwas).
```

form/6: Artikel

Bei Artikel wird das gleiche Vorgehen wie bei den Pronomen verwendet. Auch die Artikel beziehen ihre Formdaten aus Fakten des Prädikates `form_/5`. Während bei Pronomen das dritte Argument jedoch den semantischen Typ angibt, entspricht es bei Artikeln dem Artmerkmal, dem Genus des Artikels.

Artikel:

```
form(art>Def, Grundform, Genus, Form, Atom, Sem) :-
        Sem = lam(Q, lam(P, ex(X, Q*X und P*X))),
        (Def = def ; Def = indef),
        form_(art>Def, Grundform, Genus, Form, Atom).

form(art>qu, Grundform, Genus, Form, Atom, Sem) :-
        Sem = lam(Q, lam(P, qu(X, Q*X und P*X))),
        form_(art>qu, Grundform, Genus, Form, Atom).
```

Implementiert sind der definite Artikel, der indefinite und die Interrogativartikel *welcher, welche, welches*.

form/3, form/4, form/5: die nicht flektierbaren Wortarten

Die Lexikoneinträge der nicht flektierbaren Wortarten sind einfache Aufzählungen. Diese Wortarten sind Adverbien, Präpositionen und Junktoren.

Beispiel für Lexikoneinträge der Adverbien:

```
form(adv>qu, 'wo', loc, lam(P, qu(X, ort(X) und P*X))).
form(adv>qu, 'wann', Typ, lam(P, qu(X, zeit(X) und P*X))) :-
        Typ = temp_d ; Typ = semester.

form(adv>adv, 'montags', temp_d,
        lam(P, ex(X, tag(X, 'mo') und P*X))).
form(adv>adv, 'dienstags', temp_d,
        lam(P, ex(X, tag(X, 'di') und P*X))).
...
```

Die Semantik der Adverbien entspricht der Semantik, die Präpositionalphrasen und Adverbialphrasen aufweisen. Da bei NASfVI die Syntax-Terme fest verzahnt sind und nicht wie bei Montague parallel zur Syntax aufgebaut werden, muss beachtet werden, dass erst durch das Einsetzen der Semantik in den λ-Term des Vollverbs Terme entstehen, die den Kategorien entsprechen, die Montague definiert hat.

Eine kurze Überprüfung bestätigt dies:
`Term := SemLoc * lam(L, SemPa * lam(Y, halten(X, Y, L, D)))`
`Semantik` *halten* `aus Kapitel 3.1.1 = SemAg * lam(X,` *SemDies* `* lam(D, Term))`
`Semantik` *montags* `= lam(P, ex(T, tag(T, 'mo') und P*T))`
`Einsetzen der Semantik von` *montags* `in SemDies:`
`SemAg * lam(X,` *lam(P, ex(T, tag(T, 'mo') und P*T))* `* lam(D, Term))`
`= SemAg *` *lam(X, ex(T, tag(T, 'mo') und lam(D, Term)*T))*
Die Adverbialphrase erwartet in der Variabel P einen Term der Kategorie IV. Wird diese adverbiale Semantik in die Semantik-Formel des Verbs eingesetzt und eine Betareduktion durchgeführt, entsteht der Term `lam(X, ex(T, tag(T, 'mo') und lam(D, Term)*T))`. Dieser Term erwartet in der Variabel X die Kategorie e. Damit ergibt sich für diesen Teilterm die Kategorie t/e und damit IV. Der durch Einsetzen der Adverbialsemantik entstehende Term entspricht also der Montague-Kategorie IV/IV und damit der Kategorie von Adverbialphrasen IAV.

Lexikoneinträge der Präpositionen:

```
form(p>bbox, 'von', thema, dat, lam(T, lam(X, thema(X, T)))).
form(p>bbox, 'ueber', thema, akk, lam(T, lam(X, thema(X, T)))).

form(p>p, 'von', hum, dat, lam(Q, lam(P, Q*P))).
form(p>p, 'in', Typ, dat, lam(Q, lam(P, Q*P))) :-
        Typ = semester ; Typ = loc.

form(p>def, 'im', Typ, dat, lam(Q, lam(P, ex(X, Q*X und P*X)))) :-
                Typ = semester ; Typ = loc.
form(p>def, 'am', temp_d, dat, lam(Q, lam(P, ex(X, Q*X und P*X)))).
```

Lexikoneinträge der Junktoren:

```
form(junkt>con, 'und', lam(A, lam(B, lam(P, A*P und B*P)))).
form(junkt>dis, 'oder', lam(X, lam(A, lam(B, lam(P, ex(X, (A oder B)
    und P*X)))))).
```

Die Junktoren in NASfVI verbinden Phrasen miteinander. Die kopulative Konjunktion *und* ist geradeheraus definiert. Zur disjunktiven Konjunktion *oder* siehe Kapitel 3.2. Die Disjunktion muss es erlauben, die koordinierten Phrasen zu einer Phrase zusammenfassen. Zu diesem Zweck kann die in der koordinierten Phrase gebundene Variabel X von Außen bestimmt und so sichergestellt werden, dass sie mit den Individuenvariabelen in A und B übereinstimmt.

3.1.2. Die Flexion

In NASfVI ist die Flexion von Nomen und Verben implementiert. Die Flexionsprädikate kapseln dabei das Grundformenlexikon in dem Sinn, dass der Aufruf der Grundformen durch das Flexionsprädikat erfolgt und es außerdem alle Informationen aus dem Grundformen-Eintrag mit Ausnahme der Flexionseigenschaften zusammen mit der erzeugten, flektierten Vollform dem Aufrufer der Flexion verfügbar macht.
Die Nomenflexion und die Verbflexion teilen sich in NASfVI gemeinsame Prädikate. Das heißt, sie greifen teilweise auf dieselbe Funktionalität zurück. Das mit Abstand wichtigste gemeinsam genutzte Prädikat ist `besonderheit/6`. Dieses Prädikat stellt fest, ob in der momentanen Flexionssituation die Standard-Flexionsendung der Flexionsklasse genutzt werden kann, oder ob die Endung aufgrund von Besonderheiten oder Unregelmäßigkeiten abgeändert werden muss. Das erste Argument von `besonderheit/6` kodiert die zu flektierende Wortart (n für Nomen und v für Verben), das zweite den Stamm des Wortes, gefolgt von der Flexionsklasse des Wortes, den Formmerkmalen (eine Liste mit Kasus und Numerus im Fall von Nomen und eine Liste mit Person, Numerus und Tempus im Fall von Verben), und zum Schluss die wichtigsten Argumente: ein Argument mit der Standard-Flexionsendung für die gegebenen Formmerkmale und als letztes Argument den Rückgabewert - die Endung, die tatsächlich genutzt werden muss.

Flexion der Nomen

Die Flexion der Nomen erfolgt durch das Prädikat `flexion/8`. Die acht Argumente des Prädikats sind:

1. die Nomenart (n>n oder n>app),
2. die Grundform,
3. der semantische Typ des Nomens,
4. die syntaktisch-semantische Nomenvalenz,
5. der semantische λ-Term,
6. die Formmerkmale - eine Liste aus Numerus und Kasus,
7. die Vollform;

Die Nomenflexion funktioniert nach einem einfachen Schema. Zunächst werden die morphosyntaktischen Merkmale überprüft, dann die Informationen des Grundformenlexikons abgerufen, sowie aus der passenden Endungstabelle die reguläre Endung der Flexionsklasse geholt und schließlich mit `besonderheit/6` die tatsächlich zu verwendende Endung ermittelt und die Vollform erzeugt.

Flexion der Nomen im Singular

```
flexion(n>N, Grundform, Typ, Genus, Val, Sem, [sg, Cas], Vollform) :-
casus(Cas),
grundform(n>N, Grundform, Typ, Genus, [Flex, _|Zusatz], Val, Sem),
(Zusatz = [] ; Zusatz = [Besonderheit]),
endung(n, Flex, [sg, Cas], Endung_),
besonderheit(n, Grundform, Besonderheit, [sg, Cas], Endung_, Endung),
atom_concat(Grundform, Endung, Vollform).
```

Flexion der Nomen im Plural

```
flexion(n>N, Grundform, Typ, Genus, Val, Sem, [pl, Cas], Vollform) :-
casus(Cas),
grundform(n>N, Grundform, Typ, Genus, [_, Flex|Zusatz], Val, Sem),
(Zusatz = [] ; Zusatz = [Besonderheit]),
(
        (nonvar(Besonderheit), Besonderheit = umlaut:Pos)
                -> ersetze_durch_umlaut(Grundform, Pos, GF_)
                ; GF_ = Grundform
),
endung(n, Flex, [pl, Cas], Endung_),
besonderheit(n, Grundform, Besonderheit, [pl, Cas], Endung_, Endung),
atom_concat(GF_, Endung, Vollform).
```

Die Flexionsklassen können bei den Nomen zwei Muster haben:

- Deklinationsschema für Singular und Plural: [-, en]
  (Beispiel: Flexionsklasse des Nomens Veranstaltung)
- Beide Schemata und zusätzlich eine Flexionsbesonderheit: [es, e, umlaut:2]
  (Beispiel: Flexionsklasse des Nomens Raum)

Beim Zugriff auf die Grundform wird die Flexionsklasse daher in eine Liste mit zwei Elementen und eine Restliste zerlegt. Durch den Trick, die Flexionsklasse in zwei Elemente und eine Restliste aufzuspalten, funktioniert der Aufruf bei beiden Mustern von Flexionsklassen. Denn ist keine *Besonderheit* vorhanden, ist die Restliste (die Variabel *Zusatz*) schlicht leer. Im Plural führt `flexion/8` zudem eine Veränderung des Stamms durch, wenn *Besonderheit* mit `umlaut:Pos` unifiziert. In diesem Fall ruft es `ersetze_durch_umlaut/3` auf. Dieses Prädikat führt in dem als ersten Argument übergebenen Atom an der Stelle *Pos* eine Umlautung durch und gibt den neuen Stamm als drittes Argument aus. Umlautung bedeutet, dass a durch ae, o durch oe und u durch ue ersetzt wird.
Bei der Nomenflexion berücksichtigt NASfVI zudem die folgenden Sonderfälle bei Endungen durch das Prädikat `besonderheit/6`:

- n-Verdoppelung bei Endung -in

  ```
  besonderheit(n, Gf, _, [pl, _], en, nen)
          :- atom_concat(_, in, Gf), !.
  ```

  Regel: Wenn sich von der Grundform die Endung -in abspalten lässt und die Pluralendung -en verwendet werden soll, dann wird das *n* am Stammende verdoppelt

[CNOO]. Diese Regel stellt sicher, dass aus zum Beispiel *Dozentin* und der regelmäßigen Endung *-en* die korrekte Form *Dozentinnen* wird.

- e-Tilgung bei Endung -en

```
besonderheit(n, Gf, unb, [sg, _], en, Endung)
        :- atom_concat(_, err, Gf), !, (Endung=n ; Endung=en).
```

Regel: Endet ein Nomen auf -err und trägt die Besonderheit *unb* für *unb*etontes Stammende, dann sind mit der Standardendung -en die Endungen -n und -en möglich [CNOO]. Diese Regel erzeugt aus zum Beispiel *Herr* und der Endung -en die beiden Formen *Herrn* und *Herren*.

- e-Einfügung bei „Genitiv Singular"-Endung -s

```
besonderheit(n, Gf, _, [sg, gen], s, es)
        :- atom_concat(_, s, Gf), !.
```

Regel: Endet der Stamm eines Nomens auf *-s* und soll im Genitiv Singular die Endung *-s* angehängt werden, wird ein *e* eingeschoben. Diese Regel stellt sicher, dass zum Beispiel aus *Kurs* mit der Standardendung *-s* die korrekte Form *Kurses* wird.

- Keine Besonderheit

```
besonderheit(_, _, _, _, Endung, Endung).
```

Für die fallunterscheidungslose Funktionalität notwendig.

Flexion der Vollverben

Im Vergleich zu der Nomenflexion ist die Flexion der Vollverben deutlich aufwendiger. Denn die Verbflexion benötigt mehr Hilfsprädikate als die Nomenflexion. Für die Flexion der Vollverben sind die Prädikate `ablaut/5`, `stamm/3`, `flektiere_stamm/4` und natürlich `besonderheit/6` wichtig.

```
ablaut(praes, [Pers, sg], eiw, [e, _, _], i)
        :- (Pers = 2; Pers = 3), !.
ablaut(praes, [Pers, sg], uml, [V1, _, _], V)
        :- (Pers = 2; Pers = 3), umlaut(V1, V), !.
ablaut(praes, _, _, [V1, _, _], V1).
ablaut(praet, _, _, [_, V2, _], V2).
ablaut(part2, _, _, [_, _, V3], V3).
```

Das Prädikat `ablaut/5` verarbeitet die Ablautreihe von unregelmäßigen Verben und ermittelt in Abhängigkeit der Formmerkmale (eine Liste bestehend aus Person und Numerus) und der Flexionsbesonderheiten den Ablaut, der verwendet werden muss. Das erste Argument ist das Tempus, gefolgt von den Formmerkmalen, eventueller Flexionsbesonderheiten, der vollständigen Ablautreihe bestehend aus drei Vokalen und dem letzten Argument in welchem `ablaut/5` den zu verwendenden Ablaut zurückgibt. Im Allgemeinen wird im Präsens der erste Vokal der Ablautreihe verwendet. Im Präteritum der zweite

und für die Bildung des Partizip II der letzte Vokal der Ablautreihe. Ausnahmen existieren bei den Flexionsbesonderheiten *eiw* und *uml*. Bei ersterer, dem e-i-Wechsel, wird in der zweiten und dritten Person Singular Präsens nicht der erste Vokal der Ablautreihe (ein *e*), sondern stattdessen ein *i* verwendet. Bei der Flexionsbesonderheit *uml* wird in der zweiten und dritten Person Singular Präsens der erste Vokal der Ablautreihe zu einem Umlaut. Das heißt, aus a wird ae, aus o wird oe und aus u wird ue.

```
stamm(Infinitiv, Stamm, 'en')
        :- atom_concat(Stamm, 'en', Infinitiv), !.
stamm(Infinitiv, Stamm, 'n')
        :- atom_concat(Stamm, 'n', Infinitiv).
```

`stamm/3` ist ein vergleichsweise einfaches Prädikat. Es spaltet lediglich den Infinitiv in den Verbstamm und die Infinitivendung auf. Mindestens der Infinitiv oder der Stamm muss gegeben sein. `stamm/3` berücksichtigt auch die e-Tilgung bei Verben, die im Infinitiv nicht auf *-en*, sondern nur auf *-n* enden ([DUG06], 620). Im modellierten Sprachfragment ist dies bei den Verben *handeln* und *ähneln* der Fall.

```
flektiere_stamm(InfStamm, [Pers, Num, Temp],
        urg(Ablaute:Pos, FlexEig), FlexStamm) :-
        stamm(InfStamm, Stamm, _),
        ablaut(Temp, [Pers, Num], FlexEig, Ablaute, Ablaut),
        ersetze(Stamm, Pos, Ablaut, FlexStamm), !.

flektiere_stamm(InfStamm, [_, _, _], rg, FlexStamm)
        :- stamm(InfStamm, FlexStamm, _), !.
```

Das Hilfsprädikat `flektiere_stamm/4` erzeugt aus dem Infinitivstamm, den Formmerkmalen (Person, Numerus, Tempus) und der Flexionsklasse dem den Formmerkmalen entsprechenden flektierten Stamm. Bei unregelmäßigen Verben wird dazu mit `ersetze/4` an der Stelle des Ablauts der korrekte durch `ablaut/5` ermittelte Ablaut eingesetzt. Bei regelmäßigen Verben ist keine Veränderung am Stamm notwendig.
Die Verbflexion berücksichtigt die folgenden Besonderheiten im Zusammenhang mit den Flexionsendungen aus den Endungstabellen der Flexionsklassen:

- Keine Endung -tt

  ```
  besonderheit(v, haelt, urg(_,uml), [3, sg, praes], 't', '') :- !.
  ```

  Regel: Die Verbform *haelt* mit unregelmäßiger Konjugation und *uml*-Besonderheit erhält in der dritten Person Präsens kein zusätzliches *t* als Endung ([DUG06], 642).

- e-Einschub

  ```
  besonderheit(v, Stamm, FlexKlasse, [Pers, Num, Temp],
          Endung_, Endung) :-
  (atom_concat(_, 't', Stamm) ; atom_concat(_, 'd', Stamm)),
  member(Endung_, ['t', 'st']),
  \+((
  member([Pers, Num, Temp], [[2, sg, praes], [3, sg, praes]]),
  ```

```
member(FlexKlasse, [urg(_,uml), urg(_,eiw)])
)),
atom_concat('e', Endung_, Endung), !.
```

Regel: Verben mit Dentalstämmen, die auf *-t* oder *-d* enden, fügen vor die Endung ein *e* ein, wenn die reguläre Endung *-t* oder *-st* ist. Unregelmäßige Verben mit *uml/eiw*-Flexionsbesonderheit sind davon jedoch nicht betroffen ([DUG06], 617).

- e-Tilgung

```
besonderheit(v, Stamm, rg, [_, _, praes], 'en', 'n') :-
        (atom_concat(_, 'el', Stamm)
        ; atom_concat(_, 'er', Stamm)), !.
```

Regel: Verben deren Stamm auf *-el* oder *-er* enden, erhalten im Präsens keine Endung *-en*, sondern tilgen das *e* ([DUG06], 620). Diese Regel erzeugt aus zum Beispiel dem Verbstamm *handel* mit der Endung *-en* die Form *handeln* und verhindert **handelen*.

- Keine Besonderheit

```
besonderheit(_, _, _, _, Endung, Endung).
```

Für die fallunterscheidungslose Funktionalität notwendig.

```
konjugiere(Stamm, Partikel_, FlexKlasse,
        [Vst, Pers, Num, Temp], Vollform, Partikel) :-
        endung(v, FlexKlasse, [Pers, Num, Temp], Endung_),
        besonderheit(v, Stamm, FlexKlasse,
        [Pers, Num, Temp], Endung_, Endung),
        atom_concat(Stamm, Endung, Vollform_),
        (
        (Partikel_ \= '', Vst = vl)
        ->      (atom_concat(Partikel_, Vollform_, Vollform),
            Partikel = '')
        ;       (Vollform = Vollform_, Partikel = Partikel_)
        ).
```

`konjugiere/6` ist das zentrale Hilfsprädikat für die Erzeugung finiter Verbformen. Es wird mit dem flektierten Stamm, der Verbpartikel, der Flexionsklasse und den aktuellen Formmerkmalen (Verbstellung, Person, Numerus, Tempus) aufgerufen. Das Prädikat gibt in den letzten beiden Argumenten die flektierte Vollform des Verbs zurück und falls die Vollform eine abgetrennte Verbpartikel hat, wird diese im letzten Argument zurückgegeben. `konjugiere/6` ermittelt zunächst die reguläre Endung der Flexionsklasse, leitet diese an `besonderheit/6` weiter und erzeugt schließlich mit dem Stamm und der Endung die Vollform. Die Konjugation dieses Prädikats berücksichtigt die Verbstellung des Satzes. Bei Verbletztstellung wird die Partikel - falls vorhanden - der Vollform des Verbs vorangestellt. In Verberst- und Verbzweitstellung bildet das Verb *stattfinden* zum Beispiel die Form *findet statt* in der dritten Person Singular Präsens. Bei Verbletztstellung lautet dieselbe Form dagegen *stattfindet*.

Die meisten Hilfsprädikate sind nun diskutiert worden. Der Einstiegspunkt zur eigentlichen Verbflexion ist wie bei der einfacheren Nomenflexion ein Prädikat `flexion`. Bei der Verbflexion werden die finiten Verbformen, das Partizip II und der Infinitiv in jeweils eigenen `flexion/7`-Regeln erzeugt.

Erzeugung der finiten Vollverbformen

```
flexion(v>v, Infinitiv, Partikel, Val, Sem, [Vst, Pers, Num, Temp],
    Vollform) :-
        einfache_zeiten(Temp), numerus(Num), person(Pers),
        grundform(v>v, InfStamm, Partikel_, FlexKlasse, Val, Sem),
        atom_concat(Partikel_, InfStamm, Infinitiv),
        flektiere_stamm(InfStamm, [Pers, Num, Temp], FlexKlasse,
            FlexStamm),
        konjugiere(FlexStamm, Partikel_, FlexKlasse, [Vst, Pers, Num,
             Temp], Vollform, Partikel).
```

Die Vollverben sind nur bei den einfachen Zeiten Präsens und Präteritum finit. Bei den anderen Zeitformen, den zusammengesetzten Zeiten, trägt ein Hilfsverb die finiten Merkmale und das Vollverb liegt entweder als Partizip II oder als Infinitiv vor. Zunächst stellt jede `flexion/7`-Regel fest, ob die Formmerkmale gültige Werte enthalten. Danach wird auf das Grundformenlexikon zugegriffen. Die Regel für finite Vollverben erzeugt daraufhin den Infinitiv indem es die Verbpartikel und den Infinitivstamm zusammensetzt. Hat ein Verb keine Verbpartikel, so ist die Partikelangabe im Grundformenlexikon ein leeres Atom. Eine Fallunterscheidung muss daher an dieser Stelle nicht getroffen werden. Im nächsten Schritt wird der Stamm mit `flektiere_stamm/4` flektiert und die Vollform schließlich durch konjugiere/6 erzeugt.

Erzeugung des Partizips II (part2)

```
flexion(v>v, Infinitiv, '', Val, Sem, part2, Vollform) :-
        grundform(v>v, InfStamm, Partikel_, FlexKlasse, Val, Sem),
        atom_concat(Partikel_, InfStamm, Infinitiv),
        flektiere_stamm(InfStamm, [_, _, part2], FlexKlasse,
            FlexStamm),
        endung(v, FlexKlasse, part2, Endung),
        atom_concat('ge', FlexStamm, S1), % ge.fund
        atom_concat(Partikel_, S1, S2), % statt.gefund
        atom_concat(S2, Endung, Vollform). % stattgefund.en
```

Erzeugung des Infinitivs (inf)

```
flexion(v>v, Infinitiv, '', Val, Sem, inf, Infinitiv) :-
        grundform(v>v, InfStamm, Partikel, _, Val, Sem),
        atom_concat(Partikel, InfStamm, Infinitiv).
```

Flexion der Hilfsverben

Die Konjugation der Hilfsverben unterscheidet sich deutlich von denen der normalen Vollverben und wird daher in ein Prädikat `hilfsverb/3` ausgelagert. Das Prädikat `hilfs`

`verb/3` hat drei Argumente. Das erste Argument ist die Grundform des Hilfsverbs, das zweite sind die Formmerkmale der Vollform und das dritte Argument ist die Vollform des Hilfsverbs.

Flexion des Hilfsverbs *haben*

```
hilfsverb(haben, inf, haben).
hilfsverb(haben, part2, gehabt).
hilfsverb(haben, [Pers, Num, Temp], Vollform) :-
        einfache_zeiten(Temp), numerus(Num), person(Pers),
        grundform(v>aux, haben, '', rg, [], ''),
        ((Temp = praet)
                -> Stamm = hat
                ; ( (Temp = praes, Num = sg,
                (Pers = 2 ; Pers = 3)) -> Stamm = ha ; Stamm = hab )
        ),
        konjugiere(Stamm, '', rg, [_, Pers, Num, Temp], Vollform, '').
```

Das Hilfsverb *haben* wird schwach konjugiert, weist aber dennoch verschiedene Unregelmäßigkeiten auf. So werden in der zweiten und dritten Person Singular Präsens die Formen *hast* und *hat* gebildet und im Präteritum *hatte*. Die Duden-Grammatik ([DUG06], 650) gibt an, dass es sich dabei um Zusammenziehungen aus den mittelhochdeutschen Formen *hab(e)st*, *hab(e)t* und *habete* handelt. Diese starken Veränderung am Stamm des Verbs liegen außerhalb dessen, was die normale Vollverbflexion hier leisten kann. In `hilfsverb/3` wird *haben* daher in den problematischen Fällen mit einem anderem Stamm modelliert.

Flexion des Hilfsverbs *werden*

```
hilfsverb(werden, inf, werden).
hilfsverb(werden, part2, worden).
hilfsverb(werden, [2, sg, praes], wirst).
hilfsverb(werden, [3, sg, praes], wird).
hilfsverb(werden, [Pers, sg, praet], wurde) :- (Pers = 1 ; Pers = 3).
hilfsverb(werden, [Pers, Num, Temp], Vollform) :-
        einfache_zeiten(Temp), numerus(Num), person(Pers),
        \+member([Pers, Num, Temp], [[2,sg,praes], [3,sg,praes], [1,
            sg,praet], [3,sg,praet]]),
        grundform(v>aux, werden, '', FlexKlasse, [], ''),
        flektiere_stamm(werden, [Pers, Num, Temp], FlexKlasse,
            FlexStamm),
        konjugiere(FlexStamm, '', FlexKlasse, [_, Pers, Num, Temp],
            Vollform, '').
```

Das Hilfsverb *werden* wird gemäß der Flexionsklasse `urg([e,u,o]:2)` flektiert. Ausnahmen bilden jedoch auch hier die zweite und dritte Person Singular Präsens, sowie die erste und dritte Person Singular im Präteritum. Außerdem wird das Partizip II bei der Verwendung als Hilfsverb im Passiv ohne dem Präfix *ge-* gebildet([DUG06], 649).

Flexion des Hilfsverbs *sein*

```
hilfsverb(sein, inf, sein).
hilfsverb(sein, part2, gewesen).
hilfsverb(sein, [1, sg, praes], bin).
hilfsverb(sein, [2, sg, praes], bist).
hilfsverb(sein, [3, sg, praes], ist).
hilfsverb(sein, [Pers, pl, praes], sind) :- (Pers = 1 ; Pers = 3).
hilfsverb(sein, [2, pl, praes], seid).
hilfsverb(sein, [Pers, sg, praet], war) :- (Pers = 1 ; Pers = 3).
hilfsverb(sein, [2, sg, praet], warst).
hilfsverb(sein, [Pers, pl, praet], waren) :- (Pers = 1 ; Pers = 3).
hilfsverb(sein, [2, pl, praet], wart).
```

Die Konjugation des Hilfsverbs *sein* ist so stark unregelmäßig, dass die einzelnen Formen einfach nur als Fakten aufgezählt werden ([DUG06], 700).

3.1.3. Das Vollformenlexikon

Das Vollformenlexikon baut auf der Flexion der flektierbaren Wortarten und auf dem Grundformenlexikon auf. Es stellt eine einheitlichere Schnittstelle zu den flektierten Vollformen (`flexion/7` und `flexion/8`), den Hilfsverben (`hilfsverb/3`), sowie den Formen aus dem Grundformenlexikon (`form/3`, `form/4`, `form/5`, `form/6`, `form/7`) zur Verfügung. Während bei den bisherigen Prädikaten ein möglichst geringer Schreibaufwand für die Pflege des Wortschatzes durch Menschen im Vordergrund stand, soll das Vollformenlexikon eine einheitliche Schnittstelle schaffen. Die bisherigen verschiedenen Datenquellen werden im Vollformenlexikon auf die Prädikate `vollform/7`, `vollform/8` und `vollform/9` abgebildet. Zusammen mit dieser Abbildung werden auch die bisher atomaren Vollformen durch `atom_chars/2` zusätzlich in Listen von einzelnen Zeichen und Buchstaben aufgespalten. Eine solche Zerlegung der Prolog-Atome in Zeichenlisten ist zwingend erforderlich, damit die Suggest-Funktion auch innerhalb von Worten ansetzen kann.

vollform/9: Nomen und Pronomen

Die Vollformeneinträge von Nomen und Pronomen haben insgesamt neun Argumente:

1. die Wortart: n>n (einfaches Nomen), n>app (Apposition), pro>qu (Interrogativpronomen) oder pro>expl (Expletiv);
2. die Vollform als Liste von Buchstaben,
3. die Formmerkmale,
4. der semantische Typ des Eintrags,
5. das Genus,
6. der syntaktische Valenzrahmen der Nomenvalenz,
7. der λ-Term der Semantik,

8. die Grundform als Atom,
9. die Vollform als Atom;

Abbildung der Nomen

```
vollform(n>N, LForm, Formmerkmale, Typ, Genus, Val, Sem, Grundform,
    AForm) :-
        flexion(n>N, Grundform, Typ, Genus, Val_,
                 Sem, Formmerkmale, AForm),
        atom_chars(AForm, LVF),
        nur_kleinbuchstaben(LVF, LForm),
        expandiere_valenz(aktiv, Val_, Val, _).
```

`flexion/8` erzeugt die flektierten Vollformen der Nomen und stellt die Informationen aus den Grundformen zusätzlich zur Verfügung. Mit `atom_chars/2` wird eine Liste der Zeichen und Buchstaben erzeugt. Da bei Nomen auch Großbuchstaben vorkommen können, konvertiert `nur_kleinbuchstaben/2` die Zeichenliste in eine Zeichenliste mit reinen Kleinbuchstaben. Die sprachverarbeitende Komponente arbeitet bei Anfragen nur mit normalisierten Daten. Das normalisierte Format besteht nur aus Kleinbuchstaben und der Verwendung des ASCII-Zeichensatzes. Sonderzeichen wie Umlaute müssen in ASCII-Entsprechungen konvertiert werden. Die in NASfVI verwendeten ASCII-Entsprechungen für Umlaute sind ae, oe und ue. Das Prädikat `expandiere_valenz/4` expandiert die syntaktisch-semantische Valenz in rein syntaktische Phrasen für die Verwendung in der Syntax. `expandiere_valenz/4` wird in Kapitel 3.1.4 dargestellt.

Abbildung der Pronomen

```
vollform(pro>Sub, LForm, Formmerkmale, Typ, '-', Val, Sem, Grundform,
    AForm) :-
        form(pro>Sub, Grundform, Typ, Formmerkmale, AForm,
             Val_, Sem),
        atom_chars(AForm, LForm),
        expandiere_valenz(aktiv, Val_, Val, _).
```

Die Pronomen werden analog zu den Nomen in das Vollformenlexikon überführt. Der einzige Unterschied liegt darin, dass das expletive Es und die interrogativen Pronomen, die im Sprachfragment genutzt werden, kein Genus besitzen. Daher wird in dem entsprechenden Argument im Vollformeneintrag bei Pronomen lediglich ein '-' gesetzt.

vollform/7: Verben

Insgesamt sieben Argumente haben die Einträge der Vollformen im Fall von Verben:

1. die Wortart: v>v (Vollverb) oder v>aux (Hilfsverb),
2. eine Liste mit zwei Elementen:
   a) eine Liste der Buchstaben der Vollform ohne Partikel,
   b) eine Liste der Buchstaben des Partikels, falls vorhanden; sonst leere Liste;

3. die Formmerkmale,
4. die syntaktisch-semantische Valenz,
5. der semantische λ-Term des Verbs,
6. der Infinitiv,
7. eine Liste mit zwei Elementen für die Vollform des Verbs ohne Partikel und die Partikel als Atome. Ist kein Partikel vorhanden, ist das letzte Atom leer;

Für jede Verbart existieren zwei `vollform/7`-Regeln. Eine behandelt die indefiniten Verbformen: den Infinitiv und das Partizips II. Die andere die finiten Formen. Diese Trennung erfolgt, da sich die Formmerkmale der indefiniten Formen grundlegend von denen der finiten Formen unterscheiden. Anders als bei Nomen und Pronomen wird die syntaktisch-semantische Valenz der Verben nicht schon bei den Vollformen in einen rein syntaktischen Valenzrahmen übersetzt (Kapitel 3.1.4), sondern erst beim Abruf der Formen im Lexikon (Kapitel 3.1.5). Dadurch ist es möglich, dass die semantischen Angaben im Valenzrahmen je nach Genus Verbi unterschiedlich expandiert werden. Dies ist insbesondere auch für die Kongruenz zwischen der valenzgebundenen Subjektphrase und der Verbform wichtig.

Hilfsverben: indefinite Formen

```
vollform(v>aux, [LVerb, []], Form, [], '', Infinitiv, [AVerb, '']) :-
        indefinite(Form),
        hilfsverb(Infinitiv, Form, AVerb), atom_chars(AVerb, LVerb).
```

Hilfsverben: finite Formen

```
vollform(v>aux, [LVerb, []], [_Vst|[Pers, Num, Temp]], [], '',
    Infinitiv, [AVerb, '']) :-
        hilfsverb(Infinitiv, [Pers, Num, Temp], AVerb), atom_chars(
            AVerb, LVerb).
```

Vollverben: indefinite Formen

```
vollform(v>v, [LVerb, []], Form, Val, Sem, Infinitiv, [AVerb, '']) :-
        indefinite(Form),
        flexion(v>v, Infinitiv, '', Val, Sem, Form, AVerb),
        atom_chars(AVerb, LVerb).
```

Vollverben: finite Formen

```
vollform(v>v, [LVerb, LPartikel], [Vst, Pers, Num, Temp], Val, Sem,
    Infinitiv, [AVerb, APartikel]) :-
        verbstellung(_, Vst),
        flexion(v>v, Infinitiv, APartikel, Val, Sem, [Vst, Pers, Num,
            Temp], AVerb),
        atom_chars(AVerb, LVerb), atom_chars(APartikel, LPartikel).
```

vollform/8: Adverbien, Präpositionen, Junktoren, Artikel

Das Prädikat `vollform/8` repräsentiert die Vollformen aller Wortarten mit Ausnahme der Nomen und Pronomen (`vollform/9`) und der Verben (`vollform/7`). `vollform/8` hat die folgenden acht Argumente:

1. die Wortart: adv>adv (einfache Adverbien), adv>qu (interrogative Adverbien), p>p (einfache Präpositionen), p>def (definite Präpositionen), p>bbox (Präpositionen, die eine Blackbox erlauben), junkt>con und junkt>dis (Konjunktionen), art>def (definite Artikel), art>indef (indefinite Artikel), art>qu (interrogatve Artikel);
2. eine Liste mit den einzelnen Buchstaben der Vollform,
3. im Falle der Artikel die Formmerkmale, sonst eine leere Liste;
4. der semantische Typ der Vollform,
5. die Definitheit (Adverbien, Artikel), das Artmerkmal (Präpositionen) oder leer (Konjunktionen);
6. der λ-Term der Semantik,
7. die Grundform zur Vollform als Atom,
8. die Vollform als Atom;

Adverbien

```
vollform(adv>Sub, LForm, [], Typ, Def, Sem, AForm, AForm) :-
        form(adv>Sub, AForm, Typ, Sem),
        (Sub = qu -> Def = qu ; Def = ''),
        atom_chars(AForm, LForm).
```

Adverbien haben entweder eine interrogative Definitheit oder keine Definitheit.

Präpositionen

```
vollform(p>Sub, LForm, [], Typ, Art, Sem, AForm, AForm) :-
        form(p>Sub, AForm, Typ, Art, Sem),
        atom_chars(AForm, LForm).
```

Konjunktionen

```
vollform(junkt>Sub, LForm, [], Sub, '', Sem, AForm, AForm) :-
        form(junkt>Sub, AForm, Sem), atom_chars(AForm, LForm).
```

Artikel

```
vollform(art>Def, LForm, Formmerkmale, Def, Genus, Sem, Grundform,
    AForm) :-
        form(art>Def, Grundform, Genus, Formmerkmale, AForm, Sem),
        atom_chars(AForm, LForm).
```

Weitere Prädikate des Vollformenlexikons

Das Vollformenlexikon umfasst auch eine Reihe von Hilfsprädikaten. Die wichtigsten Prädikate seien kurz vorgestellt.

`zeige_formen/2`

```
zeige_formen(v>V, Infinitiv) :-
        vollform(v>V, _, Formmerk, _, _, Infinitiv, [AVerb, APartikel
            ]),
        write(Infinitiv), write('␣'), write(Formmerk), write(':␣'),
            write(AVerb),
        (APartikel \= '' -> write('␣'), write(APartikel); true),
        nl, fail.

zeige_formen(n>N, Grundform) :-
        vollform(n>N, _, Formmerk, _, _, _, _, Grundform, Vollform),
        write(Grundform), write('␣'), write(Formmerk), write(':␣'),
            write(Vollform),
        nl, fail.

zeige_formen(POS, Grundform) :-
        vollform(POS, _, Formmerk, _, _, _, Grundform, Vollform),
        write(Grundform), write('␣'), write(Formmerk), write(':␣'),
            write(Vollform),
        nl, fail.

zeige_formen(_, _).
```

Das Prädikat `zeige_formen/2` gibt alle Vollformen zu einer Wortart und/oder einer Grundform in übersichtlicher Darstellung auf der Standardausgabe aus.

Beispielaufrufe

Die folgenden Beispieleaufrufe von `zeige_formen/2` zeigen zum Beispiel die Vollformen der Nomen Raum und Herr, sowie die Vollformen der Verben stattfinden, halten und sein:

```
?- zeige_formen(_, 'Raum').
Raum [sg, nom]: Raum
Raum [sg, gen]: Raums
Raum [sg, dat]: Raum
Raum [sg, akk]: Raum
Raum [pl, nom]: Raeume
Raum [pl, gen]: Raeume
Raum [pl, dat]: Raeumen
Raum [pl, akk]: Raeume
true.

?- zeige_formen(_, 'Herr').
Herr [sg, nom]: Herr
```

```
Herr [sg, gen]: Herrn
Herr [sg, gen]: Herren
Herr [sg, dat]: Herrn
Herr [sg, dat]: Herren
Herr [sg, akk]: Herrn
Herr [sg, akk]: Herren
Herr [pl, nom]: Herren
Herr [pl, gen]: Herren
Herr [pl, dat]: Herren
Herr [pl, akk]: Herren
true.

?- zeige_formen(v>v, stattfinden).
stattfinden part2: stattgefunden
stattfinden inf: stattfinden
stattfinden [v1/v2, 1, sg, praes]: finde statt
stattfinden [v1/v2, 2, sg, praes]: findest statt
stattfinden [v1/v2, 3, sg, praes]: findet statt
stattfinden [v1/v2, 1, pl, praes]: finden statt
stattfinden [v1/v2, 2, pl, praes]: findet statt
stattfinden [v1/v2, 3, pl, praes]: finden statt
stattfinden [v1/v2, 1, sg, praet]: fand statt
stattfinden [v1/v2, 2, sg, praet]: fandest statt
stattfinden [v1/v2, 3, sg, praet]: fand statt
stattfinden [v1/v2, 1, pl, praet]: fanden statt
stattfinden [v1/v2, 2, pl, praet]: fandet statt
stattfinden [v1/v2, 3, pl, praet]: fanden statt
stattfinden [vl, 1, sg, praes]: stattfinde
stattfinden [vl, 2, sg, praes]: stattfindest
stattfinden [vl, 3, sg, praes]: stattfindet
stattfinden [vl, 1, pl, praes]: stattfinden
stattfinden [vl, 2, pl, praes]: stattfindet
stattfinden [vl, 3, pl, praes]: stattfinden
stattfinden [vl, 1, sg, praet]: stattfand
stattfinden [vl, 2, sg, praet]: stattfandest
stattfinden [vl, 3, sg, praet]: stattfand
stattfinden [vl, 1, pl, praet]: stattfanden
stattfinden [vl, 2, pl, praet]: stattfandet
stattfinden [vl, 3, pl, praet]: stattfanden
true.

?- zeige_formen(v>v, halten).
halten part2: gehalten
halten inf: halten
halten [v1/v2, 1, sg, praes]: halte
halten [v1/v2, 2, sg, praes]: haeltst
halten [v1/v2, 3, sg, praes]: haelt
halten [v1/v2, 1, pl, praes]: halten
halten [v1/v2, 2, pl, praes]: haltet
```

```
halten [v1/v2, 3, pl, praes]: halten
halten [v1/v2, 1, sg, praet]: hielt
halten [v1/v2, 2, sg, praet]: hieltest
halten [v1/v2, 3, sg, praet]: hielt
halten [v1/v2, 1, pl, praet]: hielten
halten [v1/v2, 2, pl, praet]: hieltet
halten [v1/v2, 3, pl, praet]: hielten
halten [vl, 1, sg, praes]: halte
halten [vl, 2, sg, praes]: haeltst
halten [vl, 3, sg, praes]: haelt
halten [vl, 1, pl, praes]: halten
halten [vl, 2, pl, praes]: haltet
halten [vl, 3, pl, praes]: halten
halten [vl, 1, sg, praet]: hielt
halten [vl, 2, sg, praet]: hieltest
halten [vl, 3, sg, praet]: hielt
halten [vl, 1, pl, praet]: hielten
halten [vl, 2, pl, praet]: hieltet
halten [vl, 3, pl, praet]: hielten
true.

?- zeige_formen(v>aux, sein).
sein part2: gewesen
sein inf: sein
sein [_G274, 1, sg, praes]: bin
sein [_G274, 2, sg, praes]: bist
sein [_G274, 3, sg, praes]: ist
sein [_G274, 1, pl, praes]: sind
sein [_G274, 3, pl, praes]: sind
sein [_G274, 2, pl, praes]: seid
sein [_G274, 1, sg, praet]: war
sein [_G274, 3, sg, praet]: war
sein [_G274, 2, sg, praet]: warst
sein [_G274, 1, pl, praet]: waren
sein [_G274, 3, pl, praet]: waren
sein [_G274, 2, pl, praet]: wart
true.
```

Ein weiteres wichtiges Prädikat ist `schreibe_vollformen/1`. Dieses Prädikat schreibt alle `vollform`-Fakten in von Prolog lesbarer Form in die Standardausgabe. Zusammen mit den Prädikaten `schreibe_vollformenlexikon(Datei)` und `schreibe_vollformenlexikon/0` dient es zur Erstellung und Abspeicherung eines Vollformenlexikons. Im Produktiveinsatz muss NASfVI mit einem solchen abgespeicherten Vollformenlexikon arbeiten, da das fortlaufende Erzeugen von Vollformen NASfVI bei der syntaktischen Analyse sehr stark verlangsamt.[2]

[2]So benötigt SWI-Prolog auf einem Testsystem unter gleichen Bedingungen für alle Tests der Test-Suite aus Anhang A.1 mit Verwendung des Grundformenlexikons zum Beispiel 53 Sekunden. Wird dagegen nur das erzeugte und in einer Datei abgespeicherte Vollformenlexikon verwendet, sinkt der Zeitbedarf

`schreibe_vollformen/1`

```
schreibe_vollformen(Art) :-
        (VF = vollform(Art, _, F, _, _, _, _)
                ; VF = vollform(Art, _, F, _, _, _, _, _, _)
                ; VF = vollform(Art, _, F, _, _, _, _, _)
        ),
        call(VF),
        numbervars(VF, 0, _, [singletons(true)]),
        ((F = [_, Cas], Cas \= gen)
                ; (F = [_, 3, _, _], F \= [vl, _, _, _])
                ; (F \= [_, _], F \= [_, _, _, _])
        ),
        write_term(VF, [ignore_ops(false),
        quoted(true), numbervars(true)]),
        writeln('.'), fail.

schreibe_vollformen(_).
```

Zu beachten ist, dass `schreibe_vollformen/1` das einzige Prädikat in NASfVI ist, welches in dieser Form nicht vollständig ISO-kompatibel ist. Denn das Prädikat `numbervars/4` ist nicht Teil des ISO-Standards [ISO13211-1], sondern ein für SWI-Prolog spezifisches Prädikat. Das Prädikat ist für die Funktion von `schreibe_vollformen/1` nicht zwingend erforderlich, vermeidet aber Prolog-Warnungen, wenn das erzeugte Vollformenlexikon aus einer Datei eingelesen wird.
`schreibe_vollformen/1` passt außerdem die Einträge des Vollformenlexikons an die Verwendung im modellierten Sprachfragment (Kapitel 2.2.2) an. Da das Sprachfragment keine Genitivformen verwendet und keine Verbletztstellung erlaubt, werden die entsprechenden Lexikoneinträge von `schreibe_vollformen/1` ignoriert. Dasselbe gilt für die Verbformen der ersten und zweiten Person. Denn für das Sprachfragment wird nur die dritte Person benötigt. Da die Flexion diese Formen jedoch erzeugen kann, ist es ohne Weiteres möglich, das Sprachfragment in diese Richtung zu erweitern, falls gewünscht.

3.1.4. Syntaktische Expansion der Valenz

Im Grundformenlexikon und im Falle der Verben auch im Vollformenlexikon besitzen die Valenz-tragenden Lexme eine syntaktisch-semantische Valenz. Die semantischen Bestandteile dieser Valenzrahmen können in der Syntax jedoch nicht direkt verwendet werden. Das Prädikat `expandiere_valenz/4` überführt die syntaktisch-semantische Valenz daher in einen Valenzrahmen, der rein aus syntaktischen Phrasen besteht. Dabei werden semantische Angaben wie Agens oder Patiens in Abhängigkeit des Genus Verbi durch passende syntaktische Phrasen wie Nominalphrasen oder Präpositionalphrasen ersetzt. Anders als die syntaktisch-semantische Eingabe, welche eine einfache Liste ist, gibt `expandiere_valenz/4` bei der rein syntaktischen Ausgabe eine Differenzliste aus. Dies ermöglicht eine effiziente Verknüpfung von verschiedenen Valenzanforderungen, um auf Satzebene eine „Gesamtvalenz“ zu berechnen.

für alle Tests auf 7 Sekunden.

expandiere_valenz/4

```
expandiere_valenz(GenusVerbi, SynSemValenz, SynValenz, FormSubj)
:- expandiere_valenz(GenusVerbi, SynSemValenz, SynValenz, FormSubj,
    anf(A,B), gef(C,D), _SubjRolle),
(
        (nonvar(A), A = 0)
                -> (var(C) ; C = 0)
                ; true
),
(
        (nonvar(A), A = 1)
                -> nonvar(C), A = C
                ; true
),
(
        (nonvar(B), B = 1)
                -> nonvar(D), B = D
                ; true
), !.
```

`expandiere_valenz/4` erwartet vier Argumente. Das erste Argument gibt das Genus Verbi an, welches für die Expansion der syntaktisch-semantischen Valenz berücksichtigt wird. Es hat entweder den Wert `aktiv` oder den Wert `passiv`. Die Liste der syntaktisch-semantischen Valenz ist das zweite Argument. Das dritte Argument ist die Ausgabe die Differenzliste, die aus rein syntaktischen Phrasen besteht. Das letzte Argument beinhaltet die Formmerkmale des Subjekts. Durch Unifikation mit diesen Formmerkmalen kann die Kongruenz zwischen Subjekt und finitem Verb garantiert werden. Das ermöglicht, dass ein finites Verb in seinem syntaktischen Valenzrahmen bereits die Formmerkmale des Subjekts vorgeben kann und den möglichen Inhalt eines zu analysierenden Satzes damit vorstrukturieren kann.
Die internen Terme `anf/2` und `gef/2` beschreiben Wohlgeformtheitsbedingungen, welche `expandiere_valenz/4` nach erfolgter Expansion überprüft. Das erste Argument der beiden Terme ist ein boolescher Wert für das Vorhandensein eines Agens und das zweite Argument ein boolescher Wert für das Vorhandensein eines Patiens in der Valenzangabe. Diese Kombinationen beeinflussen, wie die semantischen Angaben expandiert werden. Ein Patiens realisiert zum Beispiel das Subjekt, wenn kein Agens vorhanden ist, aber ein Akkusativobjekt, wenn ein Agens vorhanden ist. Die Werte in `anf/2` entsprechen dem Vorkommen in der syntaktisch-semantischen Valenz, während die Werte in `gef/2` dem Vorkommen in der expandierten, syntaktischen Valenz entsprechen. `expandiere_valenz/4` überprüft im Wesentlichen, ob die Angaben jeweils vereinbar sind und verhindert so, dass ein Patiens zum Beispiel als Subjekt expandiert wird, obwohl ein Agens vorhanden ist.

expandiere_valenz/7

```
expandiere_valenz(_, [], D-D, _, _, _, _) :- var(D), !.
```

```
expandiere_valenz(GenusVerbi, [H|T], [H2|T2]-D, FormSubj, Anf, Gef,
    SubjRolle) :-
        expandiere_rolle(GenusVerbi, H, H2, FormSubj, Anf, Gef,
            SubjRolle),
        expandiere_valenz(GenusVerbi, T, T2-D, FormSubj, Anf, Gef,
            SubjRolle), !.
```

Das Hilfsprädikat `expandiere_valenz/7` übersetzt jedes Element der syntaktisch-semantischen Valenz mit `expandiere_rolle/7` in eine syntaktische Phrase der expandierten, syntaktischen Valenz. Es erzeugt zudem die Differenzliste für die syntaktische Valenz. `expandiere_rolle/7` arbeitet mit sieben Argumenten: dem Genus Verbi, der zu expandieren semantischen Rolle, der expandierten semantischen Phrase, den Formmerkmalen des Subjekts, den Termen `anf/2` und `gef/2`, sowie einer Angabe, welche semantische Rolle als Subjekt expandiert. Unter semantischen Rolle werden die semantischen Angaben in der syntaktisch-semantischen Valenz verstanden. Diese im Grundformenlexikon (Kapitel 3.1.1) angegebenen Rollen können die Werte `agens, patiens, expletiv, thema` und `loc` sein.

Expansion des Agens

```
expandiere_rolle(aktiv,
        agens(Typ, Sem), [np, [Pers, Num, nom], Typ, _, _, Sem, _], [
            Pers, Num],
        anf(1,_), gef(1,_), agens).

expandiere_rolle(passiv,
        agens(Typ, Sem), ?[pp, [dat], Typ, _, Sem, _], _,
        anf(1,1), gef(1,_), patiens).
```

Im Aktiv wird das Agens immer als Subjekt-Nominalphrase im Nominativ expandiert, im Passiv dagegen immer als Dativ-Präpositionalphrase. Die Phrasen werden in der Syntax durch eine Liste ihrer Eigenschaften gekennzeichnet. Zu beachten ist, dass die Agensphrase im Passiv fakultativ ist. Sie muss im Passiv also nicht realisiert werden und kann in allen Fällen in der Syntax auch weggelassen werden. Durch diese Expansion sind damit in der Syntax zum Beispiel Sätze wie *Die Vorlesung Syntax wird von einem Dozenten gehalten* und auch Sätze wie *Wird die Vorlesung Syntax gehalten* möglich.

Expansion des Patiens im Aktiv

```
expandiere_rolle(aktiv,
        patiens(Typ, Sem), [np, [Pers, Num, nom], Typ, _, _, Sem, _],
            [Pers, Num],
        anf(0,1), gef(_,1), patiens).

expandiere_rolle(aktiv,
        patiens(Typ, Sem), [np, [_, _, akk], Typ, _, _, Sem, _], _,
        anf(1,1), gef(_,1), _).
```

Ist kein Agens vorhanden, das Verb also intransitiv, wird das Patiens im Aktiv als Subjekt-Nominalphrase expandiert. Ist dagegen ein Agens vorhanden und das Verb damit transitiv, ist nur eine Expansion als Akkusativobjekt möglich.

Expansion des Patiens im Passiv

```
expandiere_rolle(passiv,
        patiens(Typ, Sem), [np, [Pers, Num, nom], Typ, _, _, Sem, _],
             [Pers, Num],
        anf(1,1), gef(_,1), patiens).
```

Im Passiv kann das Patiens nur dann expandiert werden, wenn das Verb transitiv und damit ein Agens vorhanden ist. Ein intransitives Verb ohne Agens ist durch diese Bedingung nicht passivierbar.

Expansion des Expletivs

```
expandiere_rolle(aktiv,
        expletiv, [np, [3, sg, nom], expl, expl, _, '', _], [3, sg],
        anf(1,_), gef(1,_), expletiv).
```

Die Rolle `expletiv` ist nur als Nominalphrase und als Subjekt im Aktiv expandierbar. Bei dieser Rolle handelt es sich linguistisch nicht um eine semantische Rolle, sondern sie ist viel mehr nur eine Kurzschreibweise für die expletive Nominalphrase in NASfVI im Rahmen der Valenz.

Expansion der weiteren Rollen

```
expandiere_rolle(_, thema(Cas, Sem),
        [pp, [Cas], thema, _, Sem, _], _, _, _, _).
expandiere_rolle(_, loc(Sem),
        [_, [dat], loc, _, Sem, _], _, _, _, _).
expandiere_rolle(_, temp_d(Sem),
        [_, [dat], temp_d, _, Sem, _], _, _, _, _).

expandiere_rolle(GV, ?X, ?Y, FormSubj, Anf, Gef, SubjRolle) :-
        expandiere_rolle(GV, X, Y, FormSubj, Anf, Gef, SubjRolle).
```

Themaergänzungen werden immer als Präpositionalphrasen expandiert. Orts- und Wochentagsangaben werden dagegen in der Form von Präpositionalphrasen und Adverbialphrasen expandiert. Das bedeutet, dass die Liste zur Identifikation der Phrase nur dem Format dieser beiden Phrasentypen entspricht. Dabei ist jedoch nicht festgelegt, welche der beiden Phrasen letztlich der Fall ist. So kann zum Beispiel die Ortsangabe als Präpositionalphrase *in welchem Raum* aber auch als Adverbialphrase *wo* umgesetzt werden.

3.1.5. Lexikonzugriff: die lex-Prädikate

Das Lexikon umfasst die Vollformen des Sprachfragments und wird zentral über die Prädikate `lex/7`, `lex/8` und `lex/9` angesprochen. Diese `lex`-Prädikate entsprechen den `vollform`-Prädikaten des Vollformenlexikons (Kapitel 3.1.3) und greifen auch direkt auf diese zu. Dabei können die `vollform`-Prädikate sowohl dynamisch zur Anfragezeit gebildet werden oder auch aus einer statischen Datei stammen. In `lex` werden diese Vollformen ausgelesen und entweder unverändert an den Aufrufer von `lex` weitergegeben oder es wird aufgrund einer Eingabe eine Präfixerkennung mit den Vollformen durchgeführt. Diese Präfixerkennung, welche die Suggest-Funktionalität auf Wortebene umsetzt, ist die

zentrale Aufgabe der `lex`-Prädikate. Ist das Prolog-Flag `grammatik_test` auf den Wert `on` gesetzt, lösen die `lex`-Prädikate zudem bei unbekannten Wörtern eine Exception aus. Unbekannte Wörter sind in diesem Zusammenhang Token als Eingaben, die nicht auf die Vollformen des Lexikons abgebildet werden können. Die in dieser Situation ausgelöste Exception berichtet das nicht lexikalisch analysierbare Token zur Ermöglichung einer Fehleranalyse. Im Produktiveinsatz werden die Exceptions jedoch nicht ausgelöst, da das Flag `grammatik_test` nicht gesetzt ist und damit das Backtracking von Prolog etwas beschleunigt werden soll.
Die Präfixerkennung wird immer dann ermöglicht, wenn das zu analysierende Token während der Tokenisierung (siehe Anhang A.2) als vervollständigbar markiert wurde. Diese Markierung liegt vor, wenn das erste Zeichen des Tokens ein `#` ist. Da ein solches Zeichen nicht den Normalisierungsanforderungen entspricht, kann es nur in der Tokenisierung als Markierung gesetzt worden sein. Eine Präfixerkennung wird desweiteren nur durchgeführt, wenn das gegebene Präfix keiner lexikalischen Vollform entspricht.
Die Erkennung der Präfixe ist dabei nicht transparent. Das heißt, ein Aufrufer der `lex`-Prädikate kann und soll nicht erkennen, ob ein Token als Präfix einer Vollform oder als eine exakte Vollform erkannt worden ist.

lex/9: Nomen und Pronomen

Das Prädikat `lex/9` ist für die Abfrage von Nomen und Pronomen im Lexikon zuständig. Die neun Argumente des Prädikats entsprechen exakt den in Kapitel 3.1.3 erläuterten Argumenten von `vollform/9`. `lex/9` verwendet das Prädikat `lex_/9` für die eigentliche Erkennung und Verarbeitung der Token und ruft bei einem nicht durch `lex_/9` analysierbarem Token `existiert_nicht/3` auf, welches dann gegebenfalls eine Exception auslöst. Die funktionale Aufspaltung in die beiden Prädikate `lex/9` und `lex_/9` verhindert eine unendliche Rekursion im Zusammenhang mit `existiert_nicht/3`. Denn nur `lex/9` kann `existiert_nicht/3` aufrufen. Da `lex_/9` dies nicht tut, kann `existiert_nicht/3` problemlos `lex_/9` aufrufen, um zu überprüfen, ob ein Token im Lexikon vorhanden ist.

Einfache Nomen

```
lex(n>n, LVollform, Formmerkmale, Typ, Genus, Val, Sem, Grundform,
    AVollform) :-
        lex_(n>Sub, LVollform, Formmerkmale, Typ, Genus, Val, Sem_,
            Grundform, AVollform),
        ((Sub = app)
                -> Sem = Sem_ * ' '
                ; Sem = Sem_
        ).
```

Diese Regel schlägt einfache Nomen nach. Dabei können diese entweder direkt als n>n im Vollformenlexikon gespeichert sein, oder auch als n>app. In letzterem Fall wird die Semantik der Apposition durch Anwendung eines leeren Atoms auf die Semantik funktional der Semantik der einfachen Nomen angepasst. Die semantisch zweistellige Apposition hat daraufhin ein leeres Atom als Wert für die eigentlich geforderte Blackbox und unterscheidet sich in der Handhabung nicht mehr von gewöhnlichen einfachen Nomen. Da sich die

Möglichkeit einer Blackbox nur aus der Wortart ergibt und sich nicht in einer eventuellen Nomenvalenz widerspiegelt, müssen keine weiteren Anpassungen durchgeführt werden.

Appositionen, Pronomen, Aufruf von existiert_nicht/3

```
lex(n>app, LVollform, Formmerkmale, Typ, Genus, Val, Sem, Grundform,
    AVollform) :-
        lex_(n>app, LVollform, Formmerkmale, Typ, Genus, Val, Sem,
            Grundform, AVollform).

lex(pro>Sub, LVollform, Formmerkmale, Typ, Genus, Val, Sem, Grundform
    , AVollform) :-
        lex_(pro>Sub, LVollform, Formmerkmale, Typ, Genus, Val, Sem,
            Grundform, AVollform).

lex(POS, LVollform, _, _, _, _, _, _, AVollform)
        :- existiert_nicht(POS-lex/9, LVollform, AVollform).
```

Die Regeln von `lex_/9` versuchen Präfixe zu erkennen oder die Token exakt auf Vollformen abzubilden. Zunächst wird versucht auch als ergänzbar markierte Token exakt im Vollformenlexikon zu finden:

```
lex_(POS, ['#'|LVollform], Formmerkmale, Typ, Genus, Val, Sem,
    Grundform, AVollform) :-
        (POS = n>_ ; POS = pro>_),
        nonvar(LVollform), LVollform \= [],
        vollform(POS, LVollform, Formmerkmale, Typ, Genus, Val, Sem,
            Grundform, AVollform).
```

Schlägt diese exakte Suche fehl, wird die Präfixerkennung bemüht:

```
lex_(POS, ['#'|LVollform], Formmerkmale, Typ, Genus, Val, Sem,
    Grundform, AVollform) :-
        (POS = n>_ ; POS = pro>_),
        nonvar(LVollform), LVollform \= [],
        \+vollform(POS, LVollform, Formmerkmale, Typ, Genus, Val, Sem
            , Grundform, AVollform),
        vollform(POS, LVollform_, Formmerkmale, Typ, Genus, Val, Sem,
             Grundform, AVollform),
        praefix_von(LVollform, LVollform_).
```

Bevor mögliche Vollform-Kandidaten für die Präfixerkennung ausgewählt werden, wird sichergestellt, dass kein exakter Treffer im Vollformenlexikon existiert. Diese Vorgehensweise mit dem negierten Aufruf von `vollform/9` ist notwendig, da die kürzere und effizientere Programmierung mit Hilfe eines Cuts nicht möglich ist, ohne Alternativen bei ambigen Vollformen zu blockieren. Das Prädikat `praefix_von/2` schließlich überprüft, ob LVollform, das zu analysierende Token, ein Präfix von LVollform_, einer lexikalischen Vollform, ist.

Exakte Suche in `lex_/9` bei nicht ergänzbaren Token

```
lex_(POS, LVollform, Formmerkmale, Typ, Genus, Val, Sem, Grundform,
    AVollform) :-
```

```
        (POS = n>_ ; POS = pro>_),
        vollform(POS, LVollform, Formmerkmale, Typ, Genus, Val, Sem,
            Grundform, AVollform).
```

Token, die nicht als ergänzbar markiert sind, werden nur exakt auf die Vollformen abgebildet.

lex/7: Verben

Wie auch die `vollform/7`-Einträge in Kapitel 3.1.3 unterscheidet auch `lex/7` zwischen finiten und infiniten Verbformen. Die Unterscheidung zwischen Vollverben und Hilfsverben ist durch die Kapselung bei den `vollform/7`-Einträgen nicht mehr notwendig.
Die in Kapitel 3.1.4 beschriebene Expansion der syntaktisch-semantischen Valenz zu einer rein syntaktischen Valenz wird bei Verben in dem Prädikat `lex/7` durchgeführt.

Infinite Verbformen

```
lex(v>Sub, Zeichenlisten, [GenusVerbi, Form, [Pers, Num]], Val, Sem,
    Infinitiv, Atome) :-
        indefinite(Form),
        lex_(v>Sub, Zeichenlisten, Form, Val_, Sem, Infinitiv, Atome)
            ,
        genusverbi(GenusVerbi),
        expandiere_valenz(GenusVerbi, Val_, Val, [Pers, Num]).
```

Die Formmerkmale bei den infiniten Formen sind bei `lex/7` anders als in den bisher diskutierten Prädikaten. Die Liste der Formmerkmale besteht hier aus dem Genus Verbi, inf für den Infinitiv oder part2 für das Partizip II und das letztes Element der Liste ist eine eingebette Liste mit zwei Elementen. Obgleich bei indefiniten Verbformen nicht vorbelegt, sind diese zwei Elemente die Person und der Numerus für die Unifikation mit dem Subjekt.

Finite Verbformen

```
lex(v>Sub, Zeichenlisten, [GenusVerbi, Vst, Pers, Num, Temp], Val,
    Sem, Infinitiv, Atome) :-
        verbstellung(Vst, Vst_),
        lex_(v>Sub, Zeichenlisten, [Vst_, Pers, Num, Temp], Val_, Sem
            , Infinitiv, Atome),
        genusverbi(GenusVerbi),
        expandiere_valenz(GenusVerbi, Val_, Val, [Pers, Num]).
```

Aufruf von `existiert_nicht/3`

```
lex(v>Sub, [LVerb, _], _, _, _, _, [AVerb, _])
        :- existiert_nicht(v>Sub-lex/7, LVerb, AVerb).
```

Die Regeln von `lex_/7` sind analog zur Funktionalität der `lex/9`-Prädikate:

```
lex_(v>Sub, [['#'|LVerb], LPartikel], Formmerkmale, Val, Sem,
    Infinitiv, Atome) :-
```

```
        nonvar(LVerb), LVerb \= [],
        vollform(v>Sub, [LVerb, LPartikel], Formmerkmale, Val, Sem,
            Infinitiv, Atome).

lex_(v>Sub, [['#'|LVerb], LPartikel], Formmerkmale, Val, Sem,
    Infinitiv, Atome) :-
        nonvar(LVerb), LVerb \= [],
        \+vollform(v>Sub, [LVerb, LPartikel], Formmerkmale, Val, Sem,
             Infinitiv, Atome),
        vollform(v>Sub, [LVerb_, LPartikel], Formmerkmale, Val, Sem,
            Infinitiv, Atome),
        praefix_von(LVerb, LVerb_).

lex_(v>Sub, Zeichenlisten, Formmerkmale, Val, Sem, Infinitiv, Atom)
        :- vollform(v>Sub, Zeichenlisten, Formmerkmale, Val, Sem,
            Infinitiv, Atom).
```

lex/8: Adverbien, Präpositionen, Junktoren, Artikel

Die Prädikate `lex/8` und `lex_/8` weisen keine Besonderheiten auf und sind in ihrer Funktion völlig analog zu den bisher diskutierten `lex`-Prädikaten und werden daher an dieser Stelle nicht separat diskutiert.

3.2. Syntax

In der Syntax werden in NASfVI die Wortformen zu Konstituenten oder Phrasen zusammengefasst, Abfolgen von Phrasen als Felder analysiert und ganze Sätze auf Grundlage des Feldermodells des Deutschen (Kapitel 2.2.1) verarbeitet.

3.2.1. Eine Frage der Anfrage

Die Sprachverarbeitungskomponente verfügt über drei verschiedene Anfragemöglichkeiten:

- Parse-Anfragen: Bei diesen Anfragen werden bei Eingaben keine Präfixe zu Vollformen ergänzt, auch Sätze müssen vollständig vorliegen; Eingaben werden genau so analysiert, wie sie vorliegen;
- Suggest-Anfragen: Bei Suggest-Anfragen werden Eingaben vervollständigt, Präfixe werden zu Vollformen und Satzanfänge zu fertigen Sätzen vervollständigt; Die Ausgabe ist keine Satzanalyse, sondern eine Liste von Sätzen, die mit der bisherigen Eingabe als Präfix möglich sind;
- Beantworte-Anfragen: Diese Anfrage verarbeitet eine Eingabe zunächst als Parse-Anfrage und generiert dann mit gegebenen Antwort-Werten eine natürlichsprachige Antwort, sowie eine Satzanalyse der Eingabe und eine Satzanalyse der Antwort als Ausgabe;

Während die Parse- und Beantworte-Anfragen keine besonderen Anforderungen an die Grammatikregeln der Syntax stellen, müssen bei Suggest-Anfragen verschiedene Dinge beachtet werden. Bei dieser Anfrageart müssen Platzhalter für Eigennamen und Raumangaben generiert werden, falls diese in einem Satz erzeugt werden, aber durch die Eingabe noch nicht vorgegeben sind. Auch potentiell unendliche rekursive Strukturen, wie der Koordination, müssen an geeigneter Stelle gestoppt werden und sollten nicht unendlich generieren.
Um diesen Anforderungen gerecht zu werden, müssen die Reglen daher in die Lage versetzt werden, festzustellen, unter welchen Bedingungen sie arbeiten. Wird die Sprachverarbeitungskomponente über die in Kapitel 3.5 beschriebenen Anfrageprädikate aufgerufen, dann setzen oder entfernen diese das Faktum `suggest_modus(aus)` in der Prolog-Datenbank. Ist es gesetzt, dann wird die Analyse *nicht* im Suggest-Modus ausgeführt. Ist dagegen `suggest_modus(an)` gesetzt, werden alle Analysen im Suggest-Modus ausgeführt. Da der Suggest-Modus *während* Benutzereingaben ausgelöst wird und nicht nur an deren Ende, ist der Suggest-Modus der insgesamt am häufigsten verwendete Modus.
In der Sprachverarbeitungskomponente werden zwei Prädikate `deaktiviere_suggest/0` und `aktiviere_suggest/0` definiert, um die Sprachverarbeitung in den Suggest-Modus versetzen zu können oder diesen zu beenden:

```
deaktiviere_suggest :- suggest_modus(aus), !.
deaktiviere_suggest :- retract(suggest_modus(_)),
        asserta(suggest_modus(aus)).

aktiviere_suggest :- suggest_modus(an), !.
aktiviere_suggest :- retract(suggest_modus(_)),
        asserta(suggest_modus(an)).
```

Die Grammatikregeln liegen als *Definite Clause Grammar*-Regeln (DCG) vor. Damit die Regeln innerhalb des DCG-Formalismus abfragen können, ob das nächste Token im Rahmen des Suggest-Modus frei generiert wird, gibt es die DCG-Regel `generierung(suggest)`. Diese Regel ist sehr wichtig. Denn Beschränkungen im Suggest-Modus sind nur bei der freien Generierung von Phrasen und Wortformen notwendig. Ist eine Konstituente dagegen vollständig oder teilweise instanziiert, d. h. mindestens ein Zeichen der Konstituente ist durch die Eingabe schon vorgegeben, dann sollen grundsätzlich alle Analysemöglichkeiten zur Verfügung stehen.

```
generierung(suggest) --> {suggest_modus(aus), !, fail}.
generierung(suggest) --> wird_generiert.

wird_generiert --> [VF], {nonvar(VF), !, fail}.
wird_generiert --> {true}.
```

Die Regel `wird_generiert/0` schlägt fehl, wenn ein vorhandenes Token aus der Eingabe gelesen werden kann. Durch diesen kleinen Trick, der ausnutzt, dass Prolog bei fehlgeschlagenen Aufrufen alle Variabelbindungen des Aufrufs rückgängig macht, ist sichergestellt, dass `wird_generiert/0` und damit auch `generierung(suggest)` kein Token von dem Eingabestrom entfernen und ihn somit nicht verändern.

3.2.2. DCG-Regeln für Token

DCG-Regeln für den Lexikonzugriff

Um das Lexikon in die Arbeit mit den DCG-Regeln einzubetten, wird ein Zugriff auf die `lex`-Prädikate mittels DCG-Regeln definiert. Die DCG-Regeln für Artikel, Präpositionen, Adverbien, Koordinationen, Nomen und Pronomen lesen jeweils ein Token vom Eingabestrom und übergeben es dem jeweiligen `lex`-Prädikat.

DCG-Regel für Artikel, Präpositionen und Adverbien

```
t(Info, Info)
--> {Info = [POS, Formmerkmale, Typ, Artmerkmal, Semantik, Grundform,
     Atom]},
[VF], {lex(POS, VF, Formmerkmale, Typ, Artmerkmal, Semantik,
    Grundform, Atom)}.
```

Die Regeln sind zweistellig, um die Arbeit mit ihnen zu vereinfachen. Auf diese Weise können als erstes Argument direkt im Aufruf der Regel morphosyntaktische oder semantische Vorgaben für das zu erkennende Token gemacht werden, während das zweite Argument die vollständige Beschreibung des Tokens zurückgibt und vom Aufrufer verwendet werden kann.

DCG-Regeln für Nomen

```
n([n>app, A, hum, mask, C, D, 'Anrede', '(Anrede)'], [n>app, A, hum,
    mask, C, D, 'Anrede', '(Anrede)'])
--> generierung(suggest), !.

n(Info, Info)
--> {Info = [n>N, Formmerkmale, Typ, Genus, Val, Semantik, Grundform,
     Atom]},
[VF], {lex(n>N, VF, Formmerkmale, Typ, Genus, Val, Semantik,
    Grundform, Atom)}.
```

Die erste DCG-Regel für Nomen macht von dem Prädikat `generierung(suggest)` Gebrauch, und definiert, dass Appositionen mit dem semantischen Typ *hum* bei freier Generierung den Platzhalter *(Anrede)* erzeugen und maskulin sind. Die Angabe des Genus bewirkt, dass der generierte Satz für die generierte Apposition das generische Genus benutzt ([DUG06], 236).

DCG-Regel für Pronomen

```
pn(Info, Info)
--> {Info = [pro>Sub, Formmerkmale, Typ, '-', Val, Semantik,
    Grundform, Atom]},
[VF], {lex(pro>Sub, VF, Formmerkmale, Typ, '-', Val, Semantik,
    Grundform, Atom)}.
```

DCG-Regeln für Verben

```
v(Zeichenlisten, Formmerkmale, Val, Semantik, Infinitiv, Atome)
--> {lex(v>v, Zeichenlisten, Formmerkmale, Val, Semantik, Infinitiv,
    Atome)}.

v_aux(Infinitiv, Formmerkmale, Atom)
--> [Liste], {lex(v>aux, [Liste, []], Formmerkmale, _-_, '',
    Infinitiv, [Atom, ''])}.
```

Nachdem Vollverben und Hilfsverben in den `lex`-Prädikaten einheitlich verarbeitet werden konnten, ist für die Syntax nun wieder eine Ausdifferenzierung notwendig. Denn die Vollverben werden in NASfVI besonders behandelt. Anders als die Hilfsverben tragen die Vollverben den Hauptbestandteil der Semantik eines Satzes und strukturieren den Satz weitgehend vor. Da die Analyse eines Satzes in NASfVI nach der linken Satzklammer eine Verbvalenz benötigt, das Vollverb aber auch erst in der rechten Satzklammer stehen kann, liest die DCG-Regel `v/6` kein Token. Dadurch ist es möglich, die Regel auch ohne Vorgabe eines Vollverbs in der linken Satzklammer zu nutzen, um ein beliebiges Vollverb für die weitere Analyse auszuwählen. Dabei handelt es sich um einen Kompromiss, um die valenzgebundene Satzanalyse auch in diesen Fällen aufrecht erhalten zu können.

Die Blackbox

Eine Blackbox ist in NASfVI ein Bereich, der in seinen Bestandteilen nicht morphosyntaktisch oder lexikalisch analysiert wird. Im tokenisierten Eingabestrom besteht eine Blackbox in allen Fällen immer aus nur genau einem Token. Das stellt die in Anhang A.2 beschriebene Tokenisierung sicher.

Blackbox-Einstiegsregel

```
b(Info, Info)
--> {Info = [blackbox, Typ, Token]}, blackbox(Typ, Token).
```

Mit der Fähigkeit Blackbox-Token bilden zu können, kann NASfVI auch Eigennamen, Veranstaltungstitel, Thema-Angaben und Raumangaben verarbeiten. Diese Angaben weisen alle innere Strukturen auf, die sehr vielfältig sein können. Die Grammatik verwendet immer dann eine Blackbox, wenn die innere Struktur nicht ohne Weiteres analysiert weden kann und wenn sie auch nicht analysiert werden muss. Aus diesem Grund wird eine Blackbox semantisch auch immer als einfaches Prolog-Atom verarbeitet. Mit Ausnahme von Veranstaltungstiteln geht einer Blackbox immer ein als Blackbox-fähig ausgezeichnetes Lexem voran. Auf diese Weise kann eine Blackbox nur dann auftreten, wenn eine entsprechende Wortform dies ermöglicht. Geht kein geeignetes Wort einer Blackbox innerhalb einer Phrase voran, kann die Blackbox nur als Nominalphrase bestehend aus einem Veranstaltungstitel analysiert werden. Eine weitere Einschränkung bei der Analyse einer Blackbox ist, dass eine Blackbox niemals einer lexikalischen Wortform entsprechen darf. Ein Token kann daher nur als Blackbox analysiert werden, wenn keine gleichlautende Vollform im Lexikon existiert.

```
blackbox(_, Token) --> \+generierung(suggest), [X],
        {nonvar(X), !, nicht_lexikalisch(X), atom_chars(TokA, X),
        atom_concat('"', TokA, Tok_), atom_concat(Tok_, '"', Token)}.

blackbox(_, Token) --> \+generierung(suggest),
        {nonvar(Token), !, atom_concat(Tok_, '"', Token), atom_concat
            ('"', TokA, Tok_),
        atom_chars(TokA, X), nicht_lexikalisch(X)}, [X].
```

Die erste `blackbox`-Regel versucht ein vorhandenes Token auf dem Eingabestrom als Blackbox zu erkennen. Gelingt dies nicht, überprüft die zweite `blackbox`-Regel, ob ein Atom Token bereits als Blackbox gegeben ist. Dies ist immer dann der Fall, wenn eine Blackbox aufgrund einer atomaren „Semantik“ generiert werden soll. Die „Semantik“ einer Blackbox ist der gelesene Wert in Anführungszeichen. Bei der Eingabe sind Anführungszeichen nur notwendig, wenn sich die Blackbox über mehrere durch Leerzeichen getrennte Worte erstecken soll. Bei der Analyse werden die Anführungszeichen jedoch immer hinzugefügt. Der semantische Typ der Blackbox spielt bei den gezeigten Regeln keine Rolle.

```
blackbox(_, _) --> \+generierung(suggest), {!}.
```

Diese Regel wird nur erreicht, wenn der Suggest-Modus deaktiviert ist und sich weder ein Token auf dem Eingabestrom befindet, noch die „Semantik“ der Blackbox instanziiert ist. Dies ist nur bei der Generierung natürlichsprachiger Antwort der Fall, wo solche völlig freien Blackbox-Phrasen erzeugt und dann durch Unifikation in einem Aufruf von `member/2` mit einem Wert belegt werden. Diese Blackbox, die nur über ihren atomaren Wert - also die „Semantik“ - verfügt, wird bei der Satzgenerierung dann von der vorangegangenen, zweiten `blackbox`-Regel verarbeitet.

```
blackbox(hum, '(Nachname)') --> [[]].
blackbox(event, '(Titel)') --> [[]].
blackbox(thema, '(Thema)') --> [[]].
blackbox(loc, '(Raum)') --> [[]].
```

Befindet sich die Grammatik im Suggest-Modus und ist die Blackbox nicht instanziiert, dann werden diese Platzhalter in Abhängigkeit vom geforderten semantischen Typ generiert.

Semesterangaben

NASfVI kann die Angaben für Sommer- und Wintersemester verarbeiten, so lange sie einem definiertem Format folgen. Dieses Format besagt, dass die Angabe für ein Sommersemester aus einer Jahreszahl und die Angabe für ein Wintersemester aus zwei Jahreszahlen besteht. Die Jahreszahlen bei dem Wintersemester müssen durch das Zeichen „/“ getrennt sein. Das Format erlaubt, dass die Jahreszahlen sowohl zweistellig, als auch vierstellig sein können.

Semesterangaben-Einstiegsregel

```
a(Info, Info) --> {Info = [angabe, Typ, Token]}, angabe(Typ, Token).
```

Regeln für Sommersemester

```
angabe(semester>sose, '2011') --> generierung(suggest), {!}.
angabe(semester>sose, Atom) --> {nonvar(Atom),
        atom_chars(Atom, Zs)}, jahr(Zs-[], _), {!}.
angabe(semester>sose, Atom) --> jahr(Zs-[], _),
        {atom_chars(Atom, Zs)}.
```

Je nachdem ob `Atom` instanziiert oder eine freie Variabel ist, wird eine andere Reihenfolge bei der Verarbeitung gewählt. Das ist notwendig, da `atom_chars/2` mindestens ein instanziiertes Argument verlangt.

Regeln für Wintersemester

```
angabe(semester>wise, '2011/2012') --> generierung(suggest), {!}.
angabe(semester>wise, Atom) --> {nonvar(Atom),
        atom_chars(Atom, A)}, jahr(A-['/'|B], NumA), [['/']],
        jahr(B-[], NumB), {NumB is NumA + 1, !}.
angabe(semester>wise, Atom) --> jahr(A-['/'|B], NumA), [['/']],
        jahr(B-[], NumB), {atom_chars(Atom, A), NumB is NumA + 1}.
```

Auch bei den Regeln für Wintersemester hängt die Reihenfolge der Verarbeitung jeweils davon ab, ob `Atom` instanziiert oder eine freie Variabel ist.

Regel für beliebige Semester

```
angabe(semester>sem, Atom)
--> angabe(semester>sose, Atom) ; angabe(semester>wise, Atom).
```

Format einer Jahreszahl

```
jahr(['2', '0', Z1, Z2|D]-D, Num) --> [J],
        {praefix_von(J, ['2', '0', Z1, Z2]), !, ziffer(Z1),
        ziffer(Z2), number_chars(Num, ['2','0',Z1,Z2])}.

jahr(['2', '0', Z1, Z2|D]-D, Num) --> [J],
        {praefix_von(J, [Z1, Z2]), ziffer(Z1), ziffer(Z2),
        number_chars(Num, ['2','0',Z1,Z2])}.
```

Die erste `jahr`-Regel erkennt und erzeugt vierstellige Jahreszahlen, die zweite Regel erkennt zweistellige Jahreszahlen und bildet sie intern auf vierstellig Jahreszahlen ab. Wenn die Jahreszahl mit einer zwei beginnt, ist wegen des Cuts nur die erste Regel anwendbar.

3.2.3. Phrasen

Phrasentypen

Es gibt in NASfVI insgesamt fünf verschiedene Phrasentypen:

- Nominalphrasen (np)
- Präpositionalphrasen (pp)
- Adverbialphrasen (avdp)

- Koordinierte Nominalphrasen (cnp)
- Koordinierte Präpositionalphrasen (cpp)

Jeder Phrasentyp wird durch eine Liste beschrieben. Diese Listen enthalten als erstes Element die Abkürzung der Phrase und dann vom Phrasentyp abhängige weitere Elemente. In allen Fällen jedoch ist neben dem Phrasentyp auch der semantische Typ der Phrase, die Semantik der Phrase und eine Liste mit der Strukturanalyse der Phrase enthalten. Die Strukturanalyse besteht dabei aus einer Liste mit Beschreibungen von Token oder mit weiteren Strukturanalysen von Phrasen, die der aktuellen Phrase untergeordnet sind.

Verallgemeinerte Phrase

Die verallgemeinerte Phrase zeichnet sich dadurch aus, dass sie alle Phrasentypen erkennen kann. Anders als die DCG-Regeln der Phrasen selbst ist bei der verallgemeinerten Phrase zudem das erste Argument eine Differenzliste. Während beim Schreiben der Phrasenregeln auf keine Differenzliste bei den Beschreibungen Rücksicht genommen werden muss, erzeugt die verallgemeinerte Phrase für jede Phrase auch eine Differenzliste. Mit dieser Differenzliste können in den Feldern die einzelnen Phrasen effizient verkettet werden. Das zweite Argument ist eine Differenzliste mit der linearen, flachen Abfolge der Atome, die die Phrase enthält.

```
obj([Beschreibung|D]-D, Atome-D2) -->
        (np(Beschreibung, Atome-D2)
        ; cnp(Beschreibung, Atome-D2)
        ; advp(Beschreibung, Atome-D2)
        ; pp(Beschreibung, Atome-D2)
        ; cpp(Beschreibung, Atome-D2)
        ).
```

3.2.4. Nominalphrasen

Nominalphrasen verfügen über die folgende Beschreibung:

```
[np, Formmerkmale, Typ, Definitheit, Valenz, Semantik, Strukturanalyse]
```

Die Formmerkmale sind dabei eine Liste aus den Merkmalen dritte Person, Numerus und Kasus. Der Typ entspricht dem semantischen Typ des Nominalkerns der Phrase und Valenz der syntaktischen Valenz des Nominalkerns.

```
np([np, [3, sg, Cas], event, def, Val, Sem, [InfoBbox]], [Bbox|D]-D)
    -->
        b([blackbox, event, Bbox], InfoBbox),
        {!, vollform(n>app, _, [sg, Cas], event, _, Val, SemN, '
            Veranstaltung', _),
        Sem = lam(P, ex(X, (SemN * Bbox)*X und P*X))}.
```

Diese Nominalphrase erkennt Vorlesungstitel als Blackbox und erzeugt daraus eine direkte Nominalphrase. Da eine Blackbox über keine weitere Semantik außer ihrem atomaren

Inhalt verfügt, muss diese Regel die Semantik einer Nominalphrase konstruieren. Die Semantik der Nominalphrasen ist im Normalfall eine einfache Verkettung der Semantik ihrer Teilausdrücke.

Beispiele: “Syntax”, “Mathe 1”, “Eine Dependezsyntax des Deutschen für die maschinelle Übersetzung mit Etap-3”;

```
np([np, [3, sg, Cas], event, def, Val, Sem, [InfoArt, InfoBbox]], [
    Art, Bbox|D]-D) -->
        t([art>def, [sg, Cas], def, Genus, SemArt, _, Art], InfoArt),
        \+generierung(suggest),
        b([blackbox, event, Bbox], InfoBbox),
        {!, bindestrich_kompositum(Bbox, Titel, Nomen),
        lex(n>app, ['#'|Nomen], [sg, Cas], event, Genus, Val, SemN, _
            , _),
        Sem = SemArt * (SemN * Titel)}.
```

Bei dieser experimentellen Nominalphrase werden im wesentlichen ein Artikel und eine Blackbox erkannt. Eine Besonderheit ist jedoch, dass der Inhalt der Blackbox verarbeitet wird. Das Prädikat `bindestrich_kompositum/3` spaltet das erste Argument an einem Bindestrich in die zwei folgenden Argumente auf. Das Ergebnis ist zum Beispiel, dass ein Blackbox-Token der Form *XML-Seminar* in die beiden Token *XML* und *Seminar* gespalten wird. Wenn das zweite dieser Token als Apposition des Typs `event` analysiert werden kann, wird dessen Semantik in die Phrasensemantik übernommen.

Beispiele: die “Syntax-Vorlesung”, das “XML-Seminar”;

```
np([np, [3, sg, Cas], hum, def, _, Sem, [InfoAnrede, InfoName]], [
    Anrede, Name|D]-D) -->
        n([n>app, [sg, Cas], hum, _Genus, _, SemAnrede, _, Anrede],
            InfoAnrede),
        b([blackbox, hum, Name], InfoName),
        {!, Sem = lam(P, ex(X, (SemAnrede*Name)*X und P*X))}.
```

Diese Nominalphrase erkennt eine Anrede und eine Blackbox. Auch bei dieser Phrase muss eine Semantik bereitgestellt werden. Bei der Semantik, die diese Phrase ergänzt, handelt es sich um die Semantik eines Artikels. Erst dadurch erhält die Phrase die Semantik einer Nominalphrase.

Beispiele: Herr “Mustermann”, Professor “Mueller”;

```
np([np, [3, Num, Cas], Typ, Def, Val, Sem, [InfoArt, InfoN]], [Art, N
    |D]-D) -->
        {(Def = indef ; Def = qu)},
        gen_bedingung(Typ, hum, Gf, 'Dozent'),
        gen_bedingung(Typ, event, Gf, 'Veranstaltung'),
        t([art>Def, [Num, Cas], Def, Genus, SemArt, _, Art], InfoArt)
            ,
        n([n>n, [Num, Cas], Typ, Genus, Val, SemN, Gf, N], InfoN),
        {Sem = SemArt * SemN}.
```

Bei dieser Nominalphrase wird keine Blackbox erkannt und damit keine bestimmte Entität benannt. Daher ist Definitheit dieser Nominalphrase nicht definit, sondern entweder indefinit oder interrogativ. Das Prädikat `gen_bedingung/4` dient zur Formulierung von Bedingungen, wenn das nächste Token generiert wird. Die ersten beiden Argumente sind die Bedingung, die letzten beiden die Folge. Der Aufruf `gen_bedingung(Typ, hum, Gf, 'Dozent')` unifiziert zum Beispiel die Variabel `Gf` mit dem Wert `'Dozent'`, wenn die Variabel `Typ` mit dem Wert `hum` unifizierbar ist. Da der semantische Typ einer Phrase immer bereits durch die Verbvalenz gegeben ist, beschreiben die beiden Bedingungen dieser Phrase, dass bei freier Generierung der Phrase das Nomen *Dozent* verwendet werden soll, wenn der Typ `hum` ist, und das Nomen *Veranstaltung*, wenn der Typ `event` ist.

Beispiele: ein Dozent, eine Vorlesung, welcher Dozent, welche Vorlesung;

```
np([np, [3, sg, Cas], Typ, Def, Val, Sem, [InfoArt, InfoN, InfoBbox
    ]], [Art, N, Bbox|D]-D) -->
        (generierung(suggest)
                -> {((Typ = hum) -> fail ; true),
                        Def = def}
                ; {Def = def ; Def = indef}
        ),
        t([art>Def, [sg, Cas], Def, Genus, SemArt, _, Art], InfoArt),
        n([n>app, [sg, Cas], Typ, Genus, Val, SemN, _, N], InfoN),
        b([blackbox, Typ, Bbox], InfoBbox),
        {Sem = SemArt * (SemN * Bbox)}.
```

Diese Nominalphrase erzeugt Phrasen der Form *der Dozent "Mueller"* und definiert zwei Sonderbedingungen für den Fall der freien Generierung der Phrase. Wird die Phrase frei generiert und der Phrasentyp ist `hum`, dann blockiert die gesamte Phrase und kann nicht erzeugt werden. Dies verhindert Suggest-Vorschläge der Form *der (Anrede) (Nachname)*. Im Suggest-Modus kann daher nur die Form ohne Artikel generiert werden. Die zweite Bedingung bei freier Generierung der Phrase besagt, dass in diesem Fall nur der bestimmte Artikel verwendet werden darf. Insgesamt ist diese Nominalphrase auf definit und indefinit beschränkt.

Beispiele: der Dozent "Mueller", die Vorlesung "Syntax";

```
np([np, [3, Num, Cas], Typ, Sub, Val, Sem, [InfoPro]], [Pro|D]-D) -->
        \+generierung(suggest),
        pn([pro>Sub, [Num, Cas], Typ, '-', Val, Sem, _, Pro], InfoPro
            ).
```

Mit dieser Nominalphrase werden alle Pronomen generiert. Pronomen stellen jeweils ihre eigene Nominalphrase. Dies ist auch die einzige nicht koordinierte Nominalphrase, die einen Plural bilden kann. Um die Anzahl der Suggestions klein zu halten, werden im Suggest-Modus keine Pronomen generiert.

Beispiele: wer, was;

3.2.5. Präpositionalphrasen

Präpositionalphrasen verfügen über die folgende Beschreibung:

```
[pp, [Kasus], Typ, Definitheit, Semantik, Strukturanalyse]
```

Der Typ entspricht hier dem semantischen Typ der Präposition.

Einfache Präpositionalphrasen

```
pp([pp, [Cas], Typ, '', Sem, [InfoPraep, InfoBbox]], [Praep, Bbox|D]-
    D) -->
        t([p>bbox, [], Typ, Cas, SemPraep, _, Praep], InfoPraep),
        b([blackbox, Typ, Bbox], InfoBbox),
        {!, Sem = SemPraep * Bbox}.
```

Diese einfache Präpositionalphrase besteht nur aus einer Blackbox-einleitenden Präposition und einer Blackbox.

Beispiel: ueber “Syntax”;

```
pp([pp, [Cas], loc, def, Sem, [InfoPraep, InfoRaum, InfoBbox]], [
    Praep, Raum, Bbox|D]-D) -->
        t([p>def, [], loc, Cas, SemPraep, _, Praep], InfoPraep),
        n([n>app, [sg, Cas], loc, _, _, SemRaum, _, Raum], InfoRaum),
        b([blackbox, loc, Bbox], InfoBbox),
        {!, Sem = SemPraep * (SemRaum * Bbox)}.
```

Aus einer definiten Präposition, einer Apposition und einer Blackbox besteht diese einfache Präpositionalphrase. Der semantische Typ ist auf Ortsangaben beschränkt. Definite Präpositionen, wie sie in dieser Phrase verwendet werden, sind aus einem Artikel und einer Präposition zusammengezogene Formen wie *im*.

Beispiel: im Raum “1.14”;

```
pp([pp, [Cas], temp_d, def, Sem, [InfoPraep, InfoTag]], [Praep, Tag|D
    ]-D) -->
        \+generierung(suggest),
        t([p>def, [], temp_d, Cas, SemPraep, _, Praep], InfoPraep),
        n([n>n, [sg, Cas], temp_d, _, _, SemTag, _, Tag], InfoTag),
        {Sem = SemPraep * SemTag}.
```

Diese Präpositionalphrase besteht aus einer definiten Präpositionalphrase und einer Wochentagsangabe. Der semantische Typ ist daher auf Wochentagsangaben beschränkt. Der Unterschied zur vorangegangenen Präpositionalphrase ist die hier fehlende Blackbox.

Beispiel: am Montag;

```
pp([pp, [Cas], Typ, Def, Sem, [InfoPraep, InfoNP]], [Praep|NP]-D) -->
        {InfoNP = [np, [_, _, Cas], Typ, Def, _, SemNP, _]},
        t([p>p, []], Typ, Cas, SemPraep, _, Praep], InfoPraep),
        (np(InfoNP, NP-D) ; cnp(InfoNP, NP-D)),
        {Sem = SemPraep * SemNP}.
```

Aus einer einfachen Präposition und einer eingebetteten einfachen Nominalphrase oder einer eingebetteten koordinierten Nominalphrase besteht diese Präpositionalphrase. Die

syntaktische Valenz der Nominalphrase wird nicht berücksichtigt. Würde die Valenz von eingebetteten Nominalphrasen auf Satzebene hochgereicht, könnten unverständliche Sätze wie **Wo befindet sich der Raum ueber Syntax der Vorlesung* gebildet werden.

Beispiele: von dem Dozenten, in dem Raum "1.14", in einem Raum;

Präpositionen mit Semesterangaben

Die folgenden Präpositionen sind einfache Präpositionen, welche eine Semesterangabe beinhalten.

```
pp([pp, [dat], semester, '', Sem, [InfoPraep, 'Semesterangabe']], [
    Praep, '(Semesterangabe)'|D]-D) -->
        generierung(suggest), {!},
        t([p>def, [], semester, dat, Sem, _, Praep], InfoPraep).
```

Beispiel: im (Semesterangabe);

```
pp([pp, [Cas], semester, def, Sem, [InfoPraep, InfoN, InfoAngabe]], [
    Praep, N, Angabe|D]-D) -->
        t([p>def, [], semester, Cas, SemPraep, _, Praep], InfoPraep),
        n([n>n, [sg, Cas], semester>Subtyp, _Genus, _, SemN, _, N],
            InfoN),
        a([angabe, semester>Subtyp, Angabe], InfoAngabe),
        {Sem = SemPraep * (SemN * Angabe)}.
```

Beispiel: im Sommersemester 2009;

```
pp([pp, [Cas], semester, def, Sem, [InfoPraep, InfoArt, InfoN,
    InfoAngabe]], [Praep, Art, N, Angabe|D]-D) -->
        t([p>p, [], semester, Cas, SemPraep, _, Praep], InfoPraep),
        t([art>def, [sg, Cas], def, Genus, SemArt, _, Art], InfoArt),
        n([n>n, [sg, Cas], semester>Subtyp, Genus, _, SemN, _, N],
            InfoN),
        a([angabe, semester>Subtyp, Angabe], InfoAngabe),
        {Sem = SemPraep * (SemArt * (SemN * Angabe))}.
```

Beispiel: in dem Sommersemester 2009;

3.2.6. Adverbialphrasen

Adverbialphrasen verfügen über die folgende Beschreibung:

```
[pp, [_], Typ, Definitheit, Semantik, Strukturanalyse]
```

Adverbialphrasen haben damit grundsätzlich dieselbe Beschreibung wie Präpositionalphrasen. Dadurch können beide in den syntaktischen Valenzen von Verb und Nomen als Alternativen füreinander zugelassen werden. Dazu muss lediglich das Kürzel der Phrase eine Variabel sein, so wie es im Kapitel 3.1.4 für die semantischen Typen `loc` und `temp_d` beschrieben ist.

```
advp([advp, [_], Typ, qu, Sem, [InfoAdv]], [Adv|D]-D) -->
        (generierung(suggest)
                -> {member(Typ, [temp_d, loc])}
                ; {true}
        ),
        t([adv>qu, [], Typ, qu, Sem, _, Adv], InfoAdv).
```

Bei dieser Adverbialphrase sind die Typen während der freien Generierung auf Wochentagsangaben und Ortsangaben beschränkt. Dies verhindert, dass *wann* in der Lesart *semester* bei der Generierung berücksichtigt wird und verhindert so, dass an sich identische Sätze im Suggest-Modus doppelt generiert werden.

Beispiele: wo, wann;

```
advp([advp, [_], Typ, '', Sem, [InfoAdv]], [Adv|D]-D) -->
        \+generierung(suggest),
        t([adv>adv, [], Typ, '', Sem, _, Adv], InfoAdv).
```

Beispiele: montags, dienstags;

3.2.7. Koordination

Die sprachverarbeitende Komponente unterstützt die Koordination von Nominal- und Präpositionalphrasen. Sowohl bei koordinierten Nominalphrasen als auch bei koordinierten Präpositionalphrasen gilt in dem modellierten Fragment des Deutschen die Einschränkung, dass nur Phrasen mit identischem semantischem Typ koordiniert werden dürfen. Diese Einschränkung ist notwendig, damit der Typ der Gesamtphrase eindeutig bestimmbar ist. Bei der Definitheit der koordinierten Phrasen gilt eine ähnliche Einschränkung. Die koordinierten Phrasen müssen dieselbe Definitheit aufweisen - mit einer Einschränkung. Denn abweichend kann eine Phrase auch interrogativ sein. In diesem Fall wird die Gesamtphrase interrogativ.
Die koordinierten Gesamtphrasen sind normale Phrasen und weisen die gleiche Beschreibungen wie die entsprechenden einfachen Phrasen auf. Bei den Nominalphrasen sind die koordinierten Gesamtphrasen nicht immer im Singular, sondern sind im Falle der Konjunktion im Plural.

Koordinierte Blackbox

```
cb([blackbox, Typ, Sem, [InfoBbox]], [Bbox|D]-D) -->
        b([blackbox, Typ, Bbox], InfoBbox),
        {Sem = lam(Art, lam(N, Art * (N * Bbox)))}.

cb([blackbox, Typ, Sem, [InfoA, InfoCon, InfoB]], [BoxA, Con|BoxB]-D)
    -->
        \+generierung(suggest),
        b([blackbox, Typ, BoxA], InfoA),
        t([junkt>con, [], con, '', SemCon, _, Con], InfoCon),
        {InfoB = [blackbox, Typ, SemB, _]},
```

```
        cb(InfoB, BoxB–D),
        {Sem = lam(Art, lam(N, SemCon * (Art*(N*BoxA)) * (SemB*Art*N)
             ))}.
```

Diese Phrase koordiniert Blackbox-Phrasen miteinander. Sie ist ein Sonderfall und wird nicht eigenständig verwendet, sondern nur innerhalb einer koordinierten Nominalphrase. Die koordinierte Blackbox-Phrase kann nur mit einer Konjunktion gebildet werden. Wie jede Phrase ohne einem Kern mit vollständiger Semantik muss auch diese Phrase ihre Semantik unabhängig vom Lexikon definieren.

Beispiel: “X” und “X”;

Koordinierte Nominalphrasen

```
cnp([np, [3, pl, Cas], Typ, def, _, Sem, [InfoArt, InfoN, InfoBbox]],
    [Art, N|Bbox]–D) –->
        \+generierung(suggest),
        t([art>def, [pl, Cas], def, Genus, SemArt,_,Art], InfoArt),
        n([n>app, [pl, Cas], Typ, Genus, _Val, SemN, _, N], InfoN),
        {InfoBbox = [blackbox, Typ, SemBbox, [_,_,_|_]]},
        cb(InfoBbox, Bbox–D),
        {Sem = (SemBbox * SemArt) * SemN}.
```

Diese koordinierte Nominalphrase stellt der eigentlichen Koordination einen definiten Artikel und eine Apposition im Plural voraus. Koordiniert werden keine Nominalphrasen, sondern Blackbox-Phrasen mit dem Phrasentyp `cb`. Dieser Phrasentyp ist ein Sonderfall und wird nur in dieser koordinierten Nominalphrase verwendet. Durch die Vorgabe von drei anonymen Variabeln als Atome in der Beschreibung der koordinierten Blackbox-Phrase wird erreicht, dass die koordinierte Blackbox mindestens zwei Blackbox-Werte enthält.

Beispiel: die Professoren “Mueller” und “Schmidt”;

```
cnp([np, [3, pl, Cas], Typ, Def, _, Sem, [InfoA, InfoCon, InfoB]],
   NP_A–D) –->
        \+generierung(suggest),
        {InfoA   = [np, [3, sg, Cas], Typ, DefA, _, SemA, _]},
        {InfoB   = [np, [3, _N, Cas], Typ, DefB, _, SemB, _]},
        {InfoCon = [junkt>con, [], con, '', SemCon, _, Con]},
        np(InfoA, NP_A–[Con|NP_B]),
        t(InfoCon, InfoCon),
        (np(InfoB, NP_B–D) ; cnp(InfoB, NP_B–D)),
        {(DefA = DefB
                –> Def = DefA
                 ; ((DefA = qu ; DefB = qu) –> Def = qu ; true)
        )},
        {Sem = (SemCon * SemA) * SemB}.
```

In dieser koordinierten Nominalphrase wird eine einfache Nominalphrase mit einer weiteren einfachen Nominalphrase oder mit einer weiteren koordinierten Nominalphrase koordiniert. An dieser Stelle sei betont, dass koordinierte Phrasen dasselbe Phrasenkürzel wie

die einfachen Phraen haben. Einzig die DCG-Regeln haben unterschiedliche Funktoren, um unendliche Linksrekursion in Prolog zu vermeiden. Aus demselben Grund sind alle koordinierten Phrasen in NASfVI so implementiert, dass die letzte Phrase den Rekursionsschritt enthält.

Beispiel: Herr “Mueller” und Professor “Mustermann” und welcher Professor;

```
cnp([np, [3, sg, Cas], Typ, Def, _, Sem, [InfoA, InfoCon, InfoB]],
    NP_A-D) -->
        \+generierung(suggest), !,
        {InfoA  = [np, [3, sg, Cas], Typ, Def, _, SemA, _]},
        {InfoB  = [np, [3, sg, Cas], Typ, Def, _, SemB, _]},
        {InfoCon = [junkt>dis, [], dis, '', SemDis, _, Con]},
        np(InfoA, NP_A-[Con|NP_B]),
        t(InfoCon, InfoCon),
        (np(InfoB, NP_B-D) ; cnp(InfoB, NP_B-D)),
        {normalisiere_term(SemA, ex(X, SA)),
         normalisiere_term(SemB, ex(X, SB)),
        Sem = ((SemDis * X) * SA) * SB}.
```

Die Implementierung der Disjunktion unterscheidet sich syntaktisch und semantisch grundlegend von der der Konjunktion. Syntaktisch bildet die Disjunktion einen Singular und nicht einen Plural. In der Semantik wiederum entfernt diese koordinierte Nominalphrase zunächst den äußeren Existenzquantor der in ihr koordinierten Phrasen. In diesem Schritt werden auch die von diesen Existenzquantoren gebundenen Variabeln unifiziert. Diese unifizierte Variabel wiederum wird der Semantik der Disjunktion übergeben, wo sie in einen neuen Existenzquantor eingesetzt wird, dessen Skopus die Semantik beider Unterphrasen umfasst. Beide Unterphrasen beziehen sich nun auf dieselbe Variabel in ihrer Semantik.

Beispiel: Herr “Mueller” oder Professor “Mustermann”;

Koordinierte Präpositionalphrasen

Die beiden implementierten koordinierten Präpositionalphrasen sind völlig analog zu ihren Entsprechungen bei den koordinierten Nominalphrasen.

```
cpp([pp, [Cas], Typ, Def, Sem, [InfoA, InfoCon, InfoB]], PP_A-D) -->
        \+generierung(suggest),
        {InfoA  = [pp, [Cas], Typ, DefA, SemA, _]},
        {InfoB  = [pp, [Cas], Typ, DefB, SemB, _]},
        {InfoCon = [junkt>con, [], con, '', SemCon, _, Con]},
        pp(InfoA, PP_A-[Con|PP_B]),
        t(InfoCon, InfoCon),
        (pp(InfoB, PP_B-D) ; cpp(InfoB, PP_B-D)),
        {(DefA = DefB
                -> Def = DefA
                ; ((DefA = qu ; DefB = qu) -> Def = qu ; true)
        )},
        {Sem = (SemCon * SemA) * SemB}.
```

Beispiel: im Sommersemester 2008 und im Sommersemester 2007;

```
cpp([pp, [Cas], Typ, Def, Sem, [InfoA, InfoCon, InfoB]], PP_A–D) —>
        \+generierung(suggest),
        {InfoA  = [pp, [Cas], Typ, Def, SemA, _]},
        {InfoB  = [pp, [Cas], Typ, Def, SemB, _]},
        {InfoCon = [junkt>dis, [], dis, '', SemDis, _, Con]},
        pp(InfoA, PP_A–[Con|PP_B]),
        t(InfoCon, InfoCon),
        (pp(InfoB, PP_B–D) ; cpp(InfoB, PP_B–D)),
        {normalisiere_term(SemA, ex(X, SA)),
         normalisiere_term(SemB, ex(X, SB)),
        Sem = ((SemDis * X) * SA) * SB}.
```

Beispiel: im Raum “1.14” oder im Raum “1.13”;

3.2.8. Felder und Satzklammern

Die syntaktische Analyse von Sätzen basiert in NASfVI auf dem Feldermodell des Deutschen (siehe Kapitel 2.2.1). Die Implementierung dieser Felder und Satzklammern des Modell werden nachfolgend diskutiert. Dabei enthalten die Felder eine Folge von verallgemeinerten Phrasen (Kapitel 3.2.3).

Das Vorfeld

Das Vorfeld besteht in NASfVI entweder aus einer beliebigen Phrase oder aus einer Nominalphrase mit dem semantischen Typ *event* und einer Präpositionalphrase mit dem semantischen Typ *thema*, zum Beispiel *„eine Vorlesung über Semantik“*. Die Behandlung von beliebigen Phrasen ist im Vorfeld auf eine Phrase beschränkt, da eine Verbvalenz für die Vorstrukturierung des Satzes erst mit dem Erreichen der linken Satzklammer zur Verfügung steht. Für eine effiziente Verarbeitung der Syntax ist eine solche Vorstrukturierung der möglichen Phrasen allerdings notwendig. Um die kombinatorischen Möglichkeiten gering zu halten, kann das Vorfeld daher nur aus einer beliebigen Phrase bestehen. Die einzige Ausnahme stellt das Auftreten einer *event*-NP und einer folgenden *thema*-PP dar, da eine Trennung der beiden Phrasen durch die linke Satzklammer oft nicht wünschenswert ist.
Das erste Argument des `vorfeld/4`-Prädikats ist eine Liste mit den erkennbaren Phrasen. Diese Liste kann gegeben sein, falls der Satz eine gegebene Liste von Phrasen enthalten soll, sie kann aber auch eine freie Variabel sein. Wenn eine Liste mit den zu erkennenden Phrasen gegeben ist, erkennt das Vorfeld die Phrasen der Liste. Bei der in Kapitel 3.5.2 beschriebenen Generierung von natürlichsprachigen Antworten werden die Felder mit vorgegebenen Phrasenlisten aufgerufen. Das zweite Argument von `vorfeld/4` ist eine Differenzliste mit der Abfolge der tatsächlich im Vorfeld erkannten Phrasen. Das dritte Argument wiederum ist eine Differenzliste mit der im Vorfeld erkannten Atome und das letzte Argument ist eine Liste der im Vorfeld erkennbaren Phrasen abzüglich der erkannten Phrase.

```
vorfeld([NP, PP|Rest], [NP, PP|D]-D, A1-A3, Rest)
        --> {NP = [np, _, event|_], PP = [pp, _, thema|_]},
                np(NP, A1-A2),
                pp(PP, A2-A3).
```

Im Fall einer Nominalphrase mit dem semantischen Typ *event* und eine Präpositionalphrase mit dem semantischen Typ *event* können in exakt dieser Reihenfolge auch zwei Phrasen im Vorfeld auftreten.

```
vorfeld([Ph|Rest], [Ph|P2]-P2, A1-A2, Rest)
        --> obj([Ph|P2]-P2, A1-A2).
```

Im allgemeinen Fall ist das Vorfeld jedoch aus Gründen der Effizienz auf eine Phrase beschränkt.

Das Mittelfeld

```
mittelfeld(Phn, []-[], A1-A1, Phn) --> {true}.

mittelfeld(Phn, [P1N|P2]-P3, A1-A3, Rest) -->
        {var(P1), member(P1, Phn), normobjekt(P1, P1N)},
         obj([P1N|P2]-P2, A1-A2),
        {entferne_liste([P1N], Phn, Rest_)},
         mittelfeld(Rest_, P2-P3, A2-A3, Rest).
```

Die Argumente des `mittelfeld`-Prädikats entsprechen denen des Vorfelds. Anders als das Vorfeld verwendet das Mittelfeld `member/2` um eine zulässige Phrase auszuwählen. Dadurch können eine beliebige Anzahl an Phrasen in beliebiger Reihenfolge im Mittelfeld vorkommen. `normobjekt/2` entfernt einen eventuellen ?-Operator von der Phrasenangabe und `entferne_liste/3` entfernt das erste Argument vom zweiten und gibt als drittes Argument das Ergebnis aus.

```
mittelfeld(Temp, aktiv, Phrasen, PhnErkannt, Atome, Rest) -->
        {einfache_zeiten(Temp)}, !,
        mittelfeld(Phrasen, PhnErkannt, Atome, Rest).

mittelfeld(_, _, Phn, [P1N|P2]-P3, A1-A3, Rest) -->
        {var(P1), member(P1, Phn), normobjekt(P1, P1N)},
        obj([P1N|P2]-P2, A1-A2), {entferne_liste([P1N], Phn, Rest_)},
        \+generierung(suggest),
        mittelfeld(Rest_, P2-P3, A2-A3, Rest).
```

Diese `mittelfeld/6`-Regeln sind vergleichbar mit `mittelfeld/4`. Sie sind jedoch um Argumente für das Tempus und das Genus Verbi erweitert. Bei einfachen Zeiten (Präsens und Präteritum) ergibt sich im Aktiv kein Unterschied zu `mittelfeld/4`. Bei zusammengesetzten Zeiten wird jedoch zwingend eine Konstituente und ein folgendes Token verlangt. `mittelfeld/6` wird ausschließlich in Verberstsätzen verwendet, um zu verhindern, dass zu kurze und damit zu zeitaufwendige Eingaben analysiert werden.

Das Nachfeld

```
nachfeld --> {true}.
```

Das Nachfeld ist in dem Sprachfragment von NASfVI nicht enthalten und daher immer leer.

Die linke Satzklammer

Die linke Satzklammer `lsk/7` hat sieben Argumente. Diese Argumente sind:

- die Formmerkmale: Genus Verbi, Verbstellung, Person, Numerus, Tempus,
- die Valenz des Vollverbs im Satz,
- die Semantik des Vollverbs im Satz,
- der Infinitiv des Vollverbs im Satz,
- die Vollform des Vollverbs oder des Partikels als Zeichenliste zur Verwendung in der rechten Satzklammer,
- die Vollform des Vollverbs oder des Partikels als Atom zur Verwendung in der rechten Satzklammer,
- eine Differenzliste mit der in der linken Satzklammer erkannten Atome;

Die linke Satzklammer wählt immer ein Vollverb aus - auch wenn in der linken Satzklammer nur ein finites Hilfsverb vorhanden ist. Durch dieses Vorgehen wird stets valenzgebunden analysiert, auch wenn die eigentliche Verbvalenz erst mit der rechten Satzklammer sicher bekannt ist.

```
lsk([aktiv, Vst, Pers, Num, Temp], Val, Sem, Inf, LP, AP, [AV|Diff]-
    Diff) -->
        {einfache_zeiten(Temp)},
        [LV],
        v([LV, LP], [aktiv, Vst, Pers, Num, Temp], Val, Sem, Inf, [AV
            , AP]).
```

Die linke Satzklammer im Aktiv erkennt mit dieser Regel Vollverben.

```
lsk([aktiv, Vst, Pers, Num, Temp], Val, Sem, Inf, LV, AV, [AAux|Diff
    ]-Diff) -->
        {Vst \= vl},
        v_aux(haben, [aktiv, Vst, Pers, Num, Taux], AAux),
        {zeiten_aux-partizip(Taux, Temp)},
        v([LV, []], [aktiv, part2, [Pers, Num]], Val, Sem, Inf, [AV,
            '']).
```

Zum Erkennen des Hilfsverbs *haben* im Aktiv dient diese Regel. Sie wählt zusätzlich das Partizip II eines Vollverbs aus und macht das Vollverb der rechten Satzklammer verfügbar, damit diese es realisieren kann.

```
lsk([GV, Vst, Pers, Num, fut1], Val, Sem, Inf, LV, AV, [AAux|Diff]-
    Diff) -->
        {genusverbi(GV), Vst \= vl},
        v_aux(werden, [GV, Vst, Pers, Num, praes], AAux),
        (
         {(GV = aktiv)}
          -> v([LV, []], [aktiv, inf, [Pers, Num]], Val, Sem, Inf,
                [AV, ''])
           ; v([LV, []], [passiv, part2, [Pers, Num]], Val, Sem, Inf,
                [AV, ''])
        ).
```

Diese Regel bildet im Aktiv und Passiv das Futur 1 mit dem Hilfsverb *werden*.

```
lsk([passiv, Vst, Pers, Num, Temp], Val, Sem, Inf, LV, AV, [AAux|Diff
    ]-Diff) -->
        {Vst \= vl, einfache_zeiten(Temp)},
        v_aux(werden, [passiv, Vst, Pers, Num, Temp], AAux),
        v([LV, []], [passiv, part2, [Pers, Num]], Val, Sem, Inf, [AV,
            '']).
```

Das Passiv mit dem Hilfsverb *werden* wird bei den einfachen Zeiten mit dieser Regel für die linke Satzklammer gebildet.

```
lsk([passiv, Vst, Pers, Num, Temp], Val, Sem, Inf, LV, AV, [AAux|Diff
    ]-Diff) -->
        {Vst \= vl, (Temp = perf ; Temp = pqperf)},
        v_aux(sein, [passiv, Vst, Pers, Num, Taux], AAux),
        {zeiten_aux-partizip(Taux, Temp)},
        v([LV, []], [passiv, part2, [Pers, Num]], Val, Sem, Inf, [AV,
            '']).
```

Im Perfekt und Plusquamperfekt wird im Passiv in der linken Satzklammer das Hilfsverb *sein* durch diese Regel verwendet. Wie die anderen Hilfsverb-Regeln der linken Satzklammer wählt auch diese ein Vollverb aus, welches in der rechten Satzklammer realisiert wird.

Die rechte Satzklammer

Die Regeln der rechte Satzklammer erwarten vier Argumente: die Formmerkmale, eine zu erkennende Zeichenliste und das dazu passende Atom. Das letzte Argument ist der Rückgabewert in Form einer Differenzliste mit der in der Klammer erkannten Atome.
Die zu erkennende Zeichenliste und das dazugehörige Atom wird verwendet, wenn in der rechten Satzklammer eine Partikel oder das Partizip II des in der linken Satzklammer gewählten Vollverbs realisiert werden muss.

```
rsk([aktiv, _, _, _, Temp], [], '', A-A) -->
        {einfache_zeiten(Temp), !}.
```

Im Aktiv und ohne Partikel ist die rechte Satzklammer durch diese Regel bei einfachen Zeiten leer.

```
rsk([aktiv, Vst, _, _, Temp], LP, AP, [AP|Diff]-Diff) -->
        {Vst \= vl, einfache_zeiten(Temp)},
        ergaenze_vollform(LP).
```

Diese Regel realisiert eine Partikel, falls eine gegeben ist. Realisiert wird die Partikel durch `ergaenze_vollform/1`, was zudem eine Präfix-Erkennung durchführt, um Backtracking über das Mittelfeld hinweg zur linken Satzklammer hin zu vermeiden.

```
rsk([aktiv, _, _, _, fut1], LV, AV, [AV|Diff]-Diff) -->
        ergaenze_vollform(LV).
```

Im Futur 1 wird im Aktiv nur die Form des Vollverbs realisiert. Dies geschieht ebenfalls durch `ergaenze_vollform/1`.

```
rsk([passiv, _, _, _, fut1], LV, AV, [AV,AAux|Diff]-Diff) -->
        ergaenze_vollform(LV),
        v_aux(werden, [passiv, inf, _], AAux).
```

Im Passiv wird durch diese Regel im Futur 1 die Form des Vollverbs, sowie das Hilfsverb *werden* in der rechten Satzklammer realisiert.

```
rsk([aktiv, _, _, _, Temp], LV, AV, [AV|Diff]-Diff) -->
        {(Temp = perf ; Temp = pqperf), !},
        ergaenze_vollform(LV).

rsk([passiv, _, _, _, Temp], LV, AV, [AV|Diff]-Diff) -->
        {(Temp = praes ; Temp = praet)},
        ergaenze_vollform(LV).
```

Diese beiden Regeln realisieren das Partizip II des Vollverbs in der rechten Satzklammer. Das ist im Aktiv in den Zeiten Perfekt und Plusquamperfekt, sowie im Passiv in den Zeiten Präsens und Präteritum der Fall.

```
rsk([passiv, _, _, _, Temp], LV, AV, [AV,AAux|Diff]-Diff) -->
        {(Temp = perf ; Temp = pqperf)},
        ergaenze_vollform(LV),
        v_aux(werden, [passiv, part2, _], AAux).
```

Im Perfekt und Plusquamperfekt bildet diese Regel das Passiv mit dem Hilfsverb *werden*. Die Regel realisiert dazu nicht nur die Form des Vollverbs in der rechten Satzklammer, sondern auch das Partizip II des Hilfsverbs.

Hilfsprädikate der Satzklammern

In den Satzklammern werden zwei wichtige Hilfsprädikate verwendet:

```
ergaenze_vollform(LV) -->
        {nonvar(LV), LV \= []},
        [LV_],
        {LV_ = LV ; (LV_ = ['#'|LV__], LV__ \= [], praefix_von(LV__,
            LV))}.
```

`ergaenze_vollform/1` wird in der rechten Satzklammer verwendet, um Backtracking über das Mittelfeld zu vermeiden. Das Prädikat führt eine Präfix-Erkennung ohne Zugriff auf das Lexikon durch. Dadurch kann zum Beispiel eine Verbpartikel direkt in der rechten Satzklammer entsprechend verarbeitet werden.

```
zeiten_aux-partizip(praes, perf).
zeiten_aux-partizip(praet, pqperf).
```

`zeiten_aux-partizip/1` ordnet die Zeitformen des Hilfsverbs (erstes Argument) den Zeitformen des Satzes zu (zweites Argument):

- hat stattgefunden
- hatte stattgefunden

3.2.9. Sätze

Da NASfVI valenzgebunden und mit dem Feldermodell des Deutschen arbeitet, müssen lediglich die Positionen der Felder und Satzklammern für jede Verbstellungsmöglichkeit festgelegt werden. Die Phrasen in den Feldern können innerhalb der Felder frei gestellt werden und sind durch die Gesamtvalenz eines Satzes vorgegeben. Die Gesamtvalenz ergibt sich aus den Angaben, der Verbvalenz des Vollverbs und allen Nomenvalenzen, die in der Verbvalenz vorkommen. Die Nomenvalenz wird zu einem Zeitpunkt ermittelt, an dem die entsprechenden Nomen noch nicht analysiert worden sind. Daher wird zur Berechnung der Gesamtvalenz der semantische Typ der Nominalphrasen herangezogen. Denn dieser ist in der Verbvalenz bereits festgelegt.

```
angaben([ ?[_, [dat], semester, _, Sem, _] ], Sem).
```

In *jedem* Satz mögliche Angaben werden durch das Prädikat `angaben/2` definiert. Die Phrasen dieser Angaben sind also nicht in der Valenz des Verbs oder des Nomens angelegt. Im Sprachfragment ist die einzige solche Angabe die Angabe eines Semesters. Diese Modellierung ist gewählt worden, da sich alle Informationen eines Vorlesungsverzeichnisses auf bestimmte Semester beziehen.

s/2: Analyse von Sätzen

Das Prädikat `s/2` analysiert sowohl Verbzweit- als auch Verberstsätze. Beide Satzmuster verwenden die gleichen Argumente bestehend aus einer Analyse des Satzes und der linearen Abfolge der Atome des Satzes. Die Analyse jedes Satzes umfasst dabei:

1. die Markiertheit des Satzes,
2. den Infinitiv des Vollverbs im Satz,
3. das Genus Verbi (aktiv oder passiv),
4. die Verbstellung (v2 für Verbzweit- oder v1 für Verberststellung),

5. die Person, den Numerus und das Tempus des finiten Verbs,

6. die Felderstruktur mit einer Liste der Phrasen des Vorfelds, einer Liste der Phrasen des Mittelfelds und einer Liste für das Nachfeld,

7. die Semantik des Satzes;

Die Semantik der Sätze besteht aus einer Liste mit zwei Elementen: der Semantik des Vollverbs und der Semantik der Angaben. Diese beiden Semantik-Werte sind getrennt, da sich die Semantik der Angaben auf die gesamte Semantik des Verbs und jedes einzelne Prädikat darin bezieht und nicht ein Teil der syntaktisch-semantischen Verbvalenz ist.

```
s(Analyse, A1) -->
                {Analyse = [Markheit, [Inf, GV, v2, Pers, Num, Temp],
                     [P1, P2, []], [SemV, SemA]]},
        vorfeld(_, P1-[], A1-A2, _),
        \+generierung(suggest),
        lsk([GV, v2, Pers, Num, Temp], Verbval, SemV, Inf, LV, LA, A2
            -A3),
                {angaben(An, SemA),
                gesamtvalenz(Verbval, Val-An),
                entferne_liste(P1, Val, PhM)
                },
        mittelfeld(PhM, P2-[], A3-A4, PhN),
        rsk([GV, v2, Pers, Num, Temp], LV, LA, A4-[]),
        nachfeld,
                {leere_semantik(PhN),
                unifiziere_objval([P1, P2, []]),
                eval(Analyse, A1, Markheit)
                }.
```

Diese Regel beschreibt die Topologie eines Verbzweitsatzes mit Vorfeld, linker und rechter Satzklammer und einem Nachfeld. Die Verwendung von `\+generierung(suggest)` erzwingt dabei, dass das Vorfeld vollständig und die linke Satzklammer mindestens teilweise instanziiert sein muss. Diese Maßnahme verhindert zu kurze Eingaben im Suggest-Modus, welche ressourcenintensiv zu verarbeiten sind.
Das Prädikat `gesamtvalenz/2` berechnet die Gesamtvalenz des Satzes bestehend aus der Verbvalenz, den Valenzen der Angaben und den Nomenvalenzen. Es kopiert dazu die Verbvalenz und fügt die Valenzen der in der Verbvalenz enthaltenen Nominalphrasen hinzu. Es berücksichtigt nur die Nominalphrasen, die *direkt* aus der Verbvalenz stammen. Auf diese Weise kann die Gesamtvalenz bereits ermittelt werden, bevor das Mittelfeld und dessen Nominalphrasen analysiert worden sind. Die Gesamtvalenz kann dadurch als „Bauplan“ für das Mittelfeld verwendet werden. `entferne_liste/3` ist ein einfaches Hilfsprädikat und entfernt die erste Liste aus der zweiten Liste und gibt als drittes Argument den verbliebenen Rest zurück. Es dient dazu, erkannte Phrasen aus der Gesamtvalenz des Satzes zu entfernen, damit diese nicht mehrmals verwendet werden können. `leere_semantik/1` setzt eine „leere“ Semantik bei nicht realisierten fakultativen Phrasen. Eine leere oder neutrale Semantik ist eine Semantik, die sich mit einer

$\beta-$Reduzierung (siehe Kapitel 3.4.1) auf eine freie Variabel oder '_' reduzieren lässt. Fakultative Phrasen, die nicht realisiert worden sind, tragen dadurch inhaltlich nicht zur Semantik des Satzes bei - auch wenn sie in der Verbvalenz oder Nomenvalenz enthalten sind. `unifiziere_objval/1` unifiziert die Valenz der Phrasen mit den tatsächlich vorhandenen Phrasen im Satz. Da gesamtvalenz/2 die Valenzen der Phrasen allgemein und im Voraus berechnet, muss durch unifiziere_objval/1 die unspezifische Semantik der allgemeinen Phrasen mit der Semantik der tatsächlich vorhandenen Phrasen unifiziert werden. Das ist über die Valenz möglich, da Valenz und Semantik im Grundformenlexikon zusammenhängen (siehe Kapitel 3.1.1). Die optimalitätstheoretische Bewertung des Satzes führt `eval/3` durch: es berechnet die Markiertheit des Satzes, wie im folgenden Kapitel 3.3 gezeigt wird.

```
s(Analyse, A1) -->
                {Analyse = [Markheit, [Inf, GV, v1, Pers, Num, Temp],
                     ['', P1, []], [SemV, SemA]]},
        lsk([GV, v1, Pers, Num, Temp], Verbval, SemV, Inf, LV, AV, A1
            -A2),
        \+generierung(suggest),
                {angaben(An, SemA),
                gesamtvalenz(Verbval, Val-An)},
        mittelfeld(Temp, GV, Val, P1-[], A2-A3, PhN),
        rsk([GV, v1, Pers, Num, Temp], LV, AV, A3-[]),
        nachfeld,
                {leere_semantik(PhN),
                unifiziere_objval([[], P1, []]),
                eval(Analyse, A1, Markheit)
                }.
```

Verberstsätze werden durch diese Regel beschrieben. Bei dieser Topologie existiert kein Vorfeld.

s_gen/3: Generierung von Sätzen

Die bisherigen Satzregeln von `s/2` analysieren gegebene Sätze. Für die Generierung eines Satzes aufgrund von übergebenen Phrasen und einer übergebenen Analyse ist dagegen `s_gen/3` notwendig. Während `s_gen/3` auch bei übergebenen Phrasen mit einer freien Stellung der Phrasen generiert und die optimalitätstheoretisch optimale Stellung sucht, ist das bei `s/2` nicht der Fall. Denn bei `s/2` wäre durch die übergebenen Phrasen auch deren Stellung festgelegt. `s_gen/3` erzeugt nur Verbzweitsätze.

```
s_gen(Phrasen, Analyse, A1) -->
                {Analyse = [Markheit, [Inf, GV, v2, Pers, Num, Temp],
                     [P1, P2, []], [SemV, SemA]]},
        vorfeld(Phrasen, P1-[], A1-A2, PhM),
        lsk([GV, v2, Pers, Num, Temp], Verbval, SemV, Inf, LV, LA, A2
            -A3),
                {angaben(An, SemA),
                gesamtvalenz(Verbval, Val-An),
```

```
            unifiziere_zwingend(Phrasen, Val),
            entferne_liste(P1, Phrasen, PhM)
            },
        mittelfeld(PhM, P2-[], A3-A4, []),
        rsk([GV, v2, Pers, Num, Temp], LV, LA, A4-[]),
        nachfeld,
            {entferne_liste(Phrasen, Val, Fakultative),
            leere_semantik(Fakultative),
            unifiziere_objval([P1, P2, []]),
            eval(Analyse, A1, Markheit)
            }.
```

Das Prädikat `unifiziere_zwingend/2` unifiziert alle Elemente der ersten Liste mit Elementen der zweiten Liste. Es schlägt fehl, wenn ein Element aus der ersten Liste nicht mit einem aus der zweiten Liste unifiziert werden kann. Mit diesem Prädikat ist es möglich, festzulegen, dass die geforderten Phrasen realisiert werden müssen.
Die Verwendung von `s_gen/3` im Rahmen der Generierung natürlichsprachiger Antworten ist in Kapitel 3.5.2 dargestellt.

3.3. Optimalitätstheoretische Evaluation

Das Feldermodell des Deutschen und dessen Umsetzung in NASfVI machen keine Vorgaben über die Reihenfolge der Phrasen innerhalb der Felder. Da sich die verschiedenen Stellungen der Phrasen dennoch in ihrer Markiertheit unterscheiden können, wird im Anschluss an die Verarbeitung eines Satzes dessen Markiertheit mit dessen konkreter Stellung der Phrasen berechnet. Dieses an die Optimalitätstheorie angelehnte Verfahren geht von fest definierten *Constraints* aus, die von Sätzen verletzt werden können. Jede Verletzung von Constraints erhöht die Markiertheit des Satzes um einen definierten Wert. Im Unterschied zur Optimalitätstheorie sind alle Constraints in NASfVI gleichrangig.[3]

```
eval([_, Verbinfo, Felder, _], Atome, Markheit) :-
        findall(M, constraint(Verbinfo, Felder, Atome, M), Ms),
        sumlist(Ms, M_Summe),
        length(Atome, Len),
        Markheit is M_Summe + Len.
```

Das Prädikat `eval/3` berechnet die Markiertheit einer gegebenen Satzanalyse. Bei der Berechnung der Markiertheit werden die Merkmale des finiten Verbs, die Felderstruktur und die Atome („Token“) des dazugehörigen Satzes berücksichtigt. Das Prädikat ermittelt zunächst die einzelnen Markiertheitswerte aller Constraints, die der Satz verletzt, in der Liste `Ms`. Die Summe dieser Einzelwerte ergibt den numerischen Wert `M_Summe`. Die Markiertheit des Satzes entspricht der Summe dieses Wertes und der Anzahl der Atome des Satzes. Da die Zahl der Atome des Satzes berücksichtigt wird, ergibt sich die implizite Annahme, dass kürzere Sätze zu bevorzugen sind. Dadurch wird beispielsweise die Präpositionalphrase *„im Sommersemester 2011“* gegenüber der Präpositionalphrase *„in dem*

[3] Siehe dazu auch die Diskussion in Kapitel 2.2.1

Sommersemester 2011“ bevorzugt. Das scheint das menschliche Verhalten abzubilden.[4] Da NASfVI bei der Generierung natürlichsprachiger Antworten stets die Antwort mit der geringsten Markiertheit wählt, verhält es sich durch diese implizite Zusatzannahme wie gewünscht.

Constraints sind in NASfVI durch das Prädikat `constraint/4` definiert. Jedes Constraint erhält die Merkmale des finiten Verbs, die Felderstruktur des Satzes, sowie die Atome des Satzes als Argumente und gibt einen Wert für die Markiertheit als letztes Argument aus, falls das jeweilige Constraint verletzt ist. Bei der Implementierung der Constraints werden die Bedingungen des Sprachfragments, z. B. die sich aus den Valenzen ergebenden möglichen Kombinationen der Phrasen, ausgenutzt. Die Constraints sind in ihrer konkreten Implementierung daher also nicht allgemeingültig.

Die folgenden Constraints sind in NASfVI definiert:

- ```
  constraint([_, passiv|_], _, _, 5).
  ```

  Prinzip: Das Passiv ist markierter als das Aktiv ([DUG06], 1853). Dies wird dadurch umgesetzt, dass der Verwendung des Passivs eine Markiertheit von „5“ zugeordnet wird, dem Aktiv jedoch keine. Der konkrete Zahlenwert ist dabei willkürlich gewählt und nur im Zusammenspiel mit anderen Constraints relevant.

- ```
  constraint([_, _, v1|_], [_, [N|_], _], _, 5) :-
          N \= [np, [_, _, nom]|_].
  constraint([_, _, v2|_], [[N|_], _, _], _, 3) :-
          N \= [np, [_, _, nom]|_].
  ```

  Prinzip: Die erste Phrase eines Satzes sollte eine Nominalphrase im Nominativ sein (siehe [DUG06], 1353). Durch dieses Prinzip sind Sätze, die mit einem Akkusativobjekt oder einer Präpositionalphrase beginnen markierter als Sätze, die mit einer Nominalphrase im Nominativ beginnen. Die Verletzung dieses Prinzips ist in Verberstsätzen (Markiertheit: 5) schwerwiegender als in Verbzweitsätzen (Markiertheit: 3).

- ```
  constraint([_, _, v2|_], [[A|_], MF, _], _, 6) :-
          A \= [advp, _, _, qu|_], member([advp, _, _, qu|_], MF).
  ```

  Prinzip: Falls im Mittelfeld eine interrogative Adverbialphrase vorkommt, sollte diese bei Verbzweitsätzen die erste Phrase des Mittelfelds sein. So ist zum Beispiel die Phrasenstellung in dem Satz *„Wer hielt Syntax in welchem Semester?“* markierter als in dem Satz *„Wer hielt in welchem Semester Syntax?“*.

- ```
  constraint([_, _, v1|_], [_, MF, _], _, 6) :-
          member([advp, _, _, qu|_], MF).
  ```

[4] Beispielsweise findet Google ungefähr 1.320.000 Ergebnisse für “im Sommersemester 2011”, dagegen nur 4 Ergebnisse für “in dem Sommersemester 2011” (Quelle: Suchanfragen bei `http://www.google.de/` am 13.09.2011).

Prinzip: Interrogative Adverbialphrasen sollten nicht in Verberstsätzen verwendet werden. So ist zum Beispiel der Satz *„Fand in welchem Semester Semantik statt?“* markierter als der Satz *„In welchem Semester fand Semantik statt?“*.

- ```
  constraint(_, [_, [N|T], _], _, 5) :-
          N \= [np, _, hum|_], member([np, _, hum|_], T).
  ```

  Prinzip: Wenn im Mittelfeld eine Nominalphrase mit dem semantischen Typ „hum“ vorkommt, sollte diese die erste Phrase im Mittelfeld sein (siehe [DUG06], 1362). Durch diese Regel ist der Satz *„Hält eine Vorlesung über Syntax jemand?“* markierter als die Formulierung *„Hält jemand eine Vorlesung über Syntax?“*.

- ```
  constraint(_, [_, [N|T], _], _, 50) :-
          N \= [np, _, expl|_], member([np, _, expl|_], T).
  ```

  Prinzip: Wenn im Mittelfeld ein expletives Es vorkommt, muss es an erster Stelle stehen ([DUG06], 1356). Das ist zum Beispiel im Satz *„Gibt es eine Vorlesung über Programmierung“* der Fall, während *„Gibt eine Vorlesung über Programmierung es“* das Prinzip verletzt. Eine Verletzung dieses Prinzips wiegt mit einer Markiertheit von 50 sehr schwer.

- ```
  constraint(_, Felder, _, M) :-
          feldobjekt([np, _, event, Def|_], Felder), (
            (Def == indef, \+feldobjekt([pp, _, thema|_], Felder),
              M = 3)
          ; (Def == def, feldobjekt([pp, _, thema|_], Felder),
              M = 5)).
  ```

  Prinzip: Eine indefinite Nominalphrase des Typs „event“ sollte nur zusammen mit einer Präpositionalphrase des Typs „thema“ auftreten. Ist die Nominalphrase des Typs „event“ jedoch definit, sollte sie ohne entsprechender Präpositionalphrase auftreten. Dieses Prinzip bedeutet, dass Sätze wie *„Gibt es eine Vorlesung“* mit indefiniter event-NP und ohne einer thema-PP, sowie Sätze wie *„Gibt es die Vorlesung über Syntax“* mit definiter event-NP und mit einer thema-PP markiert sind. Das Prädikat `feldobjekt/2` unifiziert bei der Implementierung die gesuchte Phrase mit einer passenden Phrase in einem der Felder. Es spielt dabei keine Rolle, ob es sich um das Vorfeld, das Mittelfeld oder das Nachfeld handelt.

- ```
  constraint([geben|_], Felder, _, 15) :-
          feldobjekt([np, _, expl|_], Felder),
          feldobjekt([_, _, Typ|_], Felder),
          \+member(Typ, [expl, hum, event, thema]).
  ```

  Prinzip: „Gibt es“-Fragen sollten im Rahmen von NASfVI auf die semantischen Typen „hum“ und „event“ beschränkt sein. Dieses Prinzip sorgt für eine Markierung von Sätzen wie *„Gibt es einen Montag?“*. Die Sprachkomponente kann diese Sätze zwar analysieren, durch die vergleichweise hohe Markierung werden sie jedoch von der Suggest-Funktionalität ausgeschlossen.

- ```
  constraint(_, [VF, MF, NF], _, 5) :-
          \+suggest_modus(an),
          NP = [np, _, event, _, Val-[]|_],
          PP = [pp, Form, thema, _, Sem|_],
          feldobjekt(NP, [VF, MF, NF]),
          \+(\+feldobjekt(PP, [VF, MF, NF])),
          nonvar(Val),
          member_normobjekt(PP, Val),
          \+(((finde_paar(VF, NP, PP2) ; finde_paar(MF, NP, PP2) ;
              finde_paar(NF, NP, PP2)),
                  PP2 = [pp, Form, thema, _, Sem2|_],
                  Sem == Sem2)).
  ```

  Prinzip: Eine Präpositionalphrase des semantischen Typs „thema“ sollte immer direkt auf die Nominalphrase des Typs „event“ folgen, auf die sie sich bezieht. Um den Bezug festzustellen, prüft die Regel zunächst mit `member_normobjekt/2`, welche Präpositionalphrasen in der Valenz der Nominalphrase vorkommen. Anschließend wird mit `finde_paar/3` ermittelt, ob eine solche Präpositionalphrase direkt auf die Nominalphrase folgt. Im letzten Schritt wird schließlich die Vereinbarkeit der Semantik geprüft und damit der Bezug festgestellt. Bei Sätzen mit mehreren „event“-Nominalphrasen wie in dem Beispielsatz *„Welche Vorlesung ähnelt einem Seminar über Syntax?“* kann sich die Präpositionalphrase *„über Syntax“* sowohl auf *„Vorlesung“* als auch auf *„Seminar“* beziehen. Die Sprachkomponente beherrscht diese Form der Ambiguität, ordnet aber dem Bezug auf den direkten Vorgänger - im Beispiel von *„über Syntax“* auf *„Seminar“* - durch dieses Prinzip eine geringere Markiertheit zu. Diese semantische Ambiguität führt dazu, dass Sätze mit gleichen Wortfolgen je nach Lesart verschiedene Markiertheitswerte besitzen können. Um gleiche Wortfolgen nicht mehrfach mit verschiedener Markiertheit vorzuschlagen, wird das Prinzip im Suggest-Modus blockiert und nur im Parse-Modus berücksichtigt.

3.4. Semantik

Die Semantik wird in NASfVI kompositional aufgebaut. NASfVI setzt während der Syntax-Analyse die Semantik von Teilausdrücken in die Verbvalenz ein. Diese logischen Formeln werden in der semantischen Analyse normalisiert und letztlich in die Anfragesprache der Volltextsuche übersetzt.

3.4.1. Logische Formeln

Die logischen Formeln, welche die Semantik im Lexikon angeben, basieren auf der Prädikatenlogik, erweitern diese jedoch um einen interrogativen Quantor. Der interrogative Quantor fragt nach der durch ihn gebundenen Variabel und ist für Anfragen an das System von zentraler Bedeutung. Der Existenzquantor wird durch `ex/2` dargestellt, der interrogative Quantor durch `qu/2`. Die Formeln sind durch die Verwendung des Lambda-
  ```

Kalküls kompositional aufgebaut. Der $\lambda-$Operator wird durch `lam/2` dargestellt. Sowohl `ex/2` und `qu/2` als auch `lam/2` binden in ihrem ersten Argument eine Variabel. Der Skopus entspricht dabei der Formel im zweiten Argument. Der aussagenlogische Konjunktor $\wedge$ wird durch `und/2` dargestellt, der Disjunktor $\vee$ durch `oder/2`. Die logische Formel $\lambda Q \lambda P \exists x(Q * x \wedge P * x)$ für die Semantik eines Artikels wird also als `lam(Q, lam(P, ex(X, Q*X und P*X))` angegeben. Nomen und Verben, die eine Semantik tragen, verwenden in ihrem Lexikoneintrag entsprechende Grundprädikate. Das Verb *halten* (wie in *„eine Vorlesung halten“*) verwendet zum Beispiel `halten/4` in dessen Lexikoneintrag:

```
SemAg * lam(X, SemDies *
                lam(D, SemLoc *
                        lam(L, SemPa * lam(Y, halten(X, Y, L, D)))))
```

Die freie Variabel `SemAg` ist bei dem entsprechenden Lexikoneintrag über die Syntax mit der Semantik des Agens verbunden. Bei der syntaktischen Analyse wird dessen logische Formel in die Variabel eingesetzt. Auf gleiche Weise ist die Variabel `SemDies` mit der Semantik einer Zeitangabe, `SemLoc` mit der Semantik einer Ortsangabe und `SemPa` mit der Semantik des Patiens verbunden.
Das Prädikat `normalisiere_semantik/2` normalisiert die Semantik eines Satzes indem es gegebenenfalls eine $\alpha-$Konversion veranlasst und die Formeln $\beta-$reduziert. Für die $\alpha-$Konversion werden die Prädikate `alpha/3`, `alpha_liste/3` und `umbenennen/3` verwendet. Die $\beta-$Konversion wird mit dem Prädikat `beta/3` durchgeführt. Diese Prädikate entsprechen den Gegenstücken aus der Proseminarvorlesung „Computerlinguistik II“ von Dr. Hans Leiß aus dem Wintersemester 2005/06 am Centrum für Informations- und Sprachverarbeitung der LMU [LES05].

3.4.2. Suchanfragen für die Volltextsuche

Im letzten Schritt der semantischen Analyse werden die normalisierten logischen Formeln in die Anfragesprache der Volltextsuche übersetzt. NASfVI verwendet Apache Lucene für die Volltextsuche und Datenhaltung der Vorlesungsdaten. Aus diesem Grund werden die normalisierten logischen Formeln in Suchanfragen für Lucene übersetzt [LUCQ]. Da die Semantik der natürlichsprachigen Anfragen in einem getrennten Schritt von logischen Formeln in die Suchanfragen-Syntax der Volltextsuche übersetzt wird, kann die für die Volltextsuche verwendete Bibliothek ausgetauscht werden, ohne dass die Grammatik und das Lexikon geändert werden müssten.
Suchanfragen für Lucene beziehen sich stets auf in dessen Index erfasste Felder. Jedes Stichwort, das gesucht werden soll, wird einem Feld zugeordnet. Diese Zuordnung geschieht indem dem Stichwort oder Suchbegriff der Name des jeweiligen Feldes vorangestellt wird. So ermittelt die Suchanfrage `dozent:''mueller''` alle Vorlesungen, die von einem Dozenten namens Müller gehalten werden oder wurden. Die Suchangaben für Felder können kombiniert werden. Die Suchanfrage `dozent:''leiss'' titel:''syntax''` ermittelt zum Beispiel jede Vorlesung, die von einem Dozenten namens Leiss gehalten wurde und in deren Titel das Token „Syntax“ oder dessen Stamm vorkommt. Die Angaben für die einzelnen Felder können mit den logischen Operatoren `AND` und `OR` verknüpft

werden. Ist kein logischer Operator explizit angegeben, interpretiert NASfVI stets eine logische und-Verknüpfung.
Die Syntax der Suchanfragen kennt keinen interrogativen und keinen Existenzquantor. Bei der Übersetzung der logischen Formeln in die Anfragesprache von Lucene wird der Existenzquantor nicht weiter berücksichtigt. Die Felder der Prädikate, deren Variabeln jedoch von einem interrogativen Quantor gebunden sind, werden in einer Liste als gesuchte Felder gesammelt. NASfVI bezieht bei der Beantwortung einer Anfrage den Inhalt dieser Felder von Lucene. Bei der Übersetzung der logischen Formeln werden die Prädikate auf Feldernamen in Lucene abgebildet. Die Abbildung der Prädikate auf die Feldernamen von Lucene ist in Tabelle 3.2 aufgeführt. Da sich alle Datensätze im Index auf Veranstaltungen beziehen, wird bei der Beantwortung jeder Anfrage implizit eine Veranstaltung angenommen. Aus diesem Grund wird das Prädikat `halten/4` nicht in die Anfragesprache von Lucene übersetzt. Ebenfalls nicht übersetzt wird das Prädikat `aehneln/3`, das als zusätzliche Ähnlichkeitssuche behandelt wird. Die Prädikate `ort/1` und `zeit/1`, welche nur interrogativ (bei *wo* und *wann*) gebraucht werden und daher nur in der Liste der gesuchten Felder auftreten, werden nicht in den Suchanfragen verwendet. Denn Raumangaben in der Eingabe werden mit dem Prädikat `raum/2` und Tagesangaben mit Prädikat `tag/2` abgebildet. Andere Orts- und Zeitangaben sind im Sprachfragment nicht vorgesehen. Ein besonderer Fall ist das Prädikat `thema/2`. Es bezieht sich stets auf die Felder `titel` und `beschreibung`. Der Term `thema(X, ''syntax'')` wird zum Beispiel als `(titel:''syntax'' OR beschreibung:''syntax'')` übersetzt. Der Grund für diese besondere Übersetzung liegt darin, dass die Themenangabe sowohl im Titel der Veranstaltung als auch in deren Beschreibung vorkommen kann und beide danach durchsucht werden müssen.

logisches Prädikat	Feldername in Lucene
veranstaltung	titel
thema	beschreibung
ort	raum
dozent_titel	dozent

Tabelle 3.2.: Zuordnung der logischen Prädikate zu Feldern in Lucene. Nicht aufgeführte Prädikate entsprechen direkt dem Namen des Feldes in Lucene oder sind ein im Text dargestellter Sonderfall.

Die Übersetzung der Satzsemantiken mit deren logischen Formeln erfolgt durch das Prädikat `uebersetze_semantik/4`. Es erhält die Satzsemantik als erstes Argument und ermittelt **zwei** Suchanfragen, sowie eine Liste der gesuchten Felder. Die erste Suchanfrage, die das Prädikat ermittelt, ist die allgemeine Suchanfrage nach einer Veranstaltung. Falls in der Semantik das Prädikat `aehneln/3` verwendet wird, wird das Ergebnis der ersten Suchanfrage durch eine Ähnlichkeitssuche eingeschränkt. Die zweite von `uebersetze_semantik/4` erzeugte Suchanfrage führt diese Ähnlichkeitssuche durch. Ähnlichkeitssuchen sind in Kapitel 4.2.5 ausführlicher beschrieben.

Mit den so ermittelten Suchanfragen werden die Daten von Lucene durchsucht. Mit der ermittelten Liste der gesuchten Felder wiederum werden die erfragten Informationen aus dem Ergebnis extrahiert und später in eine natürlichsprachige Antwort eingefügt. Der Suchprozess ist in Kapitel 4.2 beschrieben, die Generierung von natürlichsprachigen Antworten in Kapitel 3.5.2.

3.5. Anfragen

Die sprachverarbeitende Komponente verfügt über drei verschiedene Anfragemöglichkeiten. Bei allen wird eine Eingabe übergeben, welche zunächst tokenisiert und dann syntaktisch und semantisch analysiert wird. Bei *Parse*-Anfragen wird eine Analyse des Satzes erzeugt, bei *Suggest*-Anfragen werden alle generierbaren Sätze erzeugt, die die Eingabe als Präfix beinhalten und unterhalb einer möglichst geringen Markiertheitsgrenze liegen, und bei *Beantworte*-Anfragen wird eine Analyse der Eingabe erzeugt und auf Grundlage dieser Analyse und der Anfrage eine Antwort aus der Eingabe generiert indem interrogative Phrasen durch definite Phrasen mit beim Aufruf übergebenen Werten ersetzt werden.

3.5.1. Anfrageprädikate

Vollständige Sätze werden mit den `parse`-Prädikaten analysiert. Die Eingabe ist dabei stets ein Satz in Form eines Prolog-Atoms. In Abhängigkeit von dem aufgerufenen `parse`-Prädikat wird entweder eine linguistische Analyse des Satzes erzeugt (`parse/2`) oder es werden die sich aus der Anfrage ergebenden Suchanfragen für die Volltextsuche zurückgegeben (`parse/5`). Mit dem `suggest/4`-Prädikat können dagegen Satzanfänge zu vollständigen Sätzen vervollständigt werden. Das im folgenden Kapitel 3.5.2 behandelte `beantworte/5` wiederum generiert natürlichsprachige Antworten.

parse/2: Linguistische Satzanalyse

```
parse(Eingabe, [Markheit, [Inf, GV, Vst, Pers, Num, Temp],
    Felderstruktur, Semantik]) :-
        deaktiviere_suggest,
        tokenisiere(Eingabe, 0, Token), !,
        markiertheit(Markheit),
        Sterm = s([Markheit, [Inf, GV, Vst, Pers, Num, Temp],
            Felderstruktur, Sem_], _Atome),
        phrase(Sterm, Token, []),
        normalisiere_semantik(Sem_, Semantik).
```

Das Prädikat `parse/2` analysiert einen vollständigen Satz und liefert dessen linguistische Analyse zurück. Mit `deaktiviere_suggest` wird verhindert, dass sich die Sprachverarbeitung im Suggest-Modus befindet. Die darauf folgende Tokenisierung wird im Anhang A.2 beschrieben. Mit `markiertheit(Markheit)` wird schließlich eine natürliche Zahl als Markiertheit für den Satz ausgewählt. Im Anschluss versucht `parse/2` den Satz mit der

ausgewählten Markiertheit zu analysieren. Der erste Markiertheitswert, der stets ausgewählt wird, ist 0. Schlägt der Aufruf von `s/2` mit einem gewählten Markiertheitswert fehl, wird die nächstgrößere natürliche Zahl als Markiertheit versucht. Erst wenn ein durch `max_markiertheit/1` definierter maximaler Markiertheitswert erreicht wurde, schlägt die Analyse endgültig fehl. Durch dieses Vorgehen ist sicher gestellt, dass bei der Analyse stets niedrigere Markiertheitswerte vor höheren Markiertheitswerten berücksichtigt werden. Wenn aufgrund von Backtracking mehrere Analysen von `parse/2` erzeugt werden, liefert dies Analysen immer nach Markiertheit aufsteigend. Die erste zurückgegebene Analyse ist daher stets die mit der niedrigsten Markiertheit.

```
zahl(0).
zahl(Zahl) :- zahl(Z), Zahl is Z + 1.

max_markiertheit(30).

markiertheit(M) :- max_markiertheit(Max), zahl(M),
        (M =< Max -> true ; !, fail).
```

Das Prädikat `max_markiertheit/1` definiert, dass nur Sätze bis zu einer Markiertheit von 30 analysiert werden. Dieser Wert ist willkürlich gewählt. `zahl/1` erzeugt beim Backtracking in jedem Schritt von 0 aufsteigend unendlich oft die nächstgrößere natürliche Zahl. `markiertheit/1` nutzt `zahl/1`, um die Markiertheiten bis zum definierten Maximalwert zu erzeugen.

Der nachfolgende Beispielaufruf von `parse/2` zeigt, dass das Prädikat mehrere semantische Lesarten ermitteln kann und sie entsprechend ihrer Markiertheit aufsteigend berechnet:[5]

```
?- parse('welches␣proseminar␣aehnelt␣einer␣vorlesung␣ueber␣syntax',
        [Markiertheit, _, _, [Semantik, _]]).

Markiertheit = 12,
Semantik = qu(X, veranstaltung(X, '') und typ(X, proseminar) und ex(Y
    , veranstaltung(Y, '') und typ(Y, vorlesung) und thema(Y, '"syntax
    "') und aehneln(X, Y, '_'))) ;

Markiertheit = 17,
Semantik = qu(X, veranstaltung(X, '') und typ(X, proseminar) und
    thema(X, '"syntax"') und ex(Y, veranstaltung(Y, '') und typ(Y,
    vorlesung) und aehneln(X, Y, '_'))) ;

false.
```

Es sind im Beispiel genau zwei Lesarten möglich. Die Präpositionalphrase „ueber syntax“ kann sich sowohl auf die Vorlesung als auch auf das Proseminar beziehen. Aufgrund der in Kapitel 3.3 vorgestellten Constraints hat die Lesart in der sich die Präpositional-

[5] Die Variabelbezeichnungen von Prolog wurden zugunsten einer besseren Lesbarkeit durch X und Y ersetzt, sowie zusätzliche Leerzeichen eingefügt.

phrase „ueber syntax“ auf die direkt vorangegangene Vorlesung bezieht eine geringere Markiertheit - und wird daher als erste Lösung ermittelt.

parse/5: Berechnung von Suchanfragen

```
parse(Eingabe, Tempus, Query, SimilQuery, Gesucht) :-
        parse(Eingabe, [_Markheit, [_, _, _, _, _, Tempus],
            _Felderstruktur, Semantik]),
        !,
        uebersetze_semantik(Semantik, Query, SimilQuery, Gesucht).
```

Das Prädikat `parse/5` analysiert eine Eingabe indem es `parse/2` aufruft, übersetzt aber als letzten Schritt die errechnete Semantik zu den in Kapitel 3.4.2 beschriebenen Suchanfragen für die Volltextsuche. `parse/5` ermittelt das Tempus der Anfrage, eine Liste der gesuchten Felder, die Suchanfrage für die Eingabe, sowie gegebenenfalls eine Ähnlichkeitssuche. Bei der Verarbeitung der Suchanfragen wird das Tempus der Eingabe berücksichtigt. Dieser Suchprozess ist in Kapitel 4.2 beschrieben. Im Unterschied zu `parse/2` erlaubt `parse/5` aufgrund des Cuts nach dem Aufruf von `parse/2` kein Backtracking und berechnet daher nur eine Lösung. Diese Lösung entspricht immer der ersten von `parse/2` gelieferten Analyse und damit der Lesart mit der geringsten Markiertheit.

```
?- parse('welches proseminar aehnelt einer vorlesung ueber syntax',
    Tempus, Query, SQuery, Gesucht).

Tempus = praes,
Query = 'typ:"proseminar"',
SQuery = 'typ:"vorlesung" (titel:"syntax" OR beschreibung:"syntax")',
Gesucht = [titel].
```

In diesem Beispiel wird sowohl eine normale Suchanfrage (Query) als auch eine Ähnlichkeitssuchanfrage (SQuery) berechnet. Da `parse/5` nur die Analyse mit der geringsten Markiertheit berücksichtigt, wird nur die Analyse des Satzes verarbeitet, bei der sich die Präpositionalphrase „ueber syntax“ auf die Vorlesung bezieht.

suggest/4: Vervollständigen von Satzanfängen

```
suggest(Eingabe, Toleranz, Markiertheit, Vorschlaege) :-
        aktiviere_suggest,
        tokenisiere(Eingabe, 1, Token), !,
        markiertheit(Markiertheit),
        All = (
                intervall(Markiertheit, Toleranz, M),
                phrase(s([M|_], Atome), Token, [])
        ),
        findall(Atome, All, Vorschlaege),
        \+(Vorschlaege = []), !.
```

Das Prädikat `suggest/4` ermittelt Vervollständigungen zu einem gegebenen Satzanfang. Mit `aktiviere_suggest` stellt es sicher, dass sich die Grammatik im Suggest-Modus befindet. Nur in diesem Modus kann die Grammatik unvollständige Sätze, Phrasen und

Token verarbeiten. Das Aktivieren und Deaktivieren des Suggest-Modus ist in Kapitel 3.2.1 beschrieben. Nachdem die Eingabe tokenisiert wurde, wählt `suggest/4` einen Markiertheitswert aus oder verwendet einen übergebenen Wert als Markiertheit. Die erzeugten Vervollständigungen, die als Vorschläge von `suggest/4` zurückgegeben werden, haben mindestens diese Markiertheit. Im Unterschied zu den anderen Anfrageprädikate erlaubt `suggest/4` jedoch ein Intervall an möglichen Markiertheitswerten, die durch die `Toleranz` und das Hilfsprädikat `intervall/3` erzeugt werden. Dadurch dass das Prädikat Markiertheitswerte aus einem Intervall erlaubt, nimmt die Zahl der Vorschläge und insbesondere deren syntaktische Vielfalt zu, ist aber dennoch durch den Toleranzwert begrenzt. Für den Aufruf des Prädikats `findall/3`, das alle in Frage kommenden Satzvorschläge ermittelt, wird ein Term konstruiert, der zunächst `intervall/3` aufruft, um die Markiertheitswerte innerhalb des tolerierten Intervalls zu erzeugen, und schließlich einen vervollständigten Satz mit einer der Markiertheiten unter Verwendung von `s/2` berechnet.

```
intervall(Basis, Bereich, Zahl) :-
        integer(Basis),
        integer(Bereich),
        zahl(Z),
        (Z =< Bereich -> true ; !, fail),
        Zahl is Basis + Z.
```

Das Hilfsprädikat `intervall/3` berechnet aufsteigend eine natürliche Zahl aus dem durch `Basis` und `Basis + Bereich` gegebenen Intervall.

Die folgenden beiden Beispiele zeigen, welche Vorschläge von `suggest/4` für den Satzanfang *„welches proseminar ha“* zurückgegeben werden. Im ersten Beispiel wird ein Toleranzwert von 5 für die Markiertheit gewählt, im zweiten Beispiel ein Toleranzwert von 7. Die Anzahl der von `suggest/4` vorgeschlagenen Vervollständigungen variiert stark mit dem akzeptierten Intervall der Markiertheitswerte. Bei geringen Unterschieden dominiert der Einfluss des impliziten Constraints der Satzlänge: kürzere Sätze werden bevorzugt.

```
?- suggest('welches␣proseminar␣ha', 5, _, Vs), member(V, Vs),
   concat_atom(V, '␣', S), writeln(S), fail.
welches Proseminar hat stattgefunden
welches Proseminar hatte stattgefunden
welches Proseminar handelt von (Thema)
welches Proseminar handelte von (Thema)
welches Proseminar hat (Titel) geaehnelt
welches Proseminar hatte (Titel) geaehnelt
false.

?- suggest('welches␣proseminar␣ha', 7, _, Vs), member(V, Vs),
   concat_atom(V, '␣', S), writeln(S), fail.
welches Proseminar hat stattgefunden
welches Proseminar hatte stattgefunden
welches Proseminar handelt von (Thema)
welches Proseminar handelte von (Thema)
welches Proseminar hat (Titel) geaehnelt
```

```
welches Proseminar hatte (Titel) geaehnelt
welches Proseminar hat ueber (Thema) stattgefunden
welches Proseminar hat im (Semesterangabe) stattgefunden
welches Proseminar hat von (Thema) gehandelt
welches Proseminar hatte ueber (Thema) stattgefunden
welches Proseminar hatte im (Semesterangabe) stattgefunden
welches Proseminar hatte von (Thema) gehandelt
welches Proseminar handelt ueber (Thema) von (Thema)
welches Proseminar handelt von (Thema) ueber (Thema)
welches Proseminar handelt von (Thema) im (Semesterangabe)
welches Proseminar handelt im (Semesterangabe) von (Thema)
welches Proseminar handelte ueber (Thema) von (Thema)
welches Proseminar handelte von (Thema) ueber (Thema)
welches Proseminar handelte von (Thema) im (Semesterangabe)
welches Proseminar handelte im (Semesterangabe) von (Thema)
welches Proseminar hat im Raum (Raum) stattgefunden
welches Proseminar hat in welchem Sommersemester stattgefunden
welches Proseminar hat in welchem Wintersemester stattgefunden
welches Proseminar hat in welchem Semester stattgefunden
welches Proseminar hat (Titel) im (Semesterangabe) geaehnelt
welches Proseminar hat im (Semesterangabe) (Titel) geaehnelt
welches Proseminar hatte im Raum (Raum) stattgefunden
welches Proseminar hatte in welchem Sommersemester stattgefunden
welches Proseminar hatte in welchem Wintersemester stattgefunden
welches Proseminar hatte in welchem Semester stattgefunden
welches Proseminar hatte (Titel) im (Semesterangabe) geaehnelt
welches Proseminar hatte im (Semesterangabe) (Titel) geaehnelt
welches Proseminar haelt (Nachname)
false.
```

Die Verwendung der Prädikate `member/2`, `concat_atom/3`, `writeln/1` und `fail/0` bei der Formulierung der Beispielanfragen dient ausschließlich dazu, dass SWI-Prolog jeden einzelnen Vorschlag aus der Liste der Vorschläge, die `suggest/4` zurückgibt, formatiert ausgibt.

3.5.2. Natürlichsprachige Beantwortung von Anfragen

Das Prädikat `beantworte/5` generiert eine natürlichsprachige Antwort zu einer natürlichsprachigen Eingabe. Es anaylsiert dazu die Eingabe linguistisch und verwendet diese Analyse als Grundgerüst, um durch Transformation des erhaltenen Syntaxbaums und gegebenenfalls dem Einsetzen von erfragten Werten eine natürlichsprachige Antwort zu erzeugen. Dieses Vorgehen ist möglich, da Anfragen und deren Antworten einen hinreichend ähnlichen Aufbau besitzen. Bei der Anfrage *„In welchem Raum findet eine Vorlesung über Syntax statt?“* kann zum Beispiel eine Antwort erzeugt werden, indem die interrogative Phrase *„In welchem Raum“* aus dem Vorfeld entfernt wird und durch die definite Phrase *„im Raum 1.14“* mit dem eingesetzten erfragten Wert *„1.14“* am Ende des Mittelfeldes ersetzt wird, während *„eine Vorlesung über Syntax“* unverändert in das Vorfeld übernommen wird: *„Eine Vorlesung über Syntax findet im Raum 1.14 statt.“*

```
beantworte(Anfrage, Werte, AnalyseAnfrage, [Markheit, [Inf, aktiv, v2
    , Pers, Num, Temp], Felder, Semantik], Antwort) :-
        deaktiviere_suggest,
        parse(Anfrage, AnalyseAnfrage),
        transform_analyse(AnalyseAnfrage, TransPhrasen),
        werte_einsetzen(TransPhrasen, Werte, Phrasen-[]),
        AnalyseAnfrage = [_, [Inf, _, _, Pers, _, Temp], _, _],
        markiertheit(Markheit),
        Sterm = s_gen(Phrasen, [Markheit, [Inf, aktiv, v2, Pers, Num,
            Temp], Felder, Sem_], Antwort),
        phrase(Sterm, _, []),
        normalisiere_semantik(Sem_, Semantik), !.
```

Nachdem der Suggest-Modus deaktiviert ist, wird die Eingabe mit `parse/2` linguistisch analysiert. Dann erfolgt die erste Transformation mit `transform_analyse/2`. Falls die Anfrage im Passiv gestellt wurde, werden gegebenenfalls vorkommende Präpositionalphrasen und Nominalphrasen für das Subjekt und Objekt des Satzes von dem Prädikat durch die entsprechenden aktiven Phrasen ersetzt. `transform_analyse/2` überführt zudem die Phrasen des Vorfelds, Mittelfelds und Nachfelds in eine flache Liste. Mit dieser Liste aller Phrasen und den einzusetzenden Werten ersetzt `werte_einsetzen/3` anschließend interrogative Phrasen durch definitive Entsprechungen, die die passenden Werte enthalten. Die interrogativen Phrasen werden dabei entfernt und die definiten Phrasen am Ende der Phrasenliste eingefügt. Zusammen mit Merkmalen des Verbs aus der Anfrage - dessen Infinitiv, Person und Tempus - wird die transformierte Liste der Phrasen verwendet, um mit Hilfe von `s_gen/3` eine natürlichsprachige Antwort zu generieren.

transform_analyse/2: Vom Passiv zum Aktiv

Da die Phrasen, die in der zu generierenden Antwort verwendet werden, von der Analyse der Anfrage stammen, sind die semantischen Agens- und Patiens-Rollen der Verbvalenz bereits zu syntaktischen Phrasen expandiert worden.[6] Damit wäre allerdings auch das Genus Verbi stets auf das Genus Verbi der Anfrage festgelegt. Diese Festlegung könnte dadurch umgangen werden, dass die entsprechenden syntaktischen Phrasen in die Agens- und Patiens-Rollen zurückverwandelt werden. Da, wie in Kapitel 3.3 gezeigt, das Aktiv jedoch stets weniger markiert als das Passiv ist, erzeugt `beantworte/5` dagegen immer nur Antworten im Aktiv und verwendet `transform_analyse/2`, um Agens- und Patiens-Phrasen des Passivs direkt in die entsprechenden Aktiv-Phrasen umzuwandeln.

Zusätzlich überführt `transform_analyse/2` die nach Vorfeld, Mittelfeld und Nachfeld getrennten Phrasen in eine ungetrennte Differenz-Liste aller in der Anfrage vorkommenden Phrasen. Durch diese Aufhebung der Feldergrenzen rücken nachfolgende Phrasen des Mittelfelds automatisch nach, falls am Kopf der Liste, d. h. im Vorfeld, eine interrogative Phrase entfernt wird.

```
transform_analyse([_, [_Inf, GV|_], [VF, MF, NF], _], Ph–D) :-
        transform_liste(GV, VF, VF2–MF2, Subj),
```

[6] Die Expansion in syntaktische Phrasen ist in Kapitel 3.1.4 beschrieben.

```
        transform_liste(GV, MF, MF2–NF2, Subj),
        transform_liste(GV, NF, NF2–D, Subj),
        ((GV = passiv, var(Subj))
                -> Ph = [[np, [3, sg, nom], hum, qu, _, _, [[pro>qu,
                    [sg, nom], hum, _, _, _, 'jemand', _]]]|VF2]
                ; Ph = VF2
        ).
```

Wie in Kapitel 2.2.1 gezeigt, muss bei transitiven Verben im Passiv das Agens nicht immer realisiert sein: *„Ist eine Vorlesung gehalten worden?“* Falls eine Anfrage im Passiv gestellt wurde und kein Subjekt enthält, welches in NASfVI immer dem Agens entspricht, fügt `transform_analyse/2` daher zusätzlich eine Nominalphrase im Nominativ als Subjekt ein. Mit dieser Nominalphrase, die stets aus dem Pronomen *„jemand“* besteht, können schließlich auch solche Anfrage in das Aktiv umformuliert werden: *„Jemand hat eine Vorlesung gehalten.“*

```
transform_liste(_, '', D–D, _). % kein Vorfeld vorhanden
transform_liste(_, [], D–D, _) :- (var(D) ; D = []), !.
transform_liste(GV, [H|T], [H2|T2]–D, Subj) :-
        transform(GV, H, H2, Subj),
        transform_liste(GV, T, T2–D, Subj).
```

Das Prädikat `transform_liste/4` erzeugt die Differenzlisten und transformiert rekursiv jede Phrase unter Verwendung des Prädikats `transform/4`. Das erste Argument von `transform/4` gibt das Genus Verbi an, das zweite Argument definiert die zu transformierende Phrase, das dritte Argument gibt die transformierte Phrase an (also das Ergebnis der Transformation) und das letzte Argument ist entweder eine freie Variabel oder hat den Wert `1`, falls ein Subjekt im Passiv gefunden und transformiert worden ist.

```
transform(passiv,
        [pp, [dat], _, _, _, [[p>p|_], NP]],
        [np, [Pers, Num, nom], hum, Def, _, _, Baum2], 1)
        :- NP = [np, [Pers, Num, dat], hum, Def, _, _, Baum], !,
           setze_kasus(nom, Baum, Baum2).
```

Diese Regel transformiert eine Präpositionalphrase, die mit einem Dativ gebildet worden ist, falls sie eine Nominalphrase des semantischen Typs *hum* enthält und die Anfrage im Passiv gebildet worden ist. Diese Phrasen, zum Beispiel *„von Professor Schulz“*, werden als Subjekt erkannt und entsprechend zu Nominalphrasen im Nominativ umgewandelt. Das Hilfsprädikat `setze_kasus/3` stellt sicher, dass alle Elemente, z. B. Artikel und Nomen, des übernommenen Syntaxbaums im neuen Kasus stehen.

```
transform(passiv,
        [np, [Pers, Num, nom], Typ, Def, _, _, Baum],
        [np, [Pers, Num, akk], Typ, Def, _, _, Baum2], _)
        :- !, setze_kasus(akk, Baum, Baum2).
```

Nominalphrasen werden bei Anfragen im Passiv durch diese Regel erkannt und in Akkusativobjekte umgewandelt.

```
transform(_, [np, Syn, Typ, Def, _, _, Baum],
          [np, Syn, Typ, Def, _, _, Baum], _).
transform(_, [Ph, Syn, Typ, Def, _, Baum],
          [Ph, Syn, Typ, Def, _, Baum], _).
```

Alle anderen Phrasen werden in allen anderen Fällen von `transform/4` weitgehend unverändert übernommen. Lediglich die Semantik wird für die spätere Generierung mit `s_gen/3` entfernt und durch eine freie Variabel ersetzt.

werte_einsetzen/3: Von interrogativen Phrasen zu definiten Phrasen

Im letzten Schritt werden die interrogativen Phrasen aus der Liste der Anfrage-Phrasen entfernt und durch definite Phrasen ersetzt. Diese definiten Phrasen enthalten dabei stets Blackboxen, welche jeweils erfragte Antwortwerte, z. B. einen Namen oder eine Raumangabe, enthalten. Diese Antwortwerte werden von der Volltextsuche ermittelt und beim Aufruf von `beantworte/5` übergeben. Die Einsetzung der definiten Phrasen und das Entfernen der interrogativen Phrasen erfolgt durch einen Aufruf von `werte_einsetzen/3`. Das Prädikat erwartet als erstes Argument die Liste der Phrasen, als zweites Argument die einzusetzenden Werte und gibt als drittes Argument eine Liste der Phrasen mit eingesetzten Werten zurück.

```
werte_einsetzen(X, [], X).

werte_einsetzen(Phrasen, [Wert|T], Ergebnis) :-
        ersetze_interrogativ(Phrasen, Wert, Erg_),
        werte_einsetzen(Erg_, T, Ergebnis).
```

`werte_einsetzen/3` arbeitet alle einzusetzenden Werte rekursiv ab. Die Aufrufe von `ersetze_interrogativ/3`, welche die eigentliche Aufgabe durchführen, orientieren sich also an den einzusetzenden Werten und nicht an den Phrasen. Die Reihenfolge der einzusetzenden Werte darf daher bei der Übergabe beliebig sein.
Die Liste der einzusetzenden Werte besteht aus einer Liste von Listen. Jedes Element der Liste ist selbst eine Liste mit mindestens zwei Elementen. Das erste Element entspricht immer einem Feldnamen der Volltextsuche, während die folgenden Elemente die einzusetzenden Informationen sind. `[[dozent, '"Max Mustermann‘‘, '"Manuela Musterfrau‘‘], [semester, '2009']]` ist zum Beispiel eine Liste mit zwei Werten für `dozent` und einem Wert für `semester`. Ist mehr als ein Wert pro Feld gegeben, wird eine koordinierte Phrase erzeugt.

```
ersetze_interrogativ(Phrasen, [LuceneFeld|Werte], Rest-D2) :-
        typ_entspr(Typ, LuceneFeld),
        Ph = [_, _, Typ, qu|_],
        entferne_d(Ph, Phrasen, 1, Rest-D),
        komplexe_phrase(Ph, Werte, D, D2).
```

Das Prädikat `ersetze_interrogativ/3` übersetzt unter Verwendung des Hilfsprädikats `typ_entspr/2` den jeweiligen Feldnamen der Volltextsuche in einen semantischen Typ der Grammatik. Diese Entsprechungen sind in Tabelle 3.3 aufgeführt. Mit der Variabel `Ph` wird eine beliebige interrogative Phrase definiert. Das Hilfsprädikat `entferne_d/4`

unifiziert diese beliebige interrogative Phrase mit einer konkreten interrogativen Phrase aus der Liste der Phrasen und entfernt zugleich die unifizierte Phrase aus der Liste der Phrasen. Das Prädikat `komplexe_phrase/4` schließlich erzeugt die definite Phrase mit dem jeweils einzusetzenden Wert und fügt diese Phrase an die Differenzliste der übrigen Phrasen an.

Semantischer Typ	Feldername in Lucene
hum	dozent
event	titel
thema	beschreibung
semester	semester
loc	raum
temp_d	tag

Tabelle 3.3.: Zuordnung von semantischen Typen der Grammatik zu Feldernamen der Volltextsuche.

Komplexe Phrasen

Jede mögliche interrogative Phrase wird durch einen Aufruf von `komplexe_phrase/4` unter Einsetzung von übergebenen Werten zu einer definiten Phrase übersetzt. An das Prädikat `komplexe_phrase/4` wird stets eine Liste mit einzusetzenden Atomen übergeben.

```
komplexe_phrase([advp, Syn, Typ|_], Werte, D, D2) :-
        komplexe_phrase([pp, Syn, Typ|_], Werte, D, D2).
```

Interrogative Adverbialphrasen wie *„wo“* werden auf definite Präpositionalphrasen wie *„im Raum 1.14“* abgebildet.

```
komplexe_phrase(Ph, [Atom], D, D2) :-
        einfache_phrase(Ph, Atom, Ph2),
        D = [Ph2|D2].
```

Komplexe Phrasen im eigentlichen Sinn von `komplexe_phrase/4` sind koordinierte Phrasen. Nicht koordinierte, einfache Phrasen werden durch das Prädikat `einfache_phrase/3` verarbeitet. Zur Vereinfachung des Quellcodes leitet `komplexe_phrase/4` auf das Prädikat `einfache_phrase/3` weiter. Beim Aufruf im Prädikat `ersetze_interrogativ/3` muss daher keine Fallunterscheidung zwischen komplexen und einfachen Phrasen stattfinden. Darüber hinaus können koordinierte Phrasen so beim Aufruf von `komplexe_phrase/4` sowohl rekursiv weitere Werte einfügen als auch eine einfache Phrase als Endpunkt einer Rekursion verwenden.

```
komplexe_phrase([np, [Pers, _, Cas], Typ|_], Werte, D, D2) :-
        Typ \= event,
        komplexe_phrase([blackbox, Typ, _, _], Werte, [D0], []),
        CPh = [np, [Pers, pl, Cas], Typ, def, _, _, [_Art, _N, D0]],
```

```
        !, D = [CPh|D2].

komplexe_phrase([blackbox, Typ, _, _], [Atom|Weiter], D, D2) :-
        PhraseA = [blackbox, Typ, Atom],
        komplexe_phrase([blackbox, Typ, _, _], Weiter, [PhraseB],[]),
        CPh = [blackbox, Typ, _, [PhraseA, _, PhraseB]],
        D = [CPh|D2].
```

Die erste Regel ermöglicht koordinierte Phrasen wie *„Die Dozenten X und Y“*. Sie gibt die Erzeugung eines passenden Artikels und Nomens im Plural vor, im Beispiel *„die Dozenten“*, und verwendet zur Bildung der Aufzählung *„X und Y“* die direkt folgende Regel für die Koordination von Blackbox-Werten. Aufgrund der Beschränkung `Typ \= event` werden koordinierte Vorlesungstitel nur als Blackbox-Werte koordiniert. Phrasen wie *„Die Vorlesungen “Syntax” und “Semantik”“* sind daher nicht möglich. Diese Einschränkung ist einzig stilistischen Überlegungen geschuldet.

```
komplexe_phrase([np, [Pers, _, Cas], Typ|_], [Atom|Weiter], D, D2) :-
        einfache_phrase([np, [Pers, sg, Cas], Typ, def|_], Atom,
            PhraseA),
        komplexe_phrase([np, [Pers, _,   Cas], Typ, def|_], Weiter, [
            PhraseB], []),
        CPh = [np, [Pers, pl, Cas], Typ, def, _, _, [PhraseA, _,
            PhraseB]],
        D = [CPh|D2].
```

Diese Regel ermöglicht koordinierte Nominalphrasen der Art *„Dozent X und Dozent Y“*. Sie erzeugt eine einfache Phrase, gefolgt von einer komplexen Phrase. Der Aufruf von `komplexe_phrase/4` kann sich dabei sowohl rekursiv fortsetzen als auch in eine weitere einfache Phrase münden. Da Prolog stets von Links nach Rechts arbeitet und diese Regel rechtsrekursiv ist, werden unendliche Linksrekursionen verhindert.

```
komplexe_phrase([pp, [Cas], Typ|_], [Atom|Weiter], D, D2) :-
        einfache_phrase([pp, [Cas], Typ, def|_], Atom,    PhraseA),
        komplexe_phrase([pp, [Cas], Typ, def|_], Weiter, [PhraseB],
            []),
        CPh = [pp, [Cas], Typ, def, _, [PhraseA, _, PhraseB]],
        D = [CPh|D2].
```

Auch diese Regel folgt dem eben gezeigten Schema. Sie ermöglicht koordinierte Präpositionalphrasen wie zum Beispiel *„am Montag und am Dienstag“*.

```
komplexe_phrase([pp, [Cas], Typ|_], Werte, D, D2) :-
        komplexe_phrase([np, [_, _, Cas], Typ, def|_], Werte, [NP],
            []),
        CPh = [pp, [Cas], Typ, def, _, [_P, NP]],
        D = [CPh|D2].
```

Mit dieser Regel sind Präpositionalphrasen wie *„in dem Raum A und dem Raum B“* möglich. Sie gibt dazu eine koordinierte Nominalphrase vor, im Beispiel *„dem Raum A und dem Raum B“*, und stellt dieser eine passende Präposition voran.

Einfache Phrasen

Einfache Phrasen sind nicht koordiniert und werden durch das Prädikat `einfache_phrase/3` behandelt. Da einfache Phrasen nicht koordiniert sind, erhalten sie beim Aufruf stets nur ein Atom als einzusetzenden Wert.

```
einfache_phrase([np, [Pers, _, Cas], Typ|_], Atom,
        [np, [Pers, sg, Cas], Typ, def, _, _, Baum2])
        :- np([np, [Pers, sg, Cas], Typ, def, _, _, Baum], _, _, []),
           member([blackbox, Typ, Atom], Baum),
           entferne_semantik(Baum, Baum2).
```

Diese Regel übersetzt interrogative Nominalphrasen zu definiten Nominalphrasen. Sie ruft dazu direkt `np/4` auf. Der einzusetzende Wert wird mit `member/2` in die Blackbox des Syntaxbaums der Nominalphrase übertragen. Das Hilfsprädikat `entferne_semantik/2` entfernt die Semantik-Formeln aus den Token-Einträgen, damit die Bindung über die freien Variabeln des Lexikons bei der Generierung mit `s_gen/3` problemlos möglich ist.

Die Regeln für gewöhnliche Präpositionalphrasen und Blackboxen folgen dem gleichen Vorgehen und sind daher an dieser Stelle nicht aufgeführt. Bei den Präpositionalphrasen gibt es jedoch zwei Ausnahmefälle, die keine Blackbox verwenden: Tagesangaben und Semesterangaben.

```
einfache_phrase([pp, Syn, temp_d|_], Atom, [pp, Syn, temp_d, def, _,
    [_, InfoTag]]) :-
        InfoTag = [n>n, _, temp_d, _, _, lam(X, tag(X, Atom)), _, _],
        pp([pp, Syn, temp_d, def, _, [_, InfoTag]], _, _, []).
```

Bei Tagesangaben existiert keine Blackbox in die der übergebene Wert eingesetzt werden könnte. Viel mehr entspricht der übergebene Wert der Kurzform eines Tages (mo, di, mi, do, fr, sa, so). Diese Kurzformen finden sich im Lexikon in der Semantik der entsprechenden Nomen wieder. Die Regel für Tagesangaben nutzt das, um das korrekte Nomen für den jeweiligen Tag anhand der Semantik auszuwählen.

```
einfache_phrase([pp, Syn, semester|_], Atom,
        [pp, Syn, semester, def, _, Baum2])
        :- pp([pp, Syn, semester, def, _, Baum], [_,_,Atom]-[], _,
            []),
           member([angabe, semester>_, Atom], Baum),
           entferne_semantik(Baum, Baum2).
```

Auch bei Semesterangaben wird keine Blackbox verwendet. Stattdessen sind die Jahreszahlen als jeweilige Angabe realisiert. Die Regel für Semesterangaben erzeugt daher zunächst eine entsprechende definite Präpositionalphrase und überträgt dann die Semesterangabe in das entsprechende Token.

Beispiele für natürlichsprachige Antworten

Zusammenfassend sei die Verwendung und Leistungsfähigkeit des Prädikats `beantworte/5` an den folgenden Beispielen demonstriert.

- `beantworte('fand ein seminar im sommersemester 2009 statt', _, _, _, A).`
  `A = [ein, 'Seminar', fand, im, 'Sommersemester', '2009', statt].`

  Eine einfache Anfrage, die als Verberstsatz erfolgt, führt, wenn keine einzusetzenden Werte übergeben werden, zu einer Umformulierung der Anfrage in einen Verbzweitsatz.

- `beantworte('ist syntax von herrn mustermann gehalten worden', _, _, _, A).`
  `A = ['"syntax"', hat, 'Herr', '"mustermann"', gehalten].`

  Anfragen im Passiv werden stets im Aktiv beantwortet.

- `beantworte('wurde eine vorlesung ueber sxntax gehalten', _, _, _, A).`
  `A = [jemand, hielt, eine, 'Vorlesung', ueber, '"sxntax"'].`

  Falls eine Anfrage im Passiv vorliegt und das Agens in der Anfrage nicht realisiert wurde, wird in der Antwort *„jemand“* als Subjekt eingefügt.

- `beantworte('in welchem raum findet eine vorlesung ueber syntax statt', [[raum, '"1.14"']], _, _, A).`
  `A = [eine, 'Vorlesung', ueber, '"syntax"', findet, im, 'Raum', '"1.14"', statt].`

  Übergebene Werte werden bei der Generierung der Antwort an passender Stelle eingesetzt. Die interrogative Phrase, welche den jeweiligen Wert erfragt hat, wird dabei durch eine definite Phrase mit dem erfragten Wert ersetzt.

- `beantworte('wer hielt wann ein seminar ueber syntax', [[dozent, '"Max Mustermann"'], [semester, '2009']], _, _, A).`
  `A = [ein, 'Seminar', ueber, '"syntax"', hielt, '"Max Mustermann"', im, 'Sommersemester', '2009'].`

  Bei Anfragen mit mehreren interrogativen Phrasen können auch mehrere Werte eingesetzt werden.

- `beantworte('wer hielt wann ein seminar ueber syntax', [[dozent, '"Max Mustermann"', '"Manuela Musterfrau"'], [semester, '2009']], _, _, A).`
  `A = [ein, Seminar, ueber, "syntax", hielten, die, Dozenten, "Max Mustermann", und, "Manuela Musterfrau", im, Sommersemester, 2009]`

  Falls mehrere Datensätze für dasselbe Feld übergeben werden, werden koordinierte Phrasen erzeugt, um diese aufzunehmen.

4. Der Server

Die Sprachverarbeitungskomponente ist in einer Web-Application eingebettet, welche die Server-Funktionalität von NASfVI implementiert. Im Unterschied zur Sprachverarbeitungskomponente ist der Server in Java geschrieben und wird von einem Servlet-Container ausgeführt. Er bündelt mehrere Servlets für den Zugriff auf das System, kapselt die Sprachverarbeitungskomponente, welche als externer Prozess mit SWI-Prolog ausgeführt wird, und stellt die Volltextsuche, sowie den Client für die Anzeige in Browsern bereit. Der grundsätzliche Aufbau des Servers ist in Kapitel 2.3.2 bereits vorgestellt worden.
Der Java-Code von NASfVI ist in Java-Paketen organisiert. Um möglichst weltweit eindeutige Namensräume zu ermöglichen, ist es in Java Konvention, dass die Pakete sich an der Struktur von umgekehrten Domainnamen orientieren. NASfVI folgt dieser Konvention. Es beherbergt allgemeine Hilfsklassen im Paket `de.spartusch` und spezielle Hilfsklassen, die für NASfVI entwickelt wurden, die aber weder dem Server noch dem Client zuzuordnen sind, im Paket `de.spartusch.nasfvi`. Der eigentliche Java-Code des Servers befindet sich mit der Volltextsuche und den Servlets in dem Paket `de.spartusch.nasfvi.server`. Der Code des Clients wiederum befindet sich unter `de.spartusch.nasfvi.client`.

4.1. Die Schnittstelle zwischen Java und Prolog

Für die Kommunikation zwischen Java und Prolog verwendet NASfVI, wie in Kapitel 2.3.2 beim Aufbau des Servers erwähnt, jpl von SWI-Prolog. Diese Bibliothek stellt eine Schnittstelle zu Prolog und damit zur Sprachverarbeitungskomponente zur Verfügung, so dass die Prolog-Prädikate der Sprachverarbeitung von Java aus aufgerufen werden können.
Damit SWI-Prolog als externer Prozess für die Verarbeitung des Prolog-Codes gestartet werden kann, muss die Systemeigenschaft `java.library.path` von Java einen Pfad zu SWI-Prolog beinhalten. Die Verwendung von NASfVI ist in Anhang B.1 erklärt.

4.1.1. Kapselung der Sprachverarbeitungskomponente

Die Klasse `de.spartusch.nasfvi.server.Grammar` ist eine Kapselung der Sprachverarbeitungskomponente und stellt letztlich Methoden zum Aufruf der Prädikate `suggest/4`, `parse/5` und `beantworte/5` bereit.
Ein Merkmal der natürlichsprachigen Anfragen ist, dass die Formulierungen stets einer grammatischen Zeitform entsprechen. Diese Zeiten werden, wie in Kapitel 4.2.4 gezeigt, von NASfVI bei der Beantwortung von Suchanfragen interpretiert. Aus diesem Grund

stellt die `Grammar`-Klasse außerdem Java-Konstanten für jede Zeitform bereit, die von der Sprachverarbeitungskomponente unterstützt wird:

```
public enum Tense {
        pqperf, perf, praet, praes, fut1
};
```

Die Konstanten kodieren der Reihe nach das Plusquamperfekt, das Perfekt, das Präteritum, das Präsens und das Futur 1.
Initialisiert wird die Klasse mit einer Startdatei im Konstruktor:

```
public Grammar(final File file) {
        ...
        Query consult = new Query("consult", new Term[] { new Atom(
            file.getAbsolutePath()) });

        if (!consult.hasSolution()) {
                String msg = "Consulting " + file + " failed";
                ...
                throw new RuntimeException(msg);
        }
}
```

Die Sprachverarbeitungskomponente stellt zwei Startdateien zur Verfügung. Die erste Startdatei, `start_gf`, lädt die Sprachverarbeitung mit einem Grundformenlexikon. Die zweite Startdatei, `start_vf`, verwendet dagegen ein effizienteres Vollformenlexikon. Der Konstruktor ermittelt von der jeweils gewählten Datei den absoluten Pfad und konstruiert ein neues Prolog-Atom für diesen Pfad. Prolog-Atome sind in Java stets Objekte der Klasse `jpl.Atom`. Dieses Atom wird dem Prädikat `consult/1` übergeben. Dazu wird ein Objekt der Klasse `jpl.Query` angelegt, welches als erstes Argument den Namen des Prädikats und als zweites Argument ein Array der Terme erwartet. Jeder Term des Arrays stellt ein Argument des Prädikats dar. Die Oberklasse aller Terme ist `jpl.Term`. Von dieser Klasse abgeleitet sind die beiden Klassen `jpl.Atom` für Atome und `jpl.Variable` für Prolog-Variabeln. Mit der Methode `hasSolution()` wird schließlich der Prolog-Code aufgerufen und ermittelt, ob der Aufruf des Prädikats eine Lösung in Prolog besitzt oder nicht. Kann SWI-Prolog die entsprechende Datei nicht laden, schlägt der Aufruf von `consult/1` fehl, `hasSolution()` liefert `false` als Rückgabewert zurück und die Initialisierung des gesamten Servers wird mit einem Laufzeitfehler abgebrochen.
Zur Verwendung in anderen Methoden definiert Grammar die Hilfsmethode `solve(final Query goal)`, um Prolog-Prädikate aufzurufen und um diese Aufrufe zur besseren Nachverfolgbarkeit zu loggen. Das Logging nimmt einen Großteil der Methode ein, ist im nachfolgenden Code jedoch ausgelassen. Da die Logger-Aufrufe generell nichts zur Funktionalität und damit zum Verständnis des Systems beitragen, sind auch in den anderen Code-Ausschnitten dieser Arbeit die Logger-Aufrufe zur besseren Lesbarkeit herausgekürzt.

```
@SuppressWarnings("unchecked")
private Map<String, Term> solve(final Query goal) {
        Map<String, Term> bindings =
```

```
                (Map<String, Term>) goal.oneSolution();
        ...
        return bindings;
}
```

`solve(final Query goal)` verwendet für den Aufruf der Prolog-Prädikate nicht die Methode `hasSolution()`, sondern `oneSolution()`. Diese Methode prüft nicht nur, ob eine Lösung für das Prolog-Prädikat existiert, es liefert auch eventuelle Variabelbelegungen von Prolog zurück. Diese Variabelbindungen werden als Map zurückgegeben. Die Schlüssel der Map sind die Namen der Variabeln als String und deren Werte die jeweilige Belegung als Term. Existiert keine Lösung für einen Aufruf, so gibt `oneSolution()` und damit letztlich auch `solve(final Query goal)` den Wert `null` zurück.

Kapselung von suggest/4

Die Sprachverarbeitungskomponente stellt das Prolog-Prädikat `suggest/4` zur Verfügung, um Satzanfänge zu vervollständigen. Dieses Prädikat wird in der Methode `suggest (final String input)` aufgerufen und ermöglicht es so, von Java aus Satzanfänge zu vervollständigen.

```
public final Set<String> suggest(final String input) {
        Set<String> suggestions = new TreeSet<String>();

        Term[] args = new Term[] {
                new Atom(input),
                new jpl.Integer(8),
                new Variable("Markiertheit"),
                new Variable("Vorschlaege")
        };

        Map<String, Term> bindings =
                solve(new Query("suggest", args));

        if (bindings == null) {
                return suggestions;
        }

        for (Term t : Util.listToTermArray(bindings.get("Vorschlaege"
            ))) {
                String[] tokens = Util.atomListToStringArray(t);
                suggestions.add(fromProlog(tokens).toString());
        }

        return suggestions;
}
```

Die vervollständigten Sätze werden in einem `TreeSet<String>` gesammelt. Diese Collection garantiert, dass sie keine Duplikate enthält und dass die in ihr gespeicherten Sätze lexikographisch sortiert zurückgegeben werden. Das Set wird leer erzeugt und

sollte der Aufruf von `solve` kein Ergebnis zurückliefern, gibt `suggest(final String input)` das leere Set als Ergebnis zurück. Das Prädikat `suggest/4` wird mit dem Eingabesatz als Atom, einer maximalen Markiertheit von 8, sowie den Variabeln „Markiertheit“ und „Vorschlaege“ aufgerufen. Nach einem erfolgreichen Aufruf ist die Variabel „Vorschlaege“ mit einer Liste von vervollständigten Sätzen belegt. Die Hilfsmethode `Util.listToTermArray`, die von jpl in jpl.Util bereitgestellt wird, erzeugt aus dieser Liste, welche in Form von `./2`-Termen vorliegt, ein Array von Termen. Dieses Array wird in einer `for`-Schleife durchlaufen. Jeder Satz wird dabei als `Term t` einzeln verarbeitet und von der Hilfsmethode `Util.atomListToStringArray`, die ebenfalls von jpl in jpl.Util bereitgestellt wird, von einer Liste von Atomen in ein Array von Strings überführt. Die Token des jeweiligen Satzes, die nun als ein Array von Strings vorliegen, werden von der Hilfsmethode `fromProlog(final String[] tokens)` konvertiert und schließlich als String zum Set der vervollständigten Sätze hinzugefügt.
Die Methode `fromProlog(final String[] tokens)`, die Grammar bereitstellt, konvertiert die Sätze aus dem Format, das sie in Prolog haben, in ein besser lesbares Format. So verbindet sie die einzelnen Tokens zu einer Zeichenfolge und setzt bei Blackboxen nur dann Anführungszeichen, wenn sie aus mehreren Begriffen bestehen. Sie wandelt außerdem den ersten Buchstaben der Sätze, sowie den ersten Buchstaben jeder Blackbox in Großbuchstaben um und erzeugt statt oe, ue, ae die entsprechenden Umlaute und statt ss ein scharfes-s. Gibt Prolog also „[eine, 'Vorlesung', ueber, "'syntax"', findet, statt]“ zurück, wandelt `fromProlog` diese Rückgabe in die Zeichenkette „Eine Vorlesung über Syntax findet statt“ um.

Kapselung von parse/5

Das Prädikat `parse/5` führt eine linguistische Analyse der Eingabe durch. Dessen Kapselung, die Methode `parse(final String input, final Analyzer analyzer)`, verwendet diese Analyse, um eine NQuery-Suchanfrage zu erstellen. Die Klasse NQuery für Suchanfragen ist in Kapitel 4.2.4 beschrieben.

```
public final NQuery parse(final String input, final Analyzer analyzer
    ) throws QueryNodeException {
        Term[] args = new Term[] {
                        new Atom(input),
                        new Variable("Tempus"),
                        new Variable("Query"),
                        new Variable("SimilQuery"),
                        new Variable("Gesucht")
        };

        Map<String, Term> bindings = solve(new Query("parse", args));

        if (bindings == null) {
                return null;
        }

        String tempus = bindings.get("Tempus").toString();
```

```
        String query = bindings.get("Query").toString();
        String similQuery = bindings.get("SimilQuery").toString();
        String[] fields = Util.atomListToStringArray(bindings.get("
            Gesucht"));

        return new NQuery(Tense.valueOf(tempus), query, similQuery,
            fields, analyzer);
}
```

Die Methode `parse(final String input, final Analyzer analyzer)` erwartet den zu parsenden Satz als Argument, sowie einen Analyzer. Der Analyzer wird für die Konstruktion der Suchanfrage verwendet. Analyzer sind im folgenden Kapitel 4.2.1 beschrieben.
Zunächst erzeugt die Methode ein Array der Argumente für das Prolog-Prädikat `parse/5`. Dieses Array besteht aus dem zu parsenden Satz als Prolog-Atom, sowie Variabeln für das Tempus, die Suchanfrage, die Ähnlichkeitssuche und die Liste der gesuchten Felder. Führt der Aufruf des Prädikats zu keinem Ergebnis, gibt die Methode den Wert `null` zurück. Andernfalls wird ein NQuery-Objekt mit der von Grammar definierten Konstante für das Tempus, der Suchanfrage und der Ähnlichkeitssuche als String, sowie einem Array mit den Namen der gesuchten Felder konstruiert und zurückgegeben.

Kapselung von beantworte/5

Mit der Kapselung des Prädikats `beantworte/5` können natürlichsprachige Antworten generiert werden.

```
public final String generate(final String input, final Map<String,
    Set<String>> answerValues) {
        Term[] termValues = new Term[answerValues.size()];
        int i = 0;

        for (Map.Entry<String, Set<String>> e : answerValues.entrySet
            ()) {
                String key = e.getKey();
                String[] arr = new String[e.getValue().size() + 1];
                Iterator<String> iter = e.getValue().iterator();

                arr[0] = key;
                for (int j = 1; iter.hasNext(); j++) {
                        String value = iter.next();
                        if (StringMethods.equalsOneOf(key, new String
                            []{"semester", "tag"})) {
                                arr[j] = value.toLowerCase(Locale.
                                    GERMAN);
                        } else {
                                arr[j] = "\"" + value + "\"";
                        }
                }
```

```
                termValues[i] = Util.stringArrayToList(arr);
                i++;
        }

        Term[] args = new Term[] {
                        new Atom(input),
                        Util.termArrayToList(termValues),
                        new Variable("AnalyseAnfrage"),
                        new Variable("AnalyseAntwort"),
                        new Variable("Antwort")
        };

        Map<String, Term> bindings = solve(new Query("beantworte",
            args));

        if (bindings == null) {
                throw new AssertionError("No bindings received");
        }
        ...
}
```

Die Methode `generate(final String input, final Map<String, Set<String» answerValues)` erwartet als Argumente einerseits die zu analysierende natürlichsprachige Anfrage und andererseits eine Map mit den in die Antwort einzusetzenden Werten. Diese Map stellt eine Abbildung von Feldernamen auf Sets mit den einzusetzenden Werten dar. Jedes Feld kann also mehrere einzusetzende Werte beinhalten. Wenn mehrere Werte für ein Feld vorliegen, führt das bei der Generierung der natürlichsprachigen Antwort zur Verwendung von koordinierten Phrasen. Das Hauptaugenmerk liegt in dieser Methode auf der äußeren for-Schleife, welche die Java-Map in eine Prolog-Liste in dem von der Sprachverarbeitungskomponente erwarteten Format umwandelt. Die äußere Schleife iteriert durch die Einträge der Map. Im Schleifenkörper wird unter `arr` ein temporäres Array für die Werte des jeweiligen Eintrags angelegt, sowie ein Iterator erzeugt, der durch die einzelnen Werte iteriert. Die Sprachverarbeitungskomponente erwartet als Format eine Liste der Form [feld, "'wert1"', "'wert2"']. Dementsprechend wird der Name des jeweiligen Felds in das erste Element (`arr[0]`) des temporären Arrays gespeichert und die Werte in die folgenden Elemente. Da es sich bei den Werten für die Felder „semester" und „tag" um keine Blackbox-fähigen Felder handelt, sind die Werte dieser Felder nicht in Anführungszeichen gesetzt. Bei allen anderen Feldern handelt es sich dagegen um Blackboxen. Die Werte dieser Felder werden dementsprechend in Anführungszeichen gesetzt. Mit der von jpl unter jpl.Util bereitgestellten Hilfsfunktion `Util.stringArrayToList` kann das temporäre Array `arr` in eine Prolog-Liste umgewandelt werden. Diese Feldnamen-Werte-Listen werden in dem Array `termValues` gesammelt, welches schließlich mit einer weiteren Hilfsfunktion von jpl namens `Util.termArrayToList` zu einer weiteren Prolog-Liste umgesetzt wird. Zusammen mit der natürlichsprachigen Anfrage und Variabeln für die linguistische Analyse der Anfrage und der Antwort, sowie einer Variabeln für die natürlichsprachige Antwort wird schließlich `beantworte/5` aufgerufen.

Die Belegungen dieser Variabeln dienen schließlich dazu eine Zeichenkette als Rückgabe des Methodenaufrufs von `generate(final String input, final Map<String, Set<String» answerValues)` zu berechnen. Diese Rückgabe erfolgt im JSON-Format und ist in Kapitel 4.3.3 beschrieben. Die Erzeugung dieser Rückgabe ist im hier gezeigten Quellcode-Ausschnitt der Methode nicht zu sehen.

4.1.2. Caching und Normalisierung der Eingaben

Grundsätzlich kapselt die Klasse `de.spartusch.nasfvi.server.Grammar` alle Anfragen an die Sprachverarbeitungskomponente. Es gibt jedoch zwei Funktionen, die außerdem wünschenswert sind:

- Bequeme Normalisierung. Bevor die natürlichsprachigen Anfragen an die Sprachverarbeitungskomponente geschickt werden können, müssen sie normalisiert und für Prolog aufbereitet werden. Idealerweise sind die entsprechenden Aufrufe so gekapselt, dass sich der Aufrufer nicht um diese Normalisierung kümmern muss.
- Caching. Die Sprachverarbeitung mit Prolog ist ein zeitintensiver Vorgang bei der Verarbeitung von natürlichsprachigen Anfragen. Durch Caching kann die Verarbeitung dadurch beschleunigt werden, dass einmal von der Sprachverarbeitungskomponente berechnete Ergebnisse aufbewahrt und bei identischen Anfragen wieder ausgeliefert werden. Auf diese Weise werden nachfolgende Anfragen unter Umgehung der Sprachverarbeitungskomponente direkt aus dem Cache beantwortet. Gecachte Ergebnisse müssen daher nur ein Mal berechnet werden.

Diese beiden Funktionen werden von der Klasse `de.spartusch.nasfvi.server.GrammarManager` bereitgestellt. GrammarManager kapselt dabei die `Grammar`-Klasse.
Im Konstruktor überprüft GrammarManager zunächst, ob die Startdatei `start_vf` vorhanden ist. Ist dies der Fall, dann legt GrammarManager im Attribut `grammar` ein Objekt der Klasse `Grammar` an. Der Analyzer, der im Konstruktor an GrammarManager übergeben wird, wird später für Aufrufe der `parse`-Methode von `grammar` benutzt. Dadurch muss der Analyzer nur beim Instanziieren des GrammarManagers bekannt sein und nicht mehr bei jeder Verwendung von `parse`:

```
public GrammarManager(final Resources res, final Analyzer analyzer) {
        File startFile = res.getFile("nasfi.StartFile", "/WEB-INF/
            classes/grammar/start_vf");

        if (startFile == null) {
                throw new IllegalArgumentException("Start file not
                    found");
        }

        grammar = new Grammar(startFile);
        suggestionsCache = new SoftCache<String, Set<String>>();
        this.analyzer = analyzer;
}
```

Die `GrammarManager`-Klasse legt im Konstruktor einen Cache für die Speicherung von Satzvervollständigungen (Suggestions) im Attribut `suggestionsCache` an. Der Cache ist durch die Klasse `de.spartusch.SoftCache` implementiert. Er stellt eine Abbildung von natürlichsprachigen Eingaben auf gespeicherte Vervollständigungen der Eingabe dar. Da SoftCache die starken, weichen und schwachen Referenzbindungen in Java ausnutzt,[1] wächst der Cache nur so lange, wie freier Arbeitsspeicher zur Verfügung steht. Sobald der zur Verfügung stehende Speicher knapp wird, beginnt der Garbage Collector von Java Einträge aus dem Cache zu entfernen und deren Speicher wieder freizugeben.
In der gegenwärtigen Implementierung werden nur die Satzvervollständigungen gecachet, da sie einerseits am aufwändigsten zu berechnen sind und andererseits durch die Platzhalter in den Blackboxen auch die am allgemeinsten berechneten Ergebnisse der Sprachverarbeitungskomponente sind. Bei einem großflächigeren Einsatz des Systems sollten jedoch weitere Caching-Strategien in Betracht gezogen werden.
Die Implementierung der `suggest(final String input)`-Methode in GrammarManager normalisiert zunächst die natürlichsprachige Anfrage und prüft dann den Cache:

```
public final Set<String> suggest(final String input) {
        String normInput = Grammar.toProlog(input);
        Set<String> suggestions = suggestionsCache.get(normInput);

        if (suggestions != null) {
                return suggestions;
        } else {
                suggestions = grammar.suggest(normInput);
                if (suggestions != null) {
                        suggestionsCache.put(normInput, suggestions);
                }
                return suggestions;
        }
}
```

Die von Grammar bereitgestellte Methode `toProlog(final String input)` führt eine Normalisierung der Eingabe durch. Die Methode wandelt die Eingabe in ein Format um, dass die Sprachverarbeitungskomponente für Eingaben erwartet: alle Großbuchstaben werden durch entsprechende Kleinbuchstaben ersetzt, deutsche Umlaute, sowie das scharfe-s werden in ihre ASCII-Maskierungen umgewandelt (ß in ss, ä in ae, ü in ue, ö in oe) und alle Zeichen, die weder ein Buchstabe, eine Zahl, ein doppeltes Anführungszeichen, '/', '.' oder '-' sind werden durch Leerzeichen ersetzt.
Nach dieser Normalisierung der natürlichsprachigen Anfrage wird mit der normalisierten Eingabe der Cache durchsucht. Liefert der Cache den Wert `null` für eine Eingabe zurück, so sind keine gecachten Satzvervollständigungen für diese Eingabe vorhanden. In diesem

[1] Starke Referenzen werden von dem Garbage Collector in Java nicht freigegeben, schwache in Form der Klasse `java.lang.ref.WeakReference` dagegen sofort. Weiche Referenzen in Form der Klasse `java.lang.ref.SoftReference` werden von dem Garbage Collector jedoch erst freigegeben, wenn der zur Verfügung stehende Speicher knapp wird. `SoftCache` nutzt eine `java.util.WeakHashMap`, deren Schlüssel schwach referenziert sind, und speichert die Werte weich referenziert mit starken Referenzen zu den Schlüsseln, um das gewünschte Verhalten zu erreichen.

Fall berechnet die Sprachverarbeitungskomponente die Vervollständigungen und `suggest` speichert diese, falls die Eingabe vervollständigt werden konnte, im Cache.
Die `parse`- und `generate`-Methoden der GrammarManager-Klasse führen ebenfalls eine Normalisierung der Eingaben durch und kapseln die entsprechenden Methoden der Grammar-Klasse:

```
public final NQuery parse(final String input) throws
    QueryNodeException {
        return grammar.parse(Grammar.toProlog(input), analyzer);
}

public final String generate(final String input, final Map<String,
    Set<String>> values) {
        return grammar.generate(Grammar.toProlog(input), values);
}
```

4.2. Die Volltextsuche

Die Volltextsuche wird mit der Java-Bibliothek „Lucene“ der Apache Software Foundation umgesetzt. Wie in Kapitel 2.3.1 dargestellt, basiert die Suche in Lucene auf Dokumenten und Feldern. Jedes Dokument besteht dabei aus Feldern, die gezielt durchsucht werden können, und stellt einen eigenen Datensatz dar. Die Dokumente sind im sogenannten Index erfasst und gespeichert. Bevor ein neues Dokument in den Index aufgenommen werden kann, muss es vorverarbeitet und dessen Felder tokenisiert werden. Die gleiche Vorverarbeitung ist auch bei der Analyse der Suchanfragen (Queries) notwendig, da im Index nur nach den verarbeiteten Tokens gesucht werden kann.

4.2.1. Vorverarbeitung von Dokumenten

Die in den Feldern enthaltenen Informationen der Dokumente sind in erster Linie Zeichenketten (Strings). Damit diese Zeichenketten nach einzelnen Begriffen effizient durchsucht werden können, müssen sie tokenisiert werden. Die Tokenisierung und Normalisierung geschieht durch Analyzer, welche die Vorverarbeitung auf Feldebene durchführen und in der Regel mehrere Filter zur Vorverarbeitung anwenden. Jeder Filter führt einen Arbeitsschritt, wie zum Beispiel die Normalisierung zu Kleinbuchstaben oder das Entfernen von Stoppwörtern, aus. Ein Analyzer erbt in Lucene stets von der Klasse `org.apache.lucene.analysis.Analyzer` und implementiert deren abstrakte Methode `tokenStream(String fieldName, Reader reader)`. In dieser Methode wird der Inhalt eines Feldes verarbeitet. NASfVI verwendet einen eigenen Analyzer, der durch die Klasse `de.spartusch.nasfvi.server.NAnalyzer` implementiert ist.
NAnalyzer verwendet eine eigene Stopwort-Liste[2] mit einer Vielzahl von Begriffen, die weit über die üblichen Funktionswörter hinausgehen. So befinden sich auch häufig vorkommende Adjektive, Adverbien und Verbformen in der Liste, um die Relevanz von für

[2] Die Stopwort-Liste befindet sich in `war/WEB-INF/stopwords.txt`.

Vorlesungsdaten bedeutsame, inhaltliche Nomen bei der Suche zu erhöhen. Begriffe, die sich auf der Stopwort-Liste befinden, werden bei der Indizierung nicht berücksichtigt, wodurch sie das tf-idf-Maß[3] anderer Terme nicht beeinflussen.
Außerdem verwendet NAnalyzer eine Liste zur Kompositazerlegung.[4] Diese Liste ist eine Zusammenstellung einiger einfacher Wörter mit deren Hilfe die Klasse `org.apache.lucene.analysis.compound.DictionaryCompoundWordTokenFilter` die Bestandteile von Komposita in einzelne Terme zerlegen kann. So wird zum Beispiel „computerlinguistik“ aufgrund der Einträge „computer“ und „linguistik“ in der Liste zur Kompositazerlegung mit den Termen „computerlinguistik“, „computer“ und „linguistik“ indiziert. Eine Suche nach „linguistik“ führt daher auch bei einem Vorkommen von „computerlinguistik“ zu einem Treffer.
Lucene ruft für jedes Feld eines Dokuments und jedes Feld einer Suchanfrage die Methode `tokenStream(String fieldName, Reader reader)` einzeln auf. Während `reader` den Zugriff auf die Daten des Felds ermöglicht, enthält `fieldName` den Namen des jeweiligen Feldes. Je nach Feld verwendet NAnalyzer verschiedene Filter zur Verarbeitung des Feldinhaltes. Es unterscheidet dabei drei Gruppen von Feldern:

```
@Override
public final TokenStream tokenStream(final String fieldName,
                final Reader reader) {
        if (StringMethods.equalsOneOf(fieldName, NATURAL_TEXT_FIELDS)
            ) {
                return naturalText(reader);
        } else if ("semester".equals(fieldName)) {
                return singleToken(reader);
        } else {
                return simpleText(reader);
        }
}
```

Die erste Gruppe von Feldern stellen die Felder „titel“, „beschreibung“ und „typ“ dar, welche durch die Konstante `NATURAL_TEXT_FIELDS` gegeben sind:

```
private TokenStream naturalText(final Reader reader) {
        TokenStream result = new StandardTokenizer(Version.LUCENE_33,
                reader);

        result = new StandardFilter(Version.LUCENE_33, result);
        result = new LowerCaseFilter(Version.LUCENE_33, result);
        result = new StopFilter(Version.LUCENE_33, result,
                stopWords);
```

[3] Das *tf-idf*-Maß gewichtet Terme nach deren Vorkommen in einem Dokument (*Term Frequency*) im Verhältnis zu der Anzahl an Dokumenten, in denen sie vorkommen (*Inverse Document Frequency*). Diese Gewichtung von Termen gibt damit an, wie einmalig und wie beschreibend ein Term für ein Dokument ist. Das Maß ist im Information Retrieval gängige Praxis, um die Reihenfolge von Suchergebnissen - deren Ranking - in Abhängigkeit von den verwendeten Suchbegriffen zu bestimmen. Auch Lucene verwendet das tf-idf-Maß als ein Faktor in dessen Ranking-Algorithmus ([LUC10], Seite 86).
[4] Diese Liste befindet sich in `war/WEB-INF/komposita.txt`.

```
        result = new DictionaryCompoundWordTokenFilter(Version.
            LUCENE_33, result, compounds);
        result = new SnowballFilter(result, "German2");

        return result;
}
```

Der StandardTokenizer tokenisiert zunächst den Eingabestrom. Bis auf wenige Ausnahmefälle erzeugt der StandardTokenizer immer dann ein neues Token, wenn ein Satzzeichen, ein Bindestrich oder ein Leerzeichen auftritt. Der StandardFilter arbeitet Hand in Hand mit dem StandardTokenizer und entfernt u. a. Punkte von Akronymen. Der LowerCaseFilter wiederum normalisiert alle Token zu Kleinbuchstaben.[5] Und der StopFilter entfernt die Tokens, welche Stoppwörtern entsprechen, aus dem Stream. DictionaryCompoundWordTokenFilter schließlich spaltet Komposita in mehrere Token auf. Als Letztes wird der SnowballFilter angewendet, welcher einen Stemming-Algorithmus[6] verwendet, um die flektierten Formen der Token jeweils auf einen Stamm abzubilden. Durch diesen Filter werden zum Beispiel die beiden Token „ausdrücke“ und „ausdruecke“ jeweils auf den Stamm „ausdruck“ abgebildet. Indiziert und gesucht werden also letztlich die Stämme der Token und nicht deren flektierte Formen.

Die zweite Gruppe von Feldern stellt das Feld „semester“ dar:

```
private TokenStream singleToken(final Reader reader) {
        return new KeywordTokenizer(reader);
}
```

Das Feld „semester“ wird nicht in einzelne Token zerlegt. Stattdessen wird durch den `KeywordTokenizer` der gesamte Feldinhalt als *ein* Token indiziert. Der Inhalt dieses Feldes besteht immer aus einer Jahreszahl für Sommersemester oder aus zwei Jahreszahlen der Form „2009/2010“ für Wintersemester. Würden die Wintersemester weiter zerlegt werden - im gegebenen Beispiel zu „2009“ und „2010“ - würden Suchanfragen nach Sommersemestern auch Wintersemester zum Ergebnis haben und umgekehrt. Da die Semesterangaben so jedoch stets als Ganzes indiziert werden, können sie als eindeutiges und exaktes Auswahlkriterium dienen.

Alle anderen Felder werden als dritte Gruppe verarbeitet:

```
private TokenStream simpleText(final Reader reader) {
        TokenStream result = new StandardTokenizer(Version.LUCENE_33,
                reader);

        result = new StandardFilter(Version.LUCENE_33, result);
        result = new LowerCaseFilter(Version.LUCENE_33, result);
        result = new StopFilter(Version.LUCENE_33, result,
                stopWords);
```

[5] Für eine Beschreibung dieser Filter siehe ([LUC10], Seite 118)

[6] „German2“ bezeichnet eine Abwandlung des Snowball-Stemmers für das Deutsche, welche ae, oe und ue als Umschreibung für Umlaute interpretiert. Der Algorithmus ist unter `http://snowball.tartarus.org/algorithms/german2/stemmer.html` beschrieben.

```
        result = new SnowballFilter(result, "German2");

        return result;
}
```

Diese letzte Gruppe von Feldern besteht aus den Feldern „dozent“ und „termin“. Deren Verarbeitung entspricht im Wesentlichen dem Vorgehen bei den Feldern „titel“, „beschreibung“ und „typ“. Der einzige Unterschied besteht darin, dass hier keine Kompositazerlegung durchgeführt wird. Eine Kompositazerlegung scheint insbesondere bei Eigennamen nicht notwendig. Doppelnamen, die mit einem Bindestrich geschrieben werden, werden bereits durch den StandardTokenizer erkannt und getrennt.

4.2.2. Repräsentation von Semesterangaben

Zwar wird bei jedem im Index erfassten Dokument die Semesterangabe im Feld „semester“ als Token indiziert. Doch für die Verarbeitung und Evaluierung der Semesterangaben ist eine zusätzliche Repräsentation der Semester als Objekte sinnvoll. NASfVI implementiert zu diesem Zweck die Klasse `de.spartusch.nasfvi.server.Semester`:

```
public class Semester {
        private String begin;
        private String end;
        private String canonical;
        private boolean isWinter;
        ...
```

Die Klasse berechnet für jede Semesterangabe ein auf den Tag genaues Anfangsdatum und speichert es in dem Attribut `begin`, sowie ein Enddatum gleicher Genauigkeit und speichert dieses im Attribut `end`. Das Attribut `canonical` enthält die Text-Repräsentation des jeweiligen Semesters in derselben Form wie das Feld „semester“ der Dokumente. Bei `isWinter` handelt es sich um den booleschen Wert dafür, ob das jeweilige Semester ein Wintersemester ist oder nicht. Die Klasse stellt für diese Attribute Getter[7] nach den Java-Konventionen bereit: `getBegin`, `getEnd`, `getCanonical`, sowie abweichend vom Attributnamen `isWinterSemester`.
Für den Anfang jedes Sommersemesters wird der 22. Februar angenommen und für dessen Ende der 21. Juli. Bei Wintersemestern wird der 22. Juli als Anfangsdatum angenommen und der 21. Februar als Enddatum. Diese Datumsangaben weichen von den üblichen Vorlesungszeiträumen ab. Denn der Zweck dieser Klasse ist die in Kapitel 4.2.4 gezeigte Interpretation des grammatischen Tempus bei Suchanfragen. Die gewählten Zeitpunkte unterteilen jedes Jahr in ein Sommer- und ein Wintersemester. Diese Zeitpunkte orientieren sich dabei an dem angenommenen Interesse der Studierenden für die jeweiligen Semester. Das Wintersemester beginnt also zum Beispiel im Juli. Dem liegt die Annahme

[7] Als Getter werden Methoden bezeichnet, die Werte von privaten Attributen zurückgeben und deren Bezeichnungen sich in der Regel aus den Attributnamen und einem vorangestellten `get` oder, oft bei booleschen Werten, einem vorangestellten `is` zusammensetzen. Siehe auch `http://de.wikipedia.org/wiki/Zugriffsfunktion`

zugrunde, dass sich Studenten am Ende des Sommersemesters nicht mehr für die Veranstaltungen des Sommersemesters, sondern für die des kommenden Wintersemesters interessieren. Diese Zeitpunkte sind allerdings willkürlich festgelegt. In weiteren Arbeiten muss noch evaluiert werden, ob die Zeitpunkte das studentische Interesse optimal abbilden.
Die `Semester`-Klasse kann sowohl ohne als auch mit einer Semesterangabe instanziiert werden. Wird der Konstruktor mit einer Semesterangabe aufgerufen, werden die für dieses jeweilige Semester angenommenen Daten berechnet:

```
public Semester(final String canonical) {
        // canonical == 2007/2008 or 2008
        String[] year = canonical.split("/");
        if (year.length == 2) {
                // Winter semester
                begin = year[0] + "0722"; // 22.07.
                end = year[1] + "0221"; // 21.02.
                isWinter = true;
        } else {
                // Summer semester
                begin = year[0] + "0222"; // 22.02.
                end = year[0] + "0721"; // 21.07.
                isWinter = false;
        }
        this.canonical = canonical;
}
```

Die Datumsangaben für die Semester werden als einfache Zeichenketten (Strings) gespeichert, so dass sie auch direkt im Index indiziert werden können. Das Format entspricht dabei immer `yyyyMMdd`, also der vierstelligen Jahreszahl, gefolgt von einer zweistelligen Monatsangabe und einer zweistelligen Tagesangabe. Der 22. Juli 2011 wird also als `20110722` repräsentiert. Der Vorteil dieses Formats besteht darin, dass die Datumsangaben so auch bei einfachen lexikographischen Vergleichen in die gewünschte Reihenfolge sortiert werden.
Wird der Konstruktor der Klasse ohne eine Semesterangabe aufgerufen, wird das aktuelle Semester ermittelt und die entsprechenden Daten werden berechnet:

```
public Semester() {
        Date nowDate = new Date();
        SimpleDateFormat formatter =
                new SimpleDateFormat("yyyyMMdd");
        String now = formatter.format(nowDate);
        formatter.applyPattern("yyyy");
        String year = formatter.format(nowDate);
        String summerBegin = year + "0222"; // 22.02.
        String summerEnd = year + "0721"; // 21.07.

        // Winter semester
        isWinter = true;
        if (now.compareTo(summerBegin) < 0) {
```

```
                String yearBegin =
                        Integer.toString(Integer.parseInt(year) - 1);
                begin = yearBegin + "0722";
                end = year + "0221";
                canonical = yearBegin + "/" + year;
        } else if (now.compareTo(summerEnd) > 0) {
                String yearEnd =
                        Integer.toString(Integer.parseInt(year) + 1);
                begin = year + "0722";
                end = yearEnd + "0221";
                canonical = year + "/" + yearEnd;
        } else {
                // Summer semester
                begin = summerBegin;
                end = summerEnd;
                canonical = year;
                isWinter = false;
        }
}
```

Zunächst wird dabei das aktuelle Datum ermittelt und im `yyyyMMdd`-Format in der Variabel `now` gespeichert. Als nächstes wird das aktuelle Jahr als vierstellige Zahl in `year` gespeichert und genutzt, um die Daten für den Anfang des Sommersemesters und dessen Ende im aktuellen Jahr im `yyyyMMdd`-Format zu berechnen. Diese Werte werden anschließend mit `now` verglichen, um festzustellen, in welchem Semester man sich derzeit befindet. Liegt das aktuelle Datum vor oder nach dem Sommersemester im aktuellen Jahr, so ist derzeit ein Wintersemester. Die korrekte kanonische Repräsentation des jeweiligen Wintersemesters kann durch die Unterscheidung dieser beiden Fälle problemlos berechnet werden.

4.2.3. Index und Indizierung der Dokumente

Das Herzstück der Volltextsuche ist der Suchindex. In diesem Index sind einerseits die Vorlesungsdaten gespeichert. Andererseits sind diese Daten aber vor allem auch durch den im vorausgegangenen Kapitel dargestellten Analyzer verarbeitet und entsprechend indiziert worden. Der Suchindex kann nach den so gewonnen Token durchsucht werden und er kann die zu einer Suchanfrage passenden, in ihm gespeicherten Dokumente zur Verfügung stellen.

Der Suchindex

Der Suchindex wird in Lucene durch die abstrakte Klasse `org.apache.lucene.store.Directory` bereitgestellt. Von dieser abstrakten Klasse unterstützt NASfVI die Verwendung zweier konkreter Implementierungen:

- `org.apache.lucene.store.FSDirectory`: Bei dieser Implementierung wird der Index in einem Verzeichnis des Dateisystems geschrieben. Er ist damit persistent und

über die Ausführung von NASfVI hinaus verfügbar. Mit dieser Implementierung ist es zudem möglich, den Suchindex mit einem Hilfsprogramm wie Luke[8] zu untersuchen.

- `org.apache.lucene.store.RAMDirectory`: Diese Implementierung speichert den Index nicht im Dateisystem, sondern hält ihn vollständig im Arbeitsspeicher des Computers. Kleine Indices mit nur einigen hundert Dokumenten, wie der Index von NASfVI, können so sehr schnell durchsucht und verarbeitet werden, da keine zeitaufwendigen Festplattenzugriffe notwendig sind.

Diese von Lucene zur Verfügung gestellten Implementierungen werden von NASfVI in der Klasse `de.spartusch.nasfvi.server.XmlIndex` gekapselt. Welche Implementierung XmlIndex verwendet, entscheidet sich beim Aufruf des Konstruktors:

- `XmlIndex(final File dir, final boolean newIndex, final Analyzer analyzer)`: Bei Verwendung dieses Konstruktors wird FSDirectory als Implementierung gewählt. Der Index wird in `dir` angelegt und falls `newIndex` den Wert `true` hat, wird ein bestehender Index gelöscht, bevor ein neuer angelegt wird. Andernfalls wird ein bestehender Index geöffnet und gegebenenfalls erweitert.
- `XmlIndex(final Analyzer analyzer)`: Dieser Konstruktor verwendet RAMDirectory als Implementierung für den Index.

Die `analyzer` bei den Konstruktoraufrufen werden von XmlIndex bei der Indizierung neuer Vorlesungsdaten verwendet. Jeder Konstruktoraufruf initialisiert zu diesem Zweck das Attribut `config` mit einer `org.apache.lucene.index.IndexWriterConfig`:

```
config = new IndexWriterConfig(Version.LUCENE_33, analyzer);
```

Indizierung von Veranstaltungsdaten

Für das Hinzufügen neuer Vorlesungsdaten implementiert XmlIndex einen SAX-Parser[9] für das in Kapitel 2.3.1 vorgestellte XML-Format für Vorlesungsdaten. Dieser Parser ist als private Klasse `XmlIndexHandler` innerhalb von XmlIndex implementiert und wird mit der Methode `ingest(final InputStream xmlSource)` aufgerufen. XmlIndexHandler verwendet `config`, um einen `org.apache.lucene.index.IndexWriter` zu erzeugen. Beim Parsen neuer Vorlesungsdaten werden mit IndexWriter neue Vorlesungsdokumente in den Index geschrieben.

```
private final class XmlIndexHandler extends DefaultHandler {
        private static final String NEW_DOC_TAG = "veranstaltung";
        private static final String ROOT_TAG = "veranstaltungen";
```

[8] Luke ermöglicht es Lucene-Indices zu öffnen und genauer zu untersuchen. Das Programm ist unter `http://code.google.com/p/luke/` verfügbar.

[9] Auf die grundlegende Funktionsweise von SAX-Parsern sei an dieser Stelle nicht weiter eingegangen. Eine gute Beschreibung lässt sich bei Wikipedia finden: `http://de.wikipedia.org/wiki/Simple_API_for_XML`

```
        private Document doc;
        private StringBuilder currentField;
        private IndexWriter writer;
        ...

        public XmlIndexHandler() throws IOException {
                writer = new IndexWriter(index, config);
        }
```

XmlIndexHandler baut während des Parsens jeweils ein Dokument mit Vorlesungsdaten nach dem anderen auf. Das Dokument, das aktuell aufgebaut wird, wird in dem Attribut `doc` zwischengespeichert. Da der SAX-Parser sequentiell dem XML-Dokument entsprechend Events auslöst, wird auch immer nur ein Feld gleichzeitig aufgebaut. Der Inhalt des aktuellen Feldes wird in dem Puffer `currentField` erfasst. Sobald das durch `NEW_DOC_TAG` bezeichnete Tag im XML geschlossen wird, wird das aktuelle Dokument mit `writer` in den Index geschrieben. Die Konstante `ROOT_TAG` gibt das Wurzelelement des XML-Formats an.

```
@Override
public void startElement(final String uri, final String localName,
                final String qName, final Attributes atts)
                throws SAXException {
        if (NEW_DOC_TAG.equals(localName)) {
                doc = new Document();
                currentField = null;
        } else if (!ROOT_TAG.equals(localName)) {
                if (doc == null) {
                        throw new RuntimeException("newDocumentTag␣
                            missing");
                }
                currentField = new StringBuilder();
        }
}
```

Liest der Parser ein öffnendes `NEW_DOC_TAG` ein, d. h. wird `<veranstaltung>` eingelesen, beginnt ein neuer Datensatz. XmlIndexHandler legt daher ein neues Dokument an und verwirft einen eventuell vorhandenen, alten Puffer für das aktuelle Feld. Bei jedem anderen Tag, das geöffnet wird, wird ein neuer Puffer für den Feldinhalt angelegt. Einzige Ausnahme davon ist das durch `ROOT_TAG` bezeichnete Wurzelelement - es enthält keine textuellen Daten für ein Feld.

```
@Override
public void characters(final char[] ch, final int start, final int
    length) throws SAXException {
        if (currentField == null) {
                return;
        }
        for (int i = start; i < start + length; i++) {
                currentField.append(ch[i]);
```

```
        }
}
```

Enthält ein XML-Knoten Textzeichen als Inhalt, fügt XmlIndexHandler die eingelesenen Zeichen einzeln dem Puffer für das aktuelle Feld hinzu. Die `characters`-Methode muss nicht mit einem Aufruf den gesamten textuellen Inhalt des Knotens erfassen, sondern kann von der zugrunde liegenden Implementierung auch mehrmals aufgerufen werden. Aus diesem Grund wird mit den gelesenen Zeichen nicht sofort ein Feld erzeugt, sondern die Zeichen zunächst dem Puffer hinzugefügt und auf das schließende Tag des Knotens gewartet.
Die Methode `endElement(final String uri, final String localName, final String qName)` für schließende Tags ist analog zur `startElement`-Methode für öffnende Tags aufgebaut. Wenn ein schließendes `NEW_DOC_TAG` verarbeitet wird, d. h. der Parser ein `</veranstaltung>` eingelesen hat, wird das aktuelle Dokument vervollständigt und in den Index geschrieben:

```
if (NEW_DOC_TAG.equals(localName)) {
        ...
        addDocumentId();
        addSemesterBeginEnd();
        writer.addDocument(doc);
        ...
}
```

Die Methode `addDocumentId()` fügt dem aktuellen Dokument eine eindeutige ID hinzu. Diese ID entspricht der Anzahl der bisher in dem Index vorhandenen Dokumente. Die IDs der Dokumente stellt also eine fortlaufende Nummerierung dar:

```
private void addDocumentId() throws IOException {
        String id = String.valueOf(writer.numDocs());
        Field field = new Field("id", id, Field.Store.YES, Field.
            Index.NOT_ANALYZED);
        doc.add(field);
}
```

Die ID wird in dem Feld „id“ gespeichert, ist im Index abrufbar (`Field.Store.YES`) und ist indiziert, jedoch nicht tokenisiert (`Field.Index.NOT_ANALYZED`). Ein solches ID-Feld ist notwendig, damit die im Index gespeicherten Dokumente eindeutig angesprochen und abgerufen werden können. Zwar ist diese eindeutige Adressierung von Dokumenten im Index für die gewöhnliche Suchfunktion von Lucene nicht notwendig. Sie ist für die Implementierung der im folgenden Kapitel 4.2.5 besprochenen Ähnlichkeitssuche jedoch eine Voraussetzung.
Bevor `writer` das Dokument in den Index schreiben kann, wird es mit der Methode `addSemesterBeginEnd()` noch um exakte Datumsangaben für den Anfang und Ende des jeweiligen Semesters ergänzt:

```
private void addSemesterBeginEnd() {
        Semester sem = new Semester(doc.get("semester"));
```

```
            Field field = new Field("semester_beg", sem.getBegin(), Field
                .Store.NO, Field.Index.NOT_ANALYZED);
            doc.add(field);

            field = new Field("semester_end", sem.getEnd(), Field.Store.
                NO, Field.Index.NOT_ANALYZED);
            doc.add(field);
    }
```

Die Methode legt dazu das Feld `semester_beg` für den Semesterbeginn und das Feld `semester_end` für das Semesterende an. Eine Besonderheit bei diesen beiden Feldern ist, dass sie zwar indiziert werden (`Field.Index.NOT_ANALYZED`), aber nicht mit dem Dokument gespeichert werden (`Field.Store.NO`). Das bedeutet, dass im Index zwar nach `semester_beg` und `semester_end` gesucht werden kann und die erfassten Dokumente auch entsprechend gefunden werden können. Doch die Werte dieser beiden Felder können einem Ergebnis-Dokument nicht mehr entnommen werden. Die beiden Felder werden also ausschließlich für die Indizierung der Dokumente genutzt und nicht zusammen mit den Dokumenten gespeichert.

Wenn die `endElement`-Methode ein anderes schließendes Element als das von `NEW_DOC_TAG` liest, erzeugt sie aus dem Inhalt des Puffers für das aktuelle Feld ein neues Feld und fügt es dem aktuellen Dokument hinzu. Wie schon bei der `startElement`-Methode ist auch hier `ROOT_TAG` die einzige Ausnahme, da es keinen entsprechenden textuellen Inhalt besitzt:

```
else if (!ROOT_TAG.equals(localName)) {
            String value = currentField.toString();
            Field.TermVector storeVector = Field.TermVector.NO;
            Field.Store storeField = Field.Store.YES;
            float boost = 1.0f;

            if ("titel".equals(localName)) {
                    storeVector = Field.TermVector.YES;
                    boost = 2.5f;
            } else if ("beschreibung".equals(localName)) {
                    storeVector = Field.TermVector.YES;
                    storeField = Field.Store.NO;
                    boost = 1.5f;
            }

            Field field = new Field(localName, value, storeField, Field.
                Index.ANALYZED, storeVector);
            field.setBoost(boost);
            doc.add(field);
    }
```

Eine bisher nicht gezeigte Möglichkeit bei der Erstellung von Feldern ist das Speichern von Termvektoren. Ein Termvektor ist eine Datenstruktur bestehend aus den Termen des Feldes eines Dokuments und den Termhäufigkeiten im Feld ([LUC10], Seite 44f). Jeder Term lässt sich als eine Dimension eines multidimensionalen Vektors auffassen, deren

Ausprägung der Termhäufigkeit entspricht. Mit dieser Repräsentation des Inhalts der Felder kann die inhaltliche Ähnlichkeit zweier Felder in zwei Dokumenten als Abstand der Termvektoren berechnet werden. Die in Kapitel 4.2.5 beschriebene Ähnlichkeitssuche nutzt Termvektoren, um einander ähnliche Dokumente zu finden. Da die Ähnlichkeitssuche auf den Feldern „titel“ und „beschreibung“ basiert, werden die Termvektoren für diese beiden Felder berechnet und mit dem jeweiligen Feld im Index gespeichert (`Field.TermVector.YES`). Bei allen anderen Feldern werden keine Termvektoren gespeichert (`Field.TermVector.NO`).
Der Inhalt der meisten Felder wird im Index zusammen mit dem jeweiligen Dokument gespeichert und kann daher auch in Suchergebnissen, also in den Ergebnis-Dokumenten, abgefragt werden. Da die Beschreibungen der Veranstaltungen in der implementierten Grammatik des Deutschen nicht erfragt werden können, wird das Feld „beschreibung“ als solches auch nicht gespeichert (`Field.Store.NO`). Es werden lediglich dessen Token indiziert und dessen Termvektor für die Ähnlichkeitssuche gespeichert.
Neben dem in Kapitel 4.2.1 erwähnten tf-idf-Maß ist das Boosting ein weiterer Faktor im Ranking-Algorithmus von Lucene ([LUC10], Seite 86f). In Lucene können einzelne Felder eines Dokuments „geboostet“ werden ([LUC10], Seite 49f). Der Boost ist dabei ein Faktor in Form einer Gleitkommazahl, der bei der Berechnung der Relevanz der Terme eines Feldes miteinfließt. In NASfVI erhalten die Felder „titel“ und „beschreibung“ einen besonderen Boost-Faktor, da sie thematische Treffer bei einer Suchanfrage darstellen. Wie an dem Ausschnitt aus der `endElement`-Methode zu sehen ist, erhält das Feld „beschreibung“ einen Boost-Faktor von 1.5 und das Feld „titel“ einen Boost-Faktor von 2.5, während alle anderen Felder keinen besonderen Boost, sondern einen Standard-Faktor von 1.0 erhalten. Das bedeutet, dass ein Treffer (also ein zu einer Suchanfrage passendes Token) im Titel einer Veranstaltung 2,5fach höher gewertet wird als ein Treffer in einem der Felder ohne besonderen Boost. Dokumente mit Treffern in den Feldern „titel“ oder „beschreibung“ führen daher in der Regel im Relevanzranking der Suchergebnisse und garantieren so bei Suchanfragen nach konkreten Vorlesungstiteln und bei der Suche nach Vorlesungsthemen gute Ergebnisse.

Der letzte Verarbeitungsschritt beim Parsen eines XML-Dokuments mit Veranstaltungsdaten ist mit dem Ende des XML-Dokuments erreicht. XmlIndexHandler überträgt dabei alle neuen Dokumente mit einem `commit()` in den Index und führt eine abschließende Optimierung des Index durch:

```
@Override
public void endDocument() throws SAXException {
        ...
        writer.commit();
        writer.optimize();
        ...
}
```

4.2.4. Repräsentation der Suchanfragen

Der Index kann mit Suchanfragen nach Dokumenten durchsucht werden. Die Suchanfragen sind dabei durch die Klasse `de.spartusch.nasfvi.server.NQuery` gekapselt. Wie in Kapitel 3.4.2 gezeigt, berechnet die Sprachverarbeitungskomponente bis zu zwei Suchanfragen pro natürlichsprachiger Anfrage. Diese beiden Suchanfragen werden in `de.spartusch.nasfvi.server.NQuery` erfasst:

1. Eine allgemeine Suchanfrage, die ganz allgemein eine Veranstaltung ermittelt. Diese Suchanfrage wird von NQuery in dem Attribut `query` gespeichert, welches immer gesetzt wird.

2. Eine Ähnlichkeitssuche. Die Ähnlichkeitssuche ermittelt ein Vergleichsdokument. Das Ergebnis-Dokument für die natürlichsprachige Anfrage muss sowohl die allgemeine Suchanfrage erfüllen, als auch zu dem Vergleichsdokument ähnlich sein. Eine Ähnlichkeitssuche wird nur durch das Verb *„ähneln“* in der natürlichsprachigen Anfrage ausgelöst. Die Suchanfrage der Ähnlichkeitssuche wird in dem Attribut `similQuery` gespeichert.

Analyse der Suchanfragen

Die Suchanfragen setzen sich in Lucene immer aus Unterklassen der abstrakten Klasse `org.apache.lucene.search.Query` zusammen. Es gibt Klassen wie `TermQuery` für die Terme einer Suchanfrage und Klassen wie `BooleanQuery`, die den Aufbau der Suchanfragen beschreiben - in diesem Fall durch einen booleschen Operator. Analysierte Suchanfragen sind in Lucene immer Objekte dieser Klassen. Enthält eine Suchanfrage nur einen Term, kann sie als `TermQuery` repräsentiert werden. In der Regel besitzen die Suchanfragen jedoch mehrere Terme und werden daher als `BooleanQuery` mit weiteren, untergeordneten `TermQuery`- oder `BooleanQuery`-Objekten repräsentiert.
Für die Analyse und das Parsen der beiden Suchanfragen, die von der Sprachverarbeitungskomponente als String geliefert werden, verwendet NQuery einen im Konstruktor übergebenen Analyzer. Bei diesem Analyzer muss es sich um denselben Analyzer wie bei der Indizierung handeln, damit die extrahierten Token zueinander kompatibel sind. NASfVI verwendet daher in beiden Fällen den in Kapitel 4.2.1 besprochenen NAnalyzer. Im Konstruktor instanziiert NQuery einen `org.apache.lucene.queryParser.standard.StandardQueryParser` mit dessen Hilfe die Suchanfragen unter Verwendung des Analyzers zu Query-Objekten geparset werden können:

```
StandardQueryParser qp = new StandardQueryParser(analyzer);
qp.setDefaultOperator(Operator.AND);
```

Als Standard-Operator wird `Operator.AND` gesetzt. Daher wird beim Parsen automatisch ein Und-Operator eingefügt, wenn zwischen zwei Feldern einer Suchanfrage kein Operator angegeben ist. Der Suchstring `titel:syntax dozent:mueller` wird also als `titel:syntax AND dozent:mueller` analysiert. Dieses Verhalten ist mit der Sprachverarbeitungskomponente abgestimmt, welche ausschließlich Oder-Operatoren explizit setzt.

Bei der Instanziierung von NQuery-Objekten werden die Suchanfragen aber nicht nur geparset, sondern auch für die Suche im Index aufbereitet. Eine Aufbereitung ist notwendig, da die Suchanfragen der Sprachverarbeitungskomponente die Felder „raum“ und „tag“ enthalten, der Index stattdessen aber das Feld „termin“. Wie in Kapitel 2.3.1 dargestellt, ist das eine Folge des flachen Index von Lucene in dem Felder nicht hierarchisch verschachtelt werden können. Damit die Zuordnung von Raum- zu Tagesangaben bei mehreren Terminen einer Veranstaltung im Index nicht verloren geht, werden beide Informationen in demselben Feld erfasst. Die Aufbereitung der Suchanfragen besteht daher darin, dass die Felder „raum“ und „tag“ auf „termin“ abgebildet werden. Die Namen dieser Felder definiert NQuery als Konstanten:

```
private static final String[] FIELDS_TO_COLLAPSE =
        new String[]{"raum", "tag"};
private static final String COLLAPSE_TO = "termin";
```

Die Konstante `COLLAPSE_TO` definiert auf welches Feld im Index abgebildet werden muss, während die Konstante `FIELDS_TO_COLLAPSE` die Felder angibt, die auf `COLLAPSE_TO` abgebildet werden müssen. Ein solches Definieren von Konstanten ist für feststehende Zeichenketten üblich, da der Quellcode dadurch einerseits lesbarer und andererseits auch leichter zu erweitern ist.
Diese Abbildungen werden bereits *während* des Parsens vorgenommen. Der StandardQueryParser ermöglicht es, dass durch Implementierungen des Interfaces `org.apache.lucene.queryParser.core.processors.QueryNodeProcessor` in den Aufbau des Syntaxbaums bei der Analyse von Suchanfragen eingegriffen werden kann. NASfVI definiert zu diesem Zweck die Klasse `de.spartusch.nasfvi.server.FieldsCollapsingProcessor`, welche die QueryNodeProcessor-Schnittstelle implementiert.

```
QueryNodeProcessorPipeline processors = (QueryNodeProcessorPipeline)
    qp.getQueryNodeProcessor();
processors.add(new FieldsCollapsingProcessor(FIELDS_TO_COLLAPSE,
    COLLAPSE_TO, 50));
```

Sobald der StandardQueryParser instanziiert worden ist, ermittelt NQuery dessen QueryNodeProcessorPipeline. Dieses Objekt stellt eine Liste all der QueryNodeProcessor-Objekte dar, die der StandardQueryParser derzeit verwendet. Zu dieser Liste wird der FieldsCollapsingProcessor hinzugefügt. Der FieldsCollapsingProcessor sorgt dafür, dass der StandardQueryParser von nun an bei seiner Arbeit die `FIELDS_TO_COLLAPSE`-Felder auf das `COLLAPSE_TO`-Feld abbildet. Er speichert den Inhalt der Konstanten in seinen eigenen Attributen `fieldsToCollapse` und `collapseTo`.
Der FieldsCollapsingProcessor arbeitet auf dem Syntaxbaum der Analyse. Dort sind die einzelnen Knoten noch als Unterklassen des `org.apache.lucene.queryParser.core.nodes.QueryNode`-Interfaces repräsentiert. Die Erzeugung der `Query`-Objekte erfolgt erst nachdem der QueryNode-Baum aufgebaut ist.
Wenn beim Aufbau des Baums nach unten abgestiegen wird, wird für jeden QueryNode-Knoten die Methode `preProcessNode(QueryNode node)` aufgerufen:

```
@Override
```

```
protected final QueryNode preProcessNode(final QueryNode node) throws
    QueryNodeException {
        if (node instanceof FieldQueryNode) {
                FieldQueryNode fieldNode = (FieldQueryNode) node;

                if (StringMethods.equalsOneOf(fieldNode.
                    getFieldAsString(), fieldsToCollapse)) {
                        fieldNode.setField(collapseTo);
                        collapsedNodes.add(fieldNode);
                        nodesToRemove.add(fieldNode);
                }
        }

        return node;
}
```

In dieser Methode überprüft FieldsCollapsingProcessor zunächst, ob es sich um einen Knoten mit dem Inhalt eines Feldes - Objekte der Klasse `FieldQueryNode` - handelt. Ist das der Fall und entspricht der Name des Feldes einem der `fieldsToCollapse`-Felder, so setzt er als *neuen* Namen des Feldes den Namen des `collapseTo`-Felds. Da jedes dieser Felder letztlich in *ein* `collapseTo`-Feld überführt werden soll, speichert FieldsCollapsingProcessor Referenzen zu jedem dieser Knoten in den Attributen `collapsedNodes` und `nodesToRemove`. Die Liste der `collapsedNodes` wird später in das `collapseTo`-Feld überführt. Die Liste der `nodesToRemove`-Felder dagegen sammelt Knoten, welche in der Methode `postProcessNode(final QueryNode node)` aus dem Baum entfernt werden. Diese Methode wird beim Aufsteigen in dem Baum für jeden Knoten aufgerufen:

```
@Override
protected final QueryNode postProcessNode(final QueryNode node)
    throws QueryNodeException {
        List<QueryNode> children = node.getChildren();

        if (children != null && children.removeAll(nodesToRemove)) {
                if (children.size() == 0) {
                        nodesToRemove.add(node);
                }
        }

        return node;
}
```

Die Methode fragt zunächst ab, ob dem gegenwärtig zu verarbeitendem Knoten andere Knoten als Kinder zugeordnet sind. Ist das der Fall und entfernt der Aufruf `removeAll(nodesToRemove)` einen oder mehrere dieser Knoten, wird erneut die Anzahl der Kindknoten überprüft. Denn hatte der aktuelle Knoten zunächst Kinder, dann jedoch keine mehr, kann er selbst entfernt werden, da ihm keine Feldinformationen mehr zugeordnet sind. Nach dieser Verarbeitung wird dem Syntaxbaum das `collapseTo`-Feld hinzugefügt:

```
if (!collapsedNodes.isEmpty()) {
```

```
        TokenizedPhraseQueryNode phrase = new
            TokenizedPhraseQueryNode();
        phrase.setField(collapseTo);
        phrase.set(collapsedNodes);
        SlopQueryNode slopNode = new SlopQueryNode(phrase, slop);
        ModifierQueryNode mod = new ModifierQueryNode(slopNode,
            ModifierQueryNode.Modifier.MOD_REQ);

        if (root.getChildren().size() == 0) {
                root = mod;
        } else {
                List<QueryNode> children = root.getChildren();
                children.add(mod);
        }
}
```

Zunächst wird ein TokenizedPhraseQueryNode-Knoten erzeugt, welcher als Feldnamen `collapseTo` erhält und dem die Liste der `collapsedNodes` als Kinder zugeordnet wird. Dieser Knoten für Phrasen wird wiederum einem SlopQueryNode zugeordnet. Der SlopQueryNode versieht die Phrase mit einem Slop. NQuery verwendet im Konstruktor für den Slop einen Wert von 50. Der Slop gibt den Abstand an in dem Terme voneinander entfernt stehen können, um dennoch als Phrase erkannt zu werden [LUC10]. Je größer der Slop ist, desto flexibler kann die Wortstellung und der Inhalt einer Phrase sein. Da der Inhalt der „termin“-Felder typisch kurz ist, ermöglicht ein Slop von 50 eine nahezu freie Wortstellung. Der ModifierQueryNode schließlich legt fest, dass die Terme der untergeordneten Knoten mit der `collapseTo`-Phrase und den `collapsedNodes` zwingend gefunden werden müssen. Zum Schluß wird der neu erzeugte Knoten entweder als Wurzelelement des Baums gesetzt, falls keine anderen Knoten vorhanden sind. Oder er wird zu den übrigen Kindern der Wurzel hinzugefügt.

Interpretation des Tempus der Anfragen

Wenn die Suchanfragen keine Semesterangaben enthalten, erfolgt eine Interpretation des Tempus der Suchanfragen. In diesen Fällen ruft NQuery im Konstruktor die Methode `interpretTense(final Grammar.Tense tense, final Query query)` auf, nachdem die Suchanfragen analysiert und geparset worden sind, um jede der beiden Suchanfragen entsprechend zu modifizieren. Als `tense` wird das von der Sprachkomponente ermittelte und von `Grammar` bereitgestellte Tempus der natürlichsprachigen Anfrage übergeben:

```
private static Query interpretTense(final Grammar.Tense tense, final
    Query query) {
        Semester now = new Semester();
        Query tenseQuery;

        switch(tense) {
                case pqperf:
                        int year = new GregorianCalendar().get(
                            GregorianCalendar.YEAR) - 1;
```

```
                        tenseQuery = new TermRangeQuery("semester_end
                            ", "19700101", Integer.toString(year) + "
                            0221", true, false);
                        break;
                case perf:
                        tenseQuery = new TermRangeQuery("semester_beg
                            ", "19700101", now.getBegin(), true, false
                            );
                        break;
                case praet:
                        tenseQuery = new TermRangeQuery("semester_beg
                            ", "19700101", now.getBegin(), true, true)
                            ;
                        break;
                case praes:
                        tenseQuery = new TermQuery(new Term("semester
                            ", now.getCanonical()));
                        break;
                case fut1:
                        tenseQuery = new TermRangeQuery("semester_end
                            ", now.getEnd(), "29991231", false, true);
                        break;
                default:
                        throw new AssertionError();
        }

        BooleanQuery booleanQuery = new BooleanQuery();
        booleanQuery.add(query, BooleanClause.Occur.MUST);
        booleanQuery.add(tenseQuery, BooleanClause.Occur.MUST);

        return booleanQuery;
}
```

Die Methode ermittelt zunächst das jeweils aktuelle Semester, um alle Zeiten bis auf das Plusquamperfekt releativ zu dem aktuellen Semester interpretieren zu können. Bei der Interpretation des Tempus wird eine `tenseQuery` erzeugt. Die `tenseQuery` ist eine Query, die entweder ein bestimmtes Semester angibt oder die jeweils einen Bereich für den Semesteranfang und das Semesterende von in der Suche zu berücksichtigenden Semestern definiert. Die ursprüngliche Suchanfrage wird schließlich durch eine BooleanQuery mit der Query für das Tempus kombiniert. Bei der so entstandenen neuen Suchanfrage müssen Treffer-Dokumente sowohl die ursprüngliche Suchanfrage als auch die neu hinzugekommene zeitliche Einschränkung erfüllen.
Für einige der Zeitformen wird ein TermRangeQuery-Objekt erzeugt. Objekte dieses Typs ermöglichen das Suchen innerhalb eines lexikographisch sortierten Bereichs von Werten. Das Plusquamperfekt wird z. B. so interpretiert, dass das Semesterende von zu berücksichtigenden Veranstaltungen zwischen dem 1. Januar 1970 (inklusiv) und dem Ende des letztjährigen Wintersemesters (exklusiv) liegen muss. Die Datumsangaben für Semester werden in Kapitel 4.2.2 diskutiert. Beim Perfekt dagegen muss der Semesteran-

fang vor dem Beginn des jeweils aktuellen Semesters liegen. Der aktuelle Semesteranfang ist also exklusiv, womit das früheste berücksichtigte Semester das dem aktuellen vorangegangene ist. Das Präteritum dagegen schließt den Beginn des aktuellen Semester mit ein. Die Interpretation dieser Zeiten ist weitgehend willkürlich. Es bietet sich daher an diese Interpretationen bei weiterführenden Arbeiten zu evaluieren und anzupassen.
Bei Anfragen im Präsens wiederum werden ausschließlich Veranstaltungen des jeweils aktuellen Semesters berücksichtigt. Erfolgt die Suchanfrage dagegen im Futur 1, so werden nur Veranstaltungen berücksichtigt, die auf das Ende des aktuellen Semesters folgen und spätestens am 31. Dezember 2999 enden. Wenn NASfVI über das Jahr 2999 hinaus eingesetzt werden soll, müssen die Interpretationsregeln entsprechend angepasst werden.

Interpretation der Adverbien „wo“ und „wann“

Neben den beiden Suchanfragen speichert NQuery auch die Namen der Felder, die von der natürlichsprachigen Anfrage erfragt werden. Bei diesen Feldern handelt es sich um die Felder, für die Werte ermittelt und die in die natürlichsprachige Antwort eingesetzt werden müssen. Die Namen dieser Felder werden in dem Attribut `answerFields` erfasst. Enthält die natürlichsprachige Anfrage die interrogativen Adverbien „wo“ und „wann“, so gehören die Felder „zeit“ und „ort“ zu den angefragten Feldern, wie in Kapitel 3.4.2 erklärt. Da diese beiden Felder so im Index nicht vorhanden sind, müssen sie auf vorhandene Felder abgebildet werden. NQuery verwendet im Konstruktor die Methode `mapFieldname(final String field)` für diese Abbildung:

```
private String mapFieldname(final String field) {
        if ("zeit".equals(field)) {
                if (semesterQueried || Grammar.Tense.praes.equals(
                    tense) || answerFields.contains("semester")) {
                        return "tag";
                }
                return "semester";
        } else if ("ort".equals(field)) {
                return "raum";
        }
        return field;
}
```

Das Feld „ort“ wird stets auf „raum“ abgebildet. Die Abbildung des Feldes „zeit“ ist dagegen komplexer. Das Attribut `semesterQueried` von NQuery gibt an, ob `query` eine Semesterangabe enthält. Die Zeitform der Anfrage in `query` ist dagegen im Attribut `tense` erfasst. Wenn nun eine Semesterangabe in der Suchanfrage vorliegt, die Suchanfrage im Präsens erfolgt oder nach dem Semester gefragt wird, dann wird die interrogative Frage nach einer Zeitangabe stets als Frage nach einem Wochentag interpretiert. Das Feld „zeit“ wird dann auf das Feld „tag“ abgebildet. In allen anderen Fällen wird es auf „semester“ abgebildet.
Aufgrund dieser Abbildung wird bei den folgenden Beispielen „wann“ als Frage nach einem Wochentag interpretiert, also „zeit“ auf „tag“ abgebildet:

- „Wann fand im Sommersemester 2009 Syntax statt?“
- „Wann findet Syntax statt?“
- „Wann fand in welchem Semester “Computerlinguistik 2” statt?“

In dem folgenden Beispiel wird „wann“ dagegen als Frage nach einem Semester interpretiert, also „zeit“ auf „semester“ abgebildet:

- „Wann fand Syntax statt?“

Extrahieren von Tages- und Raumangaben

Nachdem Tages- und Raumangaben zusammen in dem Feld „termin“ erfasst sind, müssen die einzelnen Angaben bei der Beantwortung von Anfragen aus diesem Feld extrahiert werden können. Wenn das Ergebnis einer Suchanfrage vorliegt, kann im Ergebnis-Dokument die jeweils gesuchte Information mit der Hilfe von zwei regulären Ausdrücken aus dem „termin“-Feld extrahiert werden. Diese Ausdrücke definiert NQuery als Konstante:

```
private static final Pattern COLLAPSED_FIELD_TAG =
        Pattern.compile("(?:^|␣)?(mo|di|mi|do|fr|sa|so)\\b", Pattern.
            CASE_INSENSITIVE);
private static final Pattern COLLAPSED_FIELD_RAUM =
        Pattern.compile("(Rechnerraum(?:␣\\w+)?)|Raum␣(.+)|(\\w+straße.*)", Pattern.CASE_INSENSITIVE);
```

Der reguläre Ausdruck `COLLAPSED_FIELD_TAG` kann Wochentagsangaben aus dem „termin“-Feld extrahieren und der reguläre Ausdruck `COLLAPSED_FIELD_RAUM` Raumangaben. Beide Ausdrücke sind für die in dieser Arbeit verwendeten Veranstaltungsdaten optimiert. Wenn NASfVI mit anderen Veranstaltungsdaten genutzt werden soll, kann eine Anpassung dieser beiden regulären Ausdrücke notwendig werden.
Für die Extraktion der Tages- und Raumangaben definiert NQuery die statische Methode `extractValue(final String field, final String value)`:

```
public static String extractValue(final String field, final String
    value) {
        Pattern pattern = null;

        if ("tag".equals(field)) {
                pattern = COLLAPSED_FIELD_TAG;
        } else if ("raum".equals(field)) {
                pattern = COLLAPSED_FIELD_RAUM;
        } else {
                throw new AssertionError();
        }

        Matcher m = pattern.matcher(value);
        if (m.find()) {
                for (int i = 1; i <= m.groupCount(); i++) {
```

```
                        String match = m.group(i);
                        if (match != null) {
                                return match;
                        }
                }
        }

        return "(Unbekannt)";
}
```

Die Methode wendet den jeweils zu `field` passenden regulären Ausdruck an und extrahiert mit diesem aus `value` die gesuchte Information. Da der reguläre Ausdruck `COLLAPSED_FIELD_RAUM` mehrere Möglichkeiten für Treffer definiert, sucht `extractValue` nach dem ersten Match, der nicht `null` ist, und gibt diesen als gefundenen Wert zurück. Kann aus einem Feld die gesuchte Information nicht extrahiert werden, gibt die Methode den Wert „(Unbekannt)“ zurück.

4.2.5. Durchsuchen des Index

Das Durchsuchen des Index ist mit Objekten der Klasse `de.spartusch.nasfvi.server.NSearcher` möglich. Zur Instanziierung eines NSearcher-Objekts muss im Konstruktor ein Objekt der Klasse `org.apache.lucene.search.IndexSearcher` übergeben werden. Der übergebene IndexSearcher wird von NSearcher in dem Attribut `searcher` gespeichert. IndexSearcher ermöglichen es Suchanfragen auf einem geöffneten Index auszuführen. Die Methode `getSearcher()` von `XmlIndex` liefert einen auf diese Art erzeugten NSearcher zurück, der den aktuellen XmlIndex durchsuchen kann.
NSearcher verwendet NQuery-Objekte für Suchanfragen. Die Methode `search(final NQuery nquery, final int offset)` von NSearcher durchsucht den Index nach den in `nquery` gekapselten Suchanfragen. Das Ergebnis einer solchen Suchanfrage wird als Objekt der Klasse `org.apache.lucene.search.TopDocs` zurückgegeben. TopDocs enthält ein Array von `org.apache.lucene.search.ScoreDoc`-Objekten, sowie die Anzahl der gefundenen Treffer-Dokumente. Jedes `ScoreDoc`-Objekt enthält die interne Dokumentennummer eines Treffer-Dokuments. Die Dokumentennummern referenzieren ein Treffer-Dokument im Index eindeutig[10] und müssen für den Zugriff auf die im Index gespeicherten Dokumente verwendet werden. Der Parameter `offset` der Suchmethode kann benutzt werden, um die Zahl der zu berücksichtigenden Suchergebnisse schrittweise zu erhöhen. Aufgrund dieses Offsets kann das Suchergebnis durchblättert werden indem die Suchanfrage weitere Treffer zurückliefern kann.

```
public final TopDocs search(final NQuery nquery, final int offset)
    throws IOException {
        Query q = nquery.getQuery();

        if (nquery.hasSimilarityQuery()) {
```

[10] Die internen Dokumentennummern können nicht für Suchanfragen verwendet werden. Daher sind die in Kapitel 4.2.3 gezeigten ID-Felder trotz der eindeutigen Dokumentennummern notwendig.

```
                ...
        }

        return search(q, offset + 5);
}
```

In der Variabel `q` wird die allgemeine Suchanfrage der NQuery gespeichert. Sofern keine Ähnlichkeitssuche durchgeführt werden muss, wird unter Verwendung der Hilfsmethode `search(final Query query, final int maxHits)` die Suchanfrage ausgeführt. Diese private Hilfsmethode von NSearcher ruft die korrespondierende Methode bei IndexSearcher auf und loggt die Suchanfrage. Der Methodenaufruf durchsucht den Index, wobei `maxHits` die maximale Anzahl an zurückzugebende Treffer angibt. Es werden also die fünf am höchsten gerankten Ergebnisse zurückgegeben. Mit jeder Erhöhung des `offset`s wird jedoch ein zusätzliches Dokument berücksichtigt. Gibt es für eine Suchanfrage weniger als fünf Ergebnis-Dokumente, so ist die Anzahl der zurückgegebenen `ScoreDocs` entsprechend geringer oder Null, falls kein Treffer für die Suchanfrage existiert.

Die Ähnlichkeitssuche

Falls das NQuery-Objekt, mit dem die Methode `search(final NQuery nquery, final int offset)` aufgerufen wurde, eine Ähnlichkeitssuche enthält, wird dies mit der Abfrage von `nquery.hasSimilarityQuery()` festgestellt und die Ähnlichkeitssuche durchgeführt. Anders als bei der allgemeinen Suchanfrage ermittelt die Ähnlichkeitssuche immer nur exakt ein Dokument:

```
Query similQuery = nquery.getSimilarityQuery();
TopDocs similDocs = search(similQuery, 1);

if (similDocs.totalHits == 0) {
        return new TopDocs(0, new ScoreDoc[0], 0f);
}
```

Dieses Dokument dient als Vergleichsdokument zu dem die Treffer der allgemeinen Suche ähnlich sein müssen. Falls die Ähnlichkeitssuche kein Treffer findet, wird die Verarbeitung der `search`-Methode sofort abgebrochen und ein leeres Ergebnis, d. h. ein TopDocs-Objekt mit einem leeren Array von ScoreDocs, zurückgegeben. Denn wenn die Ähnlichkeitssuche nicht erfüllt werden kann, kann auch die dahinter stehende natürlichsprachige Anfrage nicht mit Daten aus dem Index beantwortet werden.
Kann dagegen ein Dokument für die Ähnlichkeitssuche gefunden werden, wird es über dessen Dokumentennummer im Index abgefragt:

```
int similDocNum = similDocs.scoreDocs[0].doc;
String similId = searcher.doc(similDocNum).get("id");
Query exclude = new TermQuery(new Term("id", similId));
```

Mit dem ID-Feld des Dokuments wird schließlich eine neue Query konstruiert, die nur nach der ID des gefundenen Dokumentes sucht. Diese `exclude`-Query wird später dazu verwendet, das Dokument der Ähnlichkeitssuche aus dem allgemeinen Suchergebnis

auszuschließen. Dieser Schritt ist notwendig, da das Dokument, das mit diesem Vergleichsdokument am ähnlichsten ist, stets das Vergleichsdokument selbst ist. Würde es nicht explizit ausgeschlossen werden, würde die spätere Suche nach ähnlichen Dokumenten stets das Vergleichsdokument selbst als ähnlichstes Dokument finden.
Die Suche nach ähnlichen Dokumenten wird mit Hilfe von Suchanfragen der Klasse `org.apache.lucene.search.similar.MoreLikeThis` implementiert. Wie in Kapitel 4.2.3 erläutert, basiert die Suche nach ähnlichen Dokumenten auf den Termvektoren des Vergleichsdokuments. NSearcher definiert mit der Konstanten `SIMILARITY_FIELDS` die Felder, die für die Suche nach ähnlichen Dokumenten verwendet werden sollen:

```
private static final String[] SIMILARITY_FIELDS =
        new String[]{"titel", "beschreibung"};
```

Die MoreLikeThis-Klasse ermittelt über die Dokumentennummer des Vergleichsdokuments die Terme, mit denen das Vergleichsdokument indiziert ist, und erzeugt aus diesen eine eigene Suchanfrage:

```
MoreLikeThis mlt = new MoreLikeThis(searcher.getIndexReader());
mlt.setFieldNames(SIMILARITY_FIELDS);
Query moreLikeQuery = mlt.like(similDocNum);
```

Diese `moreLikeQuery`-Suchanfrage findet für sich alleine genommen alle Dokumente im Suchindex, die zu dem ursprünglichen Vergleichsdokument in dem Sinne ähnlich sind, dass sie möglichst viele gleiche Terme in den Feldern „titel“ und „beschreibung“ enthalten. Im letzten Schritt wird die allgemeine Suchanfrage `q` mit `moreLikeQuery` und `exclude` in einer neuen BooleanQuery vereint. Diese neue Suchanfrage ersetzt die vorherige allgemeine Suchanfrage in `q` und wird anschließend in `search(final NQuery nquery, final int offset)` ausgeführt.

```
BooleanQuery booleanQuery = new BooleanQuery();
booleanQuery.add(q, BooleanClause.Occur.MUST);
booleanQuery.add(moreLikeQuery, BooleanClause.Occur.MUST);
booleanQuery.add(exclude, BooleanClause.Occur.MUST_NOT);
q = booleanQuery;
```

Wichtig ist, dass bei `exclude` der Wert `BooleanClause.Occur.MUST_NOT` gesetzt ist, um die ID des Vergleichsdokuments von der Suche auszuschließen.

Extraktion der erfragten Informationen

Wenn ein Suchergebnis ermittelt worden ist, müssen die einzelnen Informationen für die erfragten Felder extrahiert werden. Zu diesem Zweck bietet die NSearcher-Klasse die Methode `getAnswerValues(final NQuery nquery, final TopDocs result, int offset)` an. Die Methode erwartet sowohl die für die Suchanfrage genutzte NQuery als auch das TopDocs-Suchergebnis und den Offset der Suche als Argumente. Der Rückgabewert ist vom Typ `Map<String, Set<String»` und stellt eine Abbildung von Feldernamen auf Werte der Felder dar. Das entspricht dem Format für die einzusetzenden Werte, die die `generate(final String input, final Map<String, Set<String» values)`-Methoden des GrammarManagers und der Grammar-Klasse erwarten.

Die Methode legt zunächst eine leere Map an und ermittelt die zu beantwortenden Felder der Suchanfrage:

```
Map<String, Set<String>> values = new HashMap<String, Set<String>>();
Set<String> answerFields = nquery.getFieldsToAnswer();

if (result.totalHits == 0 || answerFields.size() == 0) {
        return values;
}

if (offset >= result.scoreDocs.length) {
        offset = result.scoreDocs.length - 1;
}
```

Falls das Suchergebnis leer ist oder falls keine Felder zu beantworten sind, wird die leere Map zurückgegeben. `getAnswerValues` führt bei dem Offset eine Überprüfung durch, um sicherzustellen, dass dessen Wert nicht größer als der maximale Index im Suchergebnis ist.

```
String firstField = answerFields.iterator().next();

if (answerFields.size() == 1 && !NQuery.isFieldToCollapse(firstField)
    ) {
        // Aggregate all ScoreDocs (1 field, n documents)
        ...
} else {
        // Process first ScoreDoc only (n fields, 1 document)
        ...
}

return values;
```

Falls die Information von nur einem Feld erfragt wird und es sich bei diesem nicht um eines der auf „termin“ abgebildeten Felder handelt, werden die Treffer-Dokumente des Ergebnisses zusammengefasst. Aus diesem Grund umfasst die natürlichsprachige Antwort auf Anfragen mit nur einem zu beantwortenden Feld mehrere Suchergebnisse in einer koordinierten Phrase. Der Benutzer soll dadurch einen besseren Überblick über das Suchergebnis erhalten. Sollte die von ihm gesuchte Information nicht unter den ersten fünf ermittelten Treffern sein, kann er die Zahl der zu berücksichtigenden Dokumente schrittweise durch den Offset erhöhen.
Werden jedoch mehrere Felder angefragt, so würden aufgrund der Implementation die Felder und nicht die Treffer-Dokumente in koordinierten Phrasen zusammengefasst werden. Die Unterscheidung einzelner Treffer-Dokumente und deren Werte voneinander wäre dann nicht mehr möglich. Das gleiche Problem tritt bei Terminangaben auf. Aus diesem Grund findet eine Aggregation der Suchergebnisse nur bei einzelnen Feldern statt, die nicht auf „termin“ abgebildet werden. In den Fällen ohne Aggregation wird lediglich das erste Treffer-Dokument an der durch den Offset bezeichneten Stelle für die Antwort verwendet:

```
// Process first ScoreDoc only (n fields, 1 document)
```

```
Document doc = searcher.doc(result.scoreDocs[offset].doc);
for (String field : answerFields) {
        values.put(field, extractValues(nquery, doc, field));
}
```

Die Hilfsmethode `extractValues(final NQuery nquery, final Document doc, final String field)` extrahiert die Werte des jeweiligen Felds aus dem übergebenen Dokument. Enthält eine Veranstaltung mehrere gleichartige Daten, zum Beispiel mehrere Angaben für „termin“ oder „dozent“, so ist zunächst nicht klar, welche der Angaben zum Finden des Dokuments geführt hat. Daher nutzt die Methode die Klasse `org.apache.lucene.search.highlight.Highlighter`, um die im Dokument gematchten Angaben zu finden. Highlighter ist eigentlich für die Hervorhebung von gematchten Schlüsselwörtern in Feldern gedacht. Mit dessen Methode `getBestFragment` kann jedoch auch festgestellt werden, welche Daten Treffer beinhalten. Aus den jeweils gematchten Feldern werden schließlich die erfragten Informationen mit der in NQuery definierten Methode `extractValue(final String field, final String value)` extrahiert.
Bei der Aggregation mehrerer Treffer-Dokumente wird die Hilfsmethode `extractValues` wiederverwendet. Für die Aggregation iteriert eine for-Schleife über alle Treffer-Dokumente des Suchergebnisses:

```
// Aggregate all ScoreDocs (1 field, n documents)
Set<String> hs = new HashSet<String>(result.scoreDocs.length);
for (ScoreDoc sd : result.scoreDocs) {
        Document doc = searcher.doc(sd.doc);
        hs.addAll(extractValues(nquery, doc, firstField));
}
values.put(firstField, hs);
```

Jedes Treffer-Dokument wird aus dem Suchindex abgefragt und dessen erfragtes Feld extrahiert. Sowohl bei der Aggregation von Suchergebnissen als auch bei der Zurückgabe mehrerer Felder eines Treffer-Dokuments ist der Rückgabetyp stets `Map<String, Set<String»`, so dass in den nachfolgenden Verarbeitungsschritten für die Generierung der natürlichsprachigen Antwort nicht unterschieden werden muss, ob eine Aggregation stattgefunden hat oder nicht.

4.3. Schnittstellen des Servers nach Außen

Die Schnittstellen des Servers nach Außen werden mit Servlets bereitgestellt. Der Server selbst wird daher von einem Servlet-Container als Web-Application ausgeführt. Servlets und Web-Applications sind Technologien im Java-Umfeld und im Rahmen des Java Community Process standardisiert (siehe [JST]). Die allgemeine Funktionsweise dieser Technologien wird in Kapitel 2.3.2 diskutiert.

4.3.1. Initialisierung

Beim Starten des Servlet-Containers initialisiert der Servlet-Container die Web-Application. Bei der Initialisierung legt der Server von NASfVI einen neuen Suchindex an und bereitet

den Zugriff auf die Sprachverarbeitungskomponente vor. Dies geschieht durch die Klasse `de.spartusch.nasfvi.server.Init`:

```
public final class Init implements ServletContextListener {
        ...
        private static String luceneIndex = "nasfvi.lucene.index";
        private static String grammarManager = "nasfvi.grammar.
            manager";

        @Override
        public void contextInitialized(final ServletContextEvent
            event) {
                ServletContext context = event.getServletContext();
                Resources res = new Resources(context);
                Analyzer analyzer = new NAnalyzer(res);
                context.setAttribute(grammarManager, new
                    GrammarManager(res, analyzer));
                ...
                XmlIndex index = new XmlIndex(analyzer);
                InputStream xmlToAdd = res.getAsStream("nasfvi.
                    IndexFile", "/WEB-INF/index.xml");
                if (xmlToAdd == null) {
                        throw new IllegalArgumentException("index.xml
                            not found");
                } else {
                        index.ingest(xmlToAdd);
                }
                context.setAttribute(luceneIndex, index);
                ...
        }
        ...
}
```

Die Init-Klasse implementiert `javax.servlet.ServletContextListener`. Ihre Methode `contextInitialized(ServletContextEvent)` wird automatisch von dem Servlet-Container aufgerufen, sobald der ServletContext des Servers initialisiert worden ist. Der ServletContext bleibt während der gesamten Lebenszeit der Anwendung bestehen und eignet sich daher für das „Aufbewahren“ von Objekten, die nur einmal erzeugt und initialisiert werden sollen, um dann allen Servlets der Web-Application und damit allen Aufrufen des Servers zur Verfügung zu stehen.
Die Methode `contextInitialized(ServletContextEvent)` erzeugt zunächst ein Objekt der Klasse `de.spartusch.Resources`, welches den Zugriff auf statische Ressourcen des Servers ermöglicht. Mit Hilfe dieser Klasse kann auf die Datei `index.xml` zugegriffen werden, welche die Vorlesungsdaten in dem in Kapitel 2.3.1 beschriebenen XML-Format enthält. Die Klasse ermöglicht zudem den Zugriff auf den Prolog-Quellcode der Sprachverarbeitungskomponente. Der in Kapitel 4.1 beschriebene `de.spartusch.nasfvi.server.GrammarManager` für den Zugriff auf die Sprachverarbeitungskomponente wird in dem ServletContext als Attribut mit dem Namen „nasfvi.grammar.manager“ gespeichert. Unter

diesem Namen kann das Objekt im ServletContext wieder abgefragt werden. Die statische Methode `Init.getGrammarManager(ServletContext)` stellt auf diese Weise den ein Mal initialisierten GrammarManager zur Verfügung.
Das zweite Attribut, das im ServletContext des Servers gespeichert wird, ist der Suchindex der Volltextsuche. Die Methode `contextInitialized(ServletContextEvent)` erzeugt dazu ein Objekt der Klasse `de.spartusch.nasfvi.server.XmlIndex`, welche in Kapitel 4.2.3 vorgestellt wird. Objekte dieser Klasse kapseln den Suchindex und ermöglichen einen Zugriff auf diesen. Mit dem Aufruf `index.ingest(xmlToAdd)` parset das Objekt den Inhalt der Datei `index.xml` und fügt ihn dem tatsächlichen Index der Volltextsuche hinzu. Das XmlIndex-Objekt wird im ServletContext unter dem Namen „nasfvi.lucene.index“ gespeichert und kann mit diesem abgefragt werden. Mit der statischen Methode `getSearcher(ServletContext)` ermöglicht die Init-Klasse das Durchsuchen des Index:

```
public static NSearcher getSearcher(final ServletContext context) {
        XmlIndex index = (XmlIndex) context.getAttribute(luceneIndex)
            ;
        if (index == null) {
                throw new AssertionError(luceneIndex + "␣not␣set");
        }
        return index.getSearcher();
}
```

4.3.2. Servlet zur Bereitstellung von Satzvervollständigungen

Die Klasse `de.spartusch.nasfvi.server.Suggestlet` erweitert die Klasse `javax.servlet.http.HttpServlet` und implementiert ein Servlet als Web-Schnittstelle zur `suggest`-Methode des `GrammarManager`s. Zur Verwendung des Servlets muss es mit einer HTTP GET-Anfrage[11] aufgerufen werden und in dem Parameter `q` einen zu vervollständigenden Satz erhalten:

```
@Override
protected final void doGet(final HttpServletRequest req, final
    HttpServletResponse res) throws ServletException, IOException {
        String q = req.getParameter("q");
        if (q != null) {
                res.setCharacterEncoding("UTF-8");
                res.setContentType("application/x-suggestions+json");
                PrintWriter out = res.getWriter();
                GrammarManager manager = Init.getGrammarManager(
                    getServletContext());
                out.print("[");
                out.print(Grammar.toJsonString(q, false));
                out.print(",␣[");
                if (q.indexOf('␣') != -1) { // Minimum two tokens
```

[11] Bei HTTP GET-Anfragen werden die Parameter direkt mit der aufgerufenen URL übergeben. Siehe RFC 2616 zur Spezifikation von HTTP unter `http://tools.ietf.org/html/rfc2616`

```
                        Set<String> suggestions = manager.suggest(q);
                        Iterator<String> i = suggestions.iterator();
                        while (i.hasNext()) {
                                String s = i.next();
                                out.print(Grammar.toJsonString(s,
                                    true));
                                if (i.hasNext()) {
                                        out.print(",␣");
                                }
                        }
                }
                out.print("]]");
        }
}
```

Das Servlet antwortet in dem in Kapitel 2.3.3 gezeigten JSON-Format der OpenSearch-Spezifikation für Suggestions [OPS11]. Es wiederholt in einem JSON-Array im ersten Element den Satz, mit dem es aufgerufen wurde, und gibt im zweiten Element ein Array mit allen möglichen Vervollständigungen dieses Satzes zurück. Die Berechnung der möglichen Satzvervollständigungen erfolgt aufgrund von Performance-Überlegungen nur, wenn der zu vervollständigende Satz mindestens ein Leerzeichen enthält und damit mindestens über zwei Token verfügt. Durch diese Einschränkung sollen Berechnungen vermieden werden, die sehr zeitintensiv sind. Die Idee ist dabei ähnlich der in Kapitel 3.2.9 beschriebenen Einschränkung bei der Generierung der Satzvorschläge: Da es für kurze Satzanfänge sehr viele mögliche Ergänzungen gibt, sind sie sehr zeitintensiv bei der Berechnung und daher nach Möglichkeit zu vermeiden. Für die Berechnung der Satzvorschläge verwendet das Servlet den von der Init-Klasse bereitgestellten GrammarManager. Die Hilfsmethode `toJsonString` maskiert im Wesentlichen Anführungszeichen in den Strings, so dass die Zeichenketten als Strings im JSON-Format verwendet werden können.

4.3.3. Servlet zur Beantwortung natürlichsprachiger Anfragen

Die Schnittstelle für die natürlichsprachige Beantwortung von natürlichsprachigen Anfragen wird durch das Servlet `de.spartusch.nasfvi.server.Parselet` bereitgestellt. Wie das Servlet für die Satzvervollständigungen erweitert auch diese Klasse `javax.servlet.http.HttpServlet` und verwendet HTTP GET-Anfragen. Wird das Servlet mit einer natürlichsprachigen Anfrage im Parameter `q` aufgerufen, analysiert es die Anfrage zunächst mit der Methode `GrammarManager.parse` (vgl. Kapitel 4.1.1), durchsucht die Daten der Veranstaltungen des Vorlesungsverzeichnisses mit `NSearcher.search` (Kapitel 4.2.5) und erzeugt schließlich eine natürlichsprachige Antwort mit `GrammarManager.generate` (vgl. Kapitel 4.1.1):

```
@Override
protected final void doGet(final HttpServletRequest req, final
    HttpServletResponse res) throws ServletException, IOException {
        String q = req.getParameter("q");
        int offset = 0;
```

```
        try {
                String offsetStr = req.getParameter("offset");
                if (offsetStr != null) {
                        offset = Integer.parseInt(offsetStr);
                }
        } catch (NumberFormatException e) {
                res.sendError(HttpServletResponse.SC_BAD_REQUEST);
                return;
        }

        if (q != null) {
                GrammarManager manager = Init.getGrammarManager(
                    getServletContext());
                NSearcher searcher = Init.getSearcher(
                    getServletContext());
                NQuery nquery;

                try {
                        nquery = manager.parse(q);
                        if (nquery == null) {
                                res.sendError(HttpServletResponse.
                                    SC_BAD_REQUEST);
                                return;
                        }
                } catch (QueryNodeException e) {
                        res.sendError(HttpServletResponse.
                            SC_INTERNAL_SERVER_ERROR);
                        return;
                }

                TopDocs result = searcher.search(nquery, offset);

                res.setCharacterEncoding("UTF-8");
                res.setContentType("application/json");
                res.setStatus(HttpServletResponse.SC_OK);
                PrintWriter out = res.getWriter();

                Map<String, Set<String>> vals = searcher.
                    getAnswerValues(nquery, result, offset);
                String response = manager.generate(q, vals);

                out.print("[\n");
                out.print(searcher.toJson(nquery, result, offset));
                out.print(",\n");
                out.print(response);
                out.print("]");
        }
}
```

Da die Volltextsuche mehrere Suchergebnisse liefern kann, erwarten die Suchfunktionen für den Umgang mit diesen die Angabe eines Offset-Wertes. Dieser kann als natürliche Zahl in dem Parameter `offset` an das Servlet übermittelt werden. Wird kein Offset-Parameter angegeben, verwendet Parselet den Standardwert 0 als Offset.
Den obligatorischen Parameter `q` analysiert es dagegen mit Hilfe der Methode `parse(final String input)` des GrammarManagers und erzeugt damit eine NQuery. Die NQuery wiederum, die die in der natürlichsprachigen Anfrage geäußerten Suchanfragen enthält, erlaubt es den Suchindex mit der Methode `search(final NQuery nquery, final int offset)` des NSearchers zu durchsuchen. Die Variabel `result` verweist schließlich auf die Treffer-Dokumente der Suchanfragen. Aus diesen Treffer-Dokumenten extrahiert der Aufruf von `getAnswerValues(final NQuery nquery, final TopDocs result, int offset)` (Kapitel 4.2.5) die angefragten Informationen und `generate(final String input, final Map<String, Set<String» values)` erzeugt schließlich die natürlichsprachige Antwort auf die natürlichsprachige Anfrage.
Das Servlet kann die Bearbeitung von Anfragen aber auch mit zwei verschiedenen Fehlermeldungen abbrechen. Die Fehlermeldung `SC_BAD_REQUEST` wird zurückgegeben, wenn der `offset`-Parameter beim Aufruf zwar angegeben wurde, aber keine natürliche Zahl ist. Die gleiche Fehlermeldung wird auch zurückgegeben, wenn eine Anfrage von der Sprachverarbeitungskomponente nicht analysiert werden konnte und daher keine NQuery erzeugt werden konnte. Konnte die Sprachverarbeitungskomponente dagegen die Anfrage zwar analysieren, doch das Parsen der Suchanfragen während der Instanziierung der NQuery schlug fehl, bricht das Servlet mit der Fehlermeldung `SC_INTERNAL_SERVER_ERROR` ab. Ein Auftreten dieser Fehlermeldung würde auf einen internen Fehler verweisen. Denn die Meldung bedeutet, dass die Sprachverarbeitungskomponente Suchanfragen erzeugt hat, die Lucene nicht interpretieren konnte. Diese Fehlermeldung ist ein theoretisches Szenario und sollte beim Einsatz von NASfVI nie auftreten.

Format der Antworten

Auch die Parselet-Klasse nutzt JSON als Format für die Antworten. Die Rückgabe ist dabei stets ein Array mit zwei Elementen. Jedes der Elemente ist ein Objekt in JSON-Notation. Während Arrays in JSON mit eckigen Klammern notiert werden, werden Objekte mit geschweiften Klammern notiert. Objekte können in JSON Attribute besitzen, welche durch Kommata voneinander getrennt sind und in Form von Schlüssel-Wert-Paaren angegeben werden.[12]
Das erste Element im Array des Antwortformats beinhaltet Informationen über die ausgeführten Suchanfragen und wird von der NSearcher-Klasse erzeugt. Das Objekt beinhaltet das Attribut `Offset` mit dem jeweils bei der Anfrage übermittelten Offset-Wert und das Attribut `Hits` mit der Anzahl der Treffer-Dokumente für die Anfrage. Das Attribut `NQuery` wiederum gibt Informationen über die für die Anfrage genutzte NQuery an und wird von der NQuery-Klasse erzeugt. Der Wert des Attributs ist selbst ein Objekt und enthält die Attribute `Query` für die berechnete allgemeine Suchanfrage, `SQuery` für die

[12]Eine Grammatik und Beschreibung der JSON-Syntax ist unter `http://www.json.org/` verfügbar.

berechnete Ähnlichkeitssuche und `Fields` für ein Array mit den in der Anfrage erfragten Feldern.
Das zweite Element des Arrays wird von der `generate`-Methode der Klasse Grammar erzeugt und enthält sowohl die linguistische Analysen als auch die natürlichsprachige Antwort als Attribute. Das Attribut `AnalysisReq` stellt ein Array mit der linguistischen Analyse der natürlichsprachigen Anfrage durch die Sprachverarbeitungskomponente bereit. Dieses Array enthält die Markiertheit des Satzes, ein Array mit der Grundform des Verbs, dem Genus Verbi, der Verbstellung, der Person, dem Numerus und dem Tempus, sowie den vollständigen Syntaxbaum und die Semantik der Anfrage. Das Attribut `AnalysisAns` ist völlig analog zu `AnalysisReq` aufgebaut, enthält jedoch die linguistische Analyse der berechneten natürlichsprachigen Antwort. Das letzte Attribut schließlich lautet `Answer` und enthält die generierte, natürlichsprachige Antwort auf die Anfrage.
Dieses Antwortformat des Servlets soll mit den beiden folgenden Beispielen für Antworten des Servlets auf natürlichsprachige Anfragen verdeutlicht werden.

Beispiel: „Wer hat im Sommersemester 2009 wo Semantik gehalten?“

Das folgende Beispiel zeigt die Antwort[13] des Parselet-Servlets auf eine HTTP GET-Anfrage mit dem Satz „Wer hat im Sommersemester 2009 wo Semantik gehalten?“.


```
[{
        "NQuery": {
                "Query": "+titel:semant +semester:2009",
                "SQuery": "",
                "Fields": ["dozent", "raum"]
        },
        "Offset": 0,
        "Hits": 1
},
{
        "AnalysisReq": ["14", ["halten", "aktiv", "v2", "3", "sg", "
            perf"], [["np", ["3", "sg", "nom"], "hum", "qu", "[]-[]",
             "lam(_371, qu(_372, dozent(_372, '') und _371 * _372))",
            ["pro>qu", ["sg", "nom"], "hum", "-", "[]-[]", "lam(_371,
             qu(_372, dozent(_372, '') und _371 * _372))", "wer", "wer
            "]]]], [["pp", ["dat"], "semester", "def", "lam(_373, lam(
            _374, ex(_375, _373 * _375 und _374 * _375))) * lam(_376,
            lam(_377, semester(_377, _376))) * '2009'", ["p>def",
            "[]", "semester", "dat", "lam(_373, lam(_374, ex(_375,
            _373 * _375 und _374 * _375)))", "im", "im"], ["n>n", ["sg
            ", "dat"], "semester>sose", "neut", "_378-_378", "lam(_376
            , lam(_377, semester(_377, _376)))", "'Sommersemester'",
            "'Sommersemester'"], ["angabe", "semester>sose",
            "'2009'"]]], ["advp", ["dat"], "loc", "qu", "lam(_379, qu(
```


[13] Die hier gezeigten Antworten enthalten keine HTTP-Header und sind für eine bessere Lesbarkeit nachträglich umformatiert und gekürzt worden. Im hier gezeigten Beispiel wurde z. B. der Wert des Attributs `AnalysisAns` aus Platzgründen ausgelassen.

```
            _380, ort(_380) und _379 * _380))", [["adv>qu", "[]", "loc
            ", "qu", "lam(_379, qu(_380, ort(_380) und _379 * _380))",
             "wo", "wo"]]], ["np", ["3", "sg", "akk"], "event", "def",
             "[?[pp, [akk], thema, _381, _382, _383]]-[]", "lam(_384,
            ex(_385, lam(_386, lam(_387, veranstaltung(_387, _386) und
             _382 * _387)) * 'semantik' * _385 und _384 * _385))", [["
            blackbox", "event", "'semantik'"]]]], "[]"], ["qu(_372,
            dozent(_372, '') und qu(_380, ort(_380) und ex(_385,
            veranstaltung(_385, 'semantik') und halten(_372, _385,
            _380, '_'))))", "ex(_375, semester(_375, '2009'))"]],
        "AnalysisAns": ...,
        "Answer": "Im Sommersemester 2009 hat \"J. Schuster\"
            Semantik im Raum 1.05 und im Raum 1.14 gehalten "
}]
```


Im `Query`-Attribut der `NQuery` ist die von Lucene genutzte allgemeine Suchanfrage wiedergegeben. Diese entspricht nicht direkt der von der Sprachverarbeitungskomponente berechneten Suchanfrage. Stattdessen spiegelt sie auch den Effekt des genutzten NAnalyzers wider. So wurde in der Beispielanfrage „Semantik“ zum Stamm „semant“ gestemmt und normalisiert.

Beispiel: „Welche Seminare haben im Sommersemester 2009 dem Proseminar Syntax geähnelt?“

Das zweite Beispiel zeigt die Antwort des Servlets auf eine HTTP GET-Anfrage mit der Frage „Welche Seminare haben im Sommersemester 2009 dem Proseminar Syntax geähnelt?“[14]


```
[{
        "NQuery": {
                "Query": "+(typ:seminar) +semester:2009",
                "SQuery": "+titel:syntax +(typ:proseminar typ:seminar
                    ) +semester:2009",
                "Fields": ["titel"]
        },
        "Offset": 0,
        "Hits": 6
},
{
        "AnalysisReq": ...,
        "AnalysisAns": ...,
        "Answer": "Im Sommersemester 2009 haben dem Proseminar Syntax
            \"Eine Dependenzsyntax des Deutschen für die maschinelle
           Übersetzung mit ETAP-3\" und Grammatikinduktion und \"
           Phraseologismen in der HPSG\" und \"Mathematische
           Grundlagen der Computerlinguistik II\" und \"Lokale
           Grammatiken\" geähnelt "
```


[14] In der Antwort dieses Beispiels wurden die Werte für die Attribute `AnalysisReq` und `AnalysisAns` ausgelassen.

}]

Beachtenswert ist in diesem Beispiel die Ähnlichkeitssuche. Wie in Kapitel 4.2.5 gezeigt, ermittelt die Ähnlichkeitssuche ein Vergleichsdokument, zu dem Treffer-Dokumente der allgemeinen Suchanfrage ähnlich sein müssen. Bei der Ähnlichkeitssuche ist im Beispiel der Effekt der Kompositazerlegung des genutzten NAnalyzers zu sehen. In dieser Suchanfrage wurde für das Token „Proseminar“ das zusätzliche Token „seminar“ als Kopf ermittelt und als Alternative zum vollständigen Kompositum hinzugefügt.

5. Der Client

Der Client ist eine graphische Oberfläche, die von NASfVI für den Zugriff auf den Server mit einem Browser bereitgestellt wird. Diese graphische Oberfläche ermöglicht es Benutzern Anfragen bequem einzugeben und an den Server zu schicken. Während der Eingabe von Anfragen blendet der Client zur Unterstützung des Benutzers Satzvorschläge ein. Hat der Benutzer eine Anfrage an den Server geschickt, zeigt die graphische Oberfläche außerdem die natürlichsprachige Antwort, sowie die Suchdetails und die linguistischen Analysen der Eingabe und der generierten Antwort. Diese Funktionen stellt der Client bereit, indem er die in den vorangegangenen Kapiteln 4.3.2 und 4.3.3 gezeigten Schnittstellen des Servers im Hintergrund abfragt. So wie der Server ist auch der Client in Java geschrieben. Zentral ist dabei die Verwendung des Google Web Toolkits, welches den Java-Code für die Verwendung in Browsern nach JavaScript kompiliert. Denn technisch gesehen handelt es sich bei dem Client um eine Webseite, deren Elemente dynamisch mit JavaScript aufgebaut und angezeigt werden. Der Client wird allgemein in Kapitel 2.4 vorgestellt. Der Quellcode des Clients befindet sich in dem Java-Paket `de.spartusch.nasfvi.client`.

5.1. Das Google Web Toolkit

Mit dem Google Web Toolkit entwickelte Anwendungen können als grundlegende Typen Widgets, JavaScript-Objekte und einfache Java-Objekte verwenden. Die meisten graphischen Elemente werden als Widgets aufgebaut. Bei Widgets handelt es sich um Klassen, die von `com.google.gwt.user.client.ui.Widget` erben. Sie können im Browser dargestellt werden oder auf Browser-Events wie Mausklicks reagieren. JavaScript-Objekte werden dagegen im Java-Code von der Klasse `com.google.gwt.core.client.JavaScriptObject` ermöglicht. JavaScript-Objekte werden zur Laufzeit aus Zeichenketten, die Objekte im JSON-Format beschreiben, erzeugt und vereinfachen die Verarbeitung komplexer JSON-Formate. Anderer Java-Code und einfache Java-Objekte wiederum werden vom Google Web Toolkit direkt in JavaScript-Code übersetzt.

5.1.1. Trennung von Layout und Programmlogik

Widgets können mit dem Google Web Toolkit so geschrieben werden, dass eine Trennung zwischen Layout und Programmlogik möglich ist. Die Widget-Klassen enthalten dann nur ihre jeweilige Programmlogik, während das Layout - das heißt, statische HTML-Elemente, sowie andere enthaltene Widgets - in XML-Dateien deklarativ angegeben ist. Die XML-Dateien mit dem Layout eines Widgets müssen denselben Namen wie das Widget und die Dateiendung `.ui.xml` besitzen. Beim Übsetzen des Java-Codes zu JavaScript

berücksichtigt der Compiler des Google Web Toolkits diese deklarativen Angaben zum Layout und konstruiert das Widget entsprechend.
Grundsätzlich kann das Layout auch mit dem Google Web Toolkit prozedural im Code der Widgets aufgebaut werden, indem entsprechende Objekte und HTML-Elemente durch Methodenaufrufe erzeugt werden. Die Trennung des Layouts von der Programmlogik in den Widget-Klassen ermöglicht jedoch die deklarative Beschreibung des Layouts selbst und ist dadurch übersichtlicher und leichter anzupassen. Aufgrund dieser Vorteile geben die Widgets in NASfVI ihr jeweiliges Layout deklarativ an.
Diese Trennung von Layout und Programmlogik sei am Beispiel der Klasse `NResponse Widget` gezeigt. Das deklarative Layout des Widgets befindet sich in der Datei `NResponse Widget.ui.xml` im gleichen Paket wie die Klasse selbst. Das Wurzel-Element der XML-Datei ist `UiBinder` und bezeichnet die für die Verwendung deklarativer Layouts notwendige Komponente des Google Web Toolkits. Mit dem Element `style` können CSS-Auszeichnungen[1] für die verschiedenen Elemente des Layouts definiert werden. Diese beiden XML-Knoten sind an den XML-Namensraum `urn:ui:com.google.gwt.uibinder` gebunden. Werden Widgets im Layout verwendet, so sind diese an den XML-Namensraum `urn:import:com.google.gwt.user.client.ui` gebunden. Eingebettete HTML-Elemente verwenden dagegen keinen besonderen Namensraum. Das nachfolgende Beispiel zeigt die gekürzte Angabe des Layouts für das `NResponseWidget`, welches das Antwortformat der in Kapitel 4.3.3 beschriebenen Schnittstelle darstellt:

```
<ui:UiBinder xmlns:ui='urn:ui:com.google.gwt.uibinder' xmlns:g='
    urn:import:com.google.gwt.user.client.ui'>
<ui:style>...</ui:style>
<g:VerticalPanel width="100%">
        <g:Label stylePrimaryName="{style.answerLabel}">
                Antwort:
        </g:Label>
        <g:HTMLPanel stylePrimaryName="{style.answerBox}">
                <g:InlineLabel ui:field="answer"/>
                <g:Anchor ui:field="link" stylePrimaryName="{style.
                    link}">
                        mehr
                </g:Anchor>
        </g:HTMLPanel>
        <g:DisclosurePanel animationEnabled="true" open="false">
                <g:header>Details und Analysen</g:header>
                <g:HTMLPanel>
                <table class="{style.text-left}" cellspacing="4">
                <tr>
                <td>Suchanfrage:</td>
                <td><g:InlineLabel ui:field="query"/></td>
                </tr><tr>
                <td>Ähnlichkeit:</td>
                <td><g:InlineLabel ui:field="similQuery"/></td>
```

[1] Cascading Style Sheets ist ein Standard zur Angabe von graphischen Auszeichnungen in Web-Dokumenten. Die aktuelle Spezifikation von CSS ist unter `http://www.w3.org/TR/CSS2/` einsehbar.

```
                    </tr><tr>
                    <td>Gesuchte Felder:</td>
                    <td><g:InlineLabel ui:field="fields"/></td>
                    </tr><tr>
                    <td>Hits:</td>
                    <td><g:InlineLabel ui:field="hits"/> Treffer</td>
                    </tr><tr>
                    <td colspan="2"><g:Tree ui:field="analysisReq"/></td>
                    </tr><tr>
                    <td colspan="2"><g:Tree ui:field="analysisAns"/></td>
                    </tr>
                    </table>
            </g:HTMLPanel>
            </g:DisclosurePanel>
</g:VerticalPanel>
</ui:UiBinder>
```

Zu sehen ist, dass mehrere Widgets im Layout verwendet werden. `VerticalPanel` ordnet zum Beispiel untergeordnete Widgets vertikal an, während `Label` einen Textblock ermöglicht und `DisclosurePanel` ein aufklappbares Element erzeugt, das nur aufgeklappt untergeordnete Elemente zeigt. Die eigentlichen Inhalte werden jedoch von Widgets der Typen `InlineLabel` und `Tree` dargestellt. `InlineLabel`s enthalten einfachen Fließtext. `Tree`s dagegen aufklappbare Bäume. Die Elemente des Layouts, die in der Programmlogik von `NResponseWidget` verwendet werden müssen, haben durch das Attribut `field` einen Namen zugewiesen bekommen. Mit Hilfe dieser Namen kann im Java-Code der `NResponseWidget`-Klasse auf die Elemente des Layouts zugegriffen werden. Die Klasse definiert dazu Attribute mit denselben Namen und markiert sie mit der `@UiField`-Annotation als Widgets, die in dem Layout der Klasse vorkommen:

```
@UiField InlineLabel query;
@UiField InlineLabel similQuery;
@UiField InlineLabel hits;
@UiField InlineLabel fields;
@UiField InlineLabel answer;
@UiField Anchor link;
@UiField Tree analysisReq;
@UiField Tree analysisAns;
```

Die so definierten Attribute können innerhalb von Java als normale Objekte angesprochen werden. `NResponseWidget` prüft zum Beispiel, ob eine Suchanfrage eine Ähnlichkeitssuche enthielt und setzt in dem `InlineLabel` namens `similQuery` entsprechend entweder “keine” oder die Ähnlichkeitssuche:

```
if (response.getSimilarityQuery().isEmpty()) {
        similQuery.setText("keine");
} else {
        similQuery.setText(response.getSimilarityQuery());
}
```

Als sichtbares Ergebnis erscheint der jeweilige Text an der im Layout durch `similQuery`

definierten Stelle.
Neben dem Zugriff auf die Widgets ist es auch möglich, auf Ereignisse der Widgets zu reagieren. Mit der Annotation `@UiHandler` können Methoden markiert werden, die im Fall von bestimmten Ereignissen aufgerufen werden sollen.

```
@UiHandler("link")
protected final void nextResult(final ClickEvent event) {
        stateSetter.setHistoryState(null, nextOffset);
}
```

`NResponseWidget` definiert die Methode `nextResult(final ClickEvent event)` und bindet sie an den im Layout definierten `Anchor` namens `link`. Dass die Methode ein Argument vom Typ `ClickEvent` erwartet, ist von zentraler Bedeutung. Denn beim Kompilieren des Codes erkennt das Google Web Toolkit anhand des Argument-Typs der Methode, dass die Methode bei Klicks auf das Element ausgeführt werden soll und nicht etwa, wenn es den Fokus erhält, Tastatureingaben erfolgen oder dergleichen. Klickt ein Benutzer nun auf das entsprechende Element, wird die Methode aufgerufen, welche den Offset-Wert der Anfrage erhöht, wodurch zusätzliche Ergebnisse angezeigt werden. Die Bedeutung des Offsets für Suchanfragen wird in Kapitel 4.2.5 diskutiert.

5.2. Klassen und Interfaces des Clients

Im Folgenden werden die Klassen und Interfaces des Clients kurz vorgestellt. Der Hauptteil der Verarbeitung von Suchanfragen findet nicht im Client, sondern in dem in Kapitel 4 besprochenen Server statt.

5.2.1. Die Hauptoberfläche

Der Einstiegspunkt des Clients wird durch das Widget `de.spartusch.nasfvi.client.Main` bereitgestellt. `Main` implementiert das Interface `com.google.gwt.core.client.EntryPoint` und dessen Methode `onModuleLoad()`. Wird der Client in einem Browser aufgerufen, so wird diese Methode ausgeführt. Sie initialisiert den Client, instanziiert das `MainWidget` und zeigt es auch an. Bei dem `MainWidget` handelt es sich um ein Objekt der Klasse `de.spartusch.nasfvi.client.MainWidget`. Das `MainWidget` ist ein zusätzliches Widget, welches zudem ein Composite ist. Composites definieren graphische Elemente, die andere Widgets aufnehmen können und sind von der Widget-Unterklasse `com.google.gwt.user.client.ui.Composite` abgeleitet. Der Großteil der graphischen Oberfläche des Clients wird durch das `MainWidget` erzeugt. Es zeigt die vorgefertigten Beispiele für Suchanfragen an und es schickt eingegebene Anfragen an den Server. Es überprüft auch mit einem regulären Ausdruck, ob in einer Eingabe runde Klammern vorkommen. Ist dies der Fall, dann zeigt es eine Fehlermeldung an, die den Benutzer auffordert, die Platzhalter auszufüllen. Denn wie in Kapitel 3.2 gezeigt, erzeugt die Sprachverarbeitungskomponente im Suggest-Modus in runde Klammern gefasste Platzhalter für die Werte von Blackboxen. Zum Beispiel `(Nachname)` als Platzhalter für Eigennamen. Wählt ein Benutzer aus angezeigten Vorschlägen für Satzvervollständigungen einen

Vorschlag aus, kann es daher sein, dass Platzhalter vorkommen. Diese müssen vor dem Senden an den Server ausgefüllt werden. Die Präsenz von Platzhaltern prüft `MainWidget` mit folgendem regulären Ausdruck:

```
.*\([A–Z][a–z]+\).*
```

Findet `MainWidget` ein entsprechendes Muster in der Eingabe, wird die Fehlermeldung angezeigt. Das bedeutet auch, dass im Client keine Eingaben an den Server geschickt werden können, die ein einzelnes in runde Klammern gefasstes Wort enthalten. Da bei der Verarbeitung im Server Klammern nicht berücksichtigt werden, ist diese Einschränkung allerdings vertretbar.

Zustände des Clients

Viele Browser speichern die von Benutzern besuchten URLs in einem sogenannten Verlauf. Benutzer dieser Browser können den Verlauf nutzen, um zuvor besuchte URLs erneut aufzurufen. Doch da mit dem Google Web Toolkit entwickelte Anwendungen nur über jeweils eine URL angesprochen werden und die entsprechende Webseite dynamisch mit JavaScript modifiziert wird, sind verschiedene Zustände der Anwendungen zunächst nicht von Außen zugänglich. Bei NASfVI würde das bedeuten, dass vorangegangene Anfragen nicht mit Hilfe des Browser-Verlaufs erneut geladen werden könnten, sondern stets neu einzugeben wären. Das Google Web Toolkit ermöglicht jedoch durch die Klasse `com.google.gwt.user.client.History` eine Interaktion mit dem Verlauf, um einzelne Zustände im Verlauf abzulegen. Ein Zustand ist dabei eine Zeichenkette, die als Fragment an die URL angehängt wird. Der Client von NASfVI nutzt die `History`-Klasse, um Eingaben und Offset-Werte als Zustände abzulegen:

```
@Override
public final void setHistoryState(final String query, final int
    offset) {
        String q = query;

        if (q == null || q.isEmpty()) {
                q = suggestBox.getText();
        }

        History.newItem(URL.encodeQueryString(q)
                + "&"
                + String.valueOf(offset), true);
}
```

Diese Methode wird von `MainWidget` implementiert und sie wird durch das Interface `de.spartusch.nasfvi.client.HistoryStateSetter` vorgegeben. Die Definition als Interface ermöglicht es, die Funktionalität gezielt anderen Objekten zur Verfügung zu stellen.[2] Der Aufruf von `History.newItem` erzeugt einen neuen Zustand und speichert diesen im Verlauf des Browsers. Das wiederum veranlasst das `MainWidget`, die im Zustand gespeicherte Eingabe an den Server zu schicken. Wird die Methode ohne einer gegebenen

[2] Siehe dazu auch das entsprechende Code-Beispiel in Kapitel 5.1.1.

natürlichsprachigen Eingabe aufgerufen, d. h. hat die Variabel `query` den Wert `null`, dann wird der aktuelle Inhalt des Eingabefelds über `suggestBox` ermittelt und verwendet.
Da `MainWidget` bei der Initialisierung des Clients überprüft, ob ein Zustand übergeben worden ist, ist es auch möglich, den Zustand von Außen vorzugeben. Dazu muss der Client mit einem angehängten Fragment aus natürlichsprachiger Anfrage und Offset aufgerufen werden. Das erlaubt es schließlich auch den jeweiligen Zustand in einem Link weiterzugeben. Die in Kapitel 2.3.3 gezeigte OpenSearch-Schnittstelle nutzt das, um den Client mit einer gegebenen Suchanfrage aufzurufen. Wird NASfVI auf dem gleichen Computer wie der verwendete Browser ausgeführt, dann kann der Client zum Beispiel wie folgt aufgerufen werden:

```
http://localhost:8080/#Wer+hielt+Syntax&0
```

Beim Aufruf des Clients über diese URL wird *„Wer hielt Syntax“* an den Server geschickt und das Ergebnis angezeigt. Das Fragment folgt auf das Zeichen `#`. Es enthält die natürlichsprachige Frage und den Wert `0` für den Offset.

5.2.2. Abfragen der Satzvervollständigungen

Als Widget für die Eingabe von natürlichsprachigen Anfragen verwendet das `MainWidget` ein Objekt der Klasse `com.google.gwt.user.client.ui.SuggestBox`. Eine `SuggestBox` ist ein Textfeld, das unterhalb des Eingabefelds mehrere Vorschläge einblenden kann, die zu dem Inhalt des Felds passen. Die anzuzeigenden Vorschläge werden dabei von Objekten der Klasse `com.google.gwt.user.client.ui.SuggestOracle` ermittelt. NASfVI erweitert die `SuggestOracle`-Klasse zur Klasse `de.spartusch.nasfvi.client.Suggestions`. Diese Klasse veranlasst das Abfragen von Satzvervollständigungen beim Server und stellt sie der verwendeten `SuggestBox` zur Verfügung. Die Abfrage selbst wird von Objekten der Klasse `de.spartusch.nasfvi.client.SuggestRequest` durchgeführt. Dabei ist es nicht wünschenswert, dass zu jedem in die `SuggestBox` eingegebenen Zeichen Satzvervollständigungen berechnet werden. Denn das würde nicht nur den Server enorm belasten. Sondern es ist - da der Server eine gewisse Zeit benötigt, um Satzvervollständigungen zu berechnen - auch zu berücksichtigen, dass immer eine Verzögerung entsteht, bis die Satzvervollständigungen angezeigt werden. Tippt der Benutzer zu schnell, dann sind die eingeblendeten Satzvervollständigungen zum Zeitpunkt der Anzeige u. U. nicht mehr für die jeweils aktuelle Eingabe passend. Aus diesem Grund erweitert `SuggestRequest` die Klasse `com.google.gwt.user.client.Timer`. Denn das erlaubt eine zeitlich verzögerte Ausführung der Abfragen. `Suggestions` erzeugt zwar bei jedem eingegebenen Zeichen ein `SuggestRequest`, verzögert dessen Ausführung jedoch stets um 400 Millisekunden. Werden innerhalb dieser 400 Millisekunden weitere Zeichen in die `SuggestBox` eingegeben, wird das aktuelle `SuggestRequest`-Objekt gestoppt und durch ein neues mit der neuen Eingabe ersetzt. Aus diesem Grund wird der Server nur nach Satzvervollständigungen abgefragt, wenn der Benutzer mindestens 400 Millisekunden lang keine neuen Zeichen eingegeben hat. Der Wert von 400 Millisekunden ist willkürlich gewählt, hat sich in informellen Tests allerdings bewährt. Die Satzvervollständigungen werden an der in

Kapitel 4.3.2 besprochenen Schnittstelle von `SuggestRequest` mit einer einfachen HTTP GET-Anfrage abgefragt.

5.2.3. Verarbeitung der Antworten auf natürlichsprachige Anfragen

Hat der Benutzer eine natürlichsprachige Anfrage eingegeben und schickt diese ab, dann sendet das `MainWidget` die Anfrage im Hintergrund an den Server. Es fragt dabei die in Kapitel 4.3.3 gezeigte Schnittstelle ab. Die Antwort des Servers wird vom Client geparset und als Objekt der Klasse `de.spartusch.nasfvi.client.NResponse` repräsentiert. Bei `NResponse` handelt es sich um ein JavaScript-Objekt. Da JavaScript-Objekte zur Laufzeit aus Zeichenketten, die Objekte im JSON-Format beschreiben, erzeugt werden, sind `NResponse`-Objekte direkte Repräsentationen der Server-Antworten. Neue Instanzen der `NResponse`-Klasse werden nicht mit einem Konstruktor instanziiert, sondern mit einer statischen Methode aus dem Antwortformat geparset:

```
public static NResponse parse(final String json) {
        JSONArray array = JSONParser.parseLenient(json).isArray();

        if (array == null || array.size() != 2) {
                throw new IllegalArgumentException();
        }

        return (NResponse) array.getJavaScriptObject();
}
```

Auf einzelne Attribute von JavaScript-Objekten kann in nativen Methoden zugegriffen werden. Das Schlüsselwort `native` markiert in Java Methoden, deren Implementierung nicht in Java vorliegt.[3] Bei der Verwendung des Google Web Toolkits kennzeichnet es Methoden, die mit JavaScript implementiert sind. Der JavaScript-Code muss dabei als Kommentar auf den Methodenkopf folgen und zwischen `/*-{` und `}-*/` notiert sein. Mit nativen Methoden implementiert `NResponse` Getter für die Attribute des Antwortformats, zum Beispiel:

```
public final native String getQuery() /*-{
        return this[0].NQuery.Query;
}-*/;

public final native String getSimilarityQuery() /*-{
        return this[0].NQuery.SQuery;
}-*/;

public final native String getPlainAnswer() /*-{
        return this[1].Answer;
}-*/;
```

Die natürlichsprachige Antwort, so wie sie von der Sprachverarbeitungskomponente erzeugt und vom Server geliefert wird, wird mit der Methode `getPlainAnswer()` abgefragt.

[3] Siehe `http://java.sun.com/docs/glossary.html#N`

Die Methode `getAnswer()` dagegen modifiziert die natürlichsprachige Antwort für die Anzeige im Client:

```
public final String getAnswer() {
        String answer = getPlainAnswer().trim();

        if (getFields().length() == 0) {
                if (getHits() > 0) {
                        answer = "Ja. " + answer;
                } else {
                        answer = "Nein. " + answer;
                        if (answer.endsWith(" statt")) {
                                answer = answer.replaceAll(" statt$",
                                    " nicht statt");
                        } else {
                                answer = answer + " nicht";
                        }
                }
        }

        return answer + ". ";
}
```

Wenn die natürlichsprachige Anfrage keine Felder erfragt hat, dann muss es sich bei der Anfrage um einen Aussagesatz gehandelt haben. Zum Beispiel um einen Satz wie *„Im Wintersemester 2008/2009 fand eine Vorlesung über Grammatikformalismen statt“*. In einem solchen Fall ist die natürlichsprachige Antwort eine Wiederholung der Anfrage. Je nachdem, ob mit der erzeugten Suchanfrage Veranstaltungsdokumente gefunden werden konnten, fügt `getAnswer()` ein „Ja.“ oder ein „Nein.“ zu der natürlichsprachigen Antwort hinzu. Das soll verhindern, dass der Benutzer durch eine einfache Wiederholung der Anfrage verwirrt wird.
Konnten zu der natürlichsprachigen Anfrage keine Veranstaltungsdokumente gefunden werden, so ist sie falsch und muss in der natürlichsprachigen Antwort verneint werden. Da aber Negationen nicht Teil des in dieser Magisterarbeit implementierten Sprachfragments sind, verneint `getAnswer()` falsche Aussagen durch das Hinzufügen von „nicht“ an das Ende der Sätze. Dieses Vorgehen ist auf Aussagesätze mit fast allen Verben des Sprachfragments anwendbar. Die einzige Ausnahme stellt das Verb „stattfinden“ dar. Es wird berücksichtigt, indem bei zu verneinenden Aussagen, die auf das Partikel „statt“ enden, dieses Satzende durch „nicht statt“ ersetzt wird.

Nachdem das `MainWidget` schließlich ein `NResponse`-Objekt aus der Server-Antwort erzeugt hat, instanziiert es ein `NResponseWidget`, welches die Werte des `NResponse`-Objekts für den Benutzer des Clients darstellt. Das `NResponseWidget` überträgt dazu die vom `NResponse`-Objekt zur Verfügung gestellten Werte in Attribute, die aufgrund des deklarativen Layouts der Klasse im Browser angezeigt werden.[4]

[4]Das deklarative Layout der `NResponseWidget`-Klasse diente in Kapitel 5.1.1 als Beispiel.

6. Erweiterungsmöglichkeiten

Im Rahmen dieser Magisterarbeit wurde ein umfangreiches natürlichsprachiges Anfragesystem für Vorlesungsverzeichnisse im Internet entwickelt. Obwohl das System für eine Magisterarbeit einen großen Umfang hat, gibt es noch viele Möglichkeiten für Erweiterungen und Verbesserungen. Einige davon sollen an dieser Stelle kurz erwähnt werden.

Sprachliche Erweiterungen

Das implementierte Fragment des Deutschen behandelt zwar einige sprachliche Phänomene, doch ist leicht einsehbar, dass ein solches Fragment immer bruchstückhaft bleibt und daher stets erweitert werden kann. So könnte zum Beispiel das Lexikon vergrößert werden, um mehr universitäre Begriffe abzubilden. Aber auch weitere syntaktische Phänomene könnten zu NASfVI hinzugefügt werden. Die Grundlage für Relativsätze ist zum Beispiel mit der Unterstützung von Verbletztsätzen in der Flexion bereits vorhanden. Nicht zuletzt wäre eine Anwendung und entsprechende Erweiterung des in dieser Magisterarbeit entwickelten Systems auf andere Themengebiete als Vorlesungsverzeichnisse interessant.

Toleranz bei Eigennamen

Auch bei der Verarbeitung der natürlichsprachigen Anfragen mit Lucene sind Erweiterungen möglich. Eine zu testende Erweiterung wäre z. B. die Verwendung eines phonetischen Algorithmus bei der Indizierung von Eigennamen, um neben den exakten Schreibweisen der Eigennamen auch phonetische Repräsentationen der Eigennamen zu indizieren. Auf diese Weise könnten Eigennamen auch dann gefunden werden, wenn ein Benutzer einen gleich klingenden, aber anders geschriebenen Namen angibt. Zum Beispiel könnte die Suchanfrage nach einem Professor Mayer mit einem phonetischen Algorithmus auch auf einen Professor Meier abgebildet werden. In Lucene können die Implementierungen der phonetischen Algorithmen Metaphone und Soundex verwendet werden ([LUC10], Seite 129ff). Alternativ oder zusätzlich dazu wäre die Integration von Rechtschreibkorrekturen mit Hilfe von Lucene ([LUC10], Seite 277ff) eine sinnvolle Erweiterung des Systems. Die Interaktion mit dem Benutzer könnte zum Beispiel um Rückfragen und Schreibvorschläge ergänzt werden. Auf diese Weise könnte NASfVI einen Korrekturvorschlag anbieten, falls für eine bestimmte Anfrage mit einem Eigennamen kein Veranstaltungsdokument gefunden werden kann, für einen ähnlich geschriebenen Eigennamen dagegen aber schon.

Unterstützung gesprochener Sprache

Eine andere Erweiterungsmöglichkeit stellt wiederum die Verwendung von VoiceXML[1] oder XHTML + Voice[2] dar. Mit diesen Techniken ließe sich ein auf gesprochener Sprache basierender Client für NASfVI entwickeln. Anfragen an den Server müssten dann nicht mehr zwingend getippt werden, sondern könnten von den Benutzern auch gesprochen werden. NASfVI könnte dann nicht nur mit Browsern verwendet werden, sondern auch mit sprachbasierten Telekommunikationsdiensten.

Verbesserte Leistungsfähigkeit

Doch ungeachtet möglicher Erweiterungen ist bei einem System, das auf einem Server im Internet zur Verfügung stehen soll, die Verarbeitungsgeschwindigkeit von besonderem Interesse. Mit jpl kann allerdings nur eine SWI-Prolog-Instanz verwendet werden, was die Verwendung mehrerer Prolog-Instanzen zur zeitgleichen Verarbeitung von Anfragen erschwert. Aus diesem Grund kann es wünschenswert sein, für eine Verbesserung der Leistungsfähigkeit des Servers andere Möglichkeiten auszuschöpfen. Eine Alternative zu mehreren Instanzen wäre es, wenn NASfVI dahingehend erweitert würde, dass eine Vielzahl an möglichen Sätzen der Sprachverarbeitungskomponente im Voraus berechnet werden. Bei Anfragen könnte dann auf diese vorberechnete Liste zurückgegriffen werden, ähnlich wie die Sprachverarbeitungskomponente bereits jetzt ein Vollformenlexikon erzeugen und abspeichern kann. Es würde sich aber auch anbieten, einen Proxy-Server zu verwenden, der alle HTTP-Anfragen in einem Cache speichern kann. Mit einem solchen Proxy-Server könnten alle Anfragen an den NASfVI-Server zwischengespeichert und gegebenenfalls direkt aus dem Cache beantwortet werden, ohne dass die Antworten für jede Anfrage neu berechnet werden müssten.[3]

Alternatives Datenmodell

In der vorliegenden Form verwendet NASfVI benannte Felder, um die Veranstaltungsinformationen einzulesen und abzufragen. Alternativ ist es jedoch auch denkbar, die jeweiligen Informationen in Form von Tupeln aus Entitäten, Prädikaten und Objekten zu indizieren, um diese Beziehungen zu modellieren und auch abfragen zu können. Im Juli 2009 wurde mit SIREn (Semantic Information Retrieval Engine) eine auf Lucene basierende Erweiterung veröffentlicht,[4] die genau das ermöglicht. Durch die Unterstützung von SPARQL[5] als Anfragesprache ermöglicht SIREn eine gezielte Anfrage, sowie die Verwendung des in Tupeln modellierten Wissens ([LUC10], Seite 394ff). Eine mögliche Erweiterung von NASfVI besteht daher in der Verwendung von SIREn, um eine größere

[1] `http://www.w3.org/TR/voicexml20/`
[2] `http://www.w3.org/TR/xhtml+voice/`
[3] Ein derartiger Proxy-Server wäre zum Beispiel squid: `http://www.squid-cache.org/`.
[4] Siehe `http://lucene.472066.n3.nabble.com/ANN-SIREn-0-1-Release-td560772.html`
[5] `http://www.w3.org/TR/rdf-sparql-query/`

Anzahl an natürlichsprachigen Fragen zuzulassen. Da SIREn schema-frei ist, müssen die indizierbaren Relationen nicht im Voraus festgelegt werden, so dass der Index jederzeit um neue Relationen erweitert werden kann. Bei einer geeigneten Übertragung der Semantik von natürlichsprachigen Anfragen auf Suchanfragen von SIREn wäre es denkbar, die Beschränkung auf ein Themengebiet von NASfVI aufzugeben und mit der in dieser Magisterarbeit entwickelten Grundlage ein allgemeines Anfragesystem zu entwickeln.

Teil III.

Anhang

Anhang A.

Mehr zur Sprachverarbeitungskomponente

A.1. Das Test-Framework

SWI-Prolog unterstützt ein Framework für Unit-Tests in Prolog.[1] Mit diesem Framework können Blöcke mit einzelnen Testfällen definiert werden. Die definierten Testblöcke lassen sich getrennt oder zusammen aufrufen und durchlaufen automatisch alle definierten Testfälle. Testfälle sind Klauseln des Prädikats `test/1`[2] und müssen deterministisch beweisbar sein, damit das Framework sie als bestanden wertet. Das Argument von `test/1` identifiziert einzelne Testfälle. Das Framework nutzt diese Angabe, um fehlgeschlagene Testfälle dem Benutzer mitzuteilen. Mit Tests kann das erwartete Verhalten des Systems definiert und überprüft werden. Aufgrund möglicher Seiteneffekte durch die veränderliche Prolog-Datenbank und der Verwendung von freien Variabeln sind Tests insbesondere für die Entwicklung größerer Prolog-Anwendungen sehr nützlich. Denn sobald Veränderungen im Quellcode vorgenommen worden sind, kann mit Tests sehr schnell festgestellt werden, ob sich das System bei den einzelnen Testfällen noch wie erwartet verhält, oder ob mit den Veränderungen ein unerwartetes Verhalten eingeführt worden ist.

Der folgende Block mit der Bezeichnung `suggest` definiert zum Beispiel 12 Testfälle für `suggest`-Anfragen:

```
:- begin_tests(suggest).
test(1)  :- suggest('ueber semantik f', 0, _, _).
test(3)  :- suggest('hat eine vorlesung s', 0, _, _).
test(4)  :- suggest('fand die vorle', 0, _, _).
test(5)  :- suggest('gibt es', 0, _, _).
test(6)  :- suggest('ueber syntax haelt dozent mueller eine vorlesung
    ', 0, _, _).
test(7)  :- \+suggest('eine vorlesung findet eine vorl', 0, _, _).
test(8)  :- suggest('syntax handelt von syntax', 0, _, _).
test(9)  :- \+suggest('hat i', 0, _, _).
test(10) :- suggest('haelt herr mueller od', 0, _, _).
```

[1] Siehe `http://www.swi-prolog.org/pldoc/package/plunit.html`

[2] Es kann auch das Prädikat `test/2` mit zusätzlichen Optionen verwendet werden. Doch `test/2` wird in dieser Arbeit nicht verwendet und daher nicht weiter erläutert.

```
test(11) :- suggest('haelt herr x eine vorlesung in dem ra',
        0, _, _).
test(12) :- suggest('wann wurde montags syntax gehalten', 0, _, _).
test(13) :- suggest('welche vorlesung ae', 0, _, _).
:- end_tests(suggest).
```

Die Testfälle 7 und 9 sind negiert, da das erwartete Verhalten ist, dass die Sprachverarbeitungskomponente diese `suggest`-Anfragen nicht beweisen kann. Denn der Testfall 7 lässt sich zu keinem gültigen Satz des implementierten Sprachfragments vervollständigen. Der Testfall 9 wiederum ist zu kurz, so dass die Sprachverarbeitungskomponente dessen Verarbeitung frühzeitig abbrechen soll, um keine zu zeitintensiven Berechnungen durchzuführen.

Die Tests sind in der Datei `test` definiert und werden nicht automatisch von den Startdateien `start_gf` und `start_vf` eingebunden. Um das Test-Framework mit den für NASfVI definierten Testfällen aufrufen zu können, muss daher neben der eigentlichen Startdatei `test` hinzugefügt werden:

```
?- [start_vf, test].
```

In NASfVI sind 100 Testfälle definiert, die auf die Blöcke `syntax`, `suggest`, `semantik`, `antworten` und `fehler` verteilt sind. In jedem Block wird das Verhalten der Sprachverarbeitungskomponente auf dem jeweiligen Gebiet getestet.

Das Test-Framework führt alle Testfälle aus, wenn das Prädikat `run_tests/0` aufgerufen wird:

```
?- run_tests.
% PL-Unit: syntax ..............................................
 ............ done
% PL-Unit: suggest ............ done
% PL-Unit: semantik .......... done
% PL-Unit: antworten .............. done
% PL-Unit: fehler . done
% All 100 tests passed
true.
```

Wie zuvor erwähnt, können die Testblöcke auch gezielt aufgerufen werden. Dies geschieht mit dem Prädikat `run_tests/1`, wie das folgende Beispiel mit dem Testblock `suggest` zeigt:

```
?- run_tests(suggest).
% PL-Unit: suggest ............ done
% All 12 tests passed
true.
```

Nach jeder Veränderung des Quellcodes ist darauf zu achten, dass nach wie vor alle 100 Testfälle erfolgreich ausgeführt werden können. Beim Hinzufügen neuer Funktionen zur Sprachverarbeitungskomponente sollte sichergestellt werden, dass die neue Funktionalität auch mit neuen Testfällen abgedeckt wird.

A.2. Die Vorverarbeitung

Die Vorverarbeitung der Sprachverarbeitungskomponente tokenisiert die übergebenen Sätze. Jedes Wort wird bei der Vorverarbeitung in eine Liste von Buchstaben zerlegt. Sätze wiederum werden in Listen dieser Buchstabenlisten zerlegt. Für die Verwendung mit dem Suggest-Modus kann das letzte Wort eines Satzes zudem mit einem `#` markiert werden, wodurch dem Lexikon (siehe Kapitel 3.1.5) signalisiert wird, dass das Wort vervollständigt werden kann.
Das Prolog-Atom `'wer hielt syntax'` wird also zum Beispiel zu `[[w,e,r], [h,i,e,l, t], [s,y,n,t,a,x]]` verarbeitet. Wird ein Satz dagegen für den Suggest-Modus vorverarbeitet, so wird das jeweils letzte Wort mit einem `#` markiert. Das Atom `'wer hie'` würde in diesem Fall also zu `[[w,e,r], ['#',h,i,e]]` verarbeitet werden. Das Lexikon würde dann anhand des `#` erkennen, dass das letzte Wort zu vervollständigen ist, und es zu `[h,i,e,l,t]` vervollständigen.
Das Prädikat `tokenisiere/3` veranlasst die Vorverarbeitung der Sätze, die stets als Prolog-Atom vorliegen. Der Satz ist das erste Argument des Prädikats. Er wird mit Hilfe von `atom_chars/2` in eine Liste seiner Zeichen zerlegt. Diese Liste wird mit den in `tokenisiere/4` definierten DCG-Regeln geparset und dabei das Ergebnis der Tokenisierung aufgebaut.

```
tokenisiere(S, E, T) :-
        atom_chars(S, C), tokenisiere(E, T, C, []), !.
```

Das zweite Argument von `tokenisiere/3`, `E`, muss entweder das Zeichen `0` oder das Zeichen `1` sein. Es ist der boolesche Wert dafür, ob das jeweils letzte Wort im Satz mit einem `#` markiert werden soll oder nicht. Das letzte Argument enthält schließlich das Ergebnis der Vorverarbeitung in Form einer Liste mit zu Token zusammengefassten Buchstabenlisten, den einzelnen Wörtern und Zeichengruppen.

Gültige Zeichen

Da die Vorverarbeitung mit DCG-Regeln arbeitet, sind die für die Vorverarbeitung gültigen Zeichen festgelegt. Buchstaben, Ziffern, Leerräume (Whitespace), sowie die Zeichen `'/'`, `'“'`, `'.'` und `'-'` sind gültige Zeichen. Mit dem Prädikat `zeichen_/2` sind Hauptgruppen definiert:

```
zeichen_(buchst, C) :- alphabet(A), member(C, A).
zeichen_(ziffer, C) :- ziffern(Z), member(C, Z).
zeichen_(whitespace, C) :- whitespace(W), member(C, W).
zeichen_(separator, '/').
```

Die gültigen Zeichen der Gruppen liegen als Listen vor:

```
alphabet([a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z]).
ziffern(['0','1','2','3','4','5','6','7','8','9']).
whitespace([' ',\t,\n]).
```

Der Zugriff auf die Zeichen erfolgt mit dem Prädikat `zeichen/2`, das die Kategorien berücksichtigt:

```
zeichen(Cat, C) :- zeichen_(Cat, C), !.
```

Kann ein zu parsendes Zeichen nicht in einer passenden Kategorie gefunden werden, reagiert die Vorverarbeitung auf zwei verschiedene Weisen. Zunächst wird geprüft, ob das jeweilige Zeichen in einer anderen als der angefragten Kategorie vorkommt oder den anderen erlaubten Zeichen entspricht. Diese Prüfung stellt sicher, dass es sich um ein gültiges Zeichen handelt. In einem solchen Fall schlägt die Vorverarbeitung einfach fehl. Denn das Zeichen ist zwar gültig, doch entspricht es nicht der geforderten Kategorie:

```
zeichen(_, C) :-
        (zeichen_(buchst, C) ; zeichen_(ziffer, C) ; zeichen_(
            whitespace, C) ; zeichen_(separator, C) ; C = '"' ; C = '.
            ' ; C = '-'), !, fail.
```

Handelt es sich bei einem zu parsenden Zeichen allerdings um kein gültiges Zeichen, so löst die Vorverarbeitung eine Exception aus, die das Vorkommen des ungültigen Zeichens als `domain_error` meldet:

```
zeichen(_, C) :-
        throw(error(domain_error('gueltiges␣Zeichen', C),
        zeichen/2)).
```

Tokengruppen

Die DCG-Regeln definieren verschiedene Regel-Gruppen, die im Ergebnis eigenständige Token aufbauen. Diese Gruppen sind Wörter, die nur aus Buchstaben und Bindestrichen bestehen, alphanumerische Zeichen, das Separator-Zeichen `'/'` und durch Anführungszeichen markierte Blackboxen.
Leerräume werden bei der Vorverarbeitung zwar erkannt und verarbeitet, sie bauen aber keine Token auf und sind auch in den Token der Tokengruppen nicht repräsentiert. Die DCG-Regeln für Leerräume parsen und überprüfen die Zeichen daher nur, ohne ein Argument zurückzugeben:

```
leer --> [C], {zeichen(whitespace, C)}.
leer --> [C], {zeichen(whitespace, C)}, leer.
```

Alphanumerische Zeichen

In der Regel werden die Tokens aus alphanumerischen Zeichen aufgebaut. Die Token bestehen dann aus einer Folge von Buchstaben und Ziffern:

```
alphanum([C]) --> [C], {zeichen(buchst, C) ; zeichen(ziffer, C)}.
alphanum([C|T]) --> [C], {zeichen(buchst, C) ; zeichen(ziffer, C)},
    alphanum(T).
```

Die Regeln von `alphanum` geben, wie die Regeln der anderen Tokengruppen auch, die jeweils geparsten Zeichen in einem Argument zurück. Mit diesem Argument werden die Tokens für das Ergebnis der Tokenisierung aufgebaut.
Neben den eigentlichen alphanumerischen Zeichen erlaubt die Gruppe zusätzlich auch die Verwendung von Punkten und Bindestrichen:

```
alphanum(['.', C]) --> ['.'], [C], {zeichen(ziffer, C)}.
alphanum(['.', C|T]) -->
        ['.'], [C], {zeichen(ziffer, C)}, alphanum(T).
alphanum([C, '-'|T]) -->
        [C], ['-'], {zeichen(buchst, C) ; zeichen(ziffer, C)},
            alphanum(T).
```

Das ermöglicht das Parsen von zum Beispiel Raumangaben wie *„1.14“* und von Doppelnamen wie *„Meier-Mustermann“*. Derartige Token kommen zwar nur als Werte für syntaktische Blackboxen vor, doch diese Blackboxen müssen nur dann mit Anführungszeichen markiert werden, wenn sie Leeerräume enthalten und aus mehreren Begriffen bestehen. Blackboxen ohne Anführungszeichen, also mit nur einem Begriff, werden in der Vorverarbeitung als einfache alphanumerische Zeichengruppen tokenisiert.

Blackbox-Token

Die einfachen Blackboxen mit nur einem Begriff werden von der alphanumerischen Gruppe verarbeitet. Komplexere Blackboxen mit mehreren Begriffen werden dagegen als eigene Gruppe tokenisiert. Tokens dieser Klasse zeichnen sich dadurch aus, dass sie in der Eingabe immer mit einem Anführungszeichen eingeleitet werden:

```
blackbox_token(BBT) --> ['"'], beliebig(BBT), {!}.
```

Das Anführungszeichen selbst wird dabei nicht im erzeugten Token zurückgegeben. Nach dem Anführungszeichen können beliebige - auch nicht gültige - Zeichen folgen. Jedes folgende Zeichen wird bis zum abschließenden Anführungszeichen dem Blackbox-Token zugerechnet:

```
beliebig([]) --> ['"'].
beliebig([C|T]) --> [C], {C \= '"'}, beliebig(T).
beliebig([C]) --> [C], {C \= '"'}.
```

Da ein Blackbox-Token auch am Ende einer Eingabe stehen kann und im Suggest-Modus das abschließende Anführungszeichen unter Umständen noch nicht getippt worden ist, kann das Blackbox-Token auch auf ein Nicht-Anführungszeichen enden.

Token

Im Wesentlichen werden die Tokens von der alphanumerischen Gruppe und der Blackbox-Gruppe aufgebaut. Zur Vereinfachung werden diese beide Gruppen daher zur DCG-Regel `token` zusammengefasst:

```
token(T) --> alphanum(T) ; blackbox_token(T).
```

Endwörter

Neben der alphanumerischen Gruppe und der Blackbox-Gruppe gibt es als Sonderfall noch eine Wort-Gruppe. Wörter sind in der Vorverarbeitung eine Tokengruppe, die nur Buchstaben und Bindestriche enthalten:

```
wort([C, '-'|T]) --> [C], ['-'], {zeichen(buchst, C)}, wort(T).
wort([C]) --> [C], {zeichen(buchst, C)}.
wort([C|T]) --> [C], {zeichen(buchst, C)}, wort(T).
```

Im Unterschied zur alphanumerischer Gruppe enthalten Wörter keine Ziffern. Sie sind also ein Sonderfall der alphanumerischen Gruppe und bestehen aus einer Untermenge der alphanumerischen Zeichen. Da das Lexikon keine Ziffern enthält, werden nur diese Wörter von dem Lexikon vervollständigt. Aus diesem Grund wird das jeweils letzte Wort nicht mit der allgemeineren Gruppe für alphanumerische Zeichen verarbeitet, sondern mit der Wort-Gruppe. Nur Tokens dieser Gruppe können bei der Tokenisierung mit einem `#` markiert werden.

Trennzeichen

Neben der alphanumerischen Gruppe, der Blackbox-Gruppe und der Wort-Gruppe gibt es noch eine letzte Art von Token. Die Trennzeichen:

```
separator([C]) --> [C], {zeichen(separator, C)}.
```

Wie im Abschnitt für gültige Zeichen zu sehen ist, wird im implementierten Fragment des Deutschen ausschließlich das Zeichen `'/'` als Trennzeichen genutzt. Es dient für die Angabe von Wintersemestern. Eine Wintersemesterangabe wie *„2008/2009“* besteht also nicht aus einem, sondern aus drei Token: *2008*, */*, *2009*. Aufgrund dieser Trennung kann die Semesterangabe im Suggest-Modus während der Eingabe ergänzt werden. Denn eine fehlende Jahreszahl wird als normales Token syntaktisch ergänzt. Würde eine solche Semesterangabe dagegen als ein Token geparset werden, dann müsste das Lexikon die Vervollständigung des Tokens leisten.

Tokenisierung

Die mögliche Abfolge der Token wird durch das Prädikat `tokenisiere/4` festgelegt. Im Wesentlichen folgt ein Token auf einen Leerraum, ein Trennzeichen oder ein anderes Token:

```
tokenisiere(E, [T|TL]) --> token(T), leer, {!}, tokenisiere(E, TL).
tokenisiere(E, [T, S|TL]) -->
        token(T), separator(S), {!}, tokenisiere(E, TL).
tokenisiere(E, [T|TL]) --> token(T), tokenisiere(E, TL).
```

Wird das jeweils letzte Token einer Eingabe nicht mit einem `#` markiert, so ist der boolesche Wert dafür `0`. In diesem Fall wird als letztes Token ein einfaches Token geparset und zurückgegeben:

```
tokenisiere(0, [T]) --> token(T).
```

Wenn der boolesche Wert jedoch `1` ist und das letzte Wort mit einem `#` markiert werden soll, dann gibt es drei Möglichkeiten für das letzte Token:

- Es entspricht der Wortgruppe und kann vom Lexikon ergänzt werden. In diesem Fall wird das jeweilige Token mit einem vorangestellten `#` markiert:

```
tokenisiere(1, [['#'|T]|_]) --> wort(T).
```

- Oder es handelt sich um ein allgemeines Token, das vom Lexikon nicht ergänzt werden kann und daher nicht mit einem # markiert wird:

  ```
  tokenisiere(1, [T|_]) --> token(T).
  ```

- Oder es handelt sich um ein Trennzeichen:

  ```
  tokenisiere(1, [T|_]) --> separator(T).
  ```

Die mit `tokenisiere/4` aufgebauten Tokens werden schließlich als tokenisiertes Ergebnis der Vorverarbeitung zurückgegeben. Alle weiteren Arbeitsschritte der Sprachverarbeitumgskomponente basieren auf diesem Ergebnis.

A.3. Die Endungstabellen

Die in der Flexion der Sprachverarbeitungskomponente verwendeten grundlegenden Flexionsendungen bei Verben und Nomen sind als Fakten des Prädikates `endung/4` abgelegt. Die Argumente von `endung/4` sind wie folgt:

1. Die Wortart. Dieses Argument hat entweder den Wert `v` für Verben oder den Wert `n` für Nomen.
2. Die Flexionsklasse der Endung.
3. Die Formmerkmale. Die Formmerkmale sind bei Verben eine Liste aus Person, Numerus und Tempus. Bei Nomen sind die Formmerkmale wiederum eine Liste aus Numerus und Kasus.
4. Die Flexionsendung.

Verben

Bei Verben wird unterschieden, ob die Endungen der regelmäßigen Konjugation von schwachen Verben oder der unregelmäßigen Konjugation von starken Verben folgen. Die Flexionsendungen sind jeweils für das Singular und Plural, sowie für das Präsens, das Präteritum und das Partizip II im Indikativ angegeben. Die Endungen für Indikativ Präsens und Indikativ Präteritum sind [DUG06] (Seite 441f) entnommen.

Regelmässige Konjugation: schwache Verben

```
endung(v, rg, [1,sg,praes], 'e').
endung(v, rg, [2,sg,praes], 'st').
endung(v, rg, [3,sg,praes], 't').
endung(v, rg, [1,pl,praes], 'en').
```

```
endung(v, rg, [2,pl,praes], 't').
endung(v, rg, [3,pl,praes], 'en').

endung(v, rg, [1,sg,praet], 'te').
endung(v, rg, [2,sg,praet], 'test').
endung(v, rg, [3,sg,praet], 'te').
endung(v, rg, [1,pl,praet], 'ten').
endung(v, rg, [2,pl,praet], 'tet').
endung(v, rg, [3,pl,praet], 'ten').

endung(v, rg, part2, 't').
```

Unregelmässige Konjugation: starke Verben

```
endung(v, urg(_,_), [1,sg,praes], 'e').
endung(v, urg(_,_), [2,sg,praes], 'st').
endung(v, urg(_,_), [3,sg,praes], 't').
endung(v, urg(_,_), [1,pl,praes], 'en').
endung(v, urg(_,_), [2,pl,praes], 't').
endung(v, urg(_,_), [3,pl,praes], 'en').

endung(v, urg(_,_), [1,sg,praet], '').
endung(v, urg(_,_), [2,sg,praet], 'st').
endung(v, urg(_,_), [3,sg,praet], '').
endung(v, urg(_,_), [1,pl,praet], 'en').
endung(v, urg(_,_), [2,pl,praet], 't').
endung(v, urg(_,_), [3,pl,praet], 'en').

endung(v, urg(_,_), part2, 'en').
```

Nomen

Die Endungen der Nomen sind nach Singular und Plural getrennt gruppiert. Bei jeder Endungsgruppe sind aus Gründen der Einheitlichkeit stets alle vier Fälle als Fakt angegeben, selbst wenn sich einzelne Endungen nicht unterscheiden. Die Tabellen folgen in Inhalt und Benennung den Flexionsklassen[3] von [CNOO].

Endungen im Singular

Tabelle '-'

```
endung(n, -, [sg,nom], '').
endung(n, -, [sg,gen], '').
endung(n, -, [sg,dat], '').
endung(n, -, [sg,akk], '').
```

[3] Für eine Übersicht dieser Flexionsklassen, siehe `http://www.canoo.net/services/OnlineGrammar/InflectionRules/FRegeln-N/Texte/Flexionskl.html`

Tabelle 'es'

```
endung(n, es, [sg,nom], '').
endung(n, es, [sg,gen], 's').
endung(n, es, [sg,dat], '').
endung(n, es, [sg,akk], '').
```

Tabelle 'en'

```
endung(n, en, [sg,nom], '').
endung(n, en, [sg,gen], 'en').
endung(n, en, [sg,dat], 'en').
endung(n, en, [sg,akk], 'en').
```

Tabelle 's'

```
endung(n, s, [sg,nom], '').
endung(n, s, [sg,gen], 's').
endung(n, s, [sg,dat], '').
endung(n, s, [sg,akk], '').
```

Endungen im Plural

Tabelle 'en'

```
endung(n, en, [pl,nom], 'en').
endung(n, en, [pl,gen], 'en').
endung(n, en, [pl,dat], 'en').
endung(n, en, [pl,akk], 'en').
```

Tabelle 'e'

```
endung(n, e, [pl,nom], 'e').
endung(n, e, [pl,gen], 'e').
endung(n, e, [pl,dat], 'en').
endung(n, e, [pl,akk], 'e').
```

Tabelle '-'

```
endung(n, -, [pl,nom], '').
endung(n, -, [pl,gen], '').
endung(n, -, [pl,dat], '').
endung(n, -, [pl,dat], 'n').
endung(n, -, [pl,akk], '').
```

Anhang B.

Verwendung von NASfVI

B.1. Starten von NASfVI

NASfVI kann auf zwei Arten gestartet werden:

- Der Server von NASfVI kann in einem Servlet-Container ausgeführt werden. Dabei steht der volle in dieser Magisterarbeit beschriebene Funktionsumfang von NASfVI zur Verfügung.
- Die Sprachverarbeitungskomponente kann in Prolog getrennt aufgerufen und verwendet werden. Dabei steht nur die in den Kapiteln 2.2 und 3 beschriebene Sprachverarbeitungskomponente zur Verfügung.

In den folgenden beiden Abschnitten werden diese beiden Startmöglichkeiten beschrieben.

Den Server starten

Im Verzeichnis `war` liegen die ausführbaren Dateien und weitere Ressourcen des Servers im Web-Archive-Format bereit. Dieses Dateiformat wird von Servlet-Containern für Web-Applications genutzt. Es bündelt den Java-Code mit den statisch genutzten Ressourcen, sowie einigen Konfigurationseinstellungen.[1] Der bei NASfVI beiliegende Servlet-Container jetty[2] unterstützt dieses Dateiformat auch in entpackter Form, so wie es im Verzeichnis `war` zur Verfügung steht.
jetty befindet sich im Verzeichnis `jetty` und ist so konfiguriert, dass das Verzeichnis `war` als Web-Application genutzt wird. Daher ist es ausreichend, jetty auszuführen, um den NASfVI-Server zu starten. jetty kann mit der Datei `jetty/start.jar` gestartet werden. Dabei ist jedoch zu beachten, dass in der Java-Systemeigenschaft `java.library.path` der absolute Pfad zur jeweils zu nutzenden Installation von SWI-Prolog gesetzt sein muss. Unter Mac OS X ist ein möglicher Pfad z. B. `/opt/local/lib/swipl-5.8.3/lib/i386-darwin10.4.0/`. Unter Verwendung dieses Pfads für SWI-Prolog kann der Server z. B. wie folgt gestartet werden:

```
java -server -Djava.library.path=/opt/local/lib/swipl-5.8.3/lib/i386-
    darwin10.4.0/ -jar start.jar
```

[1] Siehe `http://en.wikipedia.org/wiki/WAR_file_format_(Sun)`
[2] Siehe `http://jetty.codehaus.org/jetty/`

Nachdem jetty und damit der Server gestartet worden ist, steht der NASfVI-Server und der Client unter `http://localhost:8080/` zur Verfügung.

Nur die Sprachverarbeitungskomponente nutzen

Der Quellcode der Sprachverarbeitungskomponente befindet sich im Verzeichnis `src/grammar`. Alle Angaben in diesem Abschnitt sind relativ zu diesem Pfad.
Zum Laden der Sprachverarbeitungskomponente können zwei verschiedene Startdateien genutzt werden: `start_gf` und `start_vf`. Die beiden Dateien unterscheiden sich nur in der Verwendung des Lexikons. Mit der Datei `start_gf` wird das Grundformenlexikon verwendet, mit der Datei `start_vf` dagegen das Vollformenlexikon in der Datei `vollformen.liste`. Beide Dateien binden die notwendigen Dateien der Sprachverarbeitungskomponente automatisch ein. Lediglich die Datei `test` für das in Anhang A.1 beschriebene Test-Framework wird nicht automatisch eingebunden.
Sobald die Sprachverarbeitungskomponente mit einer der beiden Startdateien in Prolog geladen worden ist, kann sie in Prolog verwendet werden. Die Sprachverarbeitungskomponente besitzt keinerlei Abhängigkeiten von Java oder dem restlichen Server.

B.2. Einlesen neuer Vorlesungsdaten

Wann immer in dieser Magisterarbeit Beispiele von Antworten des Servers gezeigt werden, basieren diese auf den Veranstaltungen des Centrums für Informations- und Sprachverarbeitung der Ludwig-Maximilians-Universtität München. Zur Extraktion dieser Veranstaltungsdaten aus dem Online-Vorlesungsverzeichnis der Universität[3] wurde die Klasse `de.spartusch.nasfvi.EventExtractor` entwickelt.
Mit dieser Klasse werden aus der Standardeingabe Links zu Detailseiten von Veranstaltungen im Vorlesungsverzeichnis zeilenweise ausgelesen. Jede Detailseite wird daraufhin heruntergeladen und die von NASfVI verarbeiteten Informationen werden automatisch extrahiert. Diese Daten werden schließlich in dem in Kapitel 2.3.1 beschriebenem XML-Format in die Standardausgabe geschrieben.
Um neue Vorlesungsdaten zu NASfVI hinzuzufügen, können Veranstaltungsdaten mit `EventExtractor` extrahiert werden, müssen aber manuell in die Datei `war/WEB-INF/index.xml` übertragen werden. Wie in Kapitel 4.3.1 gezeigt, ist es diese Datei, die beim Starten von NASfVI und beim Aufbau des Index im Speicher eingelesen wird.
Der `EventExtractor` kann wie folgt gestartet werden:

```
java -cp war/WEB-INF/classes/ de.spartusch.nasfvi.EventExtractor
```

Es können nicht nur Detailseiten von Veranstaltungen des Centrums für Informations- und Sprachverarbeitung verwendet werden. Der `EventExtractor` kann auch die Veranstaltungsdaten aller Institute aus dem Online-Vorlesungsverzeichnis extrahieren, sofern die Detailseiten gleich aufgebaut sind.

[3] `https://lsf.verwaltung.uni-muenchen.de`

Literaturverzeichnis

[CNOO] Bopp, Stephan (2009): *canoo.net Deutsche Wörterbücher und Grammatik*. Webseite. Universität Basel, Vrije Universiteit Amsterdam, IDSIA Lugano, Canoo Engineering AG. Basel, Schweiz. `http://www.canoo.net/`

[DUE07] Dürscheid, Christa (2007): *Syntax - Grundlagen und Theorien*, 4. Auflage. Vandenhoeck & Ruprecht GmbH & Co. KG, Göttingen

[DUG06] Duden (2006): *Duden - Die Grammatik*, 7. Auflage. Bibliographisches Institut & F. A. Brockhaus AG, Mannheim.

[DWP81] Dowty, David, Wall, Robert & Peters, Stanley (1992): *Introduction to Montague semantics*. Kluwer Academic Publishers, Dordrecht, Die Niederlande.

[FB65] Fodor, J. A. & Bever, T. G. (1965): *The Psychological Reality of Linguistic Segments*. Journal of Verbal Learning and Verbal Behavior 4, 414-420.

[GWT23] Google Inc. (2011): *Google Web Toolkit - Developer's Guide*. Dokumentation. Google Inc.
`http://code.google.com/intl/de/webtoolkit/doc/2.3/DevGuide.html`

[IRO07] Iroaie, Ana (2007): *Die Ergänzungsklassen des Nomens in Theorie und Praxis*. Zeitschrift der Germanisten Rumäniens 1-2 (29-30) 1-2 (31-32), 560-577.

[ISO13211-1] ISO/IEC JTC 1/SC 22/WG 17 (1995): *ISO/IEC 13211-1:1995, Information technology - Programming languages - Prolog - Part 1: General core*. International Organization for Standardization.

[ISO13211-2] ISO/IEC JTC 1/SC 22/WG 17 (2000): *ISO/IEC 13211-2:2000, Information technology - Programming languages - Prolog - Part 2: Modules*. International Organization for Standardization.

[ISO13211-3] ISO/IEC JTC 1/SC 22/WG 17 (2006): *ISO/IEC DTR 13211-3, Information technology - Programming languages - Prolog - Part 3: Definite clause grammar rules*. International Organization for Standardization.

[JST] Mordani et al (2006): *JSR-000154 Java Servlet 2.5 Specification*. Apache Software Foundation, Sun Microsystems Inc.
`http://jcp.org/aboutJava/communityprocess/mrel/jsr154/index.html`

[LES05] Leiß, Hans (2006): *Computerlinguistik II.* Vorlesungsfolien. Ludwig-Maximilians-Universität München, Centrum für Informations- und Sprachverarbeitung.
http://www.cis.uni-muenchen.de/~leiss/computerlinguistik-II-05-06/

[LIV08] Arrington, Michael (2008): *Interview With Barney Pell and Ramez Naam About Microsoft's Powerset Acquisition: Integration By End Of Year.* Webseite. TechCrunch.
http://www.techcrunch.com/2008/07/02/interview-with-barney-pell-and--ramez-naam-about-microsoft's-powerset-acquisition-integration-to--begin-this-year/

[LUCQ] Apache Software Foundation (2011): *Apache Lucene - Query Parser Syntax.* Dokumentation. Apache Software Foundation.
http://lucene.apache.org/java/3_1_0/queryparsersyntax.html

[LUC10] McCandless, Michael et al (2010): *Lucene in Action*, Second Edition. Manning Publications Co., Stamford.

[MON73] Montague, Richard (1973): *The Proper Treatment of Quantification in Ordinary English.* In *Approaches to Natural Language.* Jaako Hintikka, Julius Moravcsik and Patrick Suppes (Hrsg). Dordrecht, Die Niederlande, 221-242.

[OPN11] Clinton, DeWitt: *OpenSearch 1.1 (Draft 4).* A9.com Inc., Amazon.com Inc.
http://www.opensearch.org/Specifications/OpenSearch/1.1/Draft_4/

[OPS11] Clinton, DeWitt: *OpenSearch Suggestions extension 1.1 (Draft 1).* A9.com Inc., Amazon.com Inc.
http://www.opensearch.org/Specifications/OpenSearch/Extensions/Suggestions/1.1/Draft_1/

[RON01] Ronthaler, Marc (2001): *Dialogschnittstellen an Online-Informationssystemen.* Dissertation. Universität Osnabrück, Institut für Informationsverarbeitung.

[SCH08] Schwarz, Monika (2008): *Einführung in die kognitive Linguistik*, 3. Auflage. A. Francke Verlag Tübingen und Basel

[SCO09] comScore (2009): *comScore Releases February 2009 U.S. Search Engine Rankings.* Pressemitteilung. comScore, Inc.
http://www.comscore.com/press/release.asp?press=2750

[TEU79] Teubert, Wolfgang (1979): *Valenz des Substantivs. Attributive Ergänzungen und Angaben.* Pädagogischer Verlag Schwann, Düsseldorf. (Sprache der Gegenwart 49)

[XML10] Bray et al (2008): *Extensible Markup Language (XML) 1.0 (Fifth Edition).* W3C Recommendation. Textuality, Netscape, Microsoft, Sun Microsystems Inc., World Wide Web Consortium.
http://www.w3.org/TR/2008/REC-xml-20081126/