
Bachelorarbeit

im Studiengang Computerlinguistik

an der Ludwig- Maximilians- Universität München

Fakultät für Sprach- und Literaturwissenschaften

# Wittgenstein und Technik – Semantische Suche im Nachlass von Ludwig Wittgenstein

vorgelegt von
David Christian Ewald

| | |
|---|---|
| Betreuer: | M.A. Samuel Douglas Pedziwiatr |
| Prüfer: | Dr. Maximilian Hadersbeck |
| Bearbeitungszeitraum: | 19. März - 28. Mai 2018 |

## Danksagung

Zuerst möchte ich meinem Betreuer, M.A. Samuel Pedziwiatr, danken für viele hilfreiche Gespräche, Denkanstöße in die richtige Richtung und seine Unterstützung bei der Entwicklung von relevanten Ergebnissen für seine Arbeit. Diese Arbeit wäre ohne die angenehme und fruchtbare Zusammenarbeit nicht entstanden. Außerdem bin ich für den guten Kaffee zu Dank verpflichtet.

Herrn Doktor Maximilian Hadersbeck, unter dessen Aufsicht diese Arbeit verfasst wurde, danke ich für seinen Enthusiasmus für die gestellte Aufgabe, seine stete Verfügbarkeit bei Fragen und seine immer positive Einstellung. Er gab das Vorgehen und den Ansatz vor, der diese Arbeit in ihrer abschließenden Form entstehen ließ.

Ich danke meiner Familie, besonders meinen Eltern Petra und Christian, für ihre ständige und bedingungslose Unterstützung, ihr Verständnis und ihren Rückhalt, sowie für konstant hervorragenden Speis und Trank.

Ich danke Fabian Faehrmann, Harti Haslinger, Dominik Dörfler, Christian Mally und Katja Bertholdt für die richtige Ablenkung zur richtigen Zeit.

Schlussendlich möchte ich Ela Bauer danken, für ihren Beistand, ihre Zuversicht und ihre Zuneigung in jedem Augenblick.

**Selbstständigkeitserklärung**

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig angefertigt, alle Zitate als solche kenntlich gemacht sowie alle benutzten Quellen und Hilfsmittel angegeben habe.

München, den 28.05.2018

. . . . . .
David Christian Ewald

# Abstract

Diese Arbeit präsentiert eine semantische Suche im Nachlass von Ludwig Wittgenstein zum Thema ,,Wittgenstein und Technik“. Dabei ist die Aufgabe, eine semantische Suche zu diesem Gebiet zu entwickeln und mit einer Möglichkeit der Disambiguierung zu verbinden, die auf der grammatikalischen Satzstruktur aufbaut und davon ausgehend Bedeutung bewerten soll. Sie wurde am Centrum für Informations- und Sprachverarbeitung an der Ludwig-Maximilians-Universität (LMU) verfasst. Zuerst wird ein Programm entwickelt, das getaggte Sätze aus dem Nachlass extrahieren kann und diese dann einzeln parst. Aus den geparsten Sätzen werden mittels einer Suchfunktion Sätze mit Suchwörtern herausgesucht, die dann mit der jeweiligen Satznummer als Ausgabe zurückgegeben werden; diese Funktionalität ist im zweiten Programm implementiert. Mit der Satznummer und einem Suchwort lässt sich dann ein Programm aufrufen, das diesen Satz ausgibt, die syntaktischen Relationen des Suchworts zurückliefert und den Dependenzbaum zugänglich macht. Am Anfang der Arbeit steht eine Einführung und ein Überblick über relevante Werke wird gegeben. Dann wird in die verwendeten Daten und das Dependenzparsing im Allgemeinen, basierend auf der Dependenzgrammatik, eingeführt. Die Programme, die für diese Aufgabenstellung entwickelt wurden, werden vorgestellt, dann wird der Ansatz anhand einiger Beispielsätze evaluiert. Diese Evaluation stellt auch die Frage nach der Güte der Analyse und der Nutzbarkeit dieser Herangehensweise. Im Fazit wird die Arbeit zusammengefasst und ein Ausblick für weiterführende Beschäftigung mit dem Thema wird gegeben.

This thesis presents a semantic search in the ,,Nachlass“, meaning both published and unpublished works, of Ludwig Wittgenstein, for the field of ,,Technology“ (,,Technik“). The task is to develop a semantic search concerning this field of concepts and to connect it with a possibility for disambiguation based on the syntactic structure of a sentence, with the goal of judging its semantics and meaning. It has been developed at the ,,Centrum für Informations- und Sprachverarbeitung“ (Center for Information and Language Processing) at the Ludwig-Maximilians-Universität. At the outset, a program is written extracting tagged sentences from Wittgenstein's Nachlass and parsing each one independently. Following that, a search function taking words as input returns sentences containing these words and their unique sentence number and prints them out, this being the second program. Using the sentence number and a search word as input, a third program can be run, which returns the sentence, the syntactic relations of the word given as input and a visualization of a dependency tree. The thesis begins by giving an introduction and an overview of works relevant for the task. After that, a chapter regarding the resources and data being used and an introduction to dependency grammar and dependency parsing follow. The programs developed for this thesis are being presented, then an evaluation of several example sentences and of parsing quality is included. This work ends with a conclusion and an outlook.

# Inhaltsverzeichnis

# 1 Einleitung

Die Computerlinguistik ist ein vergleichsweise junges Forschungsgebiet, und dazu ein recht spezifisches, das sich dadurch auszeichnet, dass es unterschiedliche Disziplinen in gleichem Maß mit einbezieht und sich über diese definiert. Zu diesen Gebieten gehören die Informatik, die Mathematik und die Linguistik, die in dieser Kombination die maschinelle Verarbeitung von Sprache ermöglichen. Neben den theoretischen Grundlagen ist die Anwendung auf unterschiedlichen Gebieten ein hochspannendes und aktuelles Thema, da im Zuge der Digitalisierung der Welt die Interaktion von Mensch und Maschine und die Verbindung von natürlicher Sprache und Computern immer häufiger, genauer und umfassender wird. Aus diesen vielfältigen Möglichkeiten zur Anwendung und der immer weiter wachsenden Relevanz ergibt sich als ein folgerichtiger nächster Schritt die Verbindung der Computerlinguistik mit verwandten Fächern. Dabei sind hervorzuheben die ,,Digital Humanities", also die Verbindung von computerlinguistischer, statistischer und informatischer Arbeit mit den Grundlagen der Geisteswissenschaften. Diese Verbindung ist gewinnbringend, weil die ,,Digitalen Geisteswissenschaften", die auf jahrtausendealte Traditionen zurückgreifen können, damit ganz neue Werkzeuge und Methoden zum Erkenntnisgewinn zur Verfügung haben. Ein für die Geisteswissenschaften zentrales Feld, das sich durch seine Verbindung zu vielen anderen wissenschaftlichen Gebieten auszeichnet, war immer schon die Philosophie. Auch diese Wissenschaft kann von der Entwicklung der ,,Digital Humanities" profitieren, unter anderem dadurch, dass philosophische Werke und Werkzeuge durch neue Techniken zugänglich gemacht werden können. In diesem Sinne entstanden die *Wittgenstein Archives at the University of Bergen* (WAB), die auf die Wittgenstein-Forschung ausgerichtet sind. Das Centrum für Informations- und Sprachtechnologie der Ludwig-Maximilians-Universität (LMU) kooperiert mit den WAB, insbesondere im Projekt *WAST*, den *Wittgenstein Advanced Search Tools*. Im Zuge dieses Projekts entstand diese Bachelorarbeit. Relevant ist die Forschung über Ludwig Wittgenstein als eine Erkundung und Auslegung eines der größten Philosophen des vergangenen Jahrhunderts und als fruchtbare Zusammenarbeit mit der Sparte der Philosophie, besonders auch an der LMU: diese Arbeit wurde in enger Zusammenarbeit mit einem Doktoranden der Philosophie, M.A. Samuel Douglas Pedziwiatr, verfasst. Daher rührt der Fokus auf der ,,Technik" als zu bearbeitendem Gebiet, da dieses Gebiet als Begriffsklasse im Mittelpunkt der systematischen Arbeit der Dissertation von Samuel Pedziwiatr steht, die den Titel ,,Sprache, Handlung und Technik. Technische Motive in Wittgensteins Nachlass" trägt.
Das Thema dieser Arbeit ist ,,Wittgenstein und Technik", als eine semantische Suche im Nachlass Wittgensteins nach diesem Thema der Technik. Dabei liegt der Augenmerk auf einer Disambiguierung auf der Basis von syntaktischer Analyse, im Besonderen durch eine Untersuchung der Dependenzstruktur von Sätzen. Es sollen also Wörter gefunden werden, die in ihrer Semantik dem Gebiet der Technik angehören, und davon ausgehend die Sätze gesucht, in denen die Wörter vorkommen, um diese dann zu disambiguieren. Dieser Ansatz liegt im Spannungsfeld zwischen einer rein philosophischen Interpretation und einer statistischen Verarbeitung von Wittgenstein-Text. Er bezieht seinen Wert daraus, dass er auf der einen Seite den interpretatorischen Ansatz der Philosophie durch linguistische Analyse vertiefen kann und so über die Struktur auch Einblicke in die Semantik und Bedeutung zulässt. Auf der anderen Seite bildet er eine Grundlage für weiterführende, rein statistische Arbeit, die über die Kontexte von Wörtern eine Disambiguierung ermöglicht. Dabei ist von Vorteil, dass eben durch die erstellte Dependenzstruktur die Unterscheidung nicht nur auf anderen Wörtern basiert, sondern zusätzlich auf den Relationen zwischen den Wörtern, die den Kontext verfeinern und so eine exaktere Disambiguierung möglich

machen.
Als Basis dient die Arbeit über Ludwig Wittgenstein, der in der philosophischen Literatur bis heute viel Beachtung erfährt und dessen Stellenwert unumstritten ist, und das Dependenzparsing: also die automatische Konstruktion einer Dependenzstruktur zu einem Satz. Dies ist in der Computerlinguistik ein wichtiges Teilgebiet, zu dem viel Forschungsarbeit betrieben wurde und wird. Mehrere Ansätze und Modelle liefern bereits heute sehr gute und belastbare Ergebnisse, und ein Ende des Fortschritts ist nicht in Sicht. So wird die automatische Analyse immer exakter und robuster, und die Entwicklung von noch besseren Parsern hält an. Für diese Arbeit wird als Grundlage der digital zugängliche Nachlass von Ludwig Wittgenstein verwendet. Daraus werden Sätze mit Wörtern und linguistischen Annotationen extrahiert, die dann mit einem Dependenzparser geparst werden. Aus dem so geparsten Text können über eine Suchfunktion einzelne Wörter gesucht werden, dadurch Sätze mit diesen Wörtern gefunden und schließlich der geparste Satz inklusive des Dependenzbaums ausgegeben. Dabei dienen als methodische Werkzeuge vor allem der Wittgenstein-Text selbst, ein Tagger für diesen Text, und der verwendete Dependenzparser. Beispiele für Sätze, bei denen ein bestimmtes Wort zum Feld der Technik disambiguiert werden kann, finden sich bei Wittgenstein zur Genüge, anschaulich sind etwa das Wort *Prozess* im Satz ,,Während das Ergebnis nur das des physikalischen Prozesses ist.“ oder *Technik* in dem Satzteil ,,& alles auf die Technik einer Verwendung ankommt.“
Das erste Kapitel dieser Arbeit stellt die Einleitung dar. Das folgende Kapitel gibt einen Überblick über relevante Werke zu Ludwig Wittgenstein und dem Dependenzparsing, woran ein Kapitel über die verwendeten Daten anschließt, wie etwa den Nachlasstext oder die Liste mit Suchwörtern. Im vierten Kapitel wird in Dependenzgrammatik und Dependenzparsing eingeführt, das darauffolgende Kapitel dokumentiert den Versuch mit einem anderen Parser. Im Kapitel Sechs wird die Umsetzung der Aufgabe in Programmen geschildert, danach folgt ein Kapitel, in dem Ergebnisse evaluiert werden, über Beispielsätze aus einem Wittgenstein-Text. Im achten und letzten Kapitel wird ein Fazit gezogen.

# 2 Relevante Arbeiten

Wie in einer wissenschaftlichen Arbeit üblich, soll ein kurzer Überblick über für diese Arbeit relevante Werke gegeben werden, die zum Teil zitiert werden oder anderweitig eine bedeutende Grundlage darstellen. Grundsätzlich lassen sich diese relevanten Arbeiten hier in zwei Gruppen einteilen; zum einen die geisteswissenschaftlichen Hintergründe zu Ludwig Wittgenstein, zum anderen die computerlinguistische Basis, auf welcher aufbauend diese Arbeit entsteht. Einen Überblick über Wittgensteins Leben und Werk bietet die Stanford Encyclopedia of Philosophy, unter Biletzki and Matar [2014]. Der britische Philosoph Ray Monk hat unter anderem über Wittgenstein geschrieben, seine Arbeit umfasst Einführungen zu seinem Leben und Wirken und Betrachtungen Wittgensteins Philosophie betreffend. Zwei nennenswerte Veröffentlichungen sind ,,How to read Wittgenstein“ [Monk, 2005] und ,,The Duty of Genius“ [Monk, 1990]. Des weiteren hat Joachim Schulte ein Buch veröffentlicht, als ,,Eine Einführung“ gedacht [Schulte, 2016]. Von Wilhelm Vossenkuhl stammen ein Buch über ,,Ludwig Wittgenstein“ [Vossenkuhl, 2003] sowie eines über den ,,Tractatus Logico-Philosophicus“ [Vossenkuhl, 2001]. Von Wittgenstein selbst ist anzuführen sein Hauptwerk, eben dieser *Tractatus* in einer Ausgabe von 2001: [Wittgenstein, 2001]. Ein weiteres relevantes Werk des Philosophen stellen die ,,Philosophischen Untersuchungen“ dar [Wittgenstein, 2003], herausgegeben von Joachim Schulte. Für den computerlinguistischen Teil dieser Arbeit sind von Bedeutung zuerst ein Grundlagenwerk über Dependenzparsing, ,,Dependency parsing“, von Kübler et al. [2009]. Auch mehrere Veröffentlichungen von Joakim Nivre (und anderen) zum Dependenzparsing im Allgemeinen sowie dem Malt-Parser sind relevant: ,,Maltparser: A data-driven parser-generator for dependency parsing“ [Nivre et al., 2006], ,,Inductive Dependency Parsing“ [Nivre], ,,Pseudo-projective dependency parsing“ [Nivre and Nilsson, 2005], und ,,MaltParser: A language-independent system for data-driven dependency parsing“ [Nivre et al., 2007]. Außerdem zu erwähnen sind ,,Incrementality in Deterministic Dependency Parsing“ [Nivre, 2004] und ,,An Efficient Algorithm for Projective Dependency Parsing“ [Nivre, 2003]. Für den Aufbau der Bibliothek SpaCy gibt es einige wichtige Bezugswerke, darunter ,,A Dynamic Oracle for Arc-Eager Dependency Parsing“ [Goldberg and Nivre, 2012], ,,Simple and Accurate Dependency Parsing Using Bidirectional LSTM Feature Representations“ [Kiperwasser and Goldberg, 2016], ,,Stack-propagation: Improved Representation Learning for Syntax“ [Zhang and Weiss, 2016], ,,Deep multi-task learning with low level tasks supervised at lower layers“ [Søgaard and Goldberg, 2016], ,,An Improved Non-monotonic Transition System for Dependency Parsing“ [Honnibal and Johnson, 2015] und ,,A Fast and Accurate Dependency Parser using Neural Networks“ [Chen and Manning, 2014]. Eine Grundlage für Natural Language Processing mit Python bietet ,,Natural Language Processing with Python“ von Bird et al. [2009] an.

# 3 Wittgenstein-Daten

Dieses Kapitel bietet eine Einführung zu Ludwig Wittgenstein, dem Projekt WAST am CIS, den verwendeten Daten und der Wortliste für die semantische Suche.

## 3.1 Wittgenstein und der Nachlass

Ein kurzer Überblick über Ludwig Wittgensteins Leben und Werk soll hier gegeben werden, übernommen und zusammengefasst nach Biletzki and Matar [2014]. Ludwig Wittgenstein wurde am 26.4.1889 in Wien als Sohn einer Industriellenfamilie geboren. 1908 begann er an der Manchester University ein Studium der Luft- und Raumfahrttechnik bzw. des ,,Aeronautical Engineering“, wo er aufgrund seines Interesses an der Philosophie der Mathematik Kontakt zu Frege aufnahm. Auf dessen Rat hin ging er 1911 nach Cambridge, um mit Bertrand Russell studieren zu können. Mit Russell, der sich von seinem philosophischen Vermögen überzeugen ließ, führte er Gespräche über Logik und Philosophie, die beiden sollte dann eine enge Beziehung verbinden. Nach seinen Jahren in Cambridge kehrte er 1913 nach Österreich zurück und schloss sich im darauffolgenden Jahr dem österreichischen Heer an. Während des ersten Weltkriegs verfasste er Notizen und Entwürfe zu seinem ersten Hauptwerk, dem *Tractatus Logico-Philosophicus*. Nach dem Krieg wurde das Buch auf Deutsch veröffentlicht und ins Englische übertragen. 1920 betrachtete Wittgenstein seine Beschäftigung mit der Philosophie als abgeschlossen und ging in den darauffolgenden Jahren verschiedenen Tätigkeiten in und um Wien nach. Erst 1929 siedelte er wieder nach Cambridge um, um seine philosophische Tätigkeit wieder aufzunehmen, als Folge von Diskussionen mit Mitgliedern des *Wiener Kreises* über die Philosophie der Mathematik und der Naturwissenschaften. In diesen Jahren in Cambridge änderte sich seine Auffassung der Philosophie und ihrer Probleme grundlegend. Aufzeichnungen und Werke aus dieser Zeit beinhalten etwa das *Blue Book* und das *Brown Book* sowie die *Philosophische Grammatik*. In den 1930er und 1940er Jahren hielt er Seminare in Cammbridge ab und entwickelte die meisten Ideen, die er in seinem zweiten Buch, den *Philosophischen Untersuchungen*, veröffentlichen wollte. Thematisch wandte sich Wittgenstein hier der gewöhnlichen Sprache, Betrachtungen der Psychologie sowie auch einer Philosophieskepsis zu. Er entschied allerdings, dass dieses Buch erst posthum veröffentlich werden sollte. In den Folgejahren führte er seine philosophische Arbeit fort, unternahm Reisen, und kehrte schließlich nach Cambridge zurück, wo Krebs bei ihm diagnostiziert wurde. Ludwig Wittgenstein starb 1951 und gilt heute gemeinhin als einer der größten und einflussreichsten Philosophen des 20. Jahrhunderts.

Das Centrum für Informations- und Sprachverarbeitung der Ludwig-Maximilians-Universität München arbeitet in Kooperation mit den *Wittgenstein Archives at the University of Bergen* (WAB) am Projekt WAST, den *Wittgenstein Advanced Search Tools*. Das Ziel des Projekts ist die computerlinguistische Arbeit am Nachlass Wittgensteins, im besonderen, den Nachlass unterschiedlichen linguistischen Analysen zu unterziehen und zugänglich und durchsuchbar zu machen. Im Rahmen dieses Projekts entsteht auch diese Bachelorarbeit als eine Möglichkeit, den Nachlass zu untersuchen und auf geisteswissenschaftliche Erkenntnisse hinzuführen bzw. diese zu ermöglichen. Der Nachlass wird nach durch die WAB zur Verfügung gestellten Editionen verwendet, Zugang unter Pichler [2018].

Der Text, der für die Programme verwendet wird und aus dem die Ergebnisse zur Evaluation stammen, ist das Typoskript 213. Zum Text ist anzumerken, dass in der vorliegenden

Version Sätze wiederholt vorkommen, die das gleiche Siglum aufweisen, aber sich in einem oder mehreren Wörtern unterscheiden. Außerdem sind nicht alle Wörter jedes Satzes frei von Annotationen, in der getaggen XML-Version etwa sind einzelne Bindestriche Teil von Wörtern, beispielsweise im Satz mit der Nummer 3464: statt *Körper* ist das Wort *Kör<lb rend=ßhyphen>per*. Diese Tatsachen müssen bei der Textarbeit unbedingt beachtet werden, um nicht zu falschen linguistischen Schlüssen zu gelangen. Des weiteren sind auch einzelne Sätze dabei, die viel eher Bemerkungen oder Gedankenfragmente als vollständige, grammatische Sätze sind, was eine richtige und gute syntaktische Untersuchung erheblich erschwert.

## 3.2 Die Wortliste für die Semantische Suche

Eine Grundlage für die Disambiguierungsarbeit auf einem Themengebiet sind die Daten, anhand derer die semantische Arbeit verrichtet werden kann. Für das Wittgenstein-Projekt wurde von M.A. Samuel Douglas Pedziwiatr, Doktorand der Philosophie an der LMU, eine Wortliste erarbeitet, deren Begriffe als Query (bzw. als Schlagwort-Suchbegriffe) die Sätze zurückliefern, bei denen eine Disambiguierung sinnvoll und notwendig ist. Die Wortliste gliedert sich in unterschiedliche Begriffsfelder, die als Ganzes das Themengebiet „Technik“ zu skizzieren vermögen. Enthalten sind zum ersten zum Technikbegriff verwandte Begriffe sowie Komposita, wobei bei den verwandten Konzepten angenommen wird, dass sie bei Wittgenstein eine philosophische Rolle im Umfeld von „Technik“ spielen. Weiterhin sind Personennamen aufgeführt, die mit der Entwicklung der Wissenschaften oder der Technik in Europa befasst waren, oder für die die Technik bei Wittgenstein relevant war. Darauf folgen einzelne Gebiete und Fachdisziplinen. Der nächste Punkt sind Maschinenteile und technische Artefakte und Instrumente, mit motiviert durch Wittgensteins eigenen Fokus auf Technik bzw. seine Biographie. Es folgen Abstrakta sowie Sprache über Technik, beispielsweise mit technischen Prozessen, sowie sonstige Begriffe, die figurativ verwendet werden oder sich nicht exakt zuordnen lassen.

Eine Auswahl der Wortliste befindet sich im Anhang.

# 4 Dependenzgrammatik und Dependenzparsing

Das erste Kapitel soll eine Einführung in die Dependenzgrammatik und das Dependenzparsing darstellen.

## 4.1 Dependenzgrammatik

Die Idee der Dependenzgrammatik als ein Element der Linguistik kann auf eine lange Tradition zurückblicken, wie Kübler et al. [2009] schreiben, sie ,,is rooted in a long tradition, possibly going all the way back to Pāṇini's grammar of Sanskrit", und hat auch in neuerer Zeit prominente Vertreter, von denen ,,a number of different dependency grammar frameworks have been proposed, of which the most well-known are probably the Prague School's Functional Generative Description, Mel'čuk's Meaning-Text Theory, and Hudson's Word Grammar." [Kübler et al., 2009] Allerdings der ,,starting point of the modern theoretical tradition of dependency grammar is usually taken to be the work of the French linguist Lucien Tesnière, published posthumously in the late 1950s." [Kübler et al., 2009]

Eine Dependenzgrammatik ist nach Kübler et al. [2009] eine syntaktische Analyse, aufbauend auf der ,,idea that syntactic structure essentially consists of *words* linked by binary, asymmetrical relations called *dependency relations* (or *dependencies* for short)." Das heißt also, dass syntaktische Struktur im Allgemeinen von den Relationen zwischen einzelnen Wörtern festgelegt ist, und dass sich aus diesen Relationen gemeinsam mit den in ihnen enthaltenen Wörtern der Satz oder die Phrase als eine syntaktische Einheit ergibt. Die ,,dependency relation holds between a syntactically subordinate word, called the *dependent*, and another word on which it depends, called the *head*." Kübler et al. [2009] Hieraus erklärt sich die erwähnte Asymmetrie der syntaktischen Relationen: zwei Wörter stehen im einem Verhältnis, in dem eines das andere regiert, also den Kopf darstellt, und das andere von diesem Kopf bestimmt wird. Aus dieser Folge an Beziehungen oder Abhängigkeiten baut sich also der komplette Satz auf. Ein Beispiel ist abgebildet:

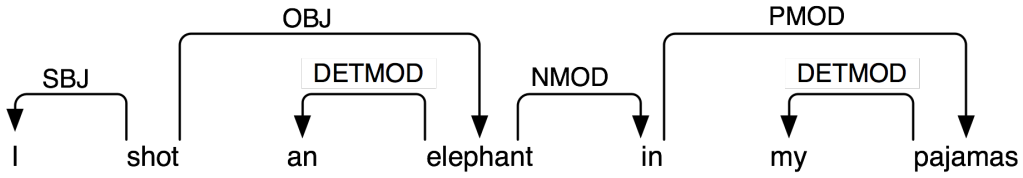


Abbildung 4.1: Ein Dependenzbaum nach [Bird et al., 2009].

Die Darstellung dieser Art der syntaktischen Analyse hat demnach die Eigenschaft, dass sie einen Graphen erzeugt bzw. mit einem Graphen gleichzusetzen ist, ,,where dependency relations are represented by arrows pointing from the head to the dependent.[...] Moreover, each arrow has a label, indicating the *dependency type*." [Kübler et al., 2009] Dies sind nun die Spezifizierungen der binären Relationen zwischen Wörtern (Knoten): Sie können als ein Pfeil betrachtet werden, dessen Fuß beim syntaktischen Kopf und dessen Spitze beim syntaktischen Dependenten liegt. Der Pfeil wird definiert durch sein Label, welches die Beziehung festhält, die zwischen den durch ihn verbunden Wörten besteht (Dependenzbögen). Im gezeigten Beispiel wäre ein solcher Pfeil etwa der von *shot* nach *I*, der mit SBJ beschriftet ist. SBJ steht für ,,Subject" oder ,,Subjekt", diese Relation bedeutet also,

dass das Subjekt *I* (der Handelnde, der die Tätigkeit ausführt) vom Prädikat *shot* (die Tätigkeit) in einer Subjektbeziehung (zum Prädikat) abhängt. Hier gilt es zu erwähnen, dass diese Richtung der Notation von Kübler et al. [2009] und Bird et al. [2009] so verwendet wird, genauso gibt es aber Varianten, bei denen der Pfeil genau umgekehrt festgelegt wird, seine Spitze dann also auf den Kopf weist.

Eine Besonderheit der hier vorgestellten Dependenzgrammatik stellt das ,,artificial word ROOT before the first word of the sentence“ [Kübler et al., 2009] dar. Dieses künstliche Wort wird aus praktischen Gründen eingeführt, denn es ,,simplifies both formal definitions and computational implementations“ [Kübler et al., 2009], wobei natürlich insbesondere die komputationelle Implementierung für die Verwendung im Dependenzparsing wichtig ist. Im gezeigten Beispiel ist kein ROOT-Wort enthalten, seine Sinnhaftigkeit und Verwendung erklären Kübler et al. [2009]: ,,In particular, we can normally assume that every real word of the sentence should have a syntactic head. Thus, instead of saying that the verb *had* lacks a syntactic head, we can say that it is a dependent of the artificial word ROOT.“ Das Wort *had* bezieht sich im Beispiel von Kübler et al. [2009] auf das Satzprädikat ohne einen syntaktischen Kopf, im gezeigten Graphen von Bird et al. [2009] wäre das fragliche Wort (das den künstlichen Kopf ROOT erhielte) demnach *shot*.

Eine relevante Unterscheidung ist diejenige von Dependenzgrammatik und Phrasenstruktur, da, wie Kübler et al. [2009] erläutern, die ,,information encoded in a dependency structure is different from the infor-mation captured in a *phrase structure* representation“, es handelt sich also um verschiedene Herangehensweisen an die Aufgabe der syntaktischen Analyse, wenngleich ,,many syntactic theories make use of hybrid representations“ [Kübler et al., 2009], es sind also keineswegs sich gänzlich ausschließende Untersuchungen von Sprache. Dependenzgrammatik und Phrasenstrukturgrammatik unterscheiden sich dadurch, dass ,,the dependency structure represents head-dependent relations between *words*, classified by *functional* categories such as subject (SBJ) and object (OBJ)“, wohingegen ,,the phrase structure represents the grouping of words into *phrases*, classified by *structural* categories such as noun phrase (NP) and verb phrase (VP).“ [Kübler et al., 2009]

Ausgehend davon, dass ,,dependency structure captures an essential element of natural language syntax, then we need some criteria for establishing dependency relations, and for distingishing the head and the dependent in these relations.“ [Kübler et al., 2009] Wenn die syntaktische Untersuchung von Sprache, hier im Besonderen eines oder mehrerer Sätze, so entscheidende oder essenzielle Einblicke in das Wesen der Sprache zulässt, so sind natürlich abgegrenzte und sinnvoll gewählte sowie logisch schlüssige Kriterien notwendig, um diese Untersuchung durchführen zu können. Nach Kübler et al. [2009] ist aufgeführt eine ,,list of some of the more common criteria that have been proposed for identifying a syntactic relation between a head *H* and a dependent *D* in a linguistic construction *C*:“

1. *H* determines the syntactic category of *C* and can often replace *C*.
2. *H* determines the semantic category of *C*; *D* gives semantic specification.
3. *H* is obligatory; *D* may be optional.
4. *H* selects *D* and determines whether *D* is obligatory or optional.
5. The form of *D* depends on *H* (agreement or government).
6. The linear position of *D* is specified with reference to *H*.

Kübler et al. [2009] merken an, dass der ,,term *construction* is used here in a non-technical sense to refer to any structural complex of linguistic expressions.“

Zu dieser Auflistung wird geschrieben, es sei ,,clear that this list contains a mix of different criteria“ und die Frage könne gestellt werden, ,,whether there is a single coherent notion of dependency corresponding to all the different criteria.“ [Kübler et al., 2009] Offenbar ist also eine abschließend gültige Definition von exakten Kriterien unmöglich,

Optionen zur Lösung dieses Problems sind die von nicht näher genannten Anderen postulierte ,,existence of several layers of dependency structure“ oder ,,the need to have different criteria for different kinds of syntactic constructions“ [Kübler et al., 2009].

Neben dieser formalen Unschärfe ist ein möglicherweise problematischer Punkt auch der der endozentrischen und exozentrischen Konstruktionen, wobei endozentrisch heißt, dass ,,the head can replace the whole without disrupting the syntactic structure“, wohingegen exozentrisch meint, dass ,,the head cannot readily replace the whole.“ [Kübler et al., 2009]. Es ist nicht bei allen Konstruktionen eindeutig, welcher Gruppe sie angehören, was eine letztgültige Analyse weiter erschwert. Diese Unterscheidung ist ,,related to the distinction between *head-complement* and *head-modifier* (or *head-adjunct*) relations“, da ,,head-complement relations are exocentric while head-modifier relations are endocentric“ [Kübler et al., 2009]. Nach Kübler et al. [2009] unterscheiden sich diese Relationen aufgrund der Valenz oder Wertigkeit, wobei die Valenz üblicherweise so aufgefasst wird, dass sie bestimmt, wieviele und welche Argumente ein syntaktischer Kopf annimmt und welche von diesen optional sind. Obwohl ,,most head-complement and head-modifier structures have a straightforward analysis“ [Kübler et al., 2009], ist bei anderen Strukturen die Einordnung schwieriger: dazu gehören ,,constructions that involve grammatical function words, [...], but also structures involving prepositional phrases.“ [Kübler et al., 2009] Ein weiteres Gebiet, über dessen Behandlung kein eindeutiger Konsens in der Dependenzgrammatik herrscht, sind Koordinationen, wie etwa im Satz ,,Sie sah den Baum und das Haus.“ (nach Kübler et al. [2009]).

## 4.2 Dependenzparsing

Ausgehend von dieser Begrifflichkeit der Dependenzgrammatik soll weiterführend auf das Dependenzparsing eingegangen werden. Dabei kann ein kurzer Überblick gegeben werden, eine genauere und formal explizite Behandlung würde den Rahmen dieser Arbeit sprengen. Nach Kübler et al. [2009] ist Dependenzparsing die ,,task of automatically analyzing the dependency structure of a given input sentence.“ Ohne zusätzliche Annahmen über die Arten der Dependenzen oder Relationen und ohne näher auf linguistische Besonderheiten des Satzes einzugehen (Kübler et al. [2009]), ist die Aufgabe also ,,to produce a labeled dependency structure [...], where the words of the sentence (including the artificial word ROOT) are connected by typed dependency relations.“ [Kübler et al., 2009] Ein Dependenzparser wird also verwendet, um einen Dependenzbaum oder Graphen automatisch zu erzeugen (wie etwa in Abbildung 1.1, mit zusätzlicher Einbindung des künstlichen Wortes ROOT), in dem alle Wörter durch Pfeile, die die Relationen darstellen, verbunden sind, und diese Pfeile mit der Art der Relation beschriftet sind. Spezifischer, ,,we can define the parsing problem as that of mapping an input sentence $S$, consisting of words $w_0 w_1 \ldots w_n$ (where $w_0 = \text{ROOT}$) to its dependency graph $G$.“ [Kübler et al., 2009] Formaler ausgedrückt, wird also eine Wortfolge auf einen Graphen abgebildet. Zwar existiert auch *unlabelled dependency parsing*, das die einzelnen Bögen nicht mit der Art der Relation versieht, auf diese Technik soll aber im Zusammenhang dieser Arbeit nicht näher eingegangen werden.

Die Heransgehensweisen, um dieses Problem zu lösen, ,,can be divided into two classes, which we will call *data-driven* and *grammar-based*, respectively.“ [Kübler et al., 2009] Eine Lösung ist ,,data-driven if it makes essential use of *machine learning* from linguistic data in order to parse new sentences“, während sie als *grammar-based* bezeichnet wird, ,,if it relies on a *formal grammar*, defining a formal language, so that it makes sense to ask whether a given input sentence is in the language defined by the grammar or not.“ [Kübler et al., 2009] Von Kübler et al. [2009] wird allerdings bemerkt, dass diese Methoden sich nicht gegenseitig ausschließen, ein Ansatz, der sowohl maschinelles Lernen wie auch eine definierte Formale Sprache verwendet, ist denkbar und möglich.

Diese beiden Klassen lassen sich jeweils weiter verfeinern. Beim datenbasierten (data-

driven) Ansatz sind insbesondere die ,,*supervised*“, also *überwachten*, Lernmethoden interessant, die davon ausgehen, dass die Sätze, die als Eingabe für das maschinelle Lernen verwendet werden, bereits mit der korrekten Annotation der Dependenzen versehen wurden (nach Kübler et al. [2009]). Bei dieser Art des Parsens ,,there are two different problems that need to be solved computationally.“ [Kübler et al., 2009] Diese beiden zu lösenden Probleme sind einmal ,,the *learning problem*, which is the task of learning a *parsing model* from a representative sample of sentences and their dependency structures.“ [Kübler et al., 2009] Das andere Problem ist das ,,*parsing problem*, which is the task of applying the learned model to the anaysis of a new sentence“ [Kübler et al., 2009], diese beiden Probleme sind also direkt verbunden: Zuerst gilt es, ein Modell aus gegebenen Daten zu lernen, danach ist die Aufgabe, eben dieses Modell auf unbekannte Daten anzuwenden. Diese beiden Probleme gelten genauso für die grammatikbasierten Lösungen, wobei hier die definierte formale Grammatik einen bedeutenden Bestandteil des zu lernenden Modells darstellt, der nur einen Teil aller möglichen Eingaben akzeptiert. Diese Grammatik kann regelbasiert sein oder ebenfalls maschinell aus linguistischen Daten erlernt, also kann auch dieser Ansatz teilweise datenbasiert sein. Bei der datenbasierten Herangehensweise sollen zwei Gruppen von Methoden erwähnt werden, ,,*transition-based* and *graph-based*“ [Kübler et al., 2009]. Der andere hauptsächliche Ansatz, der grammatikbasierte (*grammar-based*), lässt sich in die Methodengruppen ,,*context-free* and *constraint-based*, respectively“ [Kübler et al., 2009] unterteilen.

Der Begriff des *Deterministic Parsing* ist aufgrund der verbreiteten Anwendung relevant und bedeutet, dass ,,we always derive a single analysis for each input string.“ [Nivre], es wird also ein eindeutiger Dependenzbaum für einen Eingabesatz konstruiert. Genauer zu erläutern sind die

> systems that parameterize a model over the transitions of an abstract machine for deriving dependency trees, where we learn to predict the next transition given the input and the parse history, and where we predict new trees using a greedy, deterministic parsing algorithm – this is what we call transition-based parsing.

[Kübler et al., 2009] Das bedeutet also, dass ein Transitionssystem vorliegt, das Dependenzbäume ableitet, wobei die nächste Transition mittels verschiedener Parameter vorhergesagt wird, und die Parsebäume dann durch deterministisches Parsing erzeugt werden. Das ,,*transition system* is an abstract machine, consisting of a set of *configurations* (or *states*) and *transitions* between configurations.“ [Kübler et al., 2009] Das Transitionssystem selbst legt also die Übergänge zwischen Konfigurationen fest. Dabei sind Konfigurationen ,,triples consisting of a stack, an input buffer, and a set of dependency arcs.“ [Kübler et al., 2009] Der Gedanke ist, dass auf dem *Stack* die teilweise schon verarbeiteten Wörter liegen, im *Buffer* befinden sich die restlichen Wörter der Eingabe, und die Dependenzbögen sind ein zum Teil konstruierter Dependenzbaum (nach Kübler et al. [2009]). ,,Conceptually, a transition corresponds to a basic parsing action that adds an arc to the dependency tree or modifies the stack or buffer.“ [Kübler et al., 2009] Simpel ausgedrückt, ist transitionsbasiertes Parsing also ein Prozess, bei dem Wörter eines Satzes eingelesen werden und währenddessen schrittweise der Dependenzbaum aufgebaut wird (nach Kübler et al. [2009]). Da nicht jeder Schritt während des Parsingvorgangs eindeutig ist, wird speziell bei transitionsbasiertem Parsing auf einen *Guide* oder ein *Oracle* zurückgegriffen, das während des Trainings optimale Transitionssequenzen aus Dependenzbäumen des Goldstandards ableitet (die diesen produzieren) (nach Goldberg and Nivre [2012]). Diese ,,can then be used as training data for a classifier that approximates the oracle at parsing time in deterministic classifier-based parsing (Yamada and Matsumoto, 2003; Nivre et al., 2004)“ [Goldberg and Nivre, 2012], der Klassifikator legt also eine Transition an allen nichtdeterministischen Punkten (mit mehreren Optionen) während des Parsings fest. Auf diese Weise kann auch ein nichtdeterministischer Parser in einen deterministischen umgewandelt werden (nach Nivre).

## Zusammenfassung

Die Dependenzgrammatik, die auf eine lange linguistische Tradition zurückblicken kann, nimmt an, dass sich syntaktische Struktur als Verhältnis der Wörter eines Satzes zueinander auffassen lässt. Dabei werden Relationen eingeführt, die zwei Wörter in einem asymmetrischen Verhältnis verbinden, wobei ein Wort der Kopf ist und das andere als Kind von diesem Kopf abhängt. Es gibt verschiedene Typen von Relationen, die die Art der grammatikalischen Beziehung festhalten. Zwar herrscht Einigkeit über die meisten Formen von Relationen und die Analyse der meisten Beziehungen, doch gibt es genauso Fälle, in denen eine Untersuchung der Dependenzen keineswegs eindeutig und unanfechtbar ist. Auch eine allgemeingültige, abschließende Menge von Kriterien für die Bestimmung von Relationen lässt sich nicht widerspruchslos formulieren. Dennoch kann eine Dependenzgrammatik und daraus abgeleitet der syntaktische Aufbau eines Satzes als wertvolles Instrument für die linguistische Arbeit dienen. Das Dependenzparsing bezeichnet die Aufgabe, einen Dependenzbaum für einen Satz oder Korpus automatisch zu generieren. Das ist die Aufgabe eines dafür zu entwickelnden Dependenzparsers. Dabei kann ein Modell auf Daten trainiert werden (datenbasierter Ansatz) oder mit einer vordefinierten formalen Grammatik arbeiten (grammatikbasierter Ansatz). Auch ein hybrides Modell, das von beiden Ansätzen Gebrauch macht, ist möglich. Die unterschiedlichen Ansätze lassen sich noch weiter unterteilen. Das Gebiet des Dependenzparsings ist mit vielen unterschiedlichen Methoden und Architekturen für Parser ein sehr diverses, wobei auf dem heutigen Stand der Forschung bereits sehr gute Ergebnisse für unterschiedliche Sprachen erzielt werden können.

# 5 Versuch mit dem Malt-Parser

In diesem Kapitel soll ein Überblick über den Malt-Parser gegeben werden, mit dem die Bearbeitung der Aufgabe der Bachelorarbeit zunächst begonnen wurde, und es sollen die Probleme erläutert werden, aufgrund derer die Arbeit dann mit einem anderen Parser (vgl. 6) fortgesetzt und beendet wurde. Der Malt-Parser lässt sich beschreiben als ,,a data-driven parser-generator“ [Nivre et al., 2006]. Während ein ,,traditional parser-generator constructs a parser given a grammar, a data-driven parser-generator con-structs a parser given a treebank.“ [Nivre et al., 2006] Das bedeutet, dass der Malt-Parser auf Grundlage einer (annotierten) Baumbank als Eingabe arbeitet und daraus einen Parser-Generator erzeugt. Er ist eine ,,imple-mentation of *inductive dependency parsing*“ [Nivre et al., 2006], was heißt, dass ,,the syntactic analysis of a sentence amounts to the derivation of a dependency structure“ [Nivre et al., 2006], sowie ,,inductive machine learning is used to guide the parser at nondeter-ministic choice points“ [Nivre et al., 2006]. Die Methodologie lässt sich nach Nivre et al. [2006] genauer in drei Punkte unterteilen:

1. Deterministic parsing algorithms for building dependency graphs (Yamada and Matsumoto, 2003; Nivre, 2003)

2. History-based feature models for predicting the next parser action (Black et al., 1992; Magerman, 1995; Ratnaparkhi, 1997; Collins, 1999)

3. Discriminative machine learning to map histories to parser actions (Yamada and Matsumoto, 2003; Nivre et al., 2004)

Der erste Punkt heißt, dass deterministisches Dependenzparsing angewendet wird, um die Dependenzgraphen zu erstellen. Die nächsten beiden Punkte beziehen sich auf ein Feature-Modell, welches Informationen über lexikalische, Part-of-Speech und syntaktische Attribute von Token enthält und diese Features beim deterministischen Ableiten einer Dependenzstruktur verwendet (nach Nivre et al. [2006]), sowie maschinelles Lernen, das beim Parsen verwendet wird ,,to predict the next action at parsing time.“ [Nivre et al., 2006] Dabei wird also ein Klassifikator auf den Trainingsdaten trainiert. Der Malt-Parser kann dann in zwei Modi, *Learning mode* und *Parsing mode*, genutzt werden. Während ersterem werden ,,specifications of a parsing algo-rithm, a feature model and a learning algorithm“ festgelegt, die drei Komponenten haben alle mehrere wählbare Optionen, als Parsing-Algorithmus etwa *Nivre's algorithm* oder *Covington's algorithm* (Nivre et al. [2006]). Die Eingabe ist eine annotierte Baumbank. Im *Parsing mode* wird dann der während des Lernprozesses trainierte Klassifikator (mit den selben Spezifikationen wie im Lernmodus) angewandt auf ein Set an Eingabesätzen (nach Nivre et al. [2006]), das dann geparst wird. Der Malt-Parser wurde trainiert auf dem TIGER-Korpus (nach Brants et al. [2004], The TIGER Project [2003], Zugang unter TIGER Corpus, 2018), und das so erstellte Modell `deutsch` abgespeichert. Für die Anwendung des Parsers wurden Umgebungsvariablen spezifiziert, auf dem Betriebssystem Linux Ubuntu 16.04: mit `export $MALT_PARSER=HOME/maltparser-1.9.2/` wurde der Pfad für den Parser festgelegt, mit `export $MALT_MODEL=HOME/deutsch.mco` der Pfad für das auf dem deutschen Korpus trainerte Modell. In dem Integrated Development Environment PyCharm wurde ein Programm geschrieben, das einzelne Sätze mit Tags dem Malt-Parser, geladen mittels des Natural Language Toolkit für Python, als Eingabe übergibt und diese dann geparst werden sollen (nach Bird et al. [2009], unter NLTK, 2017). Es war möglich, als Ausgabe ein Objekt vom Typ `generator` zu erhalten, auf die einzelnen Analysen allerdings konnte

nie zugegriffen werden: der Versuch führte zu einem Programmabbruch mit einer Fehlermeldung `too many values to unpack.` bezogen auf die Liste mit einem Eingabesatz, der allerdings korrekt annotiert und abgespeichert (als Liste von Tupeln) war. Da keine Lösung für diesen Fehler gefunden bzw. eine korrekte Analyse dadurch unmöglich gemacht wurde, ist das Dependenzparsing mit der Bibliothek SpaCy fortgesetzt worden (vgl. Kapitel 6).

# 6 Implementierung

In diesem Kapitel wird die Implementierung der semantischen Suche behandelt, unter Erklärung der entwickelten Programme sowie der verwendeten Werkzeuge. Zusätzlich werden einige Hintergrundinformationen gegeben.

## 6.1 Python

Die Implementierung ist in der Programmiersprache Python geschrieben, die gut lesbar und nachvollziehbar ist und durch eine vergleichsweise unkomplizierte Syntax überzeugt. Python eignet sich gut für die Textverarbeitung und wird im Studium an der LMU vorrangig gelehrt. Die verwendete Version von Python ist 3.6.5, zugänglich und erhältlich unter ([Python, 2018]). Zum Zweck der Programmierung für diese Arbeit wird mit einem Integrated Development Environment, namentlich mit der PyCharm 2018.1.2 (Community Edition), gearbeitet, das für diese Programmiersprache spezialisiert ist. (Zugang unter [PyCharm, 2018]) Das IDE wird verwendet, da sich die Programmierung mit Python durch intelligente Inspektion von Code, automatische Vervollständigung, einfache Ausführung und schnelles Debugging sehr unkompliziert gestaltet. Von großem Nutzen für reibungsloses Verfassen von Programmcode ist außerdem die automatische Markierung von Fehlern. (nach PyCharm, 2018)

## 6.2 SpaCy

SpaCy ist eine speziell für Python entwickeltes Open-Source-Bibliothek, die Lösungen für Aufgaben des Natural Language Processing (NLP), also der Verarbeitung von natürlicher Sprache, anbietet. Es wird von der in Deutschland ansässigen *Explosion AI* betrieben und ist mittels Download frei verfügbar ([SpaCy, 2018]). SpaCy bietet mehrere unterschiedliche NLP-Funktionen, deren Modelle von Grund auf für diese Bibliothek entwickelt und implementiert wurden, wie etwa Named Entity Recognition, Part-of-Speech Tagging und Labelled Dependency Parsing. (nach SpaCy, 2018) Das Application Programming Interface (API) von SpaCy, also die unterschiedlichen Werkzeuge und Protokolle zum Entwickeln einer eigenen Anwendung, besteht unter anderem aus Containern sowie einer Pipeline zur Verarbeitung einer Eingabe als wichtigen Bestandteilen. Unter den Containern hervorzuheben sind der *Token*, in dem ein String bzw. eben ein Token zusammen mit Attributen, insbesondere Annotationen, gespeichert wird, sowie das *Doc*, das eine Sequenz von Token beinhaltet. Die *Processing Pipeline* kann eine Auswahl von Verarbeitungsschritten auf ein *Doc*-Objekt anwenden, wie etwa Part-of-Speech Tagging oder Labelled Dependency Parsing. (nach SpaCy API, 2018) Von Bedeutung für das Funktionieren der API sind des Weiteren die Klasse *Language*, die eine Pipeline zum Verabeiten von Text initialisiert und Daten zu Modell und Sprache enthält, sowie *Vocab*, wo das Vokabular und andere Daten einer einzelnen Sprache gespeichert werden. Diese beiden Klassen bzw. Objekte werden üblicherweise erzeugt, wenn ein Sprachmodell geladen wird. (nach SpaCy API, 2018) SpaCy stellt vortrainierte Sprachmodelle zur Verfügung, das vorliegende Programm arbeitet auf der Grundlage des Modells für Deutsch, *de_core_news_sm*, das ein Convolutional Neural Network „trained on the TIGER and WikiNER corpus“ [SpaCy Models, 2018] ist. (nach SpaCy Models, 2018)

Das Modell für Dependenzparsing ist Teil der verwendeten „Neural network model architecture“ [SpaCy API, 2018], also eines neuronalen Netzwerks, das die unterschiedlichen

Funktionen in sich vereint. Neuronale Netzwerke sind Architekturen für die Aufgabe des statistikbasierten Maschinenlernens. Der Dependenzparser ,,is still based on the work of Joakim Nivre [...], who introduced the transition-based framework [...], the arc-eager transition system, and the imitation learning objective.“ [SpaCy API, 2018] Das *arc-eager transition system* ist ein Beispiel für ,,deterministic methods for depen-dency parsing“ [Nivre, 2004], in dem ,,we need to process left-dependents bottom-up and right-dependents top-down. In this way, arcs will be added to the dependency graph as soon as the respective head and depen-dent are available, even if the dependent is not complete with respect to its own dependents.“ [Nivre, 2004] Ein weiterer Baustein ist nach Goldberg and Nivre [2012], dass ,,we introduce the concept of a dynamic parsing oracle“, das heißt, dass das Oracle nicht eine einzelne statische Sequenz an Transitionen definiert, sondern bei einzelnen Transitionen fragt, ob sie in einer bestimmten Konfiguration Gültigkeit besitzen, um den bestmöglichen Dependenzbaum zu produzieren (nach Goldberg and Nivre [2012]). Goldberg and Nivre [2012] wenden diese Entwicklung an auf das transitionsbasierte *Arc-Eager*-Dependenzparsing. Weiterhin ist eine von ,,two recent inspirations“ [SpaCy API, 2018] die Arbeit von Kiperwasser and Goldberg [2016], in dem ,,a simple and effective scheme for dependency parsing which is based on bidirectional-LSTMs“ präsentiert wird. Die BiLSTMs (ebenfalls ein Beispiel für neuronale Netzwerke) funktioneren so, dass jeder ,,sen-tence token is associated with a BiLSTM vec-tor representing the token in its sentential con-text“ [Kiperwasser and Goldberg, 2016], es wird also ein Vektor für jedes Wort, ausgehend vor allem von seinen linguistischen Eigenschaften, seiner Position und seinen syntaktischen Abhängigkeiten, erzeugt, dann ,,feature vectors are constructed by concatenating a few BiLSTM vectors.“ [Kiperwasser and Goldberg, 2016] Schließlich ,,the BiLSTM is trained with the rest of the parser in order to learn a good feature representation for the parsing problem.“ [Kiperwasser and Goldberg, 2016] Die Möglichkeit zum Parsen von Deutsch erweitert den Parser dahingehend, dass er ,,now also predicts non-projective structures, i.e., structures where arcs may cross.“ [Seeker, 2016] Diese Option wurde eingearbeitet auf Grund des ,,Pseudo-Projective Parsing“ nach Nivre and Nilsson [2005]. Das bedeutet, dass nicht-projektive Dependenzbögen im Parsingprozess ,,erhöht“ werden, also einen anderen Kopf erhalten, bis sie projektiv sind, der ursprüngliche Kopf wird aber im Label gespeichert. Die Ausgabe wird dann noch einmal durchlaufen und ordnet den ursprünglich nicht-projektiven Bögen wieder den Originalkopf zu (nach Nivre and Nilsson [2005]). Diese Skizzierung des Dependenzparsers von SpaCy ist weit entfernt von einer umfassenden und formal exakten Darstellung unter Einbeziehung sämtlicher einzelner Funktionsschritte und der theoretischen Grundlagen, da dies den Rahmen dieser Arbeit bei weitem überschreiten würde (vgl. 4.2)

## 6.3 Implementierung in Programmen

Dieser Abschnitt dokumentiert die genaue Umsetzung der Aufgabe der semantischen Suche zum Themengebiet ,,Technik“ als eine Abfolge von Programmen und erläutert Aufbau und Funktionsweise selbiger. Zu bemerken ist, dass diese Programme auf einem einzigen Text aus Wittgensteins Nachlass, dem *Typoskript 213*, entwickelt und getestet wurden. Die Aufgabe einer Erweiterung für alle Programme ist aber eine triviale, da sich über Systemargumente auch alle Texte der Reihe nach durchlaufen lassen, die für die weiteren Programme benötigten Daten können in einer anderen Datenstruktur, die mehrere Listen mit Texten enthält, serialisiert werden.

### 6.3.1 Vorverarbeitung und Parsen von Text

Dieses Programm, *parse_file.py*, ist der erste Verarbeitungsschritt einer Datei bzw. eines Textes des Wittgenstein-Nachlasses. Hier wird davon ausgegangen, dass die Datei ein Skript des Nachlasses ist, das mit dem TreeTagger von Helmut Schmid (beschrieben unter Schmid [1994] und Schmid [1995]) getaggt wurde und im XML-Format vorliegt.

Das Programm wird von der Kommandozeile aus aufgerufen und nimmt neben dem Programmnamen eine Eingabedatei und eine Ausgabedatei als Argumente, beispielsweise so: `python3 parse_file.py Ts-213_OA_NORM-expanded-tagged.xml aus.txt`

```
1 if len(sys.argv) != 3:
2     sys.stderr.write('\nError! \nVerwendung: python3
3 ____parse_file.py Eingabedatei Ausgabedatei \n\n')
4     exit()
```

Die Zeilen 2 und 3 stehen im eigentlichen Programm auf einer Zeile, die hier getrennt wurde, um die Lesbarkeit zu verbessern.

Hier wird die Anzahl der Systemargumente geprüft. Falls diese nicht mit der vorhergesehenen Verwendung übereinstimmt, wird eine Fehlermeldung ausgegeben.

Zuerst werden Python-Module geladen, die vom Programm benötigt werden.

```
1 import sys
2 import re
3 import spacy
4 from spacy.tokens import Doc
5 import pickle
```

`sys` wird benötigt, um das Programm von der Kommandozeile mit Argumenten aufrufen zu können. `re` stellt die regulären Ausdrücke bereit, die zum Suchen in Text verwendet werden. `spacy` importiert SpaCy, das den Dependenzparser zur Textanalyse bereitstellt. `Doc` aus `spacy.tokens` ist ein Container, in dem einzelne Sätze zusammenhängend zur Verarbeitung mit SpaCy aufbewahrt werden. `pickle` schließlich ermöglicht Serialisierung (vgl. 6.3.2), um von anderen Programmen benötigte Objekte speichern zu können.

Das Programm beginnt mit der Main-Funktion.

```
1 if __name__ == '__main__':
2     infile = str(sys.argv[1])
3     outfile = str(sys.argv[2])
```

Hier werden die Eingabe- und Ausgabedatei eingelesen und als String verarbeitet, um dann geöffnet werden zu können.

```
1 sentence_list = get_lines(infile)
```

Dann wird die Funktion `get_lines` aufgerufen, die die Eingabedatei zeilenweise einliest und Sätze erzeugt.

```
1 def get_lines(infile):
2     line_count = 0; lines = ['-']
3     with open(infile, 'r') as f:
4         for line in f:
5             line_count += 1; lines.append(line)
6     sentences = extract_sents(line_count, lines)
7     return sentences
```

Die Funktion erhält als Argument die Eingabedatei, über die dann zeilenweise iteriert wird. Die Variable `line_count` liest dabei die Anzahl der Zeilen ein, in der Liste `lines`

werden ab dem Listenindex 1 (äquivalent zur Zeilennummer) die einzelnen Zeilen gespeichert. Dann wird eine Satzliste mit der Funktion `extract_sents` erzeugt. Die so generierte einfache Liste mit allen Sätzen wird der Main-Funktion zurückgegeben und dort als `sentence_list` gesspeichert.

```
1  def extract_sents(linecount, lines):
2      sent = []; sentences = []; count = -1
3      sig = re.compile(r'<s (n="(.+?)"')
4  ____(ana="facs:(.+?) abnr:(\d+?) satznr:(\d+?)")>')
5      for x in range(0, linecount, 1):
6          count += 1
7          start = re.search(sig, lines[x])
8          if start:
9              sent.append(lines[x])
10             for y in range(count + 1, linecount, 1):
11                 l = re.search(sig, lines[y])
12                 if l:
13                     sentences.append(sent); sent = []
14                     break
15                 else:
16                     sent.append(lines[y])
17     return sentences
```

Die Zeilen 3 und 4 stehen im eigentlichen Programm auf einer Zeile, die hier getrennt wurde, um die Lesbarkeit zu verbessern.

Diese Funktion `extract_sents` bekommt die Zeilenanzahl und die Liste mit einzelnen Zeilen als Argumente. Es werden leere Listen und ein Count angelegt, um die einzelnen Sätze dann extrahieren und speichern zu können. Die Variable `sig` definiert einen regulären Ausdruck, mit dem nach den Satzsiglen gesucht wird. Danach wird mittels einer Integer-Variable über die Zeilen in der Liste `lines` iteriert und diese nach einem Siglum durchsucht. Wenn ein Siglum gefunden wird, so wird, wieder mittels eines Integers, über die Zeilen iteriert, die in der Liste nach dem aktuellen Eintrag (mit Siglum) folgen. Diese Iteration nun liest alle nachfolgenden Zeilen in die Liste `sent` (für den aktuellen Satz) ein – allerdings nur so lange, bis das nächste Siglum in einer Zeile entdeckt wird. Dann wird diese Zeile nicht mehr eingelesen, sondern die Liste `sent` mit dem gerade eingelesenen Satz wird an die Liste `sentences` mit allen Sätzen angehängt, dann wird diese Iteration abgebrochen: Das Programm kehrt zurück zur ersten Iteration und durchsucht zeilenweise nach Siglen, mit denen ein Satz beginnt. So werden also jede Zeile mit Siglum und alle nachfolgenden Zeilen ohne Siglum, die dann zusammen einen Satz bilden, als Liste in eine Liste mit allen Sätzen gespeichert. Diese Liste wird schließlich zurückgegeben.

```
1  text_list = make_sentences(sentence_list)
```

Die nächste Zeile der Main-Funktion ruft die Funktion `make_sentences` auf. Diese erhält die soeben erzeugte Satzliste und wandelt sie in eine neue Satzliste um, wobei aus dem XML-Format der Eingabedatei die reinen Texte der Wörter und Sätze extrahiert werden.

```
1  def make_sentences(sentence_list):
2      new_sentence_list = []; new_sentence = []
3      head = re.compile(r'<s (n="(.+?)"')
```

```
4  ␣␣␣␣(ana="facs:(.+?)␣abnr:(\d+?)␣satznr:(\d+?)")>')
5      part = re.compile(r'<w␣t="(.+?)"␣l="(.+?)">(.+?)</w>')
6      for sentence in sentence_list:
7          for word in sentence:
8              fhead = re.search(head, word)
9              fpart = re.search(part, word)
10             if fhead:
11                 new_sentence.append(fhead.group(3))
12             if fpart:
13                 new_sentence.append((fpart.group(1),
14                 fpart.group(2), fpart.group(3)))
15         new_sentence_list.append(new_sentence)
16         new_sentence = []
17     return new_sentence_list
```

Die Zeilen 3 und 4 sowie die Zeilen 13 und 14 stehen im eigentlichen Programm jeweils auf einer Zeile, die hier getrennt wurde, um die Lesbarkeit zu verbessern.

Die Funktion `make_sentences` bekommt die einfache Satzliste als Argument. Zuerst werden neue, leere Listen initialisiert, in denen dann der Text ohne die für diese Bearbeitung irrelevanten XML-Annotationen gespeichert werden kann. Als nächstes werden zwei reguläre Ausrücke definiert, die aus den gespeicherten Zeilen in den Sätzen den Satzanfang, also das Siglum (Zeilen 3 & 4), sowie die Wörter eines Satzes, annotiert mit Wortart-Tag und Lemma (Zeile 5), heraussuchen. Dann wird die Liste `sentence_list` mit allen Sätzen durchlaufen. Darin wird jede einzelne Liste, in der dann jeweils ein Satz gespeichert ist, durchlaufen, und jedes Element dieser Liste (also eine Zeile in XML) wird darauf durchsucht, ob ein Siglum oder ein tatsächliches Wort eines Satzes mit Annotationen enthalten ist. Bei einem Siglum wird die Annotation mit Skript, Absatznummer und Satznummer der neuen Liste `new_sentence` mit dem Satz hinzugefügt, bei einem Satzwort werden das Wortart-Tag, das Lemma und der Worttext selbst angehängt; das erste Element einer Satzliste ist also das Siglum. Andere Zeilen des Satzes (die dann nur XML-Spezifika beinhalten) werden nicht gespeichert. Nachdem der Satz vollständig durchlaufen wurde, wird diese neue Liste mit einem Satz hinzugefügt zu der neuen Liste `new_sentence_list` mit allen Sätzen. Diese neue Satzliste wird an die Main-Funktion zurückgegeben und in der Variable `text_list` gespeichert.

```
1  parsed_list = parse_list(text_list)
```

Im darauffolgenden Schritt wird von der Main-Funktion die erzeugte neue Satzliste an die Funktion `parse_list` übergeben, die eine Liste mit geparsten Sätzen generiert.

```
1  def parse_list(text_list):
2      parsed_list = []
3      parsed_text_list = []
4      for text in text_list:
5              parsed_text_list.append(text[0])
6              text.pop(0)
7              parsed_doc = get_parse(text)
8              parsed_text_list.append(parsed_doc)
9              parsed_list.append(parsed_text_list)
10             parsed_text_list = []
11     return parsed_list
```

Diese Funktion erhält die neue Liste mit Sätzen als Argument. Es werden leere Listen erzeugt, in die dann die geparsten Sätze gespeichert werden. Dann wird über jeden Satz in `text_list` iteriert: Das erste Element wird in der Liste für einen geparsten Satz, `parsed_text_list`, gespeichert, da es das Siglum enthält und nicht zum eigentlichen Satz gehört, der geparst wird. Dieses erste Element mit dem Siglum wird dann aus der Satzliste entfernt. Aus den verbleibenden Elementen, die den eigentlichen Satz bilden, wird von der Funktion `get_parse` ein geparstes Objekt `parsed_doc` vom Typ `Doc` mittels SpaCy erzeugt. In dieser Funktion wird also der eigentliche Parse durchgeführt. Der geparste Satz, den die Funktion zurückliefert, wird ebenfalls an `parsed_text_list` angehängt. Diese Liste (mit einem Siglum und geparsten Satz) wird dann an die Liste mit allen geparsten Sätzen, `parsed_list`, angehängt. Die Liste für einen geparsten Satz wird wieder geleert und dann in der Iteration der nächste Satz geparst und gespeichert. Zum Schluss wird die Liste mit allen geparsten Sätzen an die Main-Funktion zurückgegeben.

SpaCy wird folgendermaßen initialisiert:

```
1 nlp = spacy.load('de', disable=['tagger', 'ner'])
2 parser = nlp.parser
```

Die Klasse `Language` wird erzeugt und es wird eine Instanz `nlp` der Klasse kreiert. In die Klasse wird dann unter anderem das verwendete Modell für Deutsch geladen. Aus der Pipeline werden der Part-of-Speech Tagger und die Named Entity Recognition abgeschaltet, da diese nicht benötigt werden. Die Variable `parser` beinhaltet den Parser von SpaCy, enthalten in der Pipeline der Klasse bzw. der erzeugten Instanz.

Die Funktion `get_parse`:

```
1  def get_parse(text):
2      word_list = []; tag_list = []
3      for word in text:
4          word_list.append(word[2])
5          tag_list.append(word[0])
6      doc = Doc(nlp.vocab, words=word_list,
7      spaces=[True for x in range(len(text))])
8      for token in doc:
9          token.tag_ = tag_list[token.i]
10     parsed_doc = parser(doc)
11     for token in parsed_doc:
12         assert token.tag_ == tag_list[token.i]
13     return parsed_doc
```

Die Zeilen 6 und 7 stehen im eigentlichen Programm auf einer Zeile, die hier getrennt wurde, um die Lesbarkeit zu verbessern.

Diese Funktion erhält als Argument eine Liste mit einem Satz, mit Wörtern und Annotationen als Elementen, und gibt einen geparsten Satz zurück. Zuerst werden leere Listen erzeugt, in denen alle Wörter und Tags eines Satzes gespeichert werden. Dann werden die Satzelemente durchlaufen und aus jedem das Wort und der Tag zu der entsprechenden Liste hinzugefügt. Im nächsten Schritt wird ein `Doc`-Objekt erzeugt, in das der Satz gespeichert wird. Es wird erzeugt durch das Argument `nlp.vocab`, das das Objekt der zuvor erzeugten Instanz `nlp` zuordnet (in der Vokabular für Deutsch verwendet wird), das Argument `words`, das die Wortliste als die Wörter des Satzes festlegt, und `spaces`, das festlegt, dass die einzelnen Wörter im Satz durch Leerzeichen getrennt sind. Danach wird über die `Token`-Objekte in `doc` iteriert und jedem einzelnen Token (über den Index `i` des Tokens in `doc`) der Tag an entsprechender Stelle in der Tagliste zugewiesen. Dann wird der initialisierte Parser, als `parser`, angewendet und der geparste Satz im `Doc`-Objekt `parsed_doc` gespeichert. Zusätzlich wird noch einmal überprüft, ob die Tags im

`parsed_doc` mit denen in der Tagliste identisch sind, damit ein Fehler die Tags betreffend beim Parsen ausgeschlossen werden kann. Am Ende wird der geparste Satz zurückgegeben.

```
1     with open (outfile, 'w') as o:
2         for entry in parsed_list:
3             o.write('Siglum: '+str(entry[0])+'\n')
4             o.write('Satz: '+str(entry[1])+'\n')
5             for token in entry[1]:
6                 o.write('\nWort: '+str(token.text)+'\nRelation: '
7                 +str(token.dep_)+'\nKopf: '+str(token.head)+'\n')
8             o.write('\n\n')
```

Die Zeilen 6 und 7 stehen im eigentlichen Programm auf einer Zeile, die hier getrennt wurde, um die Lesbarkeit zu verbessern.

Dies ist der nächste Teil der Main-Funktion. Die anfangs eingelesene Ausgabedatei wird nun geöffnet und die Liste mit allen geparsten Sätzen wird satzweise durchlaufen. Für jeden Satz (der in einer Liste gespeichert ist) wird als erstes das Siglum, das erste Element in der Liste, ausgegeben. Danach wird der Satz (das zweite Listenelement) ausgegeben: Zuerst vollständig, und dann Wort für Wort, wobei jeweils das Wort selbst, seine Relation zum syntaktischen Kopf und der syntaktische Kopf in die Ausgabedatei geschrieben werden. Hierbei ist anzumerken, dass diese Informationen zur Syntax unvollständig sind und kaum zu einer manuellen syntaktischen Analyse dienen können. Allerdings geht es hier nur darum, die Möglichkeit des Parsens einer Datei zu illustrieren (und diese Datei als Text mit einigen Annotationen verfügbar zu machen), exaktere syntaktische Arbeit findet in den weiterführenden Programmen statt.

```
1     pickle.dump((parsed_list, text_list), open("save.p", "wb"))
```

Das ist die letzte Zeile der Main-Funktion und das Programmende. Hier wird die Liste mit einzelnen Satzlisten, die nur aus Wörtern und Annotationen bestehen (die Siglen wurden aus dieser Liste in `parse_list` entfernt), abgespeichert, zusammen mit der geparsten Satzliste, die aus einzelnen Listen mit jeweils dem Satzsiglum und dem geparsten Satz besteht. Dafür wird das Modul Pickle verwendet, welches Serialisierung für Python ermöglicht. Die beiden Listen `parsed_list` und `text_list` werden in die Datei *save.p* übertragen.

```
1  Siglum: ana="facs:Ts-213,Ir_abnr:4_satznr:7"
2  Satz: 3 ) Das Verstehen als Korrelat einer Erklärung .
3
4  Wort: 3
5  Relation: ROOT
6  Kopf: 3
7
8  Wort: )
9  Relation: ROOT
10 Kopf: )
11
12 Wort: Das
13 Relation: nk
14 Kopf: Verstehen
```

```
15
16 Wort: Verstehen
17 Relation: ROOT
18 Kopf: Verstehen
19
20 Wort: als
21 Relation: mnr
22 Kopf: Verstehen
23
24 Wort: Korrelat
25 Relation: nk
26 Kopf: als
27
28 Wort: einer
29 Relation: nk
30 Kopf: Erklärung
31
32 Wort: Erklärung
33 Relation: ag
34 Kopf: Korrelat
35
36 Wort: .
37 Relation: punct
38 Kopf: Verstehen
```

Hier ist ein beispielhafter Ausschnitt aus der Datei `aus.txt` dargestellt, diese Datei (die Ausgabedatei) wurde erzeugt mit dem Programmaufruf `python3 parse_file.py Ts-213_OA_NORM-expanded-tagged.xml aus.txt`. Zu sehen sind Siglum, Satz und die einzelnen Wörter mit rudimentären Annotationen.

Das ganze Programm findet sich im Anhang.

### 6.3.2 Serialisierung

Serialisierung bezeichnet die Umwandlung von Objekten oder von bestimmten Zuständen von Objekten in eine Form, die dann auf der Festplatte bzw. in einer Datei gespeichert werden können. Diese Form ist häufig ein Bytestrom, aus dem dann im dazu komplementären Prozess der Deserialisierung die urspünglichen Objekte wieder eingelesen werden (können). Ein spezielles Serialisierungsmodul für die Sprache Python heißt *Pickle* ([Pickle, 2018]), welches ,,implements binary protocols for serializing and de-serializing a Python object structure“ ([Pickle, 2018]). Die Serialisierung, genannt *Pickling*, ist hierbei der ,,process whereby a Python object hierarchy is converted into a byte stream“ [Pickle, 2018], also das Abspeichern von Objekten, um später wieder darauf zugreifen zu können. *Unpickling* heißt ,,the inverse operation, whereby a byte stream (from a binary file or bytes-like object) is converted back into an object hierarchy.“ [Pickle, 2018] Insgesamt kann also das Modul ,, transform a complex object into a byte stream and it can transform the byte stream into an object with the same internal structure.“ [Pickle, 2018] Das Datenformat, das von *Pickle* verwendet wird, ist Python-spezifisch. Das Modul kann dazu benutzt werden, die generierten Bytestöme in eine Datei zu schreiben und/oder sie in einer Datenbank zu speichern oder über ein Netzwerk zu senden (übernommen nach Pickle, 2018).

### 6.3.3 Suchfunktion im Nachlasstext

Dieses Programm *search_word.py* implementiert eine einfache, wortbasierte Suche, die auf dem ersten Programm *parse_file.py* aufbaut. Es wird davon ausgegangen, dass das erste

Programm bereits aufgerufen wurde und auf die gespeicherten Daten zugegriffen werden kann. Das Programm wird von der Kommandozeile aus mit zwei Argumenten aufgerufen werden, wobei das erste der Programmname, das zweite ein Suchwort ist, beispielsweise: `python3 search_word.py Technik`

```
1 if len(sys.argv) != 2:
2     sys.stderr.write('\nError! \nVerwendung:
3 ____python3 search_word.py Suchwort \n\n')
4     exit()
```

Die Zeilen 2 und 3 stehen im eigentlichen Programm auf einer Zeile, die hier getrennt wurde, um die Lesbarkeit zu verbessern.

Wie im ersten Programm wird auch hier die Anzahl der Systemargumente überprüft und das Programm beendet, falls die Zahl nicht gleich 2 ist.

```
1 import sys
2 import re
3 import pickle
```

In den ersten Zeilen des Programms werden Module geladen. Für das Einlesen der Argumente beim Aufruf des Programms wird `sys` gebraucht. `re` wird für reguläre Ausdrücke für die Suchfunktion verwendet, und `pickle` schließlich wird eingebunden, um die im ersten Programm gedumpten Listen wieder laden zu können.

```
1 parsed_list, text_list = pickle.load( open( "save.p", "rb" ) )
2 assert len(parsed_list) == len(text_list)
```

Hier werden `parsed_list` und `text_list` wieder deserialisiert, damit es möglich ist, sie in gleicher Form im aktuellen Programm zu verwenden. Dafür wird die Datei *save.p* geöffnet und geladen. Anschließend wird überprüft, ob beide Listen die gleiche Länge haben, um über den Index auf korrelierende Einträge (also gleiche Sätze) zugreifen zu können.

```
1 printed = False
2
3 search_string = str(sys.argv[1])
4 search_expr = re.compile(r''+search_string+'', re.IGNORECASE)
```

Jetzt wird eine Variable `printed` mit einem Boolean-Wert initialisiert, die dann in der Suchroutine verhindert, dass identische Sätze mit mehreren Vorkommen des Suchworts mehrmals ausgegeben werden. Das Suchwort wird von der Kommandozeile eingelesen und als String in der Variable `search_string` gespeichert. Dann wird dieser als regulärer Ausdruck kompiliert, die Flag stellt sicher, dass Vorkommen des Wortes unabhängig von Groß- und Kleinschreibung eine Übereinstimmung ergeben.

```
1 for x in range(len(text_list)):
2     printed = False
3     for word in text_list[x]:
4         for annotate in word:
5             appear = re.search(search_expr, annotate)
```

```
6                 if appear and printed == False:
7                     print(parsed_list[x])
8                     printed = True
```

Hier nun wird die nicht geparste Liste mit einzelnen Listen, die aus einem Satz bestehen, nach dem Suchwort durchsucht. Alle Indizes der Liste werden durchlaufen, bei jedem neuen Satz wird `printed` auf `False` gesetzt, da dieser Satz noch nicht ausgegeben wurde. Dann wird jedes einzelne Wort in der Satzliste durchlaufen, und dabei wiederum werden die einzelnen Tupel durchlaufen, die Tag, Wort und Lemma beinhalten. Bei einem Treffer (entweder Wort oder Lemma sind identisch), und falls der Satz noch nicht ausgegeben wurde, wird der Satz mit dem gleichen Index aus der geparsten Liste ausgegeben: also der identische Satz, gespeichert als Liste mit Siglum und dem Satz als Einheit. Dann wird `printed` auf `True` gesetzt, um identische Sätze mit mehreren Suchwortvorkommen nur einmal auszugeben. Das Programm gibt so alle Sätze mit Siglen aus, in denen ein Suchwort als Wort oder Lemma vorkommt.

Eine beispielhafte Ausgabe für das Suchwort „technisch“:

```
1  ['ana="facs:Ts-213,241r abnr:1209 satznr:3817"',
2   ( Sraffa ) Ein Ingenieur baut eine Brücke ; er schlägt dazu
3   in mehreren Handbüchern nach ; in technischen Handbüchern
4   und in juridischen . ]
5   ['ana="facs:Ts-213,332r abnr:1519 satznr:4912"',
6    Übrigens wären die mehreren Beispiele nur ein technisches
7    Hilfsmittel ,
8    und wenn ich einmal das Gewünschte gesehen
9    hät&lt;lb rend="shyphen"/&gt;te ,
10   so könnte ich ' s auch in einem Beispiel sehen . ]
```

In diese Ausgabe wurden zur besseren Darstellbarkeit Zeilenumbrüche eingefügt.
Zu sehen sind die Listen mit dem Siglum, gefolgt vom dazugehörigen Satz.

Das komplette Programm findet sich im Anhang.

### 6.3.4 Ausgeben der Analyse für einen Satz

Das Programm *get_parsed.py* erhält als Argument ein Suchwort und eine Satznummer und gibt eine syntaktische Analyse der dazugehörigen Dependenzstruktur aus. Dabei wird davon ausgegangen, dass das Programm *parse_file.py* bereits gelaufen ist und dieses Programm auf die erzeugten Datenstrukturen zugreifen kann. Ein beispielhafter Aufruf auf der Kommandozeile wäre `python3 get_parsed.py technischen 3817`. Dabei lässt sich auf die Ausgabe des zweiten Programms zurückgreifen, das Sätze mit einem Suchwort und der Satznummer ausgibt. Das Suchwort soll beim Aufrufen dieses Programms exakt (nicht lemmatisiert) eingegeben werden.

```
1  if len(sys.argv) != 3:
2      sys.stderr.write('\nError! \nVerwendung:
3  ␣␣␣␣python3␣search_word.py␣Suchwort␣Satznummer\n\n')
4      exit()
```

Die Zeilen 2 und 3 stehen im eigentlichen Programm auf einer Zeile, die hier getrennt wurde, um die Lesbarkeit zu verbessern.

Wieder wird das Programm bei einer falschen Zahl von Argumenten abgebrochen.

```
1 import sys
2 import pickle
3 import re
4 from spacy import displacy
```

Am Anfang werden Module geladen: `sys`, um Argumente auf der Kommandozeile zu ermöglichen, `pickle` für die geparsten Daten aus dem ersten Programm, `re` für die Suchfunktion und `displacy` (ein Teil von SpaCy) zur grafischen Darstellung eines Dependenzbaums.

```
1 parsed_list, text_list = pickle.load(open("save.p","rb"))
2 assert len(parsed_list) == len(text_list)
```

Hier werden die Satzlisten aus dem ersten Programm über pickle geladen und überprüft, ob die Listen die identische Länge haben, um über den Index auf Sätze zugreifen zu können.

Ein Problem bei der Programmierung, das nicht gelöst werden konnte, sind die fehlenden Tags nach der Deserialisierung. Wenn die geparsten Sätze wieder geladen werden, sind alle Informaionen des Parsingvorgangs noch gespeichert; die manuell eingelesenen Tags für die Sätze sind nicht mehr vorhanden – bei der Ausgabe des Tags wird ein leerer String ausgegeben. Die Funktionalität war davon nur geringfügig beeinträchtigt, da der Parse trotzdem ausgegeben wird, die syntaktischen Relationen sind korrekt zugänglich. Als Workaround wurde eine Funktion geschrieben, die den Sätzen (als geparsten Dokumenten) die Tags noch einmal zuweist.

```
1 word = str(sys.argv[1]); nr = str(sys.argv[2])
2 word_ex = re.compile(r''+word+'')
3 nr_ex = re.compile(r'satznr:'+nr+'"')
```

Hier werden Suchwort und Satznummer eingelesen und jeweils in reguläre Ausdrücke eingefasst, mit denen später das Siglum und der Satz durchsucht werden können.

```
1 for x in range(len(parsed_list)):
2     match_sent = re.search(nr_ex, parsed_list[x][0])
3     if match_sent:
4         doc = tag_assert(x)
```

Es werden die Sätze bzw. genauer die Satzsiglen durchlaufen. Bei jedem Satz mit der richtigen Satznummer wird dieser einer Funktion übergeben, die die Tags neu einliest und den geparsten Satz in der Variable `doc` speichert.

Hier ist eine Erweiterung für eine optimierte Funktionsweise nötig: Da einige Sätze bei Wittgenstein mehrmals in jeweils leicht unterschiedlichen Versionen vorkommen, die dann mit dem gleichen Siglum ausgezeichnet sind, wäre es wünschenswert, diese Sätze alle durchlaufen zu können (und in einer Liste mit geparsten Dokumenten zu speichern) oder aber ein zusätzliches Argument beim Programmaufruf zu ermöglichen, das angeben kann, welche Version des Satzes analysiert werden soll (für die erste Version zum Beispiel: `Maschine 2765 1.`).

```
1 def tag_assert(nr):
2     x = nr
3     taglist = []
4     for an_tuple in text_list[x]:
5         taglist.append(an_tuple[0])
6     doc = parsed_list[x][1]
7     for token in doc:
8         token.tag_ = taglist[token.i]
9     return doc
```

Diese Funktion erhält die gesuchte Satznummer als Argument und sucht den entsprechenden geparsten Satz. Diesem werden erneut seine Tags zugewiesen, dann wird er zurückgegeben. In der Variable bleibt (bei mehreren Versionen) die letzte Satzversion gespeichert.

```
1 xwords = []
```

Eine leere Liste wird erstellt, für jedes Vorkommen des Suchworts im gesuchten Satz.

```
1 for token in doc:
2     if re.search(word_ex, token.text):
3         print('\033[1;92m'+token.text+" -> "+token.tag_, end=" ")
4         xwords.append(token)
5     else:
6         print('\033[0m'+token.text, end=" ")
```

Der gefundene Satz wird wortweise durchlaufen und ausgegeben. Entspricht das Wort dem Suchwort, so wird es im Text markiert und mit seinem Tag versehen. Außerdem wird das Wort an die Liste mit Treffern angehängt.

```
1 for token in xwords:
2     print("\nDas Wort: "+str(token.text)+
3     "\nSyntaktischer Kopf: "+str(token.head)+
4     "\nArt der Dependenz: "+str(token.dep_)
5     +"\nSyntaktische Kinder: "+str([(child, child.dep_)
6       for child in token.children]))
```

Die Zeilen 2 bis 6 stehen im eigentlichen Programm auf einer Zeile, die hier getrennt wurde, um die Lesbarkeit zu verbessern.

Jetzt wird die Liste mit Treffern (Vorkommen des Suchworts im Satz) durchlaufen und dann syntaktische Informationen für jeden Treffer (direkte Relationen, Kopf, Kinder im Satz) ausgegeben.

```
1 print("Den Parsebaum sehen Sie im Browser auf localhost:5000\n")
2 displacy.serve(doc, style='dep')
```

Am Ende wird über das SpaCy-Modul displaCy eine Seite auf der Browseradresse `localhost:5000` erzeugt, auf der dann der Parsebaum zu sehen ist. Wegen des vorhin geschilderten Problems bei der Serialisierung der Listen ist dieser Baum nicht mit den

Part-of-Speech-Tags annotiert, er stellt aber die syntaktischen Abhängigkeiten ordnungsgemäß dar.

Eine alternative Möglichkeit ist die folgende:

```
1 svg = displacy.render(doc, style='dep')
2 output_file = 'satz.svg'
3 with open(output_file, 'w') as o:
4     o.write(svg)
```

Dadurch wird die Graphik des Dependenzbaums nicht im Browser zugänglich gemacht, sondern als eine SVG-Datei gespeichert.

Die Graphiken sind eingefügt im Kapitel über die Evaluation (vgl. 7).

```
1  python3 get_parsed.py Prozesses 2622
2 Während das Ergebnis nur das des
3  physikalischen Prozesses -> NN ist . If
4  Das Wort: Prozesses
5  Syntaktischer Kopf: das
6  Art der Dependenz: ag
7  Syntaktische Kinder: [(des, 'nk'), (physikalischen, 'nk')]
8  Den Parsebaum sehen Sie im Browser auf localhost:5000
9
10
11      Serving on port 5000...
12      Using the 'dep' visualizer
```

Dies ist eine beispielhafte Ausgabe auf dem Terminal, wobei die erste Zeile den Programmaufruf darstellt. Die Zeilen 2 und 3 stehen in der eigentlichen Ausgabe auf einer Zeile.

Das gesamte Programm findet sich im Anhang.

## 6.4 Statistische Erweiterung

Die hier dokumentierte Herangehensweise ist regelbasiert und verlangt, unter Ausgabe der syntaktischen Analyse, die manuelle Untersuchung eines einzelnen Satzes (oder mehrerer Sätze). Um weiterführend eine statistische Untersuchung von einzelnen Wörtern und Sätzen zu ermöglichen, kommt als nächster Schritt eine Implementierung von Frequenzlisten in Frage. Diese Frequenzlisten sollten in positive und negative Gruppen eingeteilt sein (positiv meint die Einordnung ins Themengebiet der Technik). Dann ließen sich für Wortvorkommen, die im Zusammenhang mit Technik stehen, aus jedem Satz die durch eine syntaktische Relation verbundenen Kontextwörter, etwa als Dictionary in Python, abspeichern. Für ein Wort ergäben sich damit zwei Listen, einmal alle Wörter aus allen Sätzen, die mit dem Suchwort syntaktisch direkt verbunden sind und zur Technik gehören, andererseits Listen über die Wörter, die mit dem Suchwort in einer nicht-technischen Verwendung verbunden sind. Danach könnte man also noch nicht eingeordnete Wörter anhand ihres Kontextes versuchen zu disambiguieren. Dazu muss gesagt werden, dass eine ausreichende Menge von solchen syntaktischen Kontexten erst (wahrscheinlich händisch) erstellt werden müsste, um dann neue Wörter anhand ihres Kontextes in eine Gruppe einzuordnen. Ein Beispiel ist der Satz „Während das Ergebnis nur das des physikalischen Prozesses ist . If“ (Das letzte Wort nach dem Punkt wird in der syntaktischen Analyse

vernachlässigt, der Satz ist aber so in der Datei aufgeführt). Hier sind die Wörter *physikalischen* und *Prozesses* durch eine Kante *nk* verbunden (mit *Prozesses* als Kopf), die sich auf ein *Noun Kernel* bezieht, hier also ein modifizierendes Adjektiv als Teil der Nominalphrase. Wenn man den Satz als positiv einordnet, wäre bei einer Disambiguierung eines anderen Wortes eine Relation vom Typ *nk* zu einem syntaktischen Kind *physikalischen* ein Indikator für eine wahrscheinliche Zugehörigkeit zur Technik. Diese Weiterführung auf der Grundlage von syntaktischer Analyse ist sinnvoll, da dabei dann von einem einfachen Bag-of-Words-Modell (also die Wörter ohne Reihenfolge und Kontext) abgewichen werden kann, um Wörter in ihren Beziehungen zu anderen Wörtern zu verwenden, und so eine exaktere Disambiguierung möglich wird. Auch eine Erweiterung zu mehreren Stufen ist denkbar, dass etwa Frequenzlisten eines Wortes auch andere Kinder des syntaktischen Kopfes einbeziehen oder den Kopf des Kopfes. Durch eine genauere Untersuchung bzw. Aufzeichnung nicht nur vom Satzinhalt, sondern auch der Satzstruktur soll so ermöglicht werden, das Themengebiet „Technik“ möglichst exakt herauszufiltern.

# 7 Evaluation

In diesem Abschnitt soll eine Evaluation der Ergebnisse der semantischen Suche sowie der verbundenen syntaktischen Analyse gegeben werden, anhand mehrerer Satzbeispiele aus dem verwendeten Typoskript (TS 213, das auch für die Entwicklung der Programme verwendet wurde). Untergliedert wird sie in positive und negative Beispiele, wobei ,,positiv" meint, dass das Wort dem semantischen Gebiet der ,,Technik" zugeordnet werden kann. Die syntaktischen Annotationen sollen auf ihre linguistische Haltbarkeit hin überprüft werden. Die vom Dependenzparser von SpaCy verwendete Annotation entspricht dem TIGER-Annotationsschema [The TIGER Project, 2003]. Erläuterungen von Dependenzen beziehen sich auf dieses Schema und die Erklärungen von Dependenztypen.

## 7.1 Positive Beispiele

Als erstes positives Beispiel soll der Satz aus dem Typoskript 213 mit der Nummer 2622 verwendet werden: ,,Während das Ergebnis nur das des physikalischen Prozesses ist . If" Dieser Satz ist in dieser Form Teil des Nachlasses, für die Analyse wird aber das letzte Wort, das sich hinter dem Satz befindet, ignoriert. Der Satz wurde mittels des Suchworts *Prozess* gefunden. Die syntaktische Satzwurzel ist hier das *ist*, das mehrere syntaktische Kinder aufweist: zum einen *Während*, das von der Wurzel in der Beziehung *cp*, also *Komplementierer*, abhängt. Diese Annotation erhält das Wort als eine satzeinleitende Konjunktion, die die Verbletztstellung auslöst [The TIGER Project, 2003]. Das Satzsubjekt ist *Ergebnis*, wobei von diesem Wort der Artikel *das* abhängt, das mit dem syntaktischen Kopf *Ergebnis* durch eine Dependenz mit dem Typ *nk* verbunden ist. Das steht für *Noun Kernel* und bezeichnet die Dependenzen in einer ,,Reihe von pronominalen, substantivischen und adjektivi-schen Kernelementen (NP kernel elements, NK)." [The TIGER Project, 2003] Der Typ *nk* tritt also bei Beziehungen zwischen den Wörtern in einer Nominalphrase auf. Das Wort *Ergebnis* selbst hängt von *ist* in der Beziehung *sb*, also *Subjekt*, ab. Das letzte syntaktische Kind von *ist* ist das zweite Vorkommen von *das* im Satz. Die Beziehung zwischen beiden Wörtern ist vom Typ *pd*, was *Prädikativ* bedeutet. The TIGER Project [2003] schreiben, dazu ,,zählen für uns nur die Phrasen NP und AP (darunter fallen auch bestimmte Parti-zipien, s.u.) bei den Verben *sein*, *bleiben*, *werden*." Von diesem zweiten Vorkommen von *das* hängen zwei Knoten ab: zum ersten das Wort *nur* in der Beziehung *mo*, was für *Modifikator* steht (das *nur* modifiziert das *das*). Das zweite syntaktische Kind dieses *das* ist *Prozesses*, die Art der Beziehung ist *ag*: *Genitivattribut*. Diese Einordnung ist sinnhaft, da es sich ja um *das (Ergebnis) des Prozesses* handelt, also einen das *Ergebnis* näher beschreibenden Genitiv. Vom Wort *Prozesses* hängen zwei Wörter ab, jeweils in der Beziehung *nk*, die also beide zu der Nominalphrase gehören: der Artikel *des* sowie das Adjektiv *physikalischen*. Diese syntaktische Analyse kann als insgesamt von guter Qualität betrachtet werden, da jede Zuordnung sowie jede annotierte Art der Dependenz sinnvoll ist und sich für jede Relation linguistische Argumente finden lassen. Ein offensichtlicher Fehler kommt nicht vor. Obwohl natürlich nicht jede Analyse zur Gänze richtig ist, demonstriert dieses Beispiel die grundsätzliche Funktionalität des verwendeten Parsers. Zwei hauptsächliche Wege stehen zur Verfügung, um den abgeleiteten Dependenzbaum zu visualieren. Dabei soll als erstes die Visualisierung auf einem Server betrachtet werden.

Die Grafik wird lediglich im Browser angezeigt, ohne allerdings als Datei verfügbar gemacht oder gespeichert zu werden. Diese Variante ist also gut geeignet, um einzelne Sätze

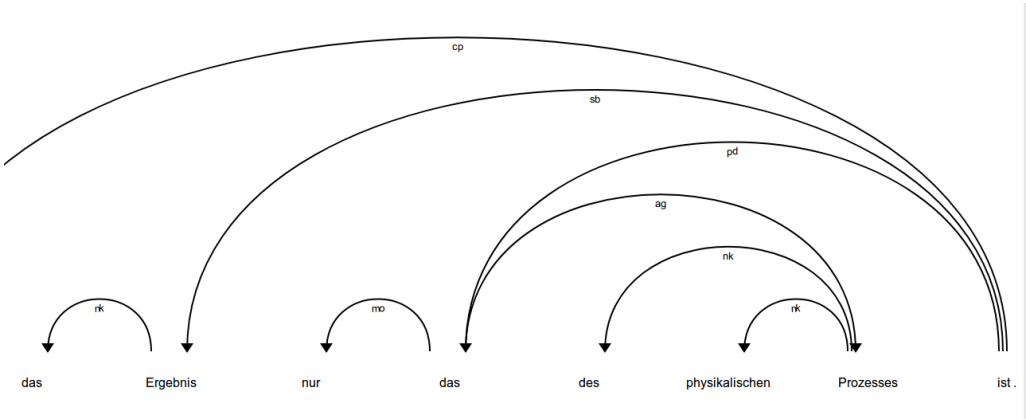


Abbildung 7.1: Ein Dependenzbaum für den Beispielsatz, erstellt mit displaCy.

zu analysieren, und lässt sich schnell und unkompliziert betrachten. Allerdings hat sie deutliche Mängel, falls größere Mengen an Daten untersucht, Sätze verglichen oder die grafischen Darstellungen weiter verwendet werden sollen. Gegen diese Art der Visualisierung spricht auch, dass nur ein Satz zur gleichen Zeit sichtbar gemacht wird und dass der Dependenzbaum auf der Browserseite sehr groß abgebildet ist (auch im Bild ersichtlich: das erste Wort, *Während*, ist auf dem Bildschirmfoto abgeschnitten). Diese fehlende Kompaktheit ist dem Gesamteindruck zusätzlich abträglich. Der Graph selbst ist schlicht und übersichtlich gehalten.

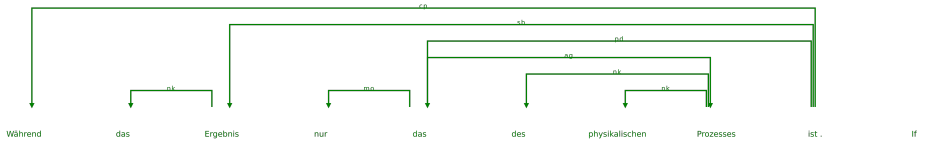


Abbildung 7.2: Ein zweiter Dependenzbaum, erstellt mit displaCy.

Dies ist die zweite Möglichkeit, eine Satzstruktur zu visualisieren. Hierbei wird mit displaCy eine SVG-Bilddatei gespeichert, die den Dependenzgraphen enthält. Dadurch werden die oben angesprochenen negativen Eigenschaften der Darstellung im Browser vermieden, da sich diese Grafik einfach für andere Zwecke weiterverwenden lässt und nicht neu erzeugt werden muss. Allerdings ist auch diese Grafik durch ihren leeren Hintergrund keineswegs eine Optimallösung, da der Dependenzbaum schwierig zu erkennen ist und das Hintergrundmuster die Betrachtung unangenehm macht. (Dieses Problem ist wegen des weißen Hintergrunds in der Arbeit nicht zu erkennen.Der ohne Optionen erzeugte Dependenzbaum wurde hier noch in der Farbe und im Kantenstil modifiziert.) Das Format als Bilddatei ist also von Vorteil, aber es wären einige optische Aufwertungen vonnöten, um die Arbeit damit und die Weiterverwendung reibungsloser und eleganter zu gestalten. Keine der beiden angebotenen Optionen ist frei von Problemen, für die Untersuchung eines einzelnen Satzes ist die Darstellung im Browser zwar ausreichend, aber kaum als umfassende Lösung zu sehen. Das Exportieren als Bilddatei ist vielversprechender, aber es müsste für eine ansprechende Präsentation das Bild grafisch anders gestaltet werden als durch den Standardaufruf von displaCy. Generell ist die Möglichkeit zur Visualisierung sehr gut,

die Umsetzung allerdings leidet unter fehlenden Optionen zur sauberen Sichtbarmachung von Dependenzstrukturen.

Dieser Satz wird hier der Technik zugeordnet. Dabei darf nicht vergessen werden, dass die Einordnung ausgehend von der Semantik in sehr wenigen Fällen exakt betrieben werden kann und sich oft Argumente sowohl für als auch wider eine Interpretation finden lassen. Dieser Satz behandelt ein *Ergebnis*, das *nur das des physikalischen Prozesses ist.* Dieses Ergebnis wird nicht näher erläutert, es *ist* nur, die Satzaussage ist demnach begrenzt und ohne den Zusammenhang anderer Sätze kaum in ihrer Gänze zu erfassen. Dennoch wurde eine Entscheidung für diese Einordnung getroffen, da eben aufgrund der limitierten Aussagekraft der gesamten Phrase einzelnen Wörtern und Satzteilen hohe Bedeutung zukommt. Das erwähnte Ergebnis wird charakterisiert als *nur das des physikalischen Prozesses*, das Satzthema also ist Konsequenz eines offenbar der Physik zuzuordnenden Prozesses, eines zielgerichteten Ablaufs. Der *physikalische Prozess* ist logischerweise ein Element der Physik, das Ergebnis, das die Physik erzeugt, wird also im Satz erörtert. Da physikalische Prozesse in Welt und Natur allgegenwärtig sind, sich der Satz genauso auf experimentelle Forschung beziehen kann, lässt sich nicht genauer bestimmen, von welcher Art von *physikalischem Prozess* gesprochen wird. Nichtsdestotrotz gehört der zugrundeliegende Ablauf zur Physik, dieser Satz ist also zweifelsohne diesem spezifischen Teilgebiet der Technik zuzuweisen. Nun hängt von der Annahme über die Terminologie ab, ob Physik als ein Teil von Technik eingeschlossen wird: diese Arbeit bejaht diese Annahme. Trotzdem gehört aber die Physik als eine Wissenschaft nicht zum alltäglicheren, engeren Technikbegriff, der eher in die Richtung der Mechanik, des Maschinenbaus oder Ingenieurwesens geht, wobei auch hier eine gewisse Unschärfe die Definitionen betreffend unvermeidlich ist. Der Satz wird also dem Themengebiet zugeschrieben, da das *Ergebnis* eines *physikalischen Prozesses* so zentral in ihm ist, wobei die Semantik der Nominalphrase *des physikalischen Prozesses* dabei ein entscheidendes Kriterium ist: dieser Ausdruck wird in dieser Arbeit als technisch aufgefasst. Zusätzlich wurde dieser Satz mit dem höchst ambivalenten Suchbegriff des *Prozesses* gefunden, dass dieser Begriff in direkter Beziehung zum Adjektiv *physikalisch* steht und von diesem modifiziert wird, ist ein zusätzliches Argument dafür, dass der Suchbegriff (und damit verbunden dieser Satz) dem Themengebiet „Technik“ entstammt.

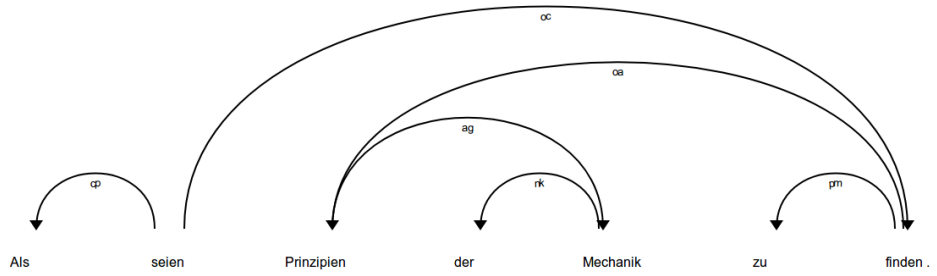


Abbildung 7.3: Ein Dependenzbaum von displaCy für das nächste Beispiel

Der abgebildete Baum ist eine Analyse des nächsten Beispiels, des Satzes mit der Nummer 1846, gefunden mit dem Suchwort *Mechanik*. Der Satz lautet: „Als seien Prinzipien der Mechanik zu finden .“ Die Wurzel des Satzes ist das Wort *seien*, von dem die Kinder *als* und *finden* abhängen. Die Beziehung zu *als* hat den Typ *cp*, also *Komplementierer*. Als satzeinleitende Konjunktion ist diese Einordnung sinnvoll. Das andere Kind, *finden*, trägt die Relation *oc*, was *clausal object* bedeutet: „VPs und Sätze, die Komplemente von NPs sind, werden als OC (clausal object) annotiert.“ [The TIGER Project, 2003] Die Verbalphrase *finden* (bzw. *zu finden*) ist also dependent von der Wurzel *seien*. Von *finden*

wiederum hängt das Wort *zu* in der Beziehung *pm* ab: ,,PM steht für morphologische Partikel“ [The TIGER Project, 2003]. Das *zu* wird hier infinitivisch gebraucht und ist daher ein solches Partikel. Außerdem hängt der Knoten *Prinzipien* von *finden* ab, das Label ist *oa*, was *Akkusativobjekt* bedeutet. Diese Annotation ergibt bezogen auf das Verb *finden* einen Sinn, da dieses den Akkusativ verlangt, im Zusammenhang des gesamten Satzes allerdings ist die Korrektheit weniger eindeutig: die Frage nach den Prinzipien kann als *wer sei zu finden?* formuliert werden, was nicht auf einen Akkusativ hinweist. Der gesamte Satz weist kein echtes Subjekt auf, was eine syntaktische Untersuchung verkompliziert, da man durchaus argumentieren kann, dass der Text in dieser Form mehr eine Bemerkung darstellt als einen grammatikalisch korrekten Satz. Diese Tatsache wirkt sich dann auf die Annotation aus, die etwa in diesem Fall der *Prinzipien* uneindeutig ist. Der Satz wirkt wie ein Klausalobjekt oder ein Modifikator eines nicht vorhandenen Hauptsatzes. Das Wort *Prinzipien* hat als syntaktisches Kind die *Mechanik*, die Art der Relation ist *ag*, also *Genitivattribut*. Diese Zuweisung ist klar und sinnvoll. Vom Wort *Mechanik* schließlich hängt der Artikel *der* ab, die Beziehung ist *nk*, da eine Nominalphrase gebildet wird. Diese Analyse weist viele schlüssige Annotationen auf, auch wenn keineswegs alle unbestreitbar richtig sind, was auch mit der fragmentellen Natur des Satzes zusammenhängen kann. Dieser nächste recht kurze Satz wird ebenfalls dem Feld der Technik zugeordnet, weil seine Aussage unklar ist: es ist nicht bekannt, was genau so ist, *als seien Prinzipien . . . zu finden.* Klar ist aber, dass diese *Prinzipien der Mechanik* etwas beeinflussen, irgendeine Entität ist, als seien sie dort zu finden oder würden sie dort gelten. Da unbekannt ist, worauf sie das bezieht, bleiben im vorliegenden Satz die *Prinzipien* zentral relevant: sie sind das Satzthema, das fast ganz ohne Kontext dargestellt ist, es lässt sich keine weiterführende Aussage ableiten. Deswegen beschränkt sich die Semantik hier auf die *Prinzipien der Mechanik*, die rein für sich genommen eindeutig der Technik entstammen: die Mechanik als so zentrales Feld des Begriffs Technik (und ihre Prinzipien) kann so eingeordnet werden. Hier ist im Hinblick auf die Disambiguierung eine mögliche Erweiterung durch Frequenzlisten interessant: ein Substantiv mit dem Genitivattribut *der Mechanik* kann durch die anzunehmende Zugehörigkeit zu dieser mit hoher Wahrscheinlichkeit der Technik zugewiesen werden.

Das nächste positive Beispiel ist der Satz mit der Satznummer 5503: ,,Angenommen , das Anziehen des Bremshebels bewirkt manchmal das Ab<lb rend=ßhyphen”/>bremsen der Maschine und manchmal nicht .“ Dieser Satz wurde mittels des Suchworts *Maschine* entdeckt. Das Wort *Abbremsen* ist hier durch eine Annotation getrennt, wird für die Zwecke dieser Analyse aber wie die Standardform *Abbremsen* behandelt. Hier liegt ein längerer Satz vor, dessen Satzwurzel das finite Verb *bewirkt* ist. Fünf unterschiedliche syntaktische Kinder dieses Worts liegen vor: als Subjekte *Angenommen* sowie *Anziehen*. Das *Anziehen* in der Relation *sb* ist korrekt, da es sich ja um das agierende Element handelt: *Das Anziehen bewirkt . . . .* Daneben ist die Einordnung des *Angenommen* fragwürdig, da zwar der Satz durch dieses Wort modifiziert wird, es aber keineswegs ein klassisches Subjekt (wie *Anziehen* etwa) ist. Es ähnelt in seiner Form einem Subjektsatz oder Modifikator, ist aber linguistisch hier nicht eindeutig. Die weiteren Dependenten sind *manchmal*, *Abbremsen* und *und*. Das Wort *manchmal* hängt in der Beziehung *mo* von der Wurzel ab, es modifiziert das *Bewirken*. Der Knoten *Abbremsen* ist als Akkusativobjekt annotiert, dies kann durch eine Frage danach veranschaulicht werden: *Wen oder was bewirkt das Anziehen?* Die letzte Relation, zu *und*, besitzt den Typ *cd*: Es hängt als koordinierende Konjunktion von *bewirkt* ab, *das Anziehen bewirkt . . . und . . . .* (nach The TIGER Project [2003]) Von diesem *und* hängt das Wort *nicht* als Konjunkt ab, von dem wiederum ein zweites *manchmal* als Modifikator abhängt. Diese Einordnung ist nachvollziehbar, allerdings ist anzumerken, dass das *nicht* auch das Wort *bewirkt* beinflusst; das ,,Abbremsen“ wird manchmal ,,nicht bewirkt“. Weiterhin hat das Wort *Anziehen* als Dependenten die Wörter *das* und *Bremshebels*. Der Typ der Relation ist beim *das* bezeichnet mit *nk*. Die Relation zu *Bremshebels*

ist annotiert mit *ag*, also *Genitivattribut*, von diesem Wort schließlich hängt der Artikel *des* als *Noun Kernel* ab. Vom Akkusativobjekt *Abbremsen* hängen zwei syntaktische Kinder ab: der Artikel *das* als *nk* und die *Maschine* als ein *Genitivattribut*. Die *Maschine* bestimmt den Artikel *einer* ebenfalls als Teil eines *Noun Kernels*. Diese Einordnungen sind als sinnvoll zu sehen. Dieser Satz wird aufgrund seiner offensichtlich direkten Beziehung zur Mechanik als positiv bewertet. Das Suchwort *Maschine* steht in unmittelbarer Relation zum *Abbremsen*, was als eindeutige physische Tätigkeit aufgefasst werden kann, eine *abbremsende Maschine* gehört höchstwahrscheinlich zur Technik im engeren, mechanischen Sinne. Auch der restliche Satzkontext unterstützt diese Einschätzung: es geht um das *Anziehen des Bremshebels*, wobei der *Bremshebel* als ein Gegenstand für sich und das *Anziehen* als Tätigkeit der Technik angehört. Schon wegen dieser enthaltenen Bedeutungen von eindeutig technischen Gegenständen lässt sich der Satz als Ganzes der Technik zuordnen, mehr noch aufgrund der dezidiert mechanischen Vorgänge. In diesem Beispiel zeigt sich, dass Frequenzlisten auch über mehr als direkte Relationen eines Suchwortes sinnvoll sein können: so ist über zwei Kanten von der *Maschine* das Wort *bewirkt*, über drei Kanten zusätzlich auch das Wort *Anziehen* entfernt.

Als letztes positives Beispiel dient der Satz, der die Nummer 3103 trägt: „Es gäbe da Zahnräder , Hebel , Stäbe , Kolben , Lager , etc. .“ Der Satz wurde über das Suchwort *Hebel* gefunden. Die Satzwurzel ist *gäbe*, von diesem Wort hängen als *Akkusativobjekt* die *Zahnräder* ab. Dieses Wort *Zahnräder* ist (über die Beziehung *cj*) mit dem Wort *Hebel* als dessen syntaktischer Kopf verknüpft, es liegt also ein *Konjunkt* vor. Diese Annotation hat in dieser Aufzählung, die als Koordination (wenngleich ohne eine Konjunktion) aufgefasst werden kann, ihre Berechtigung. Vom Wort *Hebel* hängen dann, ebenfalls sämtlich als *Konjunkt*, weitere Wörter ab: *Stäbe*, *Kolben* und *Lager*. Noch mehr als im vorangegangenen Beispiel wird dieser Satz nach den in ihm enthaltenen Begriffen in die Richtung der Technik gerückt: es liegt eine Aufzählung von eindeutig mechanischen, technischen, auch dem Maschinenbau verwandten Dingen vor, die einem alltäglichen Verständnis des Wortes *Technik* entsprechen und damit diesen Satz dem Themengebiet zuweisen.

## 7.2 Negative Beispiele

Als erstes negatives Beispiel wird der Satz mit der Satznummer 852 verwendet, der mit dem Suchwort *Zug* gefunden wurde: „( Man kann vielleicht auch von einem spezifischen Gefühl reden welches der Schachspieler bei Zügen mit dem König empfindet . )“ Hier ist die Satzwurzel das Wort *kann*, das eine Relation vom Typ *oc*, also *Klausalobjekt*, zum Wort *reden* aufweist. Von diesem Wort *reden* hängt als *Modifikator* das Wort *bei* ab, das wiederum als Dependenten das Wort *Zügen* hat. *Zügen* regiert in der Beziehung *mnr*, was *postnominaler Modifikator* [The TIGER Project, 2003] bedeutet, das Wort *mit*. Diese Annotationen sind linguistisch annehmbar, allerdings gibt es im Satz auch fragwürdige und falsche Fälle, wie etwa eine fehlende Relation vom Wort *Gefühl* zum *empfindet* oder eine Subjektbeziehung von *reden* zum Wort *Schachspieler*. Dieser Satz wird als negatives Beispiel eingeordnet, da sich hier die *Züge* nicht auf eine der Technik angehörige Sache (wie eine Eisenbahn oder einen Federzug) beziehen. Auch zum physikalischen Begriff des Zuges stehen sie nicht in Verbindung. Vielmehr geht es um Züge in einem Schachspiel, das sich der Technik nicht zuordnen lässt, auch sonst hat der Satz keine Verbindung zu diesem Feld. Anzumerken ist hier, dass für die Disambiguierungsarbeit die direkten Relationen (zu den Wörtern *bei* und *mit*) kaum aufschlussreich sind und bei der Arbeit mit Korpora als Stopwörter klassifiziert werden könnten, da sie keine aussagekräftige Semantik besitzen. Hilfreich wäre die Betrachtung weitergehender Relationen, so hat der etwa der syntaktische Kopf von *bei* (dem Kopf von *Zügen*) als andere syntaktische Kinder auch *Schachspieler*, ein Kind des syntaktischen Kindes von *Zügen* ist der *König*.

Als zweites negatives Beispiel dient der Satz mit der Nummer 4230, gefunden mit *Technik*: ,,Denn so werden charakteristische Vorstellungen die sich mit den Worten des Satzes verbinden für das Maßgebende angesehen , auch dort , wo sie es gar nicht sind & alles auf die Technik einer Verwendung ankommt . – –“ Dieser Satz wird von SpaCy beim Dependenzparsing in zwei Teile geteilt, getrennt durch das Ampersand & (allerdings trotzdem als ein Parse ausgegeben), der hintere Teil mit dem Wort *Technik* ist hier von Interesse. Die Wurzel dieses Satzteils ist *ankommt*, von der das Wort *auf* in der Beziehung *op* abhängt, das bedeutet *Präpositionalobjekt* [The TIGER Project, 2003] und ist schlüssig. Vom *auf* hängt die *Technik* ab, der Typ der Relation ist *nk*. Diese Einordung macht Sinn, weil das *ankommt* als Subjekt das Wort *alles* bestimmt, die Nominalphrase im Präpositionalobjekt ist dann *(auf) die Technik einer Verwendung*. Die *Technik* nun hat als Dependenten das Wort *Verwendung* mit der Relationsart *ag*, *Genitivattribut*. Diese Annotation veranschaulicht die Zugehörigkeit zur negativen Klasse: die Technik ist hier nicht die mechanische, Ingenieurs-Technik, sondern eine bestimmte Methode. Die Relation *ag* weist auf eine *Technik einer Verwendung*, *Technik der Verwendung* oder kürzer *Verwendungstechnik* hin, mit der Technik im hier gesuchten Sinne liegen also keine Überschneidungen vor. Auch der restliche Satz behandelt nicht Technik, sondern Sprachwissenschaft und Logik, die *Technik einer Verwendung* bezieht sich mutmaßlich auf die Verwendung von Wörtern. Durch diese Relation kann also gezeigt werden, dass hier nicht das Feld der Technik, sondern eine Vorgehensweise, eine Art der Verwendung gemeint ist.

Das letzte Beispiel wurde gefunden mit dem Suchwort *Maschine* und hat die Satznummer 3135: ,,Wie wäre es wenn ein Mensch die Sprache erfände wie man eine Maschine erfindet ? “ Die Wurzel des Satzes ist *wäre*, von der als *Subjekt* das Wort *es* abhängt. Dieses *es* hat als Dependenten das Wort *erfindet*, wobei die Beziehung *re* ist: ,,das Korrelat-es steht immer zusammen mit ei-nem satzwertigen Subjekt oder Objekt“ [The TIGER Project, 2003]. Dieses *erfindet* hängt also vom Korrelat-es als ein sogenanntes *Repeated Element* ab, das die eigentliche Bedeutung trägt (nach The TIGER Project [2003]). Dieses *erfindet* nun bestimmt die *Maschine*, die ein Akkusativobjekt darstellt. Von der Maschine hängt der Artikel *eine* als *Noun Kernel* ab. Obwohl dieser Satz als negativ eingestuft wurde, liegt hier ein nicht geringes Maß an Ambivalenz vor: Die *Maschine* als solche wird *erfunden* und weist damit durchaus auf eine Maschine im rein technischen Sinne hin, als ein Produkt des Maschinenbaus, die als eine Erfindung in die Welt tritt. Auch der direkte syntaktische Kontext (mit der Relation zu *erfindet* stützt diese Auffassung. Dagegen ließe sich einwenden, dass die Maschine hier bloße Metapher ist, mit der Sprache verglichen wird, in der Frage, ob sich diese auch erfinden lasse. Die Sprache entstammt als Phänomen nicht der Technik und beherrscht als Thema diesen Satz. Es liegt also eine Oszillation zwischen der Semantik und dem Kontext vor: die erfundene Maschine gehört zum Gebiet der Technik, im Kontext des gesamten Satzes lässt sich aber argumentieren, dass hier die Maschine nur als ein sprachliches Bild verwendet wird und deswegen keine Aussage über Technik getroffen wird: der Satz verwendet zwar eine technische Sache, behandelt diese aber nicht. Dies ist ein Grenzfall, in dem verschiedene Faktoren entscheiden, wie Syntax und Sinngehalt letztlich zu einer Einordnung führen.

## Zusammenfassung

Es wurden verschiedene Beispielsätze herangezogen und diese vor allem bei den positiven Beispielen linguistisch erläutert, mit einer Beurteilung der Güte des geparsten Satzes. Diese automatisch erstellten Dependenzstrukturen sind größtenteils als korrekt einzustufen, wobei es durchaus Fälle von Annotationen gibt, die fragwürdig oder falsch sind. Gerade bei längeren Sätzen und mehrere Wörter überspannende Bögen treten diese Fälle auf. Dennoch ist als eine Grundlage der bereitgestellte Parser verwendbar und liefert akzeptable Ergebnisse. Die Sätze wurden nach ihrer Dependenzstruktur disambiguiert, bei einigen

Beispielen war der Kontext der direkten Relationen ausreichend und sinnvoll anwendbar für diese Unterscheidung zwischen Technik und Nicht-Technik, bei anderen zeigte sich, dass dieser Ansatz für robuste Ergebnisse noch erweitert werden muss. Der Vorteil dieses Ansatzes gegenüber einer rein philosophischen Disambiguierung zeigt sich im Aufbau der kleineren Wortkontexte, die mit dem zu untersuchenden Wort über eine geringe Anzahl von Relationen verbunden sind: so offenbaren sich Konnotationen, die ein Wort in eine bestimmte semantische Richtung schieben, es lassen sich über den Kontext Satzteile mit einer abweichenden Semantik (wie etwa *Maschine erfindet*) herausgreifen, die syntaktische Struktur ordnet den zu untersuchenden Satz für eine systematische Betrachtung, und die Relationen (wie Genitive oder Adjektive in einem Noun Kernel) zum Suchwort können Aufschluss über seine Verwendung geben (*... der Mechanik*, *einer Verwendung*). Der nächste Schritt ist, über Frequenzlisten eine automatisierte Disambiguierung zu beginnen.

# 8 Fazit

In dieser Bachelorarbeit wurde zuerst in die verwendeten Ressourcen eingeführt, wie das Projekt zu Ludwig Wittgenstein als Grundlage für die Arbeit, die verwendeten Daten des Nachlasses sowie eine Wortlist mit Begriffen, die für eine Disambiguierung in Frage kommen. Es wurde ein Überblick über relevante Werke gegeben und ein Versuch mit einem Dependenzparser dokumentiert. Außerdem wurde der Aufbau der Programme erläutert, mit denen an die gestellte Aufgabe herangegangen wurde, sowie eine Evalution von gefundenen Ergebnissen durchgeführt, insbesondere mit Blick auf eine Weiterverarbeitung mit statistischen Methoden. Insgesamt war diese Arbeit dahingehend erfolgreich, dass gezeigt werden konnte, dass eine linguistische Analyse der Dependenzstrukturen von Sätzen Einblicke in den Bedeutungsgehalt zulassen und einzelne Wörter, auch und gerade in unterschiedlichen Relationen zu verschiedenen Kontexten, dadurch im Hinblich auf die Zugehörigkeit zu einem Thema genauer unterschieden werden können. Allerdings ist das mit Einschränkungen verbunden, so trägt eine (einfache) syntaktische Analyse keineswegs bei jedem Satz zu einer gewinnbringenden Konstruktion von anderen bzw. kleineren Kontexten bei. Auch ist ohne die statistische Weiterführung noch immer manuelle Interpretation notwendig. Die Arbeit zeigt nach persönlicher Einschätzung vielversprechende Ansätze für die geisteswissenschaftliche Arbeit mit Ludwig Wittgenstein, die allerdings einer weiteren Beschäftigung damit bedürfen, um in größerem Stil ertragreich eingesetzt werden zu können. Als ein Ausblick für die künftige Behandlung dieser Aufgabenstellung ist zuvorderst die Statistik zu nennen, die über die Erhebung von einzelnen Kontexten noch weit mehr zu einer umfassenden Disambiguierung von Wörtern beitragen kann. Abgesehen von den beschriebenen Frequenzlisten ist auch eine Katalogisierung von Wörtern in bestimmten Relationen oder von spezifischen Kontexten denkbar, die im Werk Wittgensteins auf eindeutige Weise verwendet werden. Darüber hinaus sind Punkte, an denen eine Weiterentwicklung möglich ist, die Arbeit mit den getaggten XML-Dateien, um diese ohne Umwege parsen zu können, die Visualisierung von Dependenzgraphen oder die Ausweitung dieser Herangehensweise auf andere semantische Gebiete.

# Literaturverzeichnis

Anat Biletzki and Anat Matar. Ludwig Wittgenstein. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, Spring 2014 edition, 2014. Abruf am 09.05.2018.

Steven Bird, Ewan Klein, and Edward Loper. *Natural Language Processing with Python*. O'Reilly Media, 2009.

Sabine Brants, Stefanie Dipper, Peter Eisenberg, Silvia Hansen, Esther König, Wolfgang Lezius, Christian Rohrer, George Smith, and Hans Uszkoreit. Tiger: Linguistic Interpretation of a German Corpus. *Journal of Language and Computation*, pages 597–620, 2004.

Danqi Chen and Christopher Manning. A fast and accurate dependency parser using neural networks. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 740–750, Doha, Qatar, October 2014. Association for Computational Linguistics.

Yoav Goldberg and Joakim Nivre. A Dynamic Oracle for Arc-Eager Dependency Parsing. In *Proceedings of COLING 2012*, pages 959–976, Mumbai, India, 2012. The COLING 2012 Organizing Committee.

Matthew Honnibal and Mark Johnson. An improved non-monotonic transition system for dependency parsing. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1373–1378, Lisbon, Portugal, September 2015. Association for Computational Linguistics.

Eliyahu Kiperwasser and Yoav Goldberg. Simple and Accurate Dependency Parsing Using Bidirectional LSTM Feature Representations. *Transactions of the Association for Computational Linguistics*, 4:313–327, 2016.

Sandra Kübler, Ryan McDonald, and Joakim Nivre. Dependency parsing. *Synthesis Lectures on Human Language Technologies*, 1(1):1–127, 2009. In Buchform verwendet.

Ray Monk. *Ludwig Wittgenstein: The Duty of Genius*. Cape, 1990. ISBN 9780224027120. URL `https://books.google.de/books?id=0JfXAAAAMAAJ`. Abruf am 10.05.2018.

Ray Monk. *How to Read Wittgenstein*. How to read. Granta Books, 2005. ISBN 9781862077249. URL `https://books.google.de/books?id=jRnXAAAAMAAJ`. Abruf am 11.05.2018.

Joakim Nivre. Inductive Dependency Parsing. Zu diesem Artikel ist kein Jahr oder weitere Informationen angegeben.

Joakim Nivre. An Efficient Algorithm for Projective Dependency Parsing. In *Proceedings of the 8th International Workshop on Parsing Technologies (IWPT 03)*, pages 149–160, Nancy, France, April 2003.

Joakim Nivre. Incrementality in Deterministic Dependency Parsing. In *Proceedings of the Workshop on Incremental Parsing: Bringing Engineering and Cognition Together*, IncrementParsing '04, pages 50–57, Barcelona, Spain, 2004. Association for Computational Linguistics.

Joakim Nivre and Jens Nilsson. Pseudo-projective dependency parsing. In *Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics*, pages 99–106, Ann Arbor, Michigan, 2005. Association for Computational Linguistics, Association for Computational Linguistics.

Joakim Nivre, Johan Hall, and Jens Nilsson. Maltparser: A data-driven parser-generator for dependency parsing. In *Proceedings of LREC*, volume 6, pages 2216–2219, Genoa, Italy, May 2006.

Joakim Nivre, Johan Hall, Jens Nilsson, Atanas Chanev, Gülşen Eryigit, Sandra Kübler, Svetoslav Marinov, and Erwin Marsi. Maltparser: A language-independent system for data-driven dependency parsing. *Natural Language Engineering*, 13(2):95–135, 2007.

NLTK, 2017. Natural Language Toolkit — NLTK 3.3 documentation, 2017. URL `https://www.nltk.org/`. Abruf am 17.05.2018.

Alois Pichler. Wittgenstein Archives at the University of Bergen (WAB): Open Access to Wittgenstein's Nachlass., 2018. URL `http://wab.uib.no/`. Abruf am 19.05.2018.

Pickle, 2018. 12.1. pickle — Python object serialization — Python 3.6.5 documentation, 2018. URL `https://docs.python.org/3/library/pickle.html`. Abruf am 16.05.2018.

PyCharm, 2018. Python IDE for Professional Developers by JetBrains, 2018. URL `https://www.jetbrains.com/pycharm/`. Abruf am 15.05.2018.

Python, 2018. Welcome to Python.org, 2018. URL `https://www.python.org/`. Abruf am 16.05.2018.

Helmut Schmid. Probabilistic Part-of-Speech Tagging Using Decision Trees. In *Proceedings of International Conference on New Methods in Language Processing*, Manchester,UK, 1994. Abruf am 11.05.2018.

Helmut Schmid. Improvements in Part-of-Speech Tagging with an Application to German. In *Proceedings of the ACL SIGDAT-Workshop*, Dublin, Ireland, 1995. Abruf am 10.05.2018.

J. Schulte. *Wittgenstein: Eine Einführung*. Reclam Universal-Bibliothek. Reclam Philipp Jun., 2016. ISBN 9783150193860. URL `https://books.google.de/books?id=RCIajwEACAAJ`. Abruf am 09.05.2018.

Wolfgang Seeker. spacy now speaks German · Blog · Explosion AI, 2016. URL `https://explosion.ai/blog/german-model`. Abruf am 17.05.2018.

Anders Søgaard and Yoav Goldberg. Deep multi-task learning with low level tasks supervised at lower layers. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 231–235, Berlin, Germany, August 2016. Association for Computational Linguistics.

SpaCy, 2018. spaCy · Industrial-strength Natural Language Processing in Python, 2018. URL `https://spacy.io/`. Abruf am 15.05.2018.

SpaCy API, 2018. Architecture · spaCy api documentation, 2018. URL `https://spacy.io/api/`. Abruf am 15.05.2018.

SpaCy Models, 2018. German · spaCy Models Documentation, 2018. URL `https://spacy.io/models/de`. Abruf am 15.05.2018.

The TIGER Project. Tiger Annotationsschema. 2003.

TIGER Corpus, 2018. Download - Institut für Maschinelle Sprachverarbeitung - Universität Stuttgart, 2018. URL `http://www.ims.uni-stuttgart.de/forschung/ressourcen/korpora/TIGERCorpus/download/start.html`. Abruf am 17.05.2018.

W. Vossenkuhl. *Ludwig Wittgenstein: Tractatus logico-philosophicus*. Klassiker Auslegen. De Gruyter, 2001. ISBN 9783050050348. URL `https://books.google.de/books?id=yNLhwVYvjX4C`. Abruf am 12.05.2018.

W. Vossenkuhl. *Ludwig Wittgenstein*. Beck Reihe. Beck, 2003. ISBN 9783406494192. URL `https://books.google.de/books?id=rS9Kefk7aXcC`. Abruf am 08.05.2018.

Ludwig Wittgenstein. *Tractatus Logico-Philosophicus*. Edition Suhrkamp. Suhrkamp, 2001. ISBN 3518100122.

Ludwig Wittgenstein. *Philosophische Untersuchungen*. Bibliothek Suhrkamp. Suhrkamp, 2003. ISBN 9783518223727. URL `https://books.google.de/books?id=lK-RSgAACAAJ`. Abruf am 10.05.2018.

Yuan Zhang and David Weiss. Stack-propagation: Improved Representation Learning for Syntax. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1557–1566, Berlin, Germany, August 2016. Association for Computational Linguistics.

# Abbildungsverzeichnis

# Anhang

Im Anhang befinden sich alle drei geschriebenen Programme komplett. Ein „//“ am Zeilenanfang signalisiert, dass diese Zeile im eigentlichen Programm keine eigene Zeile ist, sondern zur vorherigen gehört, die aus Gründen der Lesbarkeit getrennt wurde.

Das Programm `parse_file.py`:

```
1  import sys
2  import re
3  import spacy
4  from spacy.tokens import Doc
5  import pickle
6
7  if len(sys.argv) != 3:
8      sys.stderr.write('\nError! \nVerwendung:
9  // python3 parse_file.py Eingabedatei Ausgabedatei \n\n')
10     exit()
11
12 nlp = spacy.load('de', disable=['tagger', 'ner'])
13 parser = nlp.parser
14
15 def get_lines(infile):
16     line_count = 0; lines = ['-']
17     with open(infile, 'r') as f:
18         for line in f:
19             line_count += 1; lines.append(line)
20     sentences = extract_sents(line_count, lines)
21     return sentences
22
23 def extract_sents(linecount, lines):
24     sent = []; sentences = []; count = -1
25     sig = re.compile(r'<s (n="(.+?)")
26 //  (ana="facs:(.+?) abnr:(\d+?) satznr:(\d+?)")>')
27     for x in range(0, linecount, 1):
28         count += 1
29         start = re.search(sig, lines[x])
30         if start:
31             sent.append(lines[x])
32             for y in range(count + 1, linecount, 1):
33                 l = re.search(sig, lines[y])
34                 if l:
35                     sentences.append(sent); sent = []
36                     break
37                 else:
38                     sent.append(lines[y])
39     return sentences
40
41
42 def make_sentences(sentence_list):
```

```
43      new_sentence_list = []; new_sentence = []
44      head = re.compile(r'<s (n="(.+?)"
45  //  (ana="facs:(.+?) abnr:(\d+?) satznr:(\d+?)")>')
46      part = re.compile(r'<w t="(.+?)" l="(.+?)">(.+?)</w>')
47      for sentence in sentence_list:
48          for word in sentence:
49              fhead = re.search(head, word)
50              fpart = re.search(part, word)
51              if fhead:
52                  new_sentence.append(fhead.group(3))
53              if fpart:
54                  new_sentence.append((fpart.group(1)
55            //    , fpart.group(2), fpart.group(3)))
56          new_sentence_list.append(new_sentence)
57          new_sentence = []
58      return new_sentence_list
59
60
61  def parse_list(text_list):
62      parsed_list = []
63      parsed_text_list = []
64      for text in text_list:
65              parsed_text_list.append(text[0])
66              text.pop(0)
67              parsed_doc = get_parse(text)
68              parsed_text_list.append(parsed_doc)
69              parsed_list.append(parsed_text_list)
70              parsed_text_list = []
71      return parsed_list
72
73
74  def get_parse(text):
75      word_list = []; tag_list = []
76      for word in text:
77          word_list.append(word[2])
78          tag_list.append(word[0])
79      doc = Doc(nlp.vocab, words=word_list,
80   //  spaces=[True for x in range(len(text))])
81      for token in doc:
82          token.tag_ = tag_list[token.i]
83      parsed_doc = parser(doc)
84      for token in parsed_doc:
85          assert token.tag_ == tag_list[token.i]
86      return parsed_doc
87
88
89  if __name__ == '__main__':
90      infile = str(sys.argv[1])
91      outfile = str(sys.argv[2])
92      sentence_list = get_lines(infile)
93      text_list = make_sentences(sentence_list)
94      parsed_list = parse_list(text_list)
95
```

```
96      with open (outfile, 'w') as o:
97          for entry in parsed_list:
98              o.write('Siglum: '+str(entry[0])+'\n')
99              o.write('Satz: '+str(entry[1])+'\n')
100             for token in entry[1]:
101                 o.write('\nWort: '+str(token.text)+
102             //      '\nRelation: '+str(token.dep_)+
103             //      '\nKopf: '+str(token.head)+'\n')
104             o.write('\n\n')
105
106     pickle.dump((parsed_list, text_list),
107    // open("save.p", "wb"))
```

Das Programm `search_word.py`:

```
1  import sys
2  import re
3  import pickle
4
5  if len(sys.argv) != 2:
6      sys.stderr.write('\nError! \nVerwendung:
7  // python3 search_word.py Suchwort \n\n')
8      exit()
9
10 parsed_list, text_list = pickle.load(open("save.p", "rb"))
11 assert len(parsed_list) == len(text_list)
12
13 printed = False
14
15 search_string = str(sys.argv[1])
16 search_expr = re.compile(r''+search_string+'', re.IGNORECASE)
17
18
19 for x in range(len(text_list)):
20     printed = False
21     for word in text_list[x]:
22         for annotate in word:
23             appear = re.search(search_expr, annotate)
24             if appear and printed == False:
25                 print(parsed_list[x])
26                 printed = True
```

Das Programm `get_parsed.py`:

```
1  import sys
2  import pickle
3  import re
4  from spacy import displacy
5
6  if len(sys.argv) != 3:
7      sys.stderr.write('\nError! \nVerwendung:
8  // python3 search_word.py Suchwort Satznummer\n\n')
9      exit()
10
11 parsed_list, text_list = pickle.load(open("save.p","rb"))
```

```
12 assert len(parsed_list) == len(text_list)
13
14
15 word = str(sys.argv[1]); nr = str(sys.argv[2])
16 word_ex = re.compile(r''+word+'')
17 nr_ex = re.compile(r'satznr:'+nr+'" ')
18
19
20 def tag_assert(nr):
21     x = nr
22     taglist = []
23     for an_tuple in text_list[x]:
24         taglist.append(an_tuple[0])
25     doc = parsed_list[x][1]
26     for token in doc:
27         token.tag_ = taglist[token.i]
28     return doc
29
30
31
32 for x in range(len(parsed_list)):
33     match_sent = re.search(nr_ex, parsed_list[x][0])
34     if match_sent:
35         doc = tag_assert(x)
36
37 xwords = []
38
39 for token in doc:
40     if re.search(word_ex, token.text):
41         print('\033[1;92m'+token.text+" -> "+token.tag_, end=" ")
42         xwords.append(token)
43     else:
44         print('\033[0m'+token.text, end=" ")
45
46 for token in xwords:
47     print("\nDas Wort: "+str(token.text)+"\n
48 // Syntaktischer Kopf: "+str(token.head)+
49       "\nArt der Dependenz: "+str(token.dep_)+"\n
50 //    Syntaktische Kinder: "+str([(child, child.dep_)
51  // for child in token.children]))
52
53
54 print("Den Parsebaum sehen Sie im Browser auf localhost:5000\n")
55 displacy.serve(doc, style='dep')
56
57 '''
58 svg = displacy.render(doc, style='dep')
59 output_file = 'sentence.svg'
60 with open(output_file, 'w') as o:
61     o.write(svg)
62 '''
```

Eine Auswahl aus der Wortliste zum Suchen von Sätzen zur Technik:

Technik · technisch · Methode · Watson · Archimedes · Ingenieur · Mechanik · Architekt · Chemie · mathematisch · Mathematik · physikalisch · Physik · Bremse · Bremshebel · Radio · Telephon · maschinenhaft · Maschinenteil · Rechenmaschine · Zylinder · Feder · Kessel · Motor · Schraube · Barometer · Thermometer · Waage · Automobil · Lokomotive · Industrie · Automat · automatisch · Entwurf · entwerfen · Erfindung · Experiment · Hebel · Konstruktion · Werkzeug · Funktion · elektrisch · Kraft

Einige weitere Beispielsätze:

['ana=facs:Ts-213,554r abnr:2247 satznr:7415“, Man hat etwa die Vorstellung von einem Motor , der erst leer geht , und dann eine Arbeitsmaschine treibt . ]

['ana=facs:Ts-213,213r abnr:1095 satznr:3455“, Könnte eine Maschine denken ? – – – Könnte sie Schmerzen haben ? In dem Sinne in welchem der tierische Körper Schmerzen hat . – ja Wenn ich diesen eine Maschine nennen will . ]

['ana=facs:Ts-213,195r abnr:981 satznr:3138“, Uns interessiert die Sprache als Phänomen , nicht als die Maschine , die einen bestimmten Zweck erfüllt . ]

['ana=facs:Ts-213,473r abnr:2018 satznr:6554“, Zwei Farben , zwei Dampfspannungen , zwei Geschwindigkeiten , zwei elektrische Spannungen , haben nicht zugleich an einem Punkt Platz . – ]

['ana=facs:Ts-213,227r abnr:1163 satznr:3613“, Wozu berechnet er Dampfkessel und überläßt ihre Wandstärke nicht dem Zufall ? ]

['ana=facs:Ts-213,407r abnr:1822 satznr:5889“, Die Arbeit an der Philosophie ist – wie vielfach die Arbeit in [ ? ] der Architektur – eigentlich mehr eine Arbeit an Einem selbst . ]

['ana=facs:Ts-213,161r abnr:834 satznr:2621“, So , als sähen wir ein Ergebnis des logischen Prozesses . ]

['ana=facs:Ts-213,737r abnr:2682 satznr:9232“, Wie es sich nun mit derjenigen Allgemeinheit , mit den Sätzen der Mathematik verhält , die nicht von “ allen Kardinalzahlen ” , sondern , z.B. . von “ allen reellen Zahlen ” handeln , kann man nur erkennen , wenn man diese Sätze und ihre Beweise untersucht . ]

['ana=facs:Ts-213,765r abnr:2755 satznr:9549“, Könnten die Berechnungen eines Ingenieurs ergeben , daß die Stärken eines Maschinenteils bei gleichmäßig wachsender Belastung in der Reihe der Primzahlen fortschreiten müssen ? ]

['ana=facs:Ts-213,149v abnr:780 satznr:2474“, Wie seltsam : es scheint als ob zwar eine physische ( mechanische ) Führung versagen , unvorhergesehenes zulassen , könnte , aber eine Regel nicht ! ]

['ana=facs:Ts-213,702r abnr:2593 satznr:8901“, Das Verhältnis der bei& lt;lb rend=ßhyphen”/& gt;den kann man sich an der Maschine klarmachen , die Schraubenfedern erzeugt . ]

# Inhalt der beigelegten CD

- Die Arbeit als PDF-Version
- Die Arbeit als LaTeX-Version
- Die entwickelten Programme
- Zitierte Artikel
- Verwendete Abbildungen
- Die Wortliste in der Originalversion