Brandenburgische
Technische Universität
Cottbus - Senftenberg

Realisierung eines Bottom-Up-Parsers für Minimalistische Grammatiken

Bachelorarbeit

zur Erlangung des akademischen Grades
Bachelor of Science (B. Sc.) im Studienfach Informatik

vorgelegt von:	Alike Leonie Ulbricht
Matrikelnummer:	3933078
Erstgutachter:	Prof. Dr.-Ing. habil. Matthias Wolff
Zweitgutachter:	Prof. Dr. rer. nat. habil. Petra Hofstedt
eingereicht in:	Cottbus, am 29. Februar 2024

Abstract

Diese Arbeit beschäftigt sich mit Minimalistischen Grammatiken, ihrem Wortproblem und beschreibt einen neu implementierten Parser für diese.
Sie geht kurz darauf ein, welchen Hintergrund Minimalistische Grammatiken in der Forschung haben, und beschäftigt sich intensiv mit den Grundlagen für formale Sprachen, sowie von Parsern im Allgemeinen. Im letzten Abschnitt gehen wir ausführlich auf einen in Prolog geschriebenen Bottom-Up-Parser ein. Wir behandeln die Eigenschaften des Parsers, die Annahmen bzgl. der Lexika und schließen mit einem Blick auf die Laufzeit ab.

This thesis deals with Minimalist Grammars, their word problem, and describes a newly implemented parser for them. It briefly discusses the background of Minimalist Grammars in research, and deals extensively with the basics of formal languages, as well as parsers in general. In the last section, we elaborate on a Bottom-Up parser written in Prolog. We discuss the properties of the parser, the assumptions regarding the lexicons, and conclude with a look at the runtime.

Eidesstattliche Versicherung

Ich, Alike Leonie Ulbricht, Matrikel-Nr. 3933078, versichere hiermit, dass ich meine Bachelorarbeit mit dem Thema

> *Realisierung eines Bottom-Up-Parsers für Minimalistische Grammatiken*

selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, wobei ich alle wörtlichen und sinngemäßen Zitate als solche gekennzeichnet habe. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Mir ist bekannt, dass ich meine Bachelorarbeit zusammen mit dieser Erklärung fristgemäß nach Vergabe des Themas in dreifacher Ausfertigung und gebunden im Prüfungsamt der Brandenburgische Technische Universität Cottbus-Senftenberg abzugeben oder spätestens mit dem Poststempel des Tages, an dem die Frist abläuft, zu senden habe.

Cottbus, den 29. Februar 2024

Alike Leonie Ulbricht

Inhaltsverzeichnis

1. Einleitung

Sprache kann viele Facetten haben. Wir benutzen sie im Alltag, wir erkennen, wenn Sätze grammatikalisch falsch sind, der Inhalt nicht so wirklich Sinn ergibt oder ob das Gesagte in den Kontext passt. Doch wie schaffen wir das? Es ist zwar nicht das einzige Merkmal, welches uns von anderen Tieren unterscheidet, aber wohl eines der Merkmale, die uns zuerst einfallen.
Mit der Zeit wurden verschiedene Theorien und Modelle aufgestellt, zum Teil wieder verworfen oder erweitert, wie wir Sprache lernen und wie sie verwendet wird. Ein Bereich dieser Theorien, der stark von Noam Chomsky geprägt wurde, sind die generativen Grammatiken. Im Laufe der Zeit haben sich hier besonders zwei Theorien herausgebildet: Ab den 80er Jahren die "Rektions- und Bindungstheorie" *(engl. "Government and Binding Theory")* und in den 90er Jahren das "Minimalistische Programm". Das minimalistische Programm hat viele Modelle und verschiedene Umsetzungen mit unterschiedlichen Schwerpunkten.
Eine dieser Formalisierungen sind die Minimalistischen Grammatiken, auf welche wir in dieser Arbeit noch genauer eingehen wollen. Im Speziellen wollen wir einen Parser betrachten, welcher mit Hilfe der Programmiersprache Prolog Minimalistische Grammatiken umsetzt.

2. Grundlagen

Die Wissenschaft beschäftigt sich schon länger mit der Verarbeitung von Sprache. Warum schaffen es schon kleine Kinder grammatikalisch richtige Sätze zu bilden, obwohl sie die dazugehörigen Regeln nicht kennen?
Wie lernen wir Sprache? Und wie kann das mit zum Beispiel Computern reproduziert werden?

Anfang der 80er Jahren brachte Noam Chomsky die Idee einer Universellen Grammatik (UG) auf. Die Grundidee dieser ist, dass wir in unserem Erbgut eine Universelle Grammatik haben, in der das Grundgerüst aller natürlichen Sprachen codiert ist. Das mag auf den ersten Blick weit hergeholt klingen, doch basiert diese Idee auf den Eigenschaften natürlicher Sprachen. Untersuchen wir zwei beliebige Sprachen können wir bereits einige Merkmale feststellen, in denen sie sich stark ähneln oder sogar gleichen. Somit liegt die Idee nahe, dass es eine "übergeordnete" Grammatik für alles Sprachen gibt, die alle identischen Eigenschaften und für alle weiteren Eigenschaften Auswahlmöglichkeiten beinhaltet, zwischen denen man sich nur noch entscheiden muss.
So schön diese Idee auch ist, wird sie mit den Jahren immer unwahrscheinlicher Dabrowska [2015]. Abgesehen davon, dass es an allen Ecken und Enden Kritik an dieser Idee gibt, spricht gegen sie, dass bis heute keine genaue Definition der Universellen Grammatik existiert Dabrowska [2015]. Es wurde weder eine UG gefunden, der mehr als eine Hand voll zustimmen, noch konnte bis jetzt festgestellt werden, wie groß denn so eine Universelle Grammatik wäre.
Unabhängig davon, ob die Universalgrammatik existiert oder nicht, ist sie die Grundlage von Chomskys generativer Grammatik im Feld der Linguistik. So ist das Ziel der Rektions- und Bindungstheorie die Regeln der Universalgrammatik zu beschreiben.
Im Gegensatz dazu wendet sich die Weiterentwicklung, das Minimalistische Programm, stark von diesen Prinzipien und Parametern ab, ohne die Universalgrammatik ganz aus den Augen zu verlieren.

2.1. Sprachen und Grammatiken

Um im Anschluss besser über Sprache sprechen zu können, halten wir kurz ein paar Definitionen fest, die in der restlichen Arbeit verwendet werden.
Die Definition im Folgenden benutzt die Form von Noam Chomsky. Diese ist besonders in der Theoretischen Informatik weit verbreitet Hromkovi˘c [2011]; Vossen [2016]. Wir stützen uns im folgenden Verlauf auf diese Definitionen.
Dieser unterteilt anhand verschiedener Merkmale formale Sprachen in unterschiedliche Klassen. Die daraus resultierende Chomsky-Hierarchie besitzt vier verschiedene Typen, auf die wir im Zusammenhang mit Grammatiken genauer eingehen wollen.

Unabhängig von der Unterteilung in diese Klassen werden alle über ein Alphabet und dazugehörige Wörter definiert:

Definition. *Eine endliche nichtleere Menge Σ heißt **Alphabet**. Die Elemente eines Alphabets werden Buchstaben (Zeichen, Symbole) genannt.*

Definition. *Sei Σ ein Alphabet. Ein **Wort** über Σ ist eine endliche (eventuell leere) Folge von Buchstaben aus Σ. Das leere Wort λ ist die leere Buchstabenfolge. (Manchmal wird ϵ statt λ benutzt.)*

Σ^* ist die Menge aller Wörter über Σ.

Definition. *Eine **Sprache** L über einem Alphabet Σ ist eine Teilmenge von Σ^*.*

Im Konkreten gibt es verschiedene Möglichkeiten Sprachen zu definieren und zu nutzen. Alle Wörter der Sprache aufzuzählen mag eine valide Variante sein, doch ist dies besonders bei großen Sprachen sehr zeitaufwändig, wenn nicht sogar unmöglich. Typ 3 der Chomsky-Hierarchie ist die erste Klasse die wir an der Stelle genauer ansehen. Sie besteht aus den regulären Sprachen und ist damit die Sprache, die am meisten eingeschränkt ist. Reguläre Sprachen können auf verschiedene Weisen definiert werden, dazu gehören:

- Aufzählung von allen Wörtern in der Sprache
- Reguläre Ausdrücke
- Endliche Automaten

- Rechtslineare Grammatik

Reguläre Ausdrücke stellen mit Hilfe einer kurzen Formel regulären Sprachen dar. Diese können nur auf diese Sprachklasse angewendet werden. Endliche Automaten sind abstrakte Maschinen, welche nur durch ein endliches Alphabet, endlich viele Zustände sowie Zustandsübergänge zwischen diesen Zuständen arbeiten. Alle diese Beschreibungen für Sprachen sind gleichwertig, wobei diese beiden hier nicht weiter erläutert werden.

Definition. *Eine **Grammatik** G ist ein 4-Tupel G = (N, T, P, S), wobei*

(i) *N ein **Nichtterminalalphabet** ist.*

(ii) *T ein **Terminalalphabet** ist. Es gilt $N \cap T = \emptyset$.*

(iii) *$S \in N$ das **Startsymbol** ist.*
(Jede Generierung eines Wortes muss mit dem Wort S starten.)

(iv) *$P \subseteq (T \cup N)^* \cdot N \cdot (T \cup N)^* \times (T \cup N)^*$ die **Menge der Ableitungsregeln** oder auch **Produktionsregeln** von G ist. Statt $(\alpha, \beta) \in P$ schreiben wir $\alpha \to \beta$*
(Mit einer Produktionsregel $(\alpha, \beta) \in P$ kann innerhalb einem Wortes α durch β ersetzt werden.)

Definition. *Eine Grammatik G heißt regulär, rechtslinear oder Typ-3-Grammatik, wenn gilt:*

$$P \subseteq N \times (T^* \cdot N) \cup T^*$$

Beispiele für diese Sprachen sind:

L=$\{a^n | n \in \mathbb{N}\}$, mit der regulären Grammatik
$G = (\{S\}, \{a\}, \{S \to aS, S \to a\}, S)$.

L=$\{a^n b a^m | n, m \in \mathbb{N}\}$, mit der regulären Grammatik
$G = (\{S, A\}, \{a, b\}, \{S \to aS, S \to bA, A \to a\}, S)$.

Als Gegenbeispiel kann L=$\{a^n b^n | n \in \mathbb{N}\}$ genannt werden. Es gibt keinerlei Möglichkeit zu beobachten, wie viele a's geschrieben wurden. Somit kann die Anzahl an b's dahingehend auch nicht angepasst werden.

Grammatiken werden bei regulären Sprachen nicht sonderlich häufig verwendet, da reguläre Ausdrücke deutlich kompakter die gerade betrachtete Sprache beschreiben. Anders als Reguläre Sprachen bieten uns Kontextfreie Sprachen ein wenig mehr Freiheiten. Insbesondere können diese bereits für Programmiersprachen verwendet werden. Sie haben genug Komplexität um auch kompliziertere Programme zu beschreiben und sind gleichzeitig eingeschränkt genug um eine einfache Parsierung zu ermöglichen.
So gehört die zuvor genannte Sprache L=$\{a^n b^n | n \in \mathbb{N}\}$ zwar nicht zu den Regulären und kann somit nicht durch Endliche Automaten oder Regulären Ausdrücken dargestellt werden, doch genügt eine kleine Änderung am Konzept des Endlichen Automaten: Die Einführung eines Stacks, um daraus einen Kellerautomaten zu machen. Diese Veränderung ermöglicht es uns die gerade beschriebene Sprache darzustellen. Definierend besitzen kontextfreien Sprachen die Eigenschaft, dass sie durch einen Kellerautomaten oder äquivalent durch eine kontextfreie Grammatik beschrieben werden können.

Definition. *Eine Grammatik G heißt kontextfrei oder Typ-2-Grammatik, wenn gilt:*

$$P \subseteq N \times (N \cup \Sigma)^*$$

Ein Beispiel an der Stelle ist die Sprache L=$\{a^n b^n | n \in \mathbb{N}\}$ mit ihrer kontextfreien Grammatik G = $(\{S\}, \{a, b\}, \{S \to aSb, S \to ab\}, S)$.
Ein Gegenbeispiel an dieser Stelle ist die Sprache L=$\{a^n b^n c^n | n \in \mathbb{N}\}$, da wir hier die Möglichkeit benötigen mehrere Nichtterminalsymbole oder eine Kombination aus Nichtterminalen und Terminalsymbolen auf der linken Seite durch Andere zu ersetzen.

Diese Fähigkeit besitzen Sprachen des Typs 1, welche auch kontextsensitiv genannt werden.

Definition. *Eine Grammatik G heißt kontextsensitiv oder Typ-1-Grammatik, wenn gilt:*

$$P \subseteq (N \cup \Sigma)^* \cdot N \cdot (N \cup \Sigma)^* \times (N \cup \Sigma)^*$$

Das ermöglicht es uns, auch Sprachen wie L=$\{a^n b^n c^n | n \in \mathbb{N}\}$ zu beschreiben. Zum Beispiel mit der Grammatik G = $(\{S, B\}, \{a, b, c\}, \{S \rightarrow aSB, S \rightarrow aB, B \rightarrow bc, cB \rightarrow Bc\}, S)$.

2.1.1. Sprachklassen und das Wortproblem

Nachdem wir uns mit den Eigenschaften von formalen Sprachen beschäftigt haben, wollen wir einen Schritt weiter gehen und zurück zu der Frage kommen, die sich bereits am Anfang dieser Arbeit stellte. Sobald wir eine Sprache und ein Wort gegeben haben, intressiert uns ob das Wort Teil der Sprache ist, oder nicht. Diese Frage wird auch Wortproblem genannt und wurde bereits ausführlich für alle genannten Sprachklassen untersucht.

Definition. *Für eine Sprache L und ein Wort $w \in \Sigma^*$ ist das **Wortproblem**, ob $w \in L$ gilt, oder nicht.*
Für eine Sprache L im Allgemeinen ist das Wortproblem, ob es einen (terminierenden) Algorithmus gibt, der für jedes Wort $w \in \Sigma^*$ das Wortproblem löst.

Für rekursiv aufzählbare Sprachen ist das Wortproblem nur semi-entscheidbar.Es gibt einen Algorithmus der endet und bestimmen kann, wenn ein Wort in der Sprache liegt, aber nicht unbedingt endet, wenn das Wort nicht in der Sprache liegt. Konzeptionalist das Problem, dass er nicht entscheiden kann, ob es nur ein paar mehr Operationen benötigt um zu erkennen, dass das Wort doch in der Sprache ist. Hingegen sind kontextsensitive Sprachen endscheidbar und es gibt einen Algorithmus, der in maximal exponentieller Zeit bestimmen kann, ob das Wort in der Sprache liegt, oder nicht. Da sowohl kontextfreie, als auch reguläre Sprachen eine echte Teilmenge der kontextsensitiven Sprachen sind (siehe Abbildung 2.1), sind diese auch entscheidbar. In der Chomsky-Hierarchie sind auch die Minimalistischen Grammatiken zwischen den Kontextfreien und Kontextsensitiven einzuordnen, also ist das Wortproblem auch hier (in exponentieller Zeit) entscheidbar. Eine Möglichkeit dies zu sehen ist es, MGs in Multiple Kontextfreie Grammatiken (MCFGs), linear context free rewrite systems (LCFRSs) oder andere Grammatiken umzuformen, für die die Inklusion ersichtlich ist Stabler [2010].
Da es sich bei der Minimalistischen Grammatik um eine Formalisierung für natürliche Sprachen handelt, ist die entscheidbarkeit auch intuitiv nachvollziehen. Natürliche Sprache besitzt bekannter Weise nicht nur eine bestimmte Form, wie im

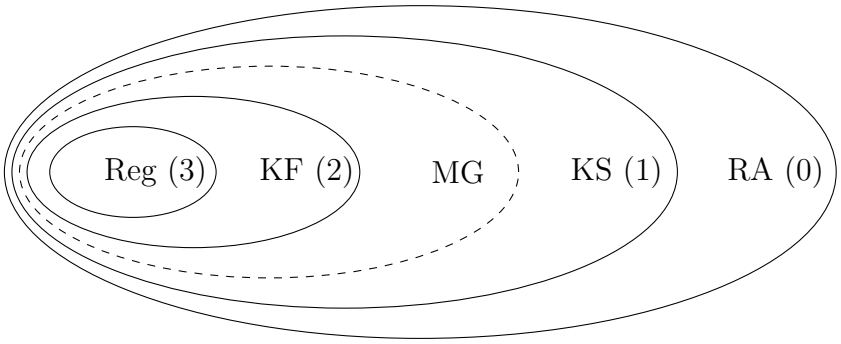

Abb. 2.1.: Einordnung von MGs in die Chomsky-Hierarchie

deutschen die Satzordnung SVO, sondern kann auch unter bestimmten Umständen andere Formen annehmen wie Fragesätze, den Einschub von Nebensätzen oder Wörter deren Gebrauch weitere Wörter oder Änderungen des Satzes nach sich ziehen. Folglich können wir nicht davon ausgehen, dass es sich bei natürlichen Sprachen um kontextfreie Sprachen handelt. Hingegen besitzen einfache Sätze deutliche Ähnlichkeiten zu kontextfreien Sprachen und könnten durch eine Kontextfreie Grammatik dargestellt werden.

2.2. Minimalistische Grammatik

Wir haben jetzt alle Begriffe und Definitionen geklärt, um uns genauer mit der Minimalistischen Grammatik auseinander zu setzen. Wie bereits beschrieben ist sie eine Formalisierung des Minimalistischen Programms von Noam Chomsky.
Minimalistische Grammatiken sind im Gegensatz zu normalen kontextfreien oder kontextsensitiven Grammatiken besonders, da sie nicht die klare Aufteilung in ein Terminalalphabet, ein Nichtterminalalphabet mit einem Startsymbol und eine Menge von Produktionen besitzen, sondern durch wenige, allgemeine Regeln definiert wird. Diese Regeln werden Merge und Move genannt.
Ergänzend dazu erhält sie ein Lexikon, welches im abstrakten Sinne dem Terminalalphabet eine Menge von Nichtterminalsymbolen zuordnet. Genauer kann eine Minimalistische Grammatik wie folgt, nach Stanojević [2016], definiert werden:

Definition. *Eine minimalistische Grammatik G ist ein 6-Tupel $G = (\Sigma, B, Typ, Lex, c, F)$, wobei:*

(i) Σ *ein nicht leeres* ***Alphabet*** *ist.*

(ii) B ***Eigenschaften***, *mit*

- $selectors = \{=f \mid f \in B\}$
- $selectees = \{f \mid f \in B\}$
- $licensors = \{+f \mid f \in B\}$
- $licensees = \{-f \mid f \in B\}$

und Syn der Vereinigung von selectors, selectees, licensors und licensees sind.

(iii) Typ eine Zuordnung zu (::) oder (:) ist.

(iv) $E = C^+$ *die Menge aller Ausdrücke mit* $C = \Sigma^* \times Typ \times Syn^*$.

(v) $Lex \subseteq E$ *ein* ***Lexikon*** *ist.*
$(\Sigma^* \times \{::\} \times (selectors \cup licensors)^* \times selectees \times licensees^*)$

(vi) $c \in B$ *ein* ***vollständiger Ausdruck*** *oder auch die* ***Startkategorie*** *ist.*

(vii) $F = \{merge, move\}$ *die* ***Funktionen*** *(F: $E^* \rightarrow E$) sind.*

Ein Wort, wie zu Beginn dieses Kapitels definiert, kann auch für längere Texte stehen, 2.1 anders als das Verständnis von Wörtern in natürlichen Sprachen. Da wir in der folgenden Arbeit meist mit natürlichen Sprachen arbeiten, werden wir in der folgenden Arbeit Satz und Wort gleichsetzen, sowie anstatt Buchstaben Wörter verwenden. Im Gegensatz zu formalen Grammatiken sind die Ableitungsregeln Merge und Move in Minimalistischen Grammatiken Bottom-Up definiert, was es uns an dieser Stelle erleichtert, da wir keine Alternative definieren müssen Stanojević [2016].

Definition. ***merge****: $(E \times E) \to E$ ist die Vereinigung der folgenden Funktionen, mit $s,\ t \in \Sigma^*,\ \cdot \in \{:, ::\},\ f \in (selectors\ \cup\ selectees),\ \gamma \in B^*,\ \delta \in licensees^+$, und $\alpha_1, \ldots, \alpha_k, \iota_1, \ldots, \iota_l\ (0\ \leq\ k, l)$:*

$$\textit{merge 1} \qquad \frac{s ::= f\gamma \qquad\qquad t \cdot f, \alpha_1, \ldots, \alpha_k}{st : \gamma, \alpha_1, \ldots, \alpha_k}$$

$$\textit{merge 2} \qquad \frac{s := f\gamma, \alpha_1, \ldots, \alpha_k \qquad\qquad t \cdot f, \iota_1, \ldots, \iota_l}{st : \gamma, \alpha_1, \ldots, \alpha_k, \iota_1, \ldots, \iota_l}$$

$$\textit{merge 3} \qquad \frac{s\cdot = f\gamma, \alpha_1, \ldots, \alpha_k \qquad\qquad t \cdot f\delta, \iota_1, \ldots, \iota_l}{st : \gamma, \alpha_1, \ldots, \alpha_k, t : \delta, \iota_1, \ldots, \iota_l}$$

Definition. ***move****: $E \to E$ ist die Vereinigung der folgenden Funktionen, mit $s,\ t \in \Sigma^*,\ \cdot \in \{:, ::\},\ f \in (licensors\ \cup\ licensees),\ \gamma \in B^*,\ \delta \in licensees^+$, und $\alpha_1, \ldots, \alpha_k, \iota_1, \ldots, \iota_l\ (0\ \leq\ k, l)$:*

$$\textit{move 1} \qquad \frac{s : +f\gamma, \alpha_1, \ldots, \alpha_{i-1},\ t : -f, \alpha_{i+1}, \ldots, \alpha_k}{ts : \gamma, \alpha_1, \ldots, \alpha_{i-1}, \alpha_{i+1}, \ldots, \alpha_k}$$

$$\textit{move 2} \qquad \frac{s : +f\gamma, \alpha_1, \ldots, \alpha_{i-1},\ t : -f\delta, \alpha_{i+1}, \ldots, \alpha_k}{s : \gamma, \alpha_1, \ldots, \alpha_{i-1},\ t : \delta, \alpha_{i+1}, \ldots, \alpha_k}$$

Dabei soll $-f$ für genau ein α_i der erste licensee sein. Diese Bedingung nennen wir auch Shortest Move Condition (SMC).

Merge und Move sind hier klar spezifiziert. Aus diesen Beschreibungen können wir Regeln ableiten. So gilt für Merge 1, dass die Seite mit dem Selektor ($\in$ *selectors*) Element des Lexikons sein muss. Ebenso darf die Kategorie ($\in$ *selectees*) keine Eigenschaften außerhalb von f besitzen. Betrachten wir die Reihenfolge innerhalb des

Satzes können wir ebenfalls beobachten, dass die Kategorie immer direkt rechts neben dem Selektor steht.
Betrachten wir Merge 2, können hier ähnliche Beobachtungen machen. So befindet sich der Teil mit der Kategorie immer links von dem Teil mit dem Selektor. Wie zuvor darf die Kategorie keine weiteren Eigenschaften außer f besitzen. Allerdings darf, im Gegensatz zu Merge 1, der Selektor mehr als ein Buchstaben des Alphabetes besitzen, da dem Merge 2 auf der Seite des Selektors mindestens ein anderer merge vorausgegangen sein muss. Buchstaben können auch Wörter natürlicher Sprachen sein.
Ein Move ohne vorheriges Merge kann nicht passieren. Die dafür erforderliche Stellung, in Form einer Kette, kann nur durch ein vorheriges Merge 3 erreicht werden. Move 1 und Move 2 können durch die Anzahl von Eigenschaften auf Seiten des Licensees unterschieden werden. Liegt mindestens eine Weitere als die gerade Betrachtete vor, so kann die Kette belassen werden und es muss auf SMC geprüft werden. Liegt hingegen nur noch eine Eigenschaft vor, so bewegt sich die Seite mit dem Licensee vor den Satzteil mit dem Licensor. Auf SMC muss hier nicht mehr geprüft werden. Merge 3, welche die Kettenstellung hervorruft, besitzt ein paar andere Abgrenzungsmöglichkeiten. Sobald der Satzteil mit der Kategorie mehr als eine Eigenschaft besitzt, können wir ausschließen, dass es sich um Merge 1 oder Merge 2 handelt. Wir können zusätzlich aussagen, dass sich die Kategorie immer vor den Selektor positionieren wird, aufgrund von Move 1 mit dem die Kettenstellung immer beendet wird. Bemerkenswert an dieser Stelle ist, dass Merge 3, im Unterschied zu Merge 1 und 2, nicht zwangsläufig zwei nebeneinander liegende Wörter miteinander verbindet. Basierend auf der Reihenfolge des gegebenen Satz kann somit nicht nachvollzogen werden, an welcher Stelle sich der zweite Ausdruck, beginnend mit der Kategorie, befindet Stanojević [2016].
Eine weitere Besonderheit von Minimalistischen Grammatiken ist die Verwendung von leeren Wörtern. Diese sind ebenso wie alle weiteren Elemente des Alphabets im Lexikon definiert. Hierbei kann es zu mehreren Vorkommen von Leeren Wörtern geben, jedes mit unterschiedlichen Eigenschaften. Jedes Element des Alphabetes ist nicht nur auf ein Vorkommen innerhalb des Lexikons beschränkt. Es kann verschiedenen Eigenschaften zugeordnet werden, um die Vielfalt einer Sprache aufzuzeigen und dabei klar zu definieren, welche Sätze Teil der Sprache sind und welche nicht. Leere Wörter unterstützen die Vielfalt einer Sprache, und vermeiden dass man einem großen Teil des Alphabetes die selben Eigenschaften geben muss. So können beliebig viele Leere Wörter zwischen zwei Wörter des Satzes hinzugefügt werden um

die gegebenen Eigenschaften des aktuellen Ausdrucks zu verändern.
Die Anzahl der möglichen Leeren Wörtern, zwischen zwei Wörtern des Satzes, ist bei den getesteten Grammatiken aufgrund der Größe des Lexikons kein Problem. Doch kann dies ohne Vorverarbeitung bei großen Lexika zu einem Problem der Laufzeit bei einigen Algorithmen kommen. Alle Lexika sind in Kombination mit den Ableitungsregeln entscheidbar, da sie [2.1.1] in den Kontextsensitiven Sprachen enthalten sind, welche entscheidbar sind. Zu beachten ist allerdings an dieser Stelle, dass viele Leere Wörter zu einer langen Laufzeit führen, wenn diese nicht zuvor ausgewertet werden. 3.4 Desweiteren gibt es die Möglichkeit von Schleifen innerhalb der leeren Wörter. Dieses Problem muss der Parser ebenfalls addressieren, sei es durch einen Zähler, der allerdings auch mögliche Ableitungen begrenzt, durch eine Vorauswertung, welche Schleifen erkennt oder intern durch die Erkennung von Schleifen.

2.3. Parser

Um ein bestimmtes Wort als Teil der Sprache zu identifizieren, muss die zum Wort gehörende Ableitung gefunden werden. Dies zu automatisieren ist das Hauptziel eines Parsers. Abbildung 2.2 ist ein Ableitungsbaum, und wird auch Syntaxbaum genannt. Dieser stellt eine Ableitung eines Wortes grafisch dar. Dabei ist die Wurzel das Startsymbol der Grammatik und alle Blätter entsprechen einem Terminalsymbol. Eine passende Linksableitung zu diesem Satz wäre:

$S \to AS \to aS \to aSA \to aBA \to abA \to aba.$

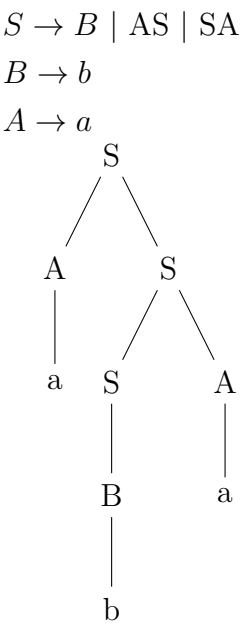

Abb. 2.2.: Syntaxbaum "*aba*"

Ableitungen können beliebig durchgeführt werden. Wird allerdings immer das am weitesten links stehende Nichtterminalsymbol ersetzt, so handelt es sich um eine Linksableitung. Analog wird bei eine Rechtsableitung immer das Nichtterminalsymbol ersetzt, was am weitesten rechts steht. Ist das Wort Teil der Sprache, so gibt es mindestens eine Ableitung und damit mindestens einen Syntaxbaum. Dabei spiegelt der Ableitungsbaum nur die Wahl der Regeln wieder und in keiner Weise in welcher Reihenfolge diese ausgewählt wurden. So können wir bereits in Abbildung 2.2 nicht nachvollziehen, ob es sich um eine Links- oder Rechtsableitung handelt. Gibt es mehrere Linksableitungen zum selben Wort, so handelt es sich um eine mehrdeutige Grammatik. Besonders bei Programmiersprachen, welche Parser zur Übersetzung in andere Sprachen nutzt, kann Mehrdeutigkeit zum Problem werden und sollte unbedingt vermieden werden. Soll ein Syntaxbaum erstellt werden, gibt es intuitiv zwei Möglichkeiten. Einerseits kann von unten nach oben gearbeitet werden, wobei die Blätter zuerst genauer betrachtet werden und anschließend Stück für Stück zu einem Baum zusammengesetzt werden, und auf der anderen Seite wird an dem Wurzelknoten begonnen und von dort aus ausprobiert, welche der Regeln anzuwenden sind um den gegebenen Satz zu erhalten. Ersteres beschreibt das Konzept eines Bottom-Up-Parsers, Letzteres das Konzept eines Top-Down Parsers.

2.3.1. Bottom-Up Analyse

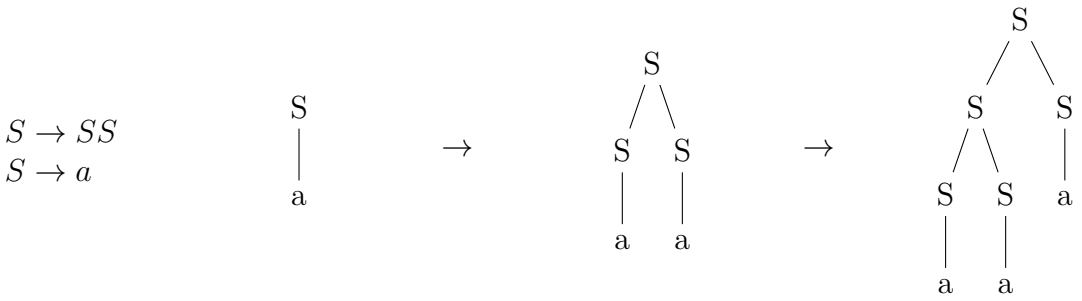

Abb. 2.3.: Bottom-Up Parsierung am Beispiel des Wortes 'aaa' der Sprache L=$\{a^n | n \in \mathbb{N}\}$

Unabhängig welche Implementierung betrachtet wird, baut der Parser der Ableitungsbaum von unten nach oben auf (Abbildung 2.3). Wird hierbei von links nach rechts von der Eingabe aus gearbeitet, so handelt es sich um eine Rechtsableitung. Dies ist der Fall, da Ableitungen vom Startsymbol aus betrachtet werden und damit das Nichtterminalsymbol am weitesten rechts zuerst ersetzt wird.
Eine einfache Implementierung eines Bottom-Up-Parsers benötigt neben der Eingabe einen Stack als Datenstruktur und nutzt folgende vier Grundfunktionen:

- **shift**. Nimm das nächste Symbol der Eingabefolge und lege es auf den Stack.
- **reduce**. Die obersten k Zeichen des Stacks bilden die rechte Seite einer Ableitung $A \to \alpha$. Ersetze α durch A auf dem Stack.
- **accept**. Die Eingabefolge ist leer und nur das Startsymbol liegt auf dem Stack. Die Eingabefolge wird akzeptiert.
- **error**. Ein Syntaxfehler liegt vor.

Aufgrund der ersten beiden Funktionen wird ein Bottom-Up-Parser auch Shift-Reduce-Parser genannt.

Eingabe	Stack	Aktion
aaba	|	shift
aba	|a	reduce $(A \to a)$
aba	|A	shift
ba	|Aa	reduce $(A \to a)$
ba	|AA	shift
a	|AAb	reduce $(B \to b)$
a	|AAB	reduce $(S \to B)$
a	|AAS	reduce $(S \to AS)$
a	|AS	reduce $(S \to AS)$
a	|S	shift
ϵ	|Sa	reduce $(A \to a)$
ϵ	|SA	reduce $(S \to SA)$
ϵ	|S	accept

Ein Beispiel mit dem Wort 'aaba' der Sprache L=$\{a^n b a^m | n, m \in \mathbb{N}\}$

$S \to AS \mid SA \mid B$
$B \to b$
$A \to a$

Ein einfacher Parser, der diesem Konzept folgt sind die LR(k)-Parser Erwig [1999]. Die Abkürzung steht für "**L**inks nach rechts lesend und eine **R**echtsableitung schreibend. Dabei auf die nächsten **k**-Zeichen voraus schauend". Das besondere bei einem LR(k)-Parser ist, dass es sich hier um eine Art Bottom-Up-Parser handelt, welcher kein Backtracking besitzt. Hierbei werden Ableitungen deutlich schneller gefunden, da keine Fehlentscheidungen getroffen werden, welche durch Backtracking rückgängig gemacht werden müssten.
Dies erfordert Vorverarbeitung, denn es muss eine Parsierungstabelle erstellt werden. Diese ordnet für jede Kombination aus Terminalsymbolen und Zuständen, in denen sich der Parser aufgrund der fortschreitendenden Analyse befinden kann, eine Funktion und Folgezustand zu. Je größer die Grammatik ist, desto größer wird auch diese Analysetabelle. Aufgrund dessen wird sie meistens durch speziell entwickelte Programme erstellt.

Eine Alternative, ohne Backtracking zu arbeiten, bietet Dynamische Programmierung. Wir können uns Vorstellen bei jeder Entscheidung die wir treffen müssen, alle Varianten auszuführen und an diesen Zwischenergebnissen parallel zu arbeiten. Sind wir am Ende angekommen können wir eine der erfolgreichen Parsierungen auswählen.

Der CYK-Algorithmus ist ein anderes Beispiel Dynamischer Programmierung Grune und Jacobs [2008]. So wird für jedes Element der Eingabefolge einzelnd Ableitungen gebildet. Anschließend werden die möglichen, nebeneinander liegenden Nichtterminalsymbole betrachtet, ob es Ableitungen gibt, welche diese produzieren. Dies wird so lange erweitert, bis das ganze Wort betrachtet wird und fest steht, ob dieses Wort geparst werden kann. Bemerkenswert ist hier, dass nach dem ersten Durchlauf nur die Nichtterminale betrachtet werden müssen und diese jede Stufe weniger werden. Aus der entstandenen Tabelle kann anschließend abgelesen werden, wie ein Syntaxbaum auszusehen hat.

Im Gegensatz dazu ist ein Algorithmus, welcher auf Tiefensuche beruht und nicht auf Breitensuche wie Dynamische Programmierung, deutlich langsamer. Dieser erfordert die Möglichkeit für Backtracking, hat aber keine hohen Ansprüche an zusätzlichen Speicherplatz.

2.3.2. Top-Down Analyse

Auch eine Top-Down-Analyse bietet die Möglichkeit Parser mit und ohne Backtracking umzusetzen. Wird ein Parser ohne Backtracking erfordert, muss in den meisten Fällen die Grammatik entsprechend angepasst werden. Ohne diese Maßnahmen können wir einen einfachen Top-Down-Parser mit Backtracking implementieren, der vom Startsymbol ausgehend eine linke Regelseite erkennt und diese durch ihre rechte Seite ersetzt.
Aufgrund der Eigenschaft vom Startsymbol aus zu arbeiten und nur mit dem Wissen der nächsten k Elemente der Eingabefolge, kann der Top-Down-Parser falsch entscheiden wenn die Grammatik Linksrekursion enthält oder nicht Linksfaktorisiert ist. Ersteres führt schnell zu Schleifen, an deren Stelle unklar ist, wann abgebrochen werden muss, oder ob dieser Weg überhaupt korrekt ist. Bei fehlender Linksfaktorisierung besitzt die Grammatik mehrere Produktionen, dessen rechte Seite den selben Präfix aufweist. Dies kann zur falschen Regelauswahl führen und somit Backtracking nach sich ziehen.

Auch hier gibt es verschiedene Umsetzungen, Beispielsweise den LL(k)-Parser doch wird darauf an dieser Stelle, aufgrund dem Schwerpunkt auf Bottom-Up-Parsierung, nicht weiter eingegangen.

2.3.3. Parser und Minimalistische Grammatiken

Abgesehen von dem allgemeinen Konzept der Top-Down und Bottom-Up-Analyse gab es in den letzte Jahren einige Implementationen und Ideen bzgl. der Umsetzung von Parsern für Minimalistische Grammiken. Darunter ist ein Inkrementeller Top-Down-Parser, welcher die Minimalistische Grammatik erst in eine equivalente Multiple Context Free Grammar umwandelt und mit dieser weiter arbeitet. Dieser wurde bereits zuvor an diesem Lehrstuhl mit Hilfe von MATLAB implementiert Puchala [2019]; Stabler [2011]. Ebenfalls sehr bekannt ist eine Mischung aus Top-Down- und Bottom-Up-Analyse. Dieser Parser nennt sich Left-Corner-Parser. Er arbeitet direkt mit Minimalistischen Grammatiken und beginnt am ersten Blatt. Von dort arbeitet er sich zum Elternknoten und schließt von dort aus auf den anderen Kindknoten, wie in Abbildung 2.4 zu erkennen. Von hier aus geht es wieder einen Knoten nach oben, von dem aus der andere Kindknoten erschlossen wird Stanojević [2018].

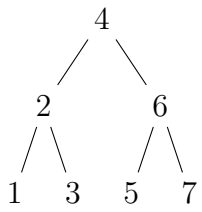

Abb. 2.4.: Aufbau-Verlauf eines Left-Corner-Parsers

3. Implementation

3.1. Aufgabenstellung

Aufgabe dieser Arbeit bestand darin, einen Bottom-Up Parser für Minimalistische Grammatiken zu schreiben in der Programmiersprache Prolog.
Es wurden zum testen sehr kleine Lexika genutzt. A.3 Diese bilden nicht alle korrekten Kombinationen ab, die eine natürliche Sprache mit diesen Wörtern besitzt, doch können viele Fälle getestet werden. Alle Lexika die mir vorlagen beinhalten nur leere Wörter dessen erste Eigenschaft ein Selektor (=X) ist. Daher wurde sowohl die Kombination aus Kategorie und Selektor, als auch nur eine Kategorie als leere Wörter nicht ausführlich getestet. Es besteht allerdings die Annahme, dass dies genauso gut wie mit reinen Selektoren funktioniert.
Zu bemerken ist ebenfalls, dass zu dem aktuellen Zeitpunkt, den 15.02.2024, der Parser noch nicht ganz vollständig ist. Sollte innerhalb der leeren Wörter Kreise enthalten sein ist es durchaus möglich, dass der Parser in diesen Kreis gerät und nicht mehr heraus kommt. Dies könnte durch die Begrenzung der Anzahl an leeren Wörter zwischen zwei Wörtern behoben werden.
Desweiteren ist der Parser noch nicht fähig, erst Teilbäume die größer als zwei Elemente sind zusammen zu setzen und anschließend alles zu verbinden. Auch den anderen Teil eines Merge 3 zu finden, wenn sie im fertigen Satz nicht direkt nebeneinander liegen, ist aktuell nicht möglich. Da beide Probleme eine große Ähnlichkeit zueinander haben, könnte ihre Lösung auf der gleichen Struktur basieren um die der Parser noch erweitert werden kann.

3.2. Architektur

Betrachten wir nun den Parser etwas genauer. Er basiert nicht direkt auf einer anderen Arbeit, da er sich das Rekursive Konzept von Prolog zu nutze macht. Bei Prolog handelt es sich um eine Logische Programmiersprache, welche in der Linguistik zur Anwendung und Prüfung von sprachwissenschaflicher Theorien verwendet wird Lehner [1992].
Daher bietet es sich ebenfalls für die Umsetzung von Minimalistischen Grammatiken an. Es war zu vermuten, dass von den drei Merge Varianten die Erste und Zweite Variante recht einfach umzusetzen ist, im Gegensatz zum Merge 3 in Kombination mit den beiden Move Varianten sowie das implementieren der leeren Wörter. Besonders kompliziert wurde der Teil von Merge 3 eingeschätzt, wo die beiden zusammengehörenden Wörter im fertigen Satz nicht mehr nebeneinander liegen.
Inspiriert wurde er von Stanojević [2016], im Bezug auf die Wahl der Parsierung zwischen chart-based und transition-based. Das Sortieren der Wörter als Vorverarbeitung, sowie die Aufteilung in typische shift und reduce Funktionen in Kombination mit verschiedenen Stacks und Buffern wurde in Betracht gezogen, allerdings aufgrund der Eigenschaften von Prolog verworfen. Inwiefern diese Konzepte den Parser beschleunigen können und ob es elegantere Varianten einer Vorverarbeitung der Eingaben gibt, gilt es noch zu untersuchen.

3.2.1. Annahmen

Fassen wir noch einmal die getätigten Annahmen in Bezug auf die Einträge des Lexikons zusammen:

- Jede Kette ($\geq$ 1) an Eigenschaften eines Wortes des Lexikons besitzt als erstes Element eine Kategorie oder einen Selektor. Lizensoren oder Lizensierer sind nicht erlaubt. Das gilt auch für leere Wörter.
- Die Startkategorie wurde gekennzeichnet.
- Durch die Aneinanderreihung von Leeren Wörtern kann kein Kreis entstehen.

Dadurch wird für jeden Satz ein Baum gefunden, der die zuvor erwähnten komplexen Strukturen nicht besitzt [3.1]. So werden *'Die Katze frisst die Maus'* 3.1 A.3 und *'fünf'* 3.2 erkannt, *'Die Katze frisst die Maus und die Maus frisst den Kaese'*

allerdings nicht, da hier erst *'Die Katze frisst die Maus'* konstruiert werden muss, bevor es mit einem Merge 2 mit *'und die Maus frisst den Kaese'* verbunden werden kann. Dafür ist der aktuelle Parser noch nicht ausgelegt.

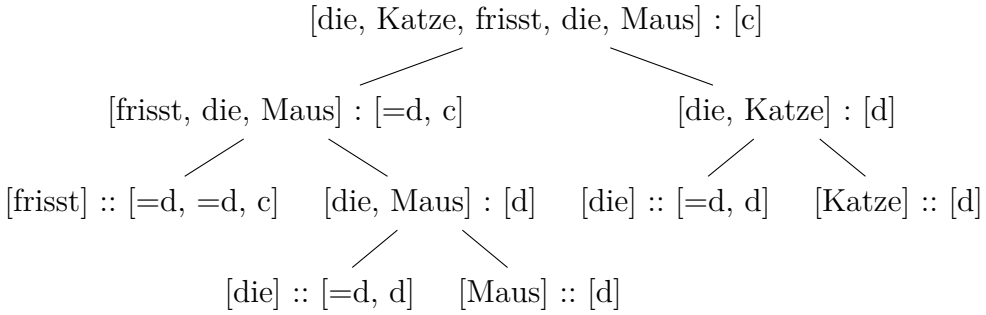

Abb. 3.1.: Ableitungsbaum 'die Katze frisst die Maus'

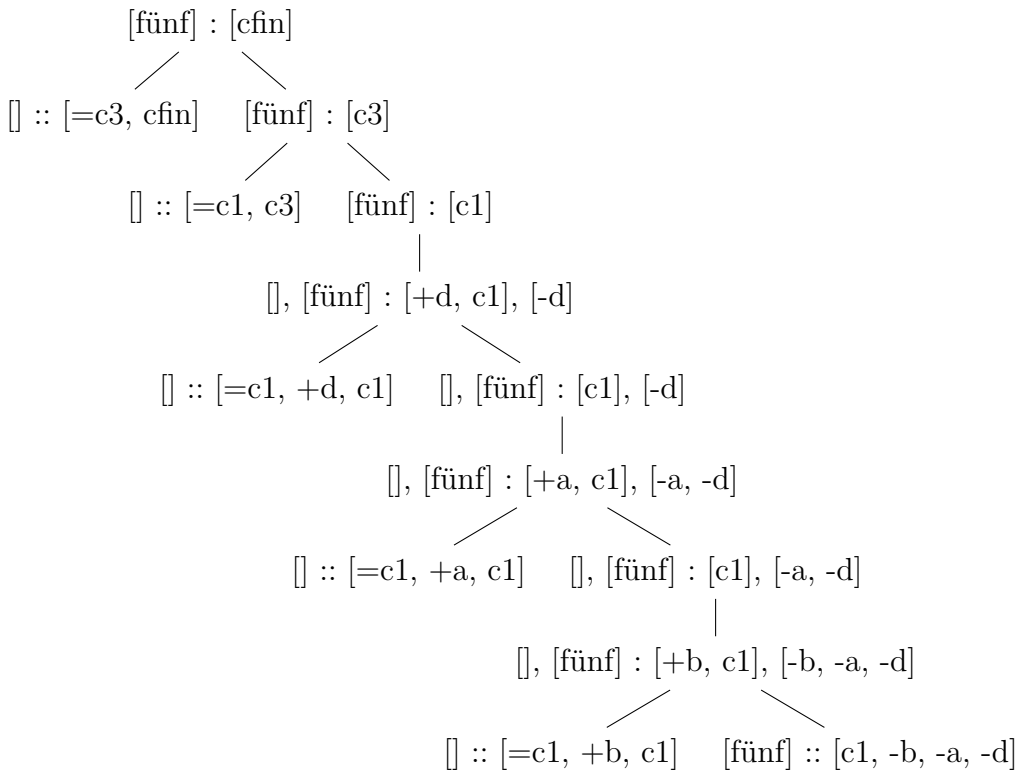

Abb. 3.2.: Ableitungsbaum 'fünf'

3.2.2. Beschreibung des Codes

Der Parser arbeitet rekursiv. Dabei geht er zuerst durch den ganzen Satz um einfache Merge 1 Ableitungen durchzuführen und schließt anschließend die Rekursionsebenen

mit Hilfe von Merge oder Move.
Innerhalb des Codes werden vier verschiedene Grundfunktionen als abstrakte Datentypen verwendet. Zwei dienen zur Ausgabe, die anderen Beiden zur speicherung der in dieser Stufe der Rekursion wichtigen Daten und der Weitergabe an die nächste äußere oder innere Stufe.

- 'li' um alle Blätter des Baumes darzustellen.
  Er besitzt ein Wort und eine Kette von Eigenschaften mit mindestens einem Element.

- 'tree' für alle inneren Knoten inklusive des Wurzelknoten des Baumes.
  Er besitzt eine Kette an bereits geparsten Satzteilen, eine Kette von zugehörigen Ketten an Eigenschaften, einen linken Teilbaum sowie einen rechten Teilbaum.

- 'wort' interne Darstellung eines Wortes, welches neu aus dem Lexikon kommt. Es besitzt ebenfalls das Wort, sowie die Eigenschaften des Wortes. Zusätzlich besitzt er die Position des Wortes innerhalb des gegebenen Satzes.

- 'zusWort' eine Abkürzung für zusammengesetztes Wort, welches für Wörter steht, die bereits mindestens einmal gemerged wurden.
  Er beinhaltet den bereits verbundenen Satzteil, die Kette an Eigenschaften dieses Satzteiles, die Nummer des kleinsten bereits verbundenen Wortes (ohne Kettenglieder), den bis jetzt zusammengesetzen Baum für spätere Darstellung, sowie eine Liste die alle Kettenglieder beinhaltet, die durch Merge 3 entstanden sind.

Der Parsierungsvorgang wird mit Hilfe der *start*-Funktion gestartet, indem ihr der zu parsende Satz übergeben wird. Prolog arbeitet mit Hilfe von pattern matching. Dies erleichtert die Unterscheidung in verschiedene Fälle, da hier von oben nach unten alle Varianten probiert werden, bis eine Variante alle Bedingungen erfüllt oder es keinen Fall mehr gibt, was zum Fehlschlagen der Anfrage führt. Die Startfunktion hat zwei solche Fälle. Sollte unser Satz aus nur einem Wort bestehen und sich bereits in der Startkategorie befinden, so benötigen wir keine weitere Verarbeitung und geben den Parsierungsbaum aus, welcher hier nur aus einem Knoten besteht. Diesen speichern wir anschließend in einer separaten Datei für spätere Verwendung. Befindet er sich nicht bereits in der Startkategorie, so wird die andere Startfunktion aufgerufen. Diese übergibt das erste Wort des Satzes zusammen mit einem leeren

Array an die Hauptfunktion *recursion*. Die Funktion (*recursion*) wird von der Startfunktion nicht mit den ersten beiden Wörtern gestartet. Dadurch ermöglicht er leere Wörter anzufügen egal wie der Parsierungsvorgang endet. Wenn dieser zum Beispiel mit einem Move 1 endet wird nicht als erstes das Move wieder aufgeteilt. Denn das kann dazu führen, dass der fast fertige Parsierungsbaum nicht gefunden wird.
Die Hauptfunktion besitzt wieder mehrere Fälle. Diese decken insbesondere ab, ob noch weitere Wörter im Satz sind, und dadurch weiterhin die Hauptfunktion aufgerufen werden muss, oder nicht. Sie probieren bei jedem Paar an Wörtern diese mit einem einfachen Merge 1 zu verbinden, womit wir kleine Zweige wie in Abbildung 3.1 ableiten können. Die Varianten der Hauptfunktionen unterscheiden sich zusätzlich darin, ob dieses Merge 1 geglückt ist, oder nicht. Ist es gelungen, erhalten wir in der nächsten inneren Rekursionsebene ein zusammengesetztes Wort und ein leeres Array. Dies hat einen einfachen Grund des Formates der Funktion *merge1*. Man könnte weitere Rückgabewerte einbauen um abzufragen ob das Merge geglückt ist, doch entsteht dadurch ein überflüssiges jonglieren an Variablen.
Da wir an dieser Stelle mit dem zusammengesetzten Wort noch nicht weiter arbeiten können, gibt es für beide Fälle, die Mitte und das Ende des Satzes, verschiedene Funktionen. Das folgende Wort liegt im fertigen Satz rechts von uns. Ist es eine Kategorie, müsste das zusammengesetzte Wort ein Selektor sein. Dieser wäre kein lexikales Element mehr, da wir bereits einmal Merge 1 verwendet haben. Daher kann es kein Merge 1 mehr sein. Merge 2 kann mit dieser Reihenfolge von Selektor und Kategorie nicht arbeiten. Tauschen wir Selektor und Kategorie, so ist der Selektor nun das lexikale Element. Merge 2 erfordert allerdings, dass wir bereits mindestens eine Ableitung ausgeführt haben müssen, auf der Seite des Selektors. Dies ist nicht der Fall. Im Gegenzug ist die Reihenfolge jetzt inkorrekt für Merge 1. Da weder Merge 1 noch Merge 2 ausgeführt werden können, aber auch nicht klar ist, ob nach der restlichen Auswertung des Satzes Merge 2 angewendet werden kann, wird dazu übergegangen erst den Rest auszuwerten, bevor an dieser Stelle andere Ableitungsregeln wie Merge 3 oder Leere Wörter ausprobiert werden.
Sind wir am Ende des Satzes angekommen, wird systematisch mit dem pattern matching probiert, welches Merge oder Move zu verwenden ist. Da Move eine Funktion auf nur einer von zwei betrachteten Teilsätze ist, und ohne ein vorheriges Move mögliche Merges nicht zu berechnen sind, wird zuerst überprüft ob diese anwendbar sind. Ist dies der Fall, wird die Funktion *sortMove* aufgerufen und wieder via pattern matching entschieden, ob ein Move 1 oder ein Move 2 vorliegt. Dafür wird die beim zusammengesetzten Wort die hinterlegte Liste durchsucht, welche durch ein Merge

3 entstanden ist. Bei dem Hinzufügen von neuen Wörtern in die Kette durch Merge 3 wird das Shortest Movement Condition überprüft. Daher können wir das erste Element der Liste nehmen, welches die richtige Eigenschaft hat. Findet ein Move 2 statt, wird das SMC ebenfalls überprüft. Schlägt es fehl, wird ein anderer Pfad für einen Ableitungsbaum ausprobiert was dazu führen kann, dass kein Syntaxbaum gefunden wird.
Ist kein Move anzuwenden, wird via pattern matching probiert, welches Move gerade zutreffend ist. Move 1 und 2 besitzen die zuvor erwähnten Einschränkungen, auf die geprüft wird. Ist es keines dieser beiden und besitzt die Seite mit der Kategorie mehr als eine Eigenschaft, so wird zuerst Merge 3 in betracht gezogen. Ist die Kategorie im fertigen Satz hinter dem Selektor, oder ein probiertes Merge 3 schlägt fehl, wird als letztes probiert, Leere Wörter hinzuzufügen. Dies als Letztes zu tun bringt uns den Vorteil, nur an Stellen Leere Wörter einzufügen, die diese wirklich brauchen und verringern so ein wenig die Laufzeit. Bei dem Fall, dass leere Wörter hinzugefügt werden müssen, wird immer das erste Wort zuerst probiert durch leere Wörter zu erweitern. Es wird nicht betrachtet, ob etwas eine Kategorie ist oder nicht, denn je nach Lexikon kann es variieren, wie leere Wörter aussehen und ob sie Selektoren oder Kategegorien sind. Daher kann eines davon nicht ausgeschlossen werden. Besitzen die verwendeten Lexika nur eine Art von leeren Wörtern, könnte der Parser daran angepasst und somit die Laufzeit verringert werden.
Wichtig an dieser Stelle ist, dass die Möglichkeit bestände, immer mehr leere Wörter zu nehmen und nie das Richtige zu erreichen, da der Parser probiert, Leere Wörter mit weiteren Leeren Wörtern zu korrigieren, die bereits zu Beginn die falsche Wahl waren (was der Parser an dieser Stelle nicht wissen kann). Dies ist durch eine einfache Abfrage gelöst. War die ursprüngliche Wahl des leeren Wortes falsch, so wird versucht an ein lexikales, leeres Wort ein weiteres leeres Wort zu setzen. Daher reicht die Abfrage, ob das Wort, welches eine Ergänzung sucht, leer und lexikal ist. Ist dies der Fall wird automatisch abgebrochen und Prolog sucht selbst nach einer Alternative.

3.2.3. Ablauf des Parsers anhand eines Beispiels

Der Vorgang des Parsers kann am Besten anhand eines einfachen Beispieles erklärt werden. Wir wählen an dieser Stelle den Satz *'fünf, undzwanzig'* mit *'fünf :: (c1, -b, -a, -d)'* und *'undzwanzig :: (=c1, +b, c2, -s0)'* als Einträge aus dem Lexikon einer

Zahlengrammatik A.3 für die deutsche Sprache.
Wir beginnen mit der Startfunktion. Es wird überprüft, ob wir nur ein Wort im Satz haben und ob dieses zufällig in der Startkategorie ist. Da dieser Fall bei unserem Satz *'fünf, undzwanzig'* nicht zutrifft, erreichen wir die zweite Startfunktion, welche den Durchlauf des eigentlichen Parsers initiiert.

Unserem Beispielsatz wird nun das erste Wort entnommen und zusammen mit dem Restsatz an die Hauptfunktion *recursion* übergeben. Diese erkennt, dass sie aktuell nur ein Wort und einen Restsatz betrachtet und somit nur beim schließen der Rekursion weiter arbeiten kann. Somit ruft sie sich selbst mit dem ersten Wort *'fünf'* und dem nächsten Wort des Satzes *'undzwanzig'* auf. Der zweite Aufruf der Funktion *recursion* erkennt, dass wir nicht beide Wörter mit der einfachsten Version von Merge 1 verbinden können. Für ein einfaches Merge 1 werden drei Bedingungen benötigt. Der Selektor ist noch ein normales Wort, die Kategorie besitzt nur eine Eigenschaft und der Selektor steht im Originalsatz links von der Kategorie. An dieser Stelle ist nur Ersteres gegeben.
Desweiteren erkennt die Funktion, dass keine weiteren Wörter Teil des Satzes sind. Wäre dies der Fall, würde es sich selbst mit dem zweiten Wort des Satzes und des nächsten Wortes des Satzes aufrufen. Sie würde erneut überprüfen, ob ein einfaches Merge 1 möglich ist, und ob der Satz nun endet. Endet er nicht, wiederholt sich dieses Spiel, bis das Ende erreicht ist. Da der übergebene Satz endlich ist, wird die Rekursion auch enden.
Nach dem erreichen des Endes des Satzes geht der Parser dazu über, alle anderen Merge Varianten zu überprüfen. Dafür wird die Funktion *sortMerge* aufgerufen, welche ebenfalls mit pattern matching die aktuellen Eigenschaften betrachtet. In unserem Fall mit *'fünf :: (c1, -b, -a, -d)'* und *'undzwanzig :: (=c1, +b, c2, -s0)'*, erkennt der Parser, dass die Kategorie mehr als nur eine Eigenschaft hat, was automatisch für Merge 3 spricht. Die Funktion *sortMerge* erreicht die Fallunterscheidung von Merge 3 und probiert hier beide zu verbinden und erhält eine Kette *'(undzwanzig; fünf) : ((+b, c2, -s0); (-b, -a, -d))'*. Diese wird nun erneut der Funktion *sortMerge* übergeben, welche überprüft ob das erste Element unserer Kette ein Move 1 oder Move 2 initiieren kann. Da dies hier der Fall ist, wird die Funktion *sortMove* aufgerufen, welche aufgrund der weiteren Eigenschaften des Elementes in der Kette einen Move 2 ausführt.

An diesem Punkt besitzt unser Parsierungsbaum bereits das Aussehen wie in Grafik 3.3 zu sehen. Der Parser wird nun versuchen den Baum durch leere Wörter zu

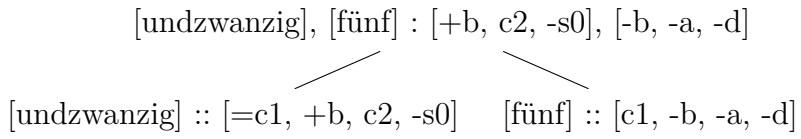

Abb. 3.3.: Syntaxbaum nach erstem Merge 3, 'fünfundzwanzig'

ergänzen um aus der Kette wieder einen vollständigen Satz zu bilden. Aufgrund der gegebenen leeren Wörter des Lexikons wird ihm dies aber nicht, mit Hilfe von der Kategorie *c2*, möglich sein. Nachdem er also alle leeren Wörter durchprobiert hat und gescheitert ist, geht er auch hier wieder zu unserem Ausgangspunkt bevor wir beide Wörter mit Merge 3 verbunden haben zurück. Um schneller zu erkennen, dass ein sofortiges Merge 3 zu einem Fehler führt, ist eine Vorverarbeitung notwendig, welche zu einem späteren Zeitpunkt im Abschnitt Zeitmessungen genauer beschriben wird.

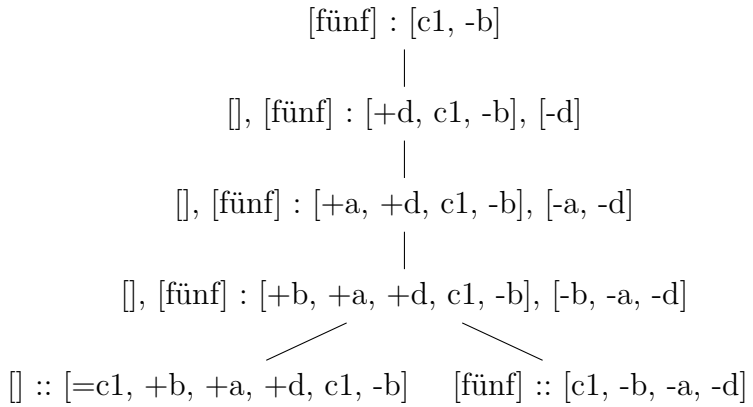

Abb. 3.4.: fünf, nach einer erfolgreichen Ableitung mit leeren Wörtern, 'fünfundzwanzig'

In der Ausgangsposition vor dem Merge 3 ist die einzige andere Möglichkeit des Parsers leere Wörter hinzuzufügen. Der Parser probiert somit zuerst, die Eigenschaften von dem Ersten der zwei Wörter anzupassen. In unserem Fall ist dies auch der korrekte Weg und der Parser erreicht schnell eine Position, in welcher sich die entstandene Kette wieder auflöst, wie in Abbildung 3.4 zu sehen. Bevor sich weitere leere Wörter mit unsere aktuellen Wörter verbinden und dadurch ihre Eigenschaften verändern, überprüft der Parser an dieser Stelle erneut, ob nun ein Merge möglich ist. Da unsere Kategorie *'fuenf : (c1, -b)'* weiterhin mehr als eine Eigenschaft hat, ist nur ein Merge 3 möglich, welches an dieser Stelle ausprobiert wird. Durch das Merge 3 beenden wir die innerste Ebene der Rekursion und erreichen die Äußere, da wir in

diesem Fall nur zwei besitzen. Hier wird mit dem zurückgegeben erneut die Funktion *'sortMerge'* aufgerufen, da wir immer noch eine Kette von Teilsätzen haben, sowie unsere Eigenschaft nicht die Startkategorie ist. Nach einem anschließenden Move 1 erhalten wir einen Parsierungsbaum wie in Grafik 3.5 zu sehen.

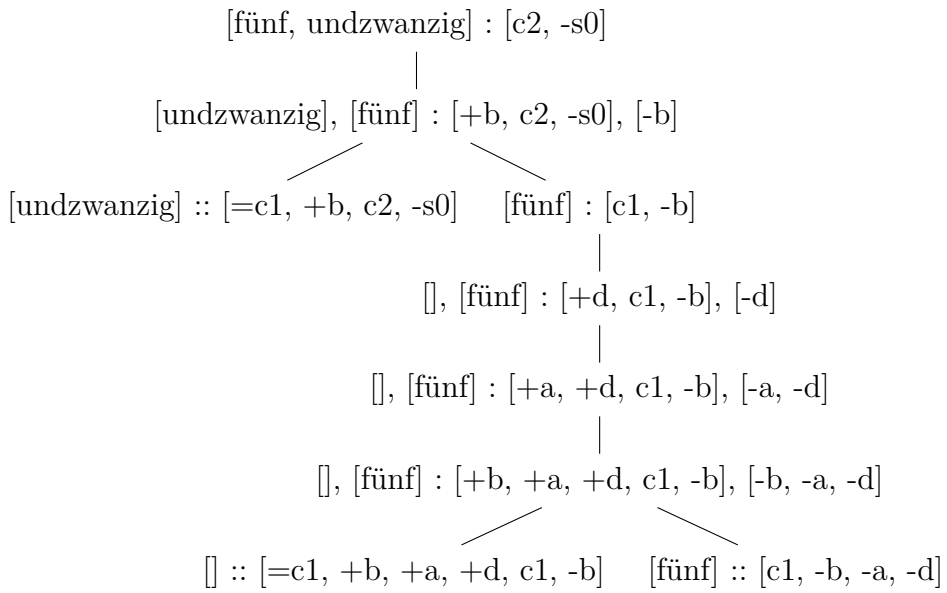

Abb. 3.5.: Ableitungsbaum am Ende des ersten Parsierungsdurchlaufes, Testen auf Startkategorie, 'fünfundzwanzig'

Der Parser wird nun aufgrund des Endes der Rekursion testen, ob wir uns in der Startkategorie befinden. Da dies noch nicht der Fall ist, geht der Parser einen Schritt zurück und fügt mit Hilfe der Funktion *sortMerge* ein weiteres leeres Wort ein. Die Überprüfung ergibt nun, dass wir uns in der Startkategorie befinden und geben den fertigen Baum zurück. Wäre dies nicht der Fall, würden weiter leere Wörter gewählt, bis die Startkategorie erreicht ist, oder es keine Kombination mehr gibt. Dann würde der Parser probieren an anderen Stellen andere Kombinationen von leeren Wörtern und den Arten von Merge zu finden um doch noch einen korrekten Parsierungsbaum zu finden. Schlagen all diese Versuche fehl, so gibt es auch keinen Parsierungsbaum und der Satz ist nicht Teil der durch das Lexikon abgebildeten Sprache. Den fertigen Baum für unser Beispiel ist in Abbildung 3.6 zu erkennen.

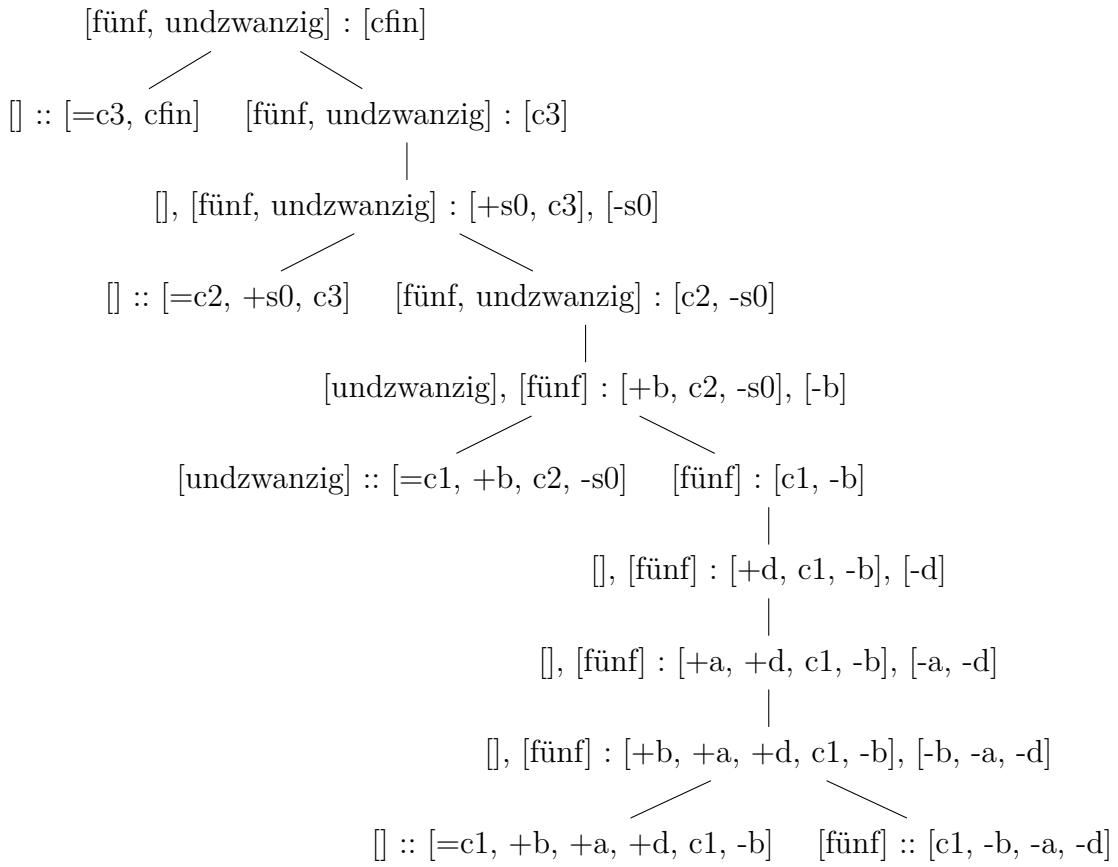

Abb. 3.6.: Fertiger Ableitungsbaum, 'fünfundzwanzig'

3.3. Weitere Bemerkungen zur Implementation

Der Parser findet (außer bei den zu Beginn angemerkten Punkten) auf jeden Fall einen Parsierungsbaum, wenn es einen gibt. Er findet den Ersten, von der Reihenfolge der Wörter im Lexikon. Er ist nicht minmal oder besitzt andere Eigenschaften. Sobald bei der Rekursion nur eines der beiden Argumente durch ein Wort oder Satzteil belegt ist, wird diese Runde übersprungen und das Wort oder Satzteil direkt an die nächste, weiter außen liegende Rekursionsebene gegeben. Wäre die Stelle ein Punkt gewesen wo weitere leere Wörter hinzugefügt werden, wie direkt am Anfang des Satzes bzw. nachdem der Satz zusammengefügt wurde allerdings zur Startkategorie noch weitere leere Wörter fehlen, so wird Prolog von selbst an diese Stelle zurückkehren und eine der letzten Regeln aufrufen, die das Hinzufügen von einem leeren Wort beinhaltet und weitere leere Wörter nach sich ziehen kann. Diese Regel ist enthalten, da nicht alle Sätze am Ende weitere leere Wörter brauchen. Auch entstehen durch die Verarbeitung von Wörtern mit Merge 1 beim rekursiven Aufruf

der Hauptfunktion weitere Lücken, die damit übersprungen werden können, da hier keine weitere Bearbeitung von nöten ist.

3.4. Zeitmessungen

Zum Vergleich, der aktuelle Algorithmus benötigt bei dem Satz *'fünf, undzwanzig'* mit der gegebenen Grammatik rund 2,35 Sekunden und rund 9 Millionen Folgerungen von Prolog bis ein korrekter Parsierungsbaum gefunden ist. Deutlich schneller ist hier *„fünf“* mit nur 0,20 Sekunden und um die 800 tausend Folgerungen. Das legt nahe, dass die Anzahl an Wörtern ausschlaggebend sein könnte bezüglich der Schnelligkeit. Allerdings benötigt *'zwei, undzwanzig'* nur 0,001 Sekunden sowie 312 Folgerungen um einen syntaktisch korrekten Parsierungsbaum zu erhalten. Unter Betrachtung der Lexika, des Vorgangs des Parsers, sowie der zurück gegebenen Parsierungsbäume können wir über die benötigte Laufzeit folgendes Aussagen:

- Sie ist Abhängig von der Sortierung von leeren Wörtern
- Zudem von der Sortierung von Wörtern sollte es je Wort verschiedene Versionen mit verschiedenen Eigenschaften haben
- Die Anzahl der Eigenschaften von allen Wörtern (Je mehr Eigenschaften desto mehr Schritte)
- Die “Wege“ zwischen Eigenschaften und Wörtern (Wie viele leere Wörter werden zwischen Wörtern benötigt und wie viele werden am Schluss benötigt, um die Startkategorie zu erreichen)

Vermutlich am Wichtigsten ist hier die zuerst genannte Eigenschaft. Davon aussgehend können wir aktuell noch die Laufzeit dadurch manipulieren, häufig genutzte leere Wörter weiter nach oben zu sortieren, oder eine Vorauswertung, welche leere Wörter mit anderen leeren Wörtern kombinierbar sind und welche Eigenschaften sie ergeben können, zu implementieren.
Je nachdem wie diese Vorauswertung ausfällt, wie groß das dazugehörige Lexikon ist und wie oft dieses Lexikon benutzt wird, sowie ob von Interesse ist, mit exakt welchen leeren Wörtern hier gearbeitet wurde, würde es sich lohnen automatisch eine zusätzliche Datei mit diesen Abkürzungen zu bilden, zusammen mit einem Verzeichnis welche leere Wörter hier verwendet wurden, um orginalgetreu den Parsierungsbaum abbilden zu können, nur schneller. Der Parser müsste entsprechend angepasst werden, dass er aus dieser Datei liest und mit dem Format arbeiten kann. Diese Vorauswertung laufen zu lassen würde je nach Größe des Lexikons vermutlich ebenfalls eine kleine Weile dauern, doch sollte dies sich bzgl. der Laufzeit beim

späteren Parsen bemerkbar machen. Denn hier könnte der Parser deutlich schneller auch Ketten von leeren Wörtern ausprobieren und verwerfen, ohne viel am restlichen Parser ändern zu müssen.
Es gibt zwar weitere Möglichkeiten zur Beschleunigung des Parsers, wie die Erweiterung auf die Betrachtung von ein oder mehr weitere Eigenschaften beim Hinzufügen von leeren Wörtern, doch diese sind nur minimal im Vergleich zu der beschriebenen Vorauswertung oder einem geeigneten Punkt im Parser wo erkennbar wäre, dass andere leere Wörter benötigt werden. Während des Parsens eine Vorauswertung zu machen bei jeder Stelle wo ein leeres Wort in Frage kommt wäre zwar auch machbar, allerdings birgt es auch hier die Gefahr, jedes mal alle Möglichkeiten durch zu gehen und dadurch eine hohe Laufzeit zu besitzen.

4. Zusammenfassung und Ausblick

Fassen wir noch einmal zusammen:

Minimalistische Grammatiken sind eine Formalisierung von Chomskys Minimalistischen Programm. Dieses ist als eine Weiterentwicklung eines Konzepts entstanden, welches als Annahme hat, dass Menschen mit einer Universalgrammatik geboren wurden. Sie bilden die natürliche Sprache ab, welche wir in unserem Alltag verwenden. Natürliche Sprache können wir zusammen mit Minimalistischen Grammatiken zwischen kontextfreien Sprachen und kontextsensitiven Sprachen anordnen. Daher können wir über sie aussagen, dass ein Algorithmus den wir implementieren immer endet und als Ergebnis sagen kann, ob ein Wort in der Sprache enthalten ist. Dieses Konzept können wir bei einem Parser verwenden, der mit Hilfe von Ableitungen der Grammatik, automatisch zu einem gegebenen Wort (synonym Satz) einen Syntaxbaum erstellt. Dafür haben wir die Möglichkeit von der Wurzel oder von den Blättern aus zu arbeiten, was uns zur Top-Down- oder Bottom-Up-Analyse bringt. In dieser Arbeit haben wir unseren Schwerpunkt auf Bottom-Up-Parser gelegt und haben uns die Unterschiede zwischen Breiten- und Tiefensuche angesehen und wie damit verschiedene Parser umzusetzen sind. Zum Ende hin haben wir uns mit der Implementation des Parsers beschäftigt, welcher Bereiche zum Ausbauen hat, und aufgrund seiner Annahmen die der Minimalistischen Grammatik entsprechen keine Vorverarbeitung benötigt. Diese kann jedoch in Zukunft hinzugefügt werden, was das Problem der aktuell langen Laufzeit bei vielen leeren Wörtern und daher falsch probierten Pfaden löst.

Literaturverzeichnis

Dabrowska 2015

Dabrowska, Ewa: What exactly is Universal Grammar, and has anyone seen it? In: *Language Sciences, a section of the journal Frontiers in Psychology* (2015), Juni 2

Erwig 1999

Erwig, Martin Ralf und G. Ralf und Güting: *Übersetzerbau, Techniken, Werkzeuge, Anwendungen*. Springer-Verlag Berlin Heidelberg, 1999. – 41–140 S. – ISBN 978–3–540–65389–9 2.3.1

Grune und Jacobs 2008

Grune, Dick ; Jacobs, Ceriel J.: *Parsing Techniques, A Practical Guide*. 2.,Auflage. Springer Science + Business Media, LLC, 2008. – 82–150 S. – ISBN 978–0–387–20248–8 2.3.1

Hromkovi˘c 2011

Hromkovi˘c, Juraj: *Theoretische Informatik, Formale Sprachen, Berechenbarkeit, Komplexitätstheorie, Algorithmik, Kommunikation und Kryptografie*. 4.,Auflage. Vieweg + Teubner Verlag, 2011. – ISBN 978–3–8348–0650–5 2.1

Lehner 1992

Lehner, Christoph: *Prolog und Linguistik*. 2.,Auflage. R. Oldenbourg Verlag GmbH, 1992. – ISBN 3–486–22144–2 3.2

Puchala 2019

Puchala, Magdalena A.: Inkrementeller Parser für minimalistische Grammatiken. (2019), Oktober 2.3.3

Stabler 2010

Stabler, Edward P.: Computational perspectives on minimalism. (2010), S. 616–641 2.1.1

Stabler 2011

Stabler, Edward P.: Top-down recognizers for MCFGs and MGs. (2011), S. 39–48 2.3.3

Stanojević 2018

Stanojević, Edward P. Miloš und Stabler S. Miloš und Stabler: A Sound and Complete Left-Corner Parsing for Minimalist Grammars. (2018), S. 65–74 2.3.3

Stanojević 2016

Stanojević, Miloš: Minimalist Grammar Transition-Based Parsing. (2016), S. 273–290. ISBN 978–3–662–53825–8 2.2, 2.2, 3.2

Vossen 2016

Vossen, Kurt-Ulrich Gottfried und W. Gottfried und Witt: *Grundkurs Theoretische Informatik, Eine anwendungsbezogene Einführung - Für Studierende in allen Informatik-Studiengängen.* 6.,Auflage. Springer Vieweg, 2016. – ISBN 978–3–8348–1770–9 2.1

Abbildungsverzeichnis

Listings

A. Anhang

A.1. Beiliegender USB-Stick

A.1.1. Inhaltsverzeichnis des USB-Sticks

1. Die gesamte Bachelorarbeit als PDF-Datei
2. Geschriebener Quellcode in Prolog
3. Verwendete Lexika in Prolog

A.2. Code

```
:- op(500, xfy, ::). % infix predicate for lexical items
:- op(500, fx, =). % for selection features


%:- [bsp_Lexika/maus].
:- [bsp_Lexika/german].
%:- [bsp_Lexika/english].

li([], Y) :- [] :: Y. % leere Wörter
li(X, Y) :- [X] :: Y.
tree(_, _, L, R) :- L, R. % X, E

wort([], Y, _) :- li([], Y). % N
wort(X, Y, _) :- li(X, Y). % N
zusWort(_, _, _, _, _). % X, E, N, TREE, M3LIST


start([X]) :-
  check(X, TREE),
  write(TREE),
  open('output.txt', write, Out),
```

```
22   write(Out, TREE),
23   close(Out).
24
25
26 start([X|SATZ]) :-
27   recursion([], wort(X, _, 1), SATZ, zusWort(A, E, B, TR, C)), % EY
28   check(zusWort(A, E, B, TR, C), TREE),
29   write(TREE),
30   open('output.txt', write, Out),
31   write(Out, TREE),
32   close(Out).
33
34
35 check(X, TREE) :- wort(X, [Y], _), Y,
36   write([Y]), write('\n');true),
37   startCategory(Y), TREE = li(X, [Y]).
38
39 check(zusWort(_, [E], _, TR, []), TREE) :- % A, B, C
40   write(E), write('\n');true),
41   startCategory(E), TREE = TR.
42
43
44
45
46 % recursions Funktionen
47
48 recursion(wort(X, E, N), Y, [], TREE_RETURN) :-
49   merge1(wort(X, E, N), Y, RETURN, RLIST),
50   sortMerge(RLIST, RETURN, TREE_RETURN).
51
52 % fuer einzelne woerter
53 recursion([], wort(X, E, N), [], TREE_RETURN) :-
54   sortMerge([], wort(X, E, N), TREE_RETURN).
55
56
57 recursion(wort(X, E, N), Y, [Z|RESTSATZ], TREE_RETURN) :-
58   merge1(wort(X, E, N), Y, RETURN, RLIST),
59   M is N+2,
60   recursion(RETURN, wort(Z, _, M), RESTSATZ, T_R), % EZ
61   sortMerge(RLIST, T_R, TREE_RETURN).
62
63 recursion(X, wort(Y, EY, NY), [Z|RESTSATZ], TREE_RETURN) :-
64   M is NY+1,
65   recursion(wort(Y, EY, NY), wort(Z, _, M), RESTSATZ, T_R), % EZ
66   sortMerge(X, T_R, TREE_RETURN).
67
68 recursion(zusWort(X, EX, NX, TR, M3LIST), wort(Y, EY, NY), [], TREE_RETURN) :-
```

```
69   sortMerge(zusWort(X, EX, NX, TR, M3LIST), wort(Y, EY, NY),TREE_RETURN).
70
71
72 merge1(wort(X, [=E|EX], N), wort(Y, [E|[]], N2), RETURN, LISTITEM) :- N2 is N+1,
73   wort(X, [=E|EX], N), wort(Y, [E|[]], (N2)) ,
74   TREE = tree([[X,Y]], [EX], li([X], [=E|EX]), li([Y], [E])),
75   RETURN = zusWort([X,Y], EX, N, TREE, []), LISTITEM = [].
76
77 merge1(X, Y, RETURN, LISTITEM) :- RETURN = Y, LISTITEM = X.
78
79
80 %------------
81
82 % move1 und move2
83 sortMerge(X, zusWort(Y, [+E|EY], NY, TR, M3LIST), RETURN) :-
84   sortMove(zusWort(Y, [+E|EY], NY, TR, M3LIST), ZUSW, []),
85   sortMerge(X, ZUSW, RETURN).
86
87 sortMerge(zusWort(Y, [+E|EY], NY, TR, M3LIST), X, RETURN) :-
88   sortMove(zusWort(Y, [+E|EY], NY, TR, M3LIST), ZUSW, []),
89   sortMerge(ZUSW, X, RETURN).
90
91
92 sortMerge(X, [], RETURN) :- RETURN = X.
93 sortMerge([], X, RETURN) :- RETURN = X.
94
95
96 %merge1
97
98 % fuer leere Woerter
99 sortMerge(wort([], [=E|EX], NX), wort(Y, [E], NY), RETURN) :-
100   NX = 0,% !,
101   wort([], [=E|EX], NX), wort(Y, [E], NY),
102   TREE = tree([Y], EX, li([], [=E|EX]), li([Y], [E])),
103   RETURN = zusWort([Y], EX, NX, TREE, []).
104
105 sortMerge(wort([], [=E|EX], NX), zusWort([], [E], _, TR, M3LIST), RETURN) :- %NY
106   NX = 0,
107   wort([], [=E|EX], NX),
108   getTreeWordPart([], M3LIST, [], TW),
109   getTreeFeatures(EX, M3LIST, [], TE),
110   TREE = tree(TW, TE, li([], [=E|EX]), TR),
111   RETURN = zusWort([], EX, NX, TREE, M3LIST).
112
113
114 sortMerge(wort([], [=E|EX], NX), zusWort(Y, [E], _, TR, M3LIST), RETURN) :- %NY
115   NX = 0,
```

```
116   wort([], [=E|EX], NX),
117   getTreeWordPart(Y, M3LIST, [], TW),
118   getTreeFeatures(EX, M3LIST, [], TE),
119   TREE = tree(TW, TE, li([], [=E|EX]), TR),
120   RETURN = zusWort(Y, EX, NX, TREE, M3LIST).
121
122 sortMerge(wort(X, [=E|EX], NX), wort(Y, [E], NY), RETURN) :-
123   NY is NX + 1,% !,
124   wort(X, [=E|EX], NX), wort(Y, [E], NY),
125   TREE = tree([X,Y], EX, li([X], [=E|EX]), li([Y], [E])),
126   RETURN = zusWort([X,Y], EX, NX, TREE, []).
127
128 sortMerge(wort(X, [=E|EX], NX), zusWort(Y, [E], NY, TR, M3LIST), RETURN) :-
129   NY is NX + 1,
130   wort(X, [=E|EX], NX),
131   getTreeWordPart([X|Y], M3LIST, [], TW),
132   getTreeFeatures(EX, M3LIST, [], TE),
133   TREE = tree(TW, TE, li([X], [=E|EX]), TR),
134   RETURN = zusWort([X|Y], EX, NX, TREE, M3LIST).
135
136
137 %merge2
138 %leere woerter
139 sortMerge(wort([], [E], NX), zusWort(Y, [=E|EY], _, TR, M3LIST), RETURN) :- %NY
140   wort([], [E], NX),
141   getTreeWordPart(Y, M3LIST, [], TW),
142   getTreeFeatures(EY, M3LIST, [], TE),
143   TREE = tree(TW, TE, TR, li([], [E])),
144   RETURN = zusWort(Y, EY, NX, TREE, M3LIST).
145
146 sortMerge(wort(X, [E], NX), zusWort(Y, [=E|EY], _, TR, M3LIST), RETURN) :- %NY
147   wort(X, [E], NX),
148   getTreeWordPart([X|Y], M3LIST, [], TW),
149   getTreeFeatures(EY, M3LIST, [], TE),
150   TREE = tree(TW, TE, TR, li([X], [E])),
151   RETURN = zusWort([X|Y], EY, NX, TREE, M3LIST).
152
153 sortMerge(zusWort(X, [E], NX, TR_X, M3LIST_X), zusWort(Y, [=E|EY], _, TR_Y, M3LIST_Y),
      RETURN ) :- %NY
154   append(X, Y, SATZTEIL),
155   append(M3LIST_X, M3LIST_Y, M3LIST),
156   getTreeWordPart(SATZTEIL, M3LIST, [], TW),
157   getTreeFeatures(EY, M3LIST, [], TE),
158   TREE = tree(TW, TE, TR_Y, TR_X),
159   RETURN = zusWort(SATZTEIL, EY, NX, TREE, M3LIST).
160
161
```

```
162
163
164 % merge3
165
166 sortMerge(wort([], [=E|EX], NX), wort(Y, [E|EY], NY), RETURN) :-
167   NX = 0, wort([], [=E|EX], NX), wort(Y, [E|EY], NY),
168   TR = tree([[], [Y]], [EX,EY], li([], [=E|EX]), li([Y], [E|EY])),
169   RETURN = zusWort([], EX, NX, TR, [zusWort([Y], EY, NY, [], [])]).
170
171 sortMerge(wort([], [E|EX], NX), wort(Y, [=E|EY], NY), RETURN) :-
172   NX < NY, wort([], [E|EX], NX), wort(Y, [=E|EY], NY),
173   TR = tree([[], [Y]], [EY,EX], li([Y], [=E|EY]), li([], [E|EX])),
174   RETURN = zusWort([Y], EY, NY, TR, [zusWort([], EX, NX, [], [])]).
175
176
177 sortMerge(wort(X, [E|EX], NX), wort(Y, [=E|EY], NY), RETURN) :-
178   NX < NY, wort(X, [E|EX], NX), wort(Y, [=E|EY], NY),
179   TR = tree([[Y, X]], [EY,EX], li([Y], [=E|EY]), li([X], [E|EX])),
180   RETURN = zusWort([Y], EY, NY, TR, [zusWort([X], EX, NX, [], [])]).
181
182
183 sortMerge(wort([], [=E|EX], NX), zusWort(Y, [E,EC|EY], NY, TR, M3LIST), RETURN) :-
184   NX = 0, smc(EC, M3LIST),
185   wort([], [=E|EX], NX),
186   getTreeWordPart(Y, M3LIST, [], TW),
187   getTreeFeatures([EC|EY], M3LIST, [], TE),
188   TREE = tree([[]|TW], [EX|TE], li([], [=E|EX]), TR),
189   RETURN = zusWort([], EX, NX, TREE, [zusWort(Y, [EC|EY], NY, [], [])|M3LIST]).
190
191
192 sortMerge(wort([], [E, EC|EX], NX), zusWort(Y, [=E|EY], NY, TR, M3LIST), RETURN) :-
193   NX < NY, smc(EC, M3LIST),
194   wort([], [E, EC|EX], NX),
195   getTreeWordPart([], M3LIST, [], TW),
196   getTreeFeatures([EC|EX], M3LIST, [], TE),
197   TREE = tree([Y|TW], [EY|TE], TR, li([], [E, EC|EX])),
198   RETURN = zusWort(Y, EY, NY, TREE, [zusWort([], [EC|EX], NX, [], [])|M3LIST]).
199
200
201 sortMerge(wort(X, [E, EC|EX], NX), zusWort(Y, [=E|EY], NY, TR, M3LIST), RETURN) :-
202   NX < NY, smc(EC, M3LIST),
203   wort(X, [E, EC|EX], NX),
204   getTreeWordPart([X], M3LIST, [], TW),
205   getTreeFeatures([EC|EX], M3LIST, [], TE),
206   TREE = tree([Y|TW], [EY|TE], TR, li([X], [E, EC|EX])),
207   RETURN = zusWort(Y, EY, NY, TREE, [zusWort([X], [EC|EX], NX, [], [])|M3LIST]).
208
```

```
209
210 sortMerge(zusWort(X, [E, EC|EX], NX, TR, M3LIST), wort(Y, [=E|EY], NY), RETURN) :-
211   NX < NY, smc(EC, M3LIST),
212   wort(Y, [=E|EY], NY),
213   getTreeWordPart(X, M3LIST, [], TW),
214   getTreeFeatures([EC|EX], M3LIST, [], TE),
215   TREE = tree([[Y]|TW], [EY|TE], li([Y], [=E|EY]), TR),
216   RETURN = zusWort([Y], EY, NY, TREE, [zusWort(X, [EC|EX], NX, [], [])|M3LIST]).
217
218
219 sortMerge(zusWort(X, [E, EC|EX], NX, TR_X, M3LIST_X), zusWort(Y, [=E|EY], NY, TR_Y,
      M3LIST_Y), RETURN) :-
220   NX < NY,
221   append(M3LIST_Y, M3LIST_X, M3LIST),
222   smc(EC, M3LIST),
223   getTreeWordPart([Y, X], M3LIST, [], TW),
224   getTreeFeatures([EY, [EC|EX]], M3LIST, [], TE),
225   TREE = tree(TW, TE, TR_Y, TR_X),
226   RETURN = zusWort(Y, EY, NY, TREE, [zusWort(X, [EC|EX], NX, [], [])|M3LIST]).
227
228 % -
229
230
231 sortMerge(X, Y, RETURN) :-
232   X \= wort([], _, _), Y \= wort([], _, _),
233   emptyWords([], X, RET1), sortMerge(RET1, Y, RET2), sortMerge([], RET2, RETURN).
234
235 sortMerge(X, Y, RETURN) :-
236   X \= wort([], _, _), Y \= wort([], _, _),
237   emptyWords([], Y, RET1), sortMerge(X, RET1, RET2), sortMerge([], RET2, RETURN).
238
239
240 smc(_, []) :- !, true. %E
241 smc(E, [zusWort(_,[E|_],_,_,_)|_]) :- !, false.
242 smc(E, [A|NEXT]) :- !, smc(E, NEXT).
243
244
245 % gibt Liste aller "Wörter" der M3LIST zurück
246 getTreeWordPart(Z, [], LIST, RET) :-
247   reverse(LIST, L), RET = [Z|L].
248 getTreeWordPart(Z, [zusWort(X, _, _, _, _)|L], LIST, RET) :-
249   getTreeWordPart(Z, L, [X|LIST], RET).
250
251 % gibt Liste aller Eigenschaften der M3LIST zurück
252 getTreeFeatures(Z, [], LIST, RET) :-
253   reverse(LIST, L), RET = [Z|L].
254 getTreeFeatures(Z, [zusWort(_, E, _, _, _)|L], LIST, RET) :-
```

```
255   getTreeFeatures(Z, L, [E|LIST], RET).
256
257
258
259 % move1
260 sortMove(zusWort(X, [+E|EX], NX, TR, [zusWort(Y, [-E], NY, _, _)|FOLLOWLIST]), RET,
        PRIORLIST) :-
261   NX > NY,
262   append(Y, X, Z), % zusammenfügen der Listen von X und Y, wenn X oder Y leer sind ([]),
          dann wird das raus gefiltert
263   append(PRIORLIST, FOLLOWLIST, LIST),
264   getTreeWordPart(Z, LIST, [], TWP),
265   getTreeFeatures(EX, LIST, [], TE),
266   TREE = tree(TWP, TE, TR, e),
267   RET = zusWort(Z, EX, NY, TREE, LIST).
268
269 sortMove(zusWort(X, [+E|EX], NX, TR, [zusWort(Y, [-E], NY, _, _)|FOLLOWLIST]), RET,
        PRIORLIST) :-
270   NX = NY,
271   append(Y, X, Z),
272   append(PRIORLIST, FOLLOWLIST, LIST),
273   getTreeWordPart(Z, LIST, [], TWP),
274   getTreeFeatures(EX, LIST, [], TE), % kann das überhaupt auftreten?
275   TREE = tree(TWP, TE, TR, e),
276   RET = zusWort(Z, EX, NY, TREE, LIST).
277
278 % move2
279
280 sortMove(zusWort(X, [+E|EX], NX, tree(T, ET, LT, RT),[zusWort(Y, [-E|EY], NY, _, _)|
        FOLLOWLIST]), RET, PRIORLIST) :-
281   NX > NY,
282   append([zusWort(Y, EY, NY, [], [])|FOLLOWLIST], PRIORLIST, LIST),
283   getTreeFeatures(EX, LIST, [], TE),
284   TREE = tree(T, TE, tree(T, ET, LT, RT), e),
285   RET = zusWort(X, EX, NY, TREE,LIST).
286
287 sortMove(zusWort(X, [+E|EX], NX, tree(T, ET, LT, RT), [zusWort(Y, [-E|EY], NY, _, _)|
        FOLLOWLIST]), RET, PRIORLIST) :-
288   NX = NY,
289   append([zusWort(Y, EY, NY, [], [])|FOLLOWLIST], PRIORLIST, LIST),
290   getTreeFeatures(EX, LIST, [], TE), % kann das überhaupt auftreten?
291   TREE = tree(T, TE, tree(T, ET, LT, RT), e),
292   RET = zusWort(X, EX, NY, TREE,LIST).
293
294 % weiter nach richtiger Eigenschaft suchen
295 sortMove(zusWort(X, EX, NX, TR_X, [W1|W]), RET, LIST) :-
296   sortMove(zusWort(X, EX, NX, TR_X, W), RET, [W1|LIST]).
```

```
297
298
299
300 % leere Woerter
301
302
303 emptyWords([], wort(Y, [E|EY], NY), RETURN) :-
304   wort(Y, [E|EY], NY),
305   sortMerge(wort([], [=E|_], 0), wort(Y, [E|EY], NY), zusWort(A, B, _, C, D)),
306   RETURN = zusWort(A, B, NY, C, D).
307
308 emptyWords([], zusWort(Y, [E|EY], NY, TR, LI), RETURN) :-
309   sortMerge(wort([], [=E|_], 0), zusWort(Y, [E|EY], NY, TR, LI), zusWort(A, B, _, C, D)),
310   RETURN = zusWort(A, B, NY, C, D).
311
312 emptyWords([], wort(Y, [=E|EY], NY), _, RETURN) :- %X
313   wort(Y, [=E|EY], NY),
314   sortMerge(wort([], [E|_], 0), wort(Y, [=E|EY], NY), zusWort(A, B, _, C, D)),
315   RETURN = zusWort(A, B, NY, C, D).
316
317
318 emptyWords([], zusWort(Y, [=E|EY], NY, TR, LI), _, RETURN) :- %X
319   sortMerge(wort([], [E|_], 0), zusWort(Y, [=E|EY], NY, TR, LI), zusWort(A, B, _, C, D)),
320   RETURN = zusWort(A, B, NY, C, D).
321
322 emptyWords([], [], zusWort(Y, [E|EY], NY, TR, L), RETURN) :-
323   sortMerge(wort([], [=E|_], 0), zusWort(Y, [E|EY], NY, TR, L), zusWort(A, B, _, C, D)),
324   RETURN = zusWort(A, B, NY, C, D).
325
326 emptyWords([], [], zusWort(Y, [=E|EY], NY, TR, L), RETURN) :-
327   sortMerge(wort([], [E|_], 0), zusWort(Y, [=E|EY], NY, TR, L), zusWort(A, B, _, C, D)),
328   RETURN = zusWort(A, B, NY, C, D).
```

A.3. Verwendete Lexika

```
1 %   File   : maus.pl
2 %   Author : Johannes Kuhn
3 % purpose : simple MG-Lexicon to build sentences with the Mouse-Cheese-Scenario
4 :- op(500, xfy, ::). % infix predicate for lexical items
5 :- op(500, fx, =). % for selection features
6 % Bsp mit Maus und Käse
7 startCategory(c).
8
9 %[] :: ([=d,+wh,d]).
```

```
['Maus'] :: ([d]).
['Kaese'] :: ([n]).
['Katze'] :: ([d]).
['Hund'] :: ([d,-w]).
['weis nix'] :: ([=d,+w,c]).
[frisst] :: ([=d,=d,c]).
[wer] :: ([d,-wh]).
[und] :: ([=c,=c,c]).
[wen] :: ([d,-wh]).
[fressen] :: ([=d,=d,v]).


[der] :: ([=d,d]).
[den] :: ([=d,d]).
[den] :: ([=n,d]).
[die] :: ([=d, d]).
[kennt] :: ([=d,=d,c]).
['Postbote'] :: ([d]).
['Besitzer'] :: ([d]).
['Frau'] :: ([d]).
['Mann'] :: ([d]).
['?'] :: ([=c,+f,d]).
['Satz: '] :: ([=d,c]).

[] :: ([=v,c]).
[] :: ([=v,+wh,c]).




% Bsp.:
% ['Katze',frisst,'Maus',und,'Maus',frisst,'Kaese']
% ['Maus',frisst,'Kaese']
% ['Maus',weis,wer,'Kaese',frisst]
```

```
startCategory(cfin).
[drei] :: ([c1, -c, -b, -a]).
[] :: ([=c3, cfin]).
[zwei] :: ([c1, -b]).
[vier] :: ([c1, -b, -a, -d]).
[fünf] :: ([c1, -b, -a, -d]).
[sechs] :: ([c1, -b]).
[sieben] :: ([c1, -b]).
[acht] :: ([c1, -b, -a, -d]).
[neun] :: ([c1, -b, -a, -d]).
[zehn] :: ([c2, -e, -s0]).
[elf] :: ([c2, -e, -s0]).
[zwölf] :: ([c2, -e, -s0]).
```

```
[sechzehn] :: ([c2, -e, -s0]).
[siebzehn] :: ([c2, -e, -s0]).
[zwanzig] :: ([c2, -e, -s0]).
[einundzwanzig] :: ([c2, -e, -s0]).
[sechzig] :: ([c2, -e, -s0]).
[einundsechzig] :: ([c2, -e, -s0]).
[siebzig] :: ([c2, -e, -s0]).
[einundsiebzig] :: ([c2, -e, -s0]).
[] :: ([=c1, c2, -e]).
[eins] :: ([c1]).
[hundert] :: ([c3]).
[hundert] :: ([=c3e, c3]).
[] :: ([=c2, +e, c3e]).
[hundert] :: ([=c1, +b, c3]).
[hundert] :: ([=c3e, =c2b, c3]).
[] :: ([=c1, +b, c3b]).
[] :: ([=c2, +s0, c3]).
[zehn] :: ([=c1, +a, c2, -s0]).
[undzwanzig] :: ([=c1, +b, c2, -s0]).
[ßig] :: ([=c1, +c, c2, -s0]).
[einund] :: ([=c21s, c2, -s0]).
[ßig] :: ([=c1, +c, c21s]).
[und] :: ([=c22s, =c2b, c2, -s0]).
[] :: ([=c1, +b, c2b]).
[ßig] :: ([=c1, +c, c22s]).
[zig] :: ([=c1, +d, c2, -s0]).
[einund] :: ([=c23s, c2, -s0]).
[zig] :: ([=c1, +d, c23s]).
[und] :: ([=c24s, =c2b, c2, -s0]).
[zig] :: ([=c1, +d, c24s]).
[undsechzig] :: ([=c1, +b, c2, -s0]).
[undsiebzig] :: ([=c1, +b, c2, -s0]).
[] :: ([=c1, c3]).
[] :: ([=c1, +d, c1]).
[] :: ([=c1, +b, c1]).
[] :: ([=c1, +c, c1]).
[] :: ([=c2, +s0, c2]).
[] :: ([=c1, +a, c1]).
[] :: ([=c2, +e, c2]).
[] :: ([=c2, +e, +s0, c2, -e]).
[] :: ([=c1, +c, +b, +a, c1, -c]).
[] :: ([=c1, +b, +a, +d, c1, -b]).
[] :: ([=c1, +b, +a, c1, -b]).
[] :: ([=c1, +a, +d, c1, -a]).
```

```
startCategory(cfin).
[eight] :: ([c1, -b]).
```

```
[] :: ([=c3, cfin]).
[one] :: ([c1]).
[two] :: ([c1]).
[three] :: ([c1]).
[four] :: ([c1, -a]).
[five] :: ([c1]).
[six] :: ([c1, -d, -a]).
[seven] :: ([c1, -d, -a]).
[nine] :: ([c1, -d, -a]).
[ten] :: ([c2, -e, -s0]).
[eleven] :: ([c2, -e, -s0]).
[twelve] :: ([c2, -e, -s0]).
[thirteen] :: ([c2, -e, -s0]).
[fifteen] :: ([c2, -e, -s0]).
[twenty] :: ([c2, -e, -s0]).
[thirty] :: ([c2, -e, -s0]).
[forty] :: ([c2, -e, -s0]).
[fifty] :: ([c2, -e, -s0]).
[] :: ([=c1, c2, -e]).
[hundred] :: ([=cnix, =c1, c3]).
[] :: ([cnix]).
[hundred_and] :: ([=c3e, =c1, c3]).
[] :: ([=c2, +e, c3e]).
[] :: ([=c2, +s0, c3]).
[teen] :: ([=c1, +a, c2, -s0]).
[een] :: ([=c1, +b, c2, -s0]).
[twenty_] :: ([=c1, c2, -s0]).
[thirty_] :: ([=c1, c2, -s0]).
[forty_] :: ([=c1, c2, -s0]).
[fifty_] :: ([=c1, c2, -s0]).
[ty] :: ([=c1, +d, c2, -s0]).
[ty_] :: ([=c1, =c2d, c2, -s0]).
[] :: ([=c1, +d, c2d]).
[y] :: ([=c1, +b, c2, -s0]).
[y_] :: ([=c1, =c2b, c2, -s0]).
[] :: ([=c1, +b, c2b]).
[] :: ([=c1, c3]).
[] :: ([=c1, +d, c1]).
[] :: ([=c1, +b, c1]).
[] :: ([=c2, +s0, c2]).
[] :: ([=c1, +a, c1]).
[] :: ([=c2, +e, c2]).
[] :: ([=c2, +e, +s0, c2, -e]).
[] :: ([=c1, +d, +a, c1, -d]).
```