Brandenburgische
Technische Universität
Cottbus - Senftenberg

Realisierung eines Top-Down-Parsers für Minimalistische Grammatiken

Bachelorarbeit

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

vorgelegt von:	Niklas J. Arlt
Matrikelnummer:	3741435
Erstgutachter:	Prof. Dr.-Ing. habil. Matthias Wolff
Zweitgutachter:	Prof. Dr. rer. nat. habil. Petra Hofstedt
eingereicht in:	Cottbus, am 02. Mai 2024

Eidesstattliche Versicherung

Ich, Niklas J. Arlt, Matrikel-Nr. 3741435, versichere hiermit, dass ich meine Bachelorarbeit mit dem Thema

Realisierung eines Top-Down-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 02. Mai 2024

Niklas J. Arlt

Inhaltsverzeichnis

1. Abstract

Diese Arbeit stellt die Funktionsweise und Implementierung eines Top-Down-Parsers für Minimalistische Grammatiken vor. Dafür beschäftigt sie sich zunächst mit den grundlegenden Begriffen der Sprachtheorie. Weiterführend werden Minimalistische Grammatiken im Detail beschrieben und eine allgemeine Übersicht über derzeitige Parser erstellt. Im Hauptteil wird die Funktionsweise eines selbst implementierten Top-Down-Parsers in der Sprache Prolog erläutert, sowie seine Laufzeit anhand eines Tests analysiert.

This paper presents the functionality and implementation of a top-down parser for minimalist grammars. To this end, it first deals with the basic concepts of language theory. It then goes on to describe minimalist grammars in detail and provides a general overview of current parsers. In the main part, the functionality of a self-implemented top-down parser in the Prolog language is explained and its runtime is analyzed a test.

2. Motivation

Ziel dieser Arbeit ist die Erstellung eines Top-Down-Parsers für minimalistische Grammatiken (MG's). Das Konzept der Minimalistischen Grammatiken wurde von Edward P. Stabler Stabler [1996] entwickelt, um eine Möglichkeit zu schaffen, menschliche Sprache für den Computer verständlich umzusetzen.
Bisher basieren Künstliche Intelligenzen (KI's) vor allem auf der Auswertung von großen Datenmengen, um daraus Antworten für zukünftige Eingaben zu generieren. Das Programm "rät" also auf Grundlage von vielen bekannten Daten die statistisch wahrscheinlichste Antwort. Menschliche Sprache eignet sich für ein solches Modell nicht, da Wörter oft viele unterschiedliche Bedeutungen haben können. Es ist für KI's nicht nur schwer solche Unterschiede zu erkennen, sondern auch sie differenziert zu betrachten.
Die Forschung am Lehrstuhl für Kommunikationstechnik der BTU Cottbus-Senftenberg beschäftigt sich mit den möglichen Anwendungsgebieten von Minimalistischen Grammatiken in diesem Bereich. Sie erforscht dafür nicht nur die Möglichkeit, MG's für die Syntaxanalyse von Sätzen einzusetzen, sondern möchte ihre Struktur zur Verarbeitung der Semantik gebrauchen.
Diese Arbeit setzt sich dafür zunächst intensiv mit der Struktur von Minimalistischen Grammatiken auseinander und beschreibt dann die Implementierung eines Parsers zur Verarbeitung von MG's in der Programmiersprache Prolog.

3. Grundlagen

3.1. Sprache

Für den Menschen ist Sprache ein grundlegendes Prinzip, es ist eines der Konzepte, die uns einen evolutionären Vorteil gegenüber anderen Tieren verschafft. Ein Computer kennt dieses Konzept jedoch nicht, daher braucht es eine klare Struktur, die definiert, was eine Sprache ist.
Dies ist allerdings kompliziert, da verschiedene Sprachen unterschiedlich strukturiert sind und oft Sonderfälle und Ausnahmen auftreten. Außerdem verwenden verschiedene Sprachen auch oftmals verschiedene Alphabete, was weitere Verständnisprobleme aufwirft.
Daher müssen diese Begriffe zunächst spezifiziert werden.

Definition Alphabet:
Eine endliche, nichtleere Menge Σ wird Alphabet genannt. Die Elemente dieser Menge heißen Buchstaben.

Definition Wort:
Sei Σ ein Alphabet. Ein Wort w über dem Alphabet Σ ist eine Folge von Buchstaben aus Σ. Eine leere Buchstabenfolge, also ein leeres Wort, wird mit ϵ bezeichnet.
Σ^* ist die Menge aller Wörter über Σ.

Definition Sprache:
L ist eine Sprache über einem Alphabet Σ genau dann wenn $L \subseteq \Sigma^*$

Mit dieser sehr allgemeinen Definition einer Sprache können nun zum Beispiel für ein Alphabet $\Sigma = \{a, b, c\}$ folgende Sprachen definiert werden:

- $L_1 = \{aaabbc, cbabc, ababa, ccc\}$
- $L_2 = \{w \in \Sigma^* | \text{w enthält genau drei a's}\}$
- $L_3 = \{a^n b^m | n, m \in \mathbb{N}\}$

Während diese Beispiele alle gültige Sprachen darstellen, so ist ihre Definition für Computer kaum verständlich. Auch sind komplexe Sprachen, z.B. eine unendliche Sprache die keiner festen Struktur folgt, so nicht darstellbar. Es bedarf also eines einheitlichen Konzeptes, das es ermöglicht, Sprachen eindeutig zu definieren und sinnvoll auswertbar zu machen.
Zu diesem Zweck wurde das Prinzip der Grammatiken eingeführt.

3.2. Grammatiken

Grammatiken sind allgemein eine Struktur, die Wörter einer Sprache durch eine Menge von Ableitungsregeln darstellt.

Definition Grammatik: *Eine Grammatik G ist ein Vier-Tupel mit*

$$G = (N, T, S, P)$$

Hierbei ist:

- N *ein* ***Nichtterminalalphabet***
- T *ein* ***Terminalalphabet***
- $S \in N$ *ein* ***Startsymbol***
- P *eine Menge an* ***Produktionsregeln*** *mit*

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

Hierbei muss weiterhin gelten, dass das Nichtterminalalphabet und das Terminalalphabet disjunkt sind ($N \cap T = \emptyset$).
Die Produktionsregeln, auch Ableitungsregeln genannt, sind also Zwei-Tupel (α, β). *Oft werden sie jedoch als* $\alpha \to \beta$ *geschrieben, da so deutlicher gezeigt wird, dass* α

auf β abgeleitet wird.

Wenden wir dieses Konzept auf die oben genannten Sprachen 3.1 an, so erhalten wir folgende Grammatiken G_1, G_2 und G_3:

$L_1 = \{aaabbc, cbabc, ababa, ccc\}$

$G_1 = (\{S\}, \{a, b, c\}, S, P)$ mit
$P = S \to aaabbc|cbabc|ababa|ccc$

$L_2 = \{w \in \Sigma^* |$w enthält genau drei a's$\}$

$G_2 = (\{S, X\}, \{a, b, c\}, S, P)$ mit
$P = S \to XaXaXaX$
$X \to bX|cX|b|c|\varepsilon$

$L_3 = \{a^n b^m | n, m \in \mathbb{N}\}$

$G_3 = (\{S, A, B\}, \{a, b, c\}, S, P)$ mit
$P = S \to AB$
$A \to aA|a$
$B \to bB|b$

Beispielhaft kann das Wort w = aaabb, welches Teil der Sprache L_3 ist, durch folgende Ableitungsregeln dargestellt werden 3.1:

String	angewendete Regel
S	S → AB
AB	A → aA
aAB	A → aA
aaAB	A → a
aaaB	B → bB
aaabB	B → b
aaabb	fertig

Tab. 3.1.: Ableitungsregeln für das Wort aaabb

Wörter, die nicht Teil der Sprache sind, lassen sich entsprechend nicht durch Anwendung der Ableitungsregeln erzeugen.

3.3. Einteilung der Grammatiken nach Chomsky

Grammatiken wurden von Chomsky in vier Klassen unterteilt Chomsky [1959], die sogenannte Chomsky-Hierarchie. Diese Klassen unterscheiden sich in der Striktheit ihrer Produktionsregeln. Je höher ihr Typ, desto größer die Beschränkung.
Während Grammatiken eines hohen Typs also weniger Sprachen darstellen können, sind sie durch einfachere Algorithmen umsetzbar und können somit deutlich schneller berechnet werden.

Definition Typ-0-Grammatik:
Als unbeschränkte Grammatik oder Typ-0-Grammatik gelten alle Grammatiken, deren Produktionsregeln die Form

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

besitzen.
Typ-0-Grammatiken besitzen als einzige Einschränkung, dass die linke Regelseite mindestens ein Nichtterminalsybol enthalten muss. Da dies der allgemeinen Defintion einer Grammatik entspricht, ist jede formale Grammatik von Typ 0.

Definition Typ-1-Grammatik:
Als kontextsensitive Grammatik oder Typ-1-Grammatik gelten alle Grammatiken, deren Produktionsregeln die Form

$$P \subseteq v \cdot N \cdot w \times v \cdot (T \cup N)^* \cdot w \text{ mit } v, w \in (T \cup N)^*$$

besitzen.
Bei allen Klassen ab den Typ-1-Grammatiken wird in jeder Produktionsregel lediglich ein einzelnes Nichtterminalsymbol abgeleitet. Im Falle der Typ-1-Grammatiken können sich davor und dahinter noch weitere Symbole befinden, welche jedoch nicht verändert werden. Aufgrund dieser Eigenschaft, dass das Nichtterminalsymbol abhängig von seinem Kontext unterschiedlich abgeleitet werden kann, werden diese Grammatiken als kontextsensitiv bezeichnet.

Definition Typ-2-Grammatik:

Als kontextfreie Grammatik oder Typ-2-Grammatik gelten alle Grammatiken, deren Produktionsregeln die Form

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

besitzen.
Bei Typ-2-Grammatiken besteht die linke Regelseite nur aus einem Nichtterminalsymbol. Dieses kann jedoch weiterhin auf beliebige Terme abgeleitet werden. Der Unterschied zu den Typ-1-Grammatiken besteht also im Fehlen des "Kontexts", daher werden Typ-2-Grammatiken auch kontextfrei genannt.

Definition Typ-3-Grammatik:
Als reguläre Grammatik oder Typ-3-Grammatik gelten alle Grammatiken, deren Produktionsregeln die Form

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

besitzen.
Typ-3-Grammatiken haben die größten Einschränkungen. Die linke Regelseite enthält wieder nur ein Nichtterminalsymbol und die Rechte besteht entweder nur aus Terminalsymbolen oder aus Terminalsymbolen und einem Nichtterminalsymbol dahinter. Somit kann bei jedem Ableitungsschritt nur ein einziges Nichtterminalsymbol existieren, welches sich immer ganz rechts vom gesamten Term befindet. Deswegen werden diese Grammatiken auch rechtslinear genannt.

Da nun alle Sprachklassen ausführlich definiert wurden, stellt sich automatisch die Frage, welcher Klasse die Minimalistischen Grammatiken angehören. Intuitiv ist davon auszugehen, dass die Ableitung einzelner Wörter in Sätzen natürlicher Sprache von ihrem Kontext abhängig ist, was der Klasse der kontextsensitiven Grammatiken entspräche. Dennoch folgen viele vor allem einfache Sätze sehr konkreten Regeln (im Deutschen wäre das z.B. die Satzstellung Subjekt-Verb-Objekt), was mehr auf kontextfreie Grammatiken hindeutet.
Tatsächlich bilden die Minimalistischen Grammatiken eine Teilmenge der kontextsensitiven Grammatiken, welche die kontextfreien Grammatiken voll umfasst. Dies wurde gezeigt, indem eine Äquivalenz auf MCFGs (Multiple kontextfreie Grammatiken) gezeigt wurde Stabler [2010].

4. Minimalistische Grammatiken

4.1. Definition

Nun sind alle Grundlagen erläutert, um die Minimalistischen Grammatiken einordnen zu können. Die Funktionalität ist zunächst gleich. Ein Wort ist dann Teil einer Sprache, wenn eine Folge von Ableitungen für dieses Wort existiert. Allerdings weist ihre Struktur deutliche Unterschiede zu formalen Grammatiken auf.
Der erste Unterschied von Minimalistischen Grammatiken zu formalen Grammatiken ist die Tatsache, dass in Minimalistischen Grammatiken Wörter zu Sätzen zusammengefügt werden und in formalen Grammatiken Buchstaben zu Wörtern.
Unsere Grundterme sind demnach Wörter (Kombinationen von Terminalsymbolen), während ein "Wort" einem Satz entspricht. Dieser Umstand kann verwirrend sein, muss aber für den Verlauf der Arbeit dringend beachtet werden.
Der wichtigste Unterschied besteht in der Beschränkung auf lediglich fünf Ableitungsfunktionen. Dies wird durch zusätzliche Funktionalität bei den Nichtterminalen, hier Features genannt, umgesetzt. Features sind hierfür in vier Kategorien unterteilt:

- **Kategorie** : Standard-Feature, ohne Vorzeichen.
- **Selektor**: Gegenpart zur Kategorie, gekennzeichnet durch ein = als Vorzeichen.
- **Lizenzgeber**: Gegenpart zum Lizenznehmer, gekennzeichnet durch ein + als Vorzeichen.
- **Lizenznehmer**: Gegenpart zum Lizenzgeber, gekennzeichnet durch ein - als Vorzeichen.

Hierbei ist es wichtig, dass Kategorie und Selektor, sowie Lizenzgeber und Lizenznehmer stets in Paaren auftreten. Zur Verarbeitung werden nämlich stets zwei Features mit dem selben Featurenamen und gegensätzlichen Vorzeichen benötigt. Eine Liste dieser Features wird Featureliste genannt.

Jedem Terminalsymbol, in diesem Fall also den einzelnen Wörtern, wird mindestens eine solche Featureliste zugewiesen, welche die benötigten Ableitungsregeln für dieses Wort angibt. Diese Zuweisungen werden dann in einem Lexikon aufgeführt, welches einen großen Teil der Definition einer Sprache ausmacht.

```
[eins] :: ([c1]).       % 1             %c1
[zwei] :: ([c1, -unU]). % 2
[drei] :: ([c1, -ssi, -zeh, -unU]). % 3
[vier] :: ([c1, -zi, -zeh, -unU]). % 4
[fuenf] :: ([c1, -zi, -zeh, -unU]). % 5
[sechs] :: ([c1, -unU]). % 6
[sieben] :: ([c1, -unU]). % 7
[acht] :: ([c1, -zi, -zeh, -unU]). % 8
[neun] :: ([c1, -zi, -zeh, -unU]). % 9
[zehn] :: ([c2, -tausU]). % 10
```

Abb. 4.1.: Ausschnitt aus dem Lexikon "Zahlen_avec_epsilon"

Die Darstellung im Lexikon 4.1 entspricht der Grundform der Terme, welche mit den später aufgeführten Ableitungsregeln abgeleitet werden.
Ein Term besteht hierbei zunächst aus einer Liste an Terminalsymbolen, im weiteren Verlauf als Exponent bezeichnet, gefolgt von einem Typindikator. Ein ”::” gibt hierbei einen lexikalen Term an, während ”:” für einen zusammengesetzten bzw. phrasalen Term steht.
In manchen Stellen dieser Arbeit wird außerdem ein ”·” verwendet. An den entsprechenden Stellen sind dann sowohl lexikale als auch phrasale Terme erlaubt.
Diese Terme lassen sich hierbei beliebig aneinanderketten, was ebenfalls valide Terme erzeugt. Die dabei entstehenden Ketten werden Ausdrücke genannt. Dabei kann nur der erste Term, die Head-Chain, weitere Ableitungsregeln einleiten. Alle weiteren Terme, Sub-Chains genannt, werden durch move-Operationen verarbeitet.

Die folgende Definition für Minimalistische Grammatiken wurde von Stanojević und Stabler [2018] adaptiert:
Eine Minimalistische Grammatik G ist ein 5-Tupel

$$G = (\Sigma, B, Lex, C, \{\text{merge,move}\})$$

Hierbei ist:

- Σ - das **Vokabular**, die Menge der Terminalsymbole,
- B - die **Basisfeatures**, die Menge der Nichtterminalsymbole,
- Lex - das **Lexikon**, die Menge der Grundterme, bestehend aus Σ und B,
- C $\in$ B - eine **Startkategorie**,
- {merge, move} - die **Generatorfunktionen**, fünf Regeln, welche die Ableitungsregeln darstellen

Die Basisfeatures werden mit einem Vorzeichen verbunden, um die Selektoren ($=f|f \in B$), Kategorien ($f|f \in B$), Lizenzgeber ($+f|f \in B$) und Lizenznehmer ($-f|f \in B$) zu erzeugen. Die Vereinigung all dieser **Features** wird mit F bezeichnet. Sei $T = \{:, ::\}$ zwei **Typen** zur Unterscheidung von lexikalen und phrasalen Strukturen.
$\mathbb{C} = \{\Sigma^* \cdot T \cdot F^*\}$ ist die Menge der **Terme**, auch **Chains** genannt. Eine Kette aus diesen ist ein **Ausdruck** $E = \mathbb{C}^+$. Er besteht aus einer Head-Chain und, falls vorhanden, deren Sub-Chains.
Das **Lexikon** ist nun eine endliche Teilmenge der Terme $Lex \subset \{\Sigma^* \cdot \{::\} \cdot F^*\}$. Das leere Wort wird mit ϵ gekennzeichnet.
Die Menge $S(G) = closure(Lex, \{merge, move\})$ beschreibt die Menge aller möglichen **Strukturen** die durch Anwenden der merge- und move-Operationen auf Terme aus dem Lexikon erzeugt werden können.
Eine **Sprache** $L(G) = \{s|s \cdot C \in S(S)\}$ ist die Menge aller Sätze s die als Ausdruck $s \cdot C$ in $S(G)$ auftauchen, wobei $\cdot = \{:, ::\}$ ein Typ und C die Startkategorie ist.
Da Σ und B indirekt über das Lexikon definiert sind und die Ableitungsfunktionen immer gleich bleiben, lässt sich in der Praxis eine Grammatik allein durch ein Lexikon und eine Startkategorie darstellen.

Generatorfunktionen

Folgend werden die Funktionen für merge und move erläutert.
Seien dafür [s] und [t] Exponenten, c ein Feature-Name, γ und δ Featurelisten und $a_1, ..., a_k$ und $b_1, ..., b_l$ mögliche Sub-Chains.

Mergefunktionen

Die Mergefunktionen erhalten zwei Ausdrücke als Eingabe und leiten sie auf einen Ausdruck ab. Sie werden durch einen Selektor initiiert.
Folgende drei merge-Funktionen werden hier benötigt:

Merge 1

Merge 1 hat die größten Einschränkungen. Der intiierende Term muss hierfür ein lexikaler Term sein, während die Featureliste des zweiten Terms nur eine Kategorie und keine anderen Features enthalten darf. Dabei wird das zweite Wort an das Erste gehängt, und die Sub-Chains hinten angehängt.

$$\frac{[s] :: [= c, \gamma] \qquad [t] \cdot [c], a_1, ..., a_k}{[st] : [\gamma], a_1, ..., a_k}$$

Merge 2

Merge 2 funktioniert ähnlich wie Merge 1, verwendet als initiierenden Term nun aber einen phrasalen Term. Dabei wird nun das zweite Wort von vorne an das Erste konkateniert, während die Sub-Chains erneut hinten angehängt werden.

$$\frac{[s] : [= c, \gamma], a_1, ..., a_k \qquad [t] \cdot [c], b_1, ..., b_l}{[ts] : [\gamma], a_1, ..., a_k, b_1, ..., b_l}$$

Merge 3

Merge 3 wird verwendet, wenn der zweite Term nach der Kategorie noch weitere Features besitzt. In diesem Fall wird der zweite Term als Sub-Chain an den Ersten angehängt, ebenso wie seine eigenen Sub-Chains.

$$\frac{[s] : [= c, \gamma], a_1, ..., a_k \qquad [t] \cdot [c, \delta], b_1, ..., b_l}{[s] : [\gamma], a_1, ..., a_k, [t] : [\delta], b_1, ..., b_l}$$

Movefunktionen

Movefunktionen erhalten nur einen Ausdruck als Eingabe und verarbeiten die Head-Chain und eine ihrer Sub-Chains. Das bedeutet, dass mindestens eine Sub-Chain existieren muss, Merge 3 muss also mindestens einmal ausgeführt worden sein. Sie werden durch einen Lizenznehmer initiiert.
Folgende zwei move Funktionen werden benötigt:

Move 1

Move 1 wird eingesetzt, wenn die entsprechende Sub-Chain nur noch einen Lizenznehmer in der Featureliste aufweist. Das entsprechende Wort wird nun vorn an das initiierende konkateniert und die Sub-Chain wird gelöscht.

$$\frac{[s] : [+c, \gamma], a_1, ..., a_k, [t] : [-c], b_1, ..., b_l}{[ts] : [\gamma], a_1, ..., a_k, b_1, ..., b_l}$$

Move 2

Wenn die Featureliste der entsprechenden Sub-Chain noch weitere Features enthält, so wird Move 2 angewendet. Dies entfernt lediglich den Lizenzgeber und Lizenznehmer.

$$\frac{[s] : [+c, \gamma], a_1, ..., a_k, [t] : [-c, \delta], b_1, ..., b_l}{[s] : [\gamma], a_1, ..., a_k, [t] : [\delta], b_1, ..., b_l}$$

Shortest Movement Constraint

Das Shortest Movement Constraint, kurz SMC, ist eine zusätzliche Regel für Minimalistische Grammatiken. Es bezieht sich dabei auf eine Problematik der Sub-Chains, welche durch mehrfache Anwendung von merge3 entstanden sind. Man betrachte beispielhaft den Ausdruck [a]:[+f,c],[b]:[-f],[c]:[-f]. Das erste Feature der Head-Chain ist ein Lizenzgeber, daher muss hier eine Move-Operation angewendet werden. Der Lizenzgeber heißt +f, daher wird ein Lizenznehmer -f benötigt. Dieser existiert nun aber in zwei Sub-Chains. Welche von beiden muss hier nun verwendet werden? Linguistisch macht es Sinn, die Sub-Chains so schnell wie möglich zu moven. Allerdings ist es hier egal, welche Sub-Chain gewählt wird, da die jeweils andere damit stets langsamer verarbeitet wird. Bisher konnte hierfür keine zufriedenstellende Lösung gefunden werden. Deshalb wurde beschlossen, solche Ausdrücke nicht zuzulassen.

4.2. Ein Beispiel-Ableitungsvorgang

Tabelle 4.1 zeigt den Ableitungsvorgang für den Satz [drei,hundert,zwei,und,vier,zig] aus dem Lexikon "Zahlen_sans_epsilon" A.4. Man kann gut erkennen, dass der abgeleitete Ausdruck jeweils im nächsten Schritt wieder verwendet wird. Alle neu hinzugefügten Ausdrücke stammen dabei aus dem Lexikon, wie man gut am Typindikator :: sehen kann.
Nach dem letzten Ableitungsschritt bleibt die Kategorie $c4$ übrig, was im Lexikon "Zahlen_sans_epsilon" der Startkategorie entspricht. [drei,hundert,zwei,und,vier,zig] ist also ein gültiger Satz dieser Grammatik.

Ableitungsschritt	angewendete Regel
$\dfrac{[zig] :: [= c1, +zi, cundZIG] \quad [vier] :: [c1, -zi]}{[zig] : [+zi, cundZIG], [vier] : [-zi]}$	merge3
$\dfrac{[zig] : [+zi, cundZIG], [vier] : [-zi]}{[vier, zig] : [cundzig]}$	move1
$\dfrac{[und] :: [= cundZIG, = cun, c2] \quad [vier, zig] : [cundzig]}{[und, vier, zig] : [= cun, c2]}$	merge1
$\dfrac{[und, vier, zig] : [= cun, c2] \quad [zwei] :: [cun]}{[zwei, und, vier, zig] : [c2]}$	merge2
$\dfrac{[hundert] :: [= c2, = cun, c4] \quad [zwei, und, vier, zig] : [c2]}{[hundert, zwei, und, vier, zig] : [= cun, c4]}$	merge1
$\dfrac{[hundert, zwei, und, vier, zig] : [= cun, c4] \quad [drei] :: [cun]}{[drei, hundert, zwei, und, vier, zig] : [c4]}$	merge2

Tab. 4.1.: Ableitung des "Satzes" [drei,hundert,zwei,und,vier,zig] mithilfe des Lexikons "Zahlen_sans_epsilon"

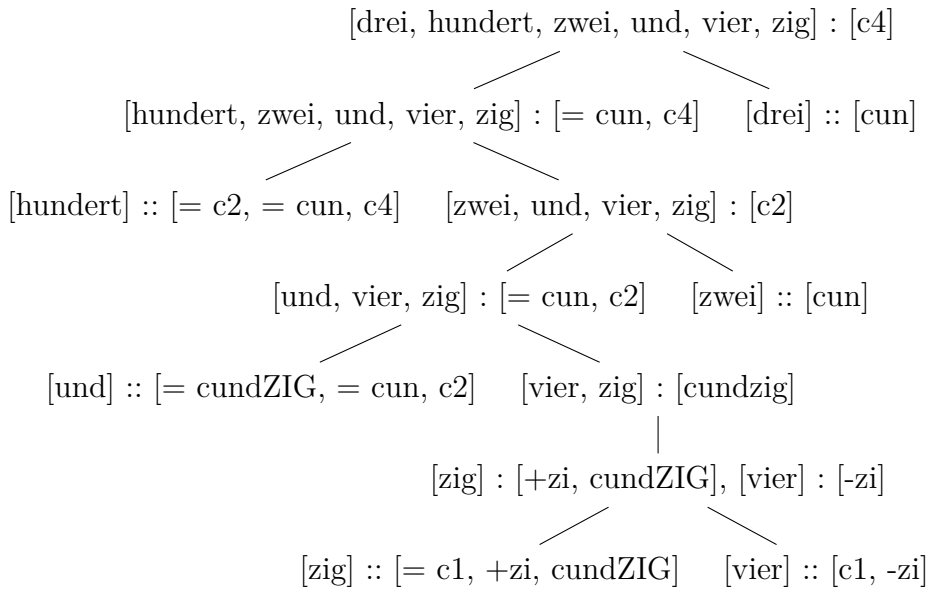

Abb. 4.2.: Ableitungsbaum für [drei, hundert, zwei, und, vier, zig]

Diese Folge von Ableitungen lässt sich grafisch in Form eines Ableitungsbaumes darstellen. Für das oben gezeigte Beispiel sieht der Ableitungsbaum wie folgt aus 4.2.

4.3. Strukturanalyse

Basierend auf den gezeigten Definitionen lassen sich nun einige Aussagen über die Struktur Minimalistischer Grammatiken treffen.

(1) Betrachten wir alle Funktionen, die zwei Exponenten s und t mergen, so sind dies merge1, merge2 und move1. Dabei werden sie in merge2 und move1 zu ts und in merge1 zu st verbunden. Merge1 ist also die einzige Funktion, um den Ausdruck mit der Kategorie hinten anzuhängen.

(2) Move1 und move2 erhalten nur einen Teilbaum als Eingabe und arbeiten mit dessen Sub-Chains. Um solche Sub-Chains zu erhalten, muss merge3 ausgeführt werden. Das bedeutet, dass merge3 mindestens einmal ausgeführt werden muss, bevor move-Operationen möglich sind.

Spezieller fügt merge3 exakt eine Sub-Chain hinzu und move1 verbraucht exakt eine. Da am Anfang keine Sub-Chains vorhanden sind und ein akzeptierter Satz keine enthalten darf, müssen ebenso viele merge3-Funktionen wie move1-Funktionen durchgeführt werden.
Move2 verändert die Anzahl der Sub-Chains nicht.

(3) Move1 und move2 beeinflussen als einzige Funktionen bereits existierende Sub-Chains. Da beide Funktionen nur Lizenzgeber der Head-Chain und Lizenznehmer der Sub-Chains verarbeiten, können andere Features innerhalb einer Sub-Chain nicht mehr verwendet werden. Um akzeptierte Sätze zu erzeugen, dürfen also nur Lizenznehmer in den Featurelisten der Sub-Chains auftauchen.

(4) Tatsächlich sind move1 und move2 auch die einzigen Funktionen, die Lizenzgeber und Lizenznehmer verarbeiten. Demnach müssen alle Lizenznehmer zunächst durch merge3 in einer Sub-Chain landen und können dann erst verarbeitet werden. Daraus können wir schlussfolgern, dass Lizenznehmer in Featurelisten nur ganz hinten auftauchen können.

(5) Betrachten wir nun alle Stellen, an denen Kategorien verarbeitet werden. Bei merge1 und merge2 dürfen sich hinter der Kategorie keine weiteren Features befinden, die Kategorie ist also das letzte Element der Featureliste. Im Falle von merge3 wird der entsprechende Term in eine Sub-Chain umgewandelt, welche nach (3) nur Lizenznehmer enthalten dürfen. Daraus können wir folgern, dass jede Featureliste nur eine Kategorie enthalten kann.

(6) Bei jedem merge werden zwei einzelne Ausdrücke kombiniert und dafür ein Selektor und eine Kategorie entfernt. Um also n lexikale Terme zu einem Satz zu verbinden werden mindestens $n - 1$ Kategorien benötigt. Zusammen mit der Startkategorie, die am Ende übrig bleiben muss, ergibt das mindestens n Kategorien. Zusammen mit (5) bedeutet das, dass jeder lexikale Term genau eine Kategorie besitzen muss.

(7) Aus (2) können wir ebenso folgern, dass move1 und move2 nicht als erste Operation auf einem lexikalen Term ausgeführt werden können. Dementsprechend muss eine Featureliste stets mit einem Selektor oder einer Kategorie beginnen.

(8) Mit (4), (6) und (7) können nun recht genaue Aussagen über die Reihenfolge der Features in einer Featureliste getroffen werden. Wir beginnen mit exakt einer Kategorie. Alle Lizenznehmer müssen sich hinter dieser befinden. Davor können in beliebiger Reihenfolge Lizenzgeber und Selektoren auftauchen, allerdings muss es sich beim ersten Feature dann um einen Selektor handeln.
Hier eine Auflistung aller möglichen gültigen Featurelisten:

I $[c]$ - nur eine Kategorie. Kann von merge1 und merge2 verwendet werden.

II $[c, -l_1, ..., -l_k]$ - eine Kategorie mit k Lizenznehmern. Kann durch ein merge3 gemerged werden. Die entstandene Sub-Chain muss danach durch k-1 move2 und ein move1 aufgelöst werden.

III $[= c_1, c_2]$ - ein Selektor und eine Kategorie. Selektiert I. für ein merge1 oder II für ein merge3. Verhält sich danach wie I.

IV $[= c_1, c_2, -l_1, ..., -l_k]$ - ein Selektor, eine Kategorie und beliebig viele Lizenznehmer. Selektiert I für ein merge1 oder II für ein merge3. Verhält sich danach wie II.

V $[= c_1, = c_2..., = c_n, +l_1, ..., +l_m, c_{n+1}]$ - ein Selektor, eine beliebige Anzahl Selektoren und Lizenzgeber (in beliebiger Reihenfolge) und danach eine Kategorie. Selektiert I für ein merge1 oder II für ein merge3. Bearbeitet danach alle Selektoren mit merge2 oder merge3, alle Lizenzgeber mit move1 und move2 auf den Sub-Chains. Sollten die Sub-Chains den benötigten Lizenznehmer nicht enthalten, so ist der Term ungültig. Verhält sich danach wie II.

VI $[= c_1, = c_2..., = c_n, +l_1, ..., +l_m, c_{n+1}, -l_{m+1}, ..., -l_o]$ - ein Selektor, eine beliebige Anzahl Selektoren und Lizenzgeber (in beliebiger Reihenfolge), eine Kategorie und beliebig viele Lizenznehmer. Selektiert I für ein merge1 oder II für ein merge3. Bearbeitet danach alle Selektoren mit merge2 oder merge3, alle Lizenzgeber mit move1 und move2 auf den Sub-Chains. Sollten die Sub-Chains

den benötigten Lizenznehmer nicht enthalten, so ist der Term ungültig. Verhält sich danach wie II.

An dieser Stelle sei zu notieren, dass nach der Definition der Minimalistischen Grammatiken die hier gezeigten Aussagen nicht eingehalten werden *müssen*. Allerdings werden Featurelisten, die sich nicht an diese Aussagen halten, nie für korrekte Sätze genutzt werden können.

Schleifen in leeren Wörtern

Leere Wörter verhalten sich generell anders als in formalen Grammatiken. Zum einen besitzen sie auch eine Featureliste, wodurch viele verschiedene leere Wörter auftreten können. Zum anderen ist es möglich, leere Wörter miteinander zu mergen, um neue leere Wörter zu erzeugen. Dies kann jedoch erhebliche Probleme verursachen. Man betrachte den Term []:[=c1,c2]. Dieser selektiert zunächst einen anderen Term mit der Kategorie c1 und besitzt dann selbst die Kategorie c2. Man kann dieses leere Wort also verwenden, um einen Term [x]:[c1] zu [x]:[c2] zu ändern. Wenn nun ein zweites leeres Wort []:[=c2,c1] existiert, was der exakten Umkehroperation entspricht, dann entsteht eine Schleife. Wir können nun beliebig oft diese beiden leeren Wörter hintereinander anwenden, ohne den Ausdruck zu verändern. Ein Parser kann hier in eine Endlosschleife geraten. Im weiteren Verlauf dieser Arbeit wird vorausgesetzt, dass die verwendeten Lexika keine derartigen Schleifen enthalten.

5. Parser

5.1. Das Wortproblem

Definition Das Wortproblem:
Das Wortproblem für ein Wort $w \in \Sigma^*$ eines Alphabets Σ und einer zugehörigen Sprache $L \subseteq \Sigma^*$ ist das Entscheidungsproblem, ob w in L enthalten ist ($w \in L$) oder nicht.
Das Wortproblem einer Sprache L im Allgemeinen ist die Frage, ob ein Algorithmus existiert, der für jedes $w \in \Sigma^*$ entscheiden kann, ob es in L liegt oder nicht.

Das Wortproblem stellt eine der entscheidenden Fragen für Sprachen dar. Um für ein Wort zu entscheiden, ob es Teil der Sprache ist, muss also ein entsprechender Ableitungsbaum gefunden werden. Dies lässt sich mithilfe eines Parsers umsetzen.
Ein Parser, meist Teil eines Compilers, ist ein Programm, das eine Eingabe auf syntaktische Korrektheit überprüft und in eine neue Struktur überführt. Er kann also entscheiden, ob ein Wort Teil einer Sprache ist und wandelt die Eingabe dafür in einen Ableitungsbaum um.

Generell gibt es zwei Ansätze für Parser. Zum einen kann mit den einzelnen Terminalsymbolen der Eingabe begonnen werden und man versucht diese so lange zu kombinieren, bis man die gesamte Eingabe erhält. Diese Variante nennt sich Bottom-Up-Analyse. Zum anderen kann mit der gesamten Eingabe begonnen werden und man versucht, durch Anwendung der Ableitungsregeln alle Terminalsymbole zu erreichen. Diese Variante heißt Top-Down-Analyse.

5.2. Bottom-Up-Parser

Bottom-Up-Parser beginnen bei den Blättern und ermitteln die Struktur des Ableitungsbaumes durch iteratives Zusammenfügen der Teilbäume. Eine der wichtigsten Arten von Bottom-Up-Parsern sind shift-reduce-Parser.
Zur Umsetzung eines shift-reduce-Parsers wird ein Stack verwendet. Auf diesen werden nun stückweise, beginnend mit dem ganz rechten, die Zeichen der Eingabe geschrieben. Nach jedem dieser Schritte wird nun überprüft, ob sich rechte Regelseiten der Ableitungsregeln auf dem Stack befinden. Ist dies der Fall, so werden sie durch ihre entsprechenden linken Regelseiten ersetzt. Sollte sich nach dem vollständigen Einlesen und Ersetzen der Eingabe nur noch das Startsymbol auf dem Stack befinden, so wird das Wort akzeptiert.

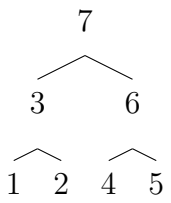

Abb. 5.1.: Ablauf eines Bottom-Up-Parsers

5.3. Top-Down-Parser

Im Gegensatz zum Bottom-Up-Parsing beginnt ein Top-Down-Parser bei der Wurzel und arbeitet sich zu den Blättern vor. Dafür wird die gesamte Struktur des Baumes hypothetisiert und iterativ verfeinert, bis man an bekannten Aussagen (den Blättern) ankommt.
Ein mögliches Verfahren zur Umsetzung eines Top-Down-Parsers sind LL(k)-Parser. Sie beginnen beim Startsymbol und entscheiden durch geschicktes Vorausschauen auf die weitere Eingabe, welche Ableitungsregeln angewendet werden müssen. Eine weitere relevante Art von Top-Down-Parsern sind recursive descent Parser. Hierbei werden die Ableitungsregeln sehr direkt implementiert und das Programm überprüft durch Rekursion und Backtracking, welche Regeln angewendet werden müssen.

Die Implementierung des Parsers dieser Arbeit folgt dem Prinzip eines recursive decent Parsers.

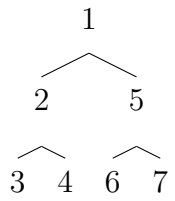

Abb. 5.2.: Ablauf eines Top-Down-Parsers

Es existieren noch weitere Parser wie z.B. left-corner-Parser, doch auf die wird in dieser Arbeit nicht weiter eingegangen.

5.4. Parser für Minimalistische Grammatiken

Das Parsieren von Minimalistischen Grammatiken ist mit einigen Herausforderungen verbunden. Vor allem die strukturellen Gegebenheiten erfordern Anpassungen. So sind die Produktionsregeln bei Minimalistischen Grammatiken bottom-up definiert, während sie bei herkömmlichen Grammatiken top-down sind. Diese Eigenschaft kommt dem Top-Down-Parser zu gute, da er die Ableitungsregeln sozusagen ”rückwärts” anwenden kann. Dies ist bei formalen Grammatiken nur den Bottom-Up-Parsern möglich.
Außerdem werden mögliche Lookahead-Verfahren erschwert, da leere Wörter ebenfalls ermöglicht werden müssen. Da nicht bekannt ist, an welchen Stellen sie sich befinden können, müssen hier alle Möglichkeiten in Betracht gezogen werden.

Es existieren bereits vielfältige Implementierungen von Parsern für MG's. Frühe Konzepte Harkema [2005]; Stabler [2001] waren zunächst bottom-up, der erste Top-Down-Parser wurde 2013 erstellt Stabler [2013]. Neuere Parser enthalten einen transition based Bottom-Up-Parser Stanojević [2016], sowie einen LC-Parser Stanojević und Stabler [2018], eine Mischung aus Top-Down und Bottom-Up Ansätzen.
In Arbeiten an diesem Lehrstuhl wurden bereits ein Umwandler auf MCFGs nach

einem Konzept von Stabler in MATLAB Puchala [2019] und ein Bottom-Up-Parser in Prolog Ulbricht [2024] erstellt.

6. Implementierung

6.1. Aufgabenstellung

Die Aufgabenstellung für diese Arbeit ist die Implementierung eines Top-Down-Parsers für Minimalistische Grammatiken in der Programmiersprache Prolog.
Prolog besitzt einige Eigenschaften, die es stark von herkömmlichen Programmiersprachen unterscheidet. Diese müssen beachtet werden, um ein effektives Programm zu erstellen.
Prolog ist eine logische Programmiersprache. Wir geben also keinen Algorithmus direkt vor, sondern treffen logische Aussagen, welche durch die Sprache ausgewertet werden. Eine Aussage in Prolog sieht im Allgemeinen wie folgt aus:

$$Aussage(Parameter) :- Bedingung1(Parameter), Bedingung2(Parameter)...$$

Dabei ist die Aussage unter den gegebenen Parametern wahr, wenn alle Bedingungen wahr sind. Sind keine Bedingungen gestellt, so ist die Aussage immer wahr.
Prolog überprüft dafür rekursiv die Wahrheit der einzelnen Bedingungen, um dann die Aussage zu bestätigen. Diese Eigenschaft eignet sich hervorragend für einen Top-Down-Parser. Da die Ableitungsregeln Bottom-Up definiert sind, ist jeder Baum korrekt, wenn seine Teilbäume korrekt sind und die Regeln einer der Ableitungsregeln eingehalten werden. Dies lässt sich in Prolog relativ einfach umsetzen.
Weiterhin überprüft Prolog alle möglichen Aussagen, bis für eine alle Bedingungen erfüllt sind. Diese Form von integriertem Backtracking ermöglicht es, einen Parser mit erschöpfender Suche mit wenig Aufwand zu erstellen.

Eine weitere Spezifikation der Aufgabenstellung ist der Fokus auf die bereitgestellten Zahlwortgrammatiken. Die Forschung am Lehrstuhl konzentriert sich zurzeit auf die Konstruktion von Zahlwörtern aus ihren Bestandteilen.

Zahlwortgrammatiken sind zum einen ein guter Einstieg, da Zahlwörter in den meisten Sprachen ähnlich konstruiert werden. Darüber hinaus folgen Zahlwörter einer strikteren Definition als generelle Sätze. Einzelne Wortteile nehmen keinen Einfluss auf entfernte Wortteile, wodurch die move-Operationen in ihrer eigentlichen Funktion nicht benötigt werden. Daher können sie für eine einfachere Kategorisierung "zweckentfremdet" werden, was die Erstellung des Lexikons vereinfacht.
Die Zahlwortgrammatiken lagen für diese Arbeit in zwei verschiedenen Varianten vor. Die eine wurde ohne Verwendung von leeren Wörtern erstellt, die andere mit. Die Variante mit leeren Wörtern erzeugt zwar ein deutlich kleineres Lexikon, allerdings sind leere Wörter aufgrund der Problematiken aus 4.3 schwerer zu verarbeiten.

6.2. Ansatz

Um die Herausforderungen von leeren Wörtern zu umgehen, wurde zunächst nur mit Lexika ohne leere Wörter gearbeitet.
Das erste Konzept sah vor, jeden Baum rekursiv über seine Teilbäume und die Definitionen der merge- und move- Operationen zu definieren. Da die Produktionsregeln Bottom-Up definiert sind, müssen sie erst für eine Verwendung in Top-Down angepasst werden.
Dafür werden die Definitionen aus 4.1 genommen und umgedreht. Die Darstellung erlaubt es, alle nötigen Bedingungen zu erkennen.

Verarbeitung lexikaler Terme
Zunächst benötigt die Rekursion einen Abbruchfall. Da wir rekursiv auf Bäumen arbeiten möchten, entspricht der Abbruchfall den Blättern. In unserem Fall sind das die lexikalen Terme. In den gegebenen Lexika liegen diese in der Form

[Exponent] :: [Featureliste]

vor. Wir können also "::" als Infix-Operator verwenden und das Lexikon dient dann als Definition für alle möglichen Blätter eines Baumes.
Die rekursive Funktion muss nun lediglich bei Eingaben ohne Sub-Chains und nur einem Wort im Exponenten diese auf Einträge im Lexikon überprüfen. Sollte der

Ausdruck nicht im Lexikon stehen, so kann es sich nicht um einen gültigen Baum handeln. Merge1, Merge2 und Move1 sind nicht mehr möglich, da das einzelne Wort nicht mehr zerlegt werden kann. Merge3 und move2 sind unmöglich, da keine Sub-Chains mehr existieren (siehe Erläuterungen der einzelnen Funktionen unten). Das Programm kann hier also abbrechen und Prolog sucht mit Backtracking an anderer Stelle weiter.

merge1:

$$\frac{[st] : [\gamma], a_1, ..., a_k}{[s] :: [= c, \gamma] \qquad [t] \cdot [c], a_1, ..., a_k}$$

Wir starten hier also mit einem Ausdruck mit dem Exponenten st, der Featureliste γ und den Sub-Chains a_1 bis a_k.
Zuerst müssen wir st in zwei Teile s und t aufteilen. Da der Term mit dem s ein lexikaler Term sein muss, darf s nur ein einzelnes Wort enthalten. Wir können die Liste st somit in ihren head und ihren tail zerlegen.
Die Featureliste γ wird in ihrer Gesamtheit an den vorderen Ausdruck angehängt. Als neuer head wird noch ein Selektor $= c$ ergänzt. Da zu diesem Zeitpunkt nicht bekannt ist, welcher Selektor gewählt werden muss, müssen hier alle möglichen Selektoren durchgetestet werden. Prolog übernimmt diesen Part automatisch, wenn $= c$ als variable Größe belassen wird. Der zweite Ausdruck erhält den verbleibenden Exponenten t, als Featureliste die zum Selektor $= c$ gehörende Kategorie c und sämtliche Sub-Chains vom st-Ausdruck, da der s-Ausdruck als lexikaler Term keine Sub-Chains enthalten darf.

merge2:

$$\frac{[ts] : [\gamma], a_1, ..., a_k, b_1, ..., b_l}{[s] : [= c, \gamma], a_1, ..., a_k \qquad [t] \cdot [c], b_1, ..., b_l}$$

merge2 ähnelt merge1 in vielen Stellen. Die hauptsächlichen Unterschiede sind zum Einen, dass ts im Exponenten steht statt st wie bei merge1. Das ist zwar bei der Erzeugung eines Satzes relevant, macht aber für den Parser kaum einen Unterschied. Zum Anderen muss der Ausdruck mit dem Exponenten s nun ein phrasaler Term sein. Da wir leere Wörter an dieser Stelle noch nicht betrachten, muss s also mehr als ein Wort enthalten. Darüber hinaus ist jedoch nicht eindeutig, welche Zerlegung von ts hier die korrekte ist, beziehungsweise ob mehrere Zerlegungen möglich sind. Daher müssen wir mit Prolog hier alle möglichen Zerlegungen überprüfen. Deswei-

teren werden auch die Sub-Chains aufgeteilt. Hierbei ist es jedoch gestattet, dass $a_1, ..., a_k$ oder $b_1, ..., b_l$ leer sind, daher ist eine andere Zerlegungsfunktion nötig.

merge3:

$$\frac{[s] : [\gamma], a_1, ..., a_k, [t] : [\delta], b_1, ..., b_l}{[s] : [= c, \gamma], a_1, ..., a_k \qquad [t] \cdot [c, \delta], b_1, ..., b_l}$$

Bei merge3 muss kein Exponent zerlegt werden, da die Funktion den Ausdruck mit Exponenten t als Sub-Chain an den Ausdruck mit Exponenten s hängt. Effektiv wird also nur die Liste der Sub-Chains zerlegt, wobei wir herausfinden müssen, welche Sub-Chain nun die Head-Chain des neuen t-Ausdrucks wird. Durch den Aufbau der merge3-Funktion ist dann jedoch ersichtlich, wie die übrigen Sub-Chains verteilt werden müssen. Lediglich der korrekte Selektor und seine entsprechende Kategorie müssen noch ermittelt werden.

move1:

$$\frac{[ts] : [\gamma], a_1, ..., a_k, b_1, ..., b_l}{[s] : [+c, \gamma], a_1, ..., a_k, [t] : [-c], b_1, ..., b_l}$$

Bei move1 müssen analog zu merge2 der Exponent ts und die Liste der Sub-Chains zerlegt werden. Im Unterschied zu merge2 werden sie dann aber nicht in zwei Ausdrücke zerlegt, sondern der neue Ausdruck mit dem Exponenten t wird als neue Sub-Chain genau am Trennpunkt der alten Sub-Chain-Liste eingefügt. Die zugehörigen Lizenzgeber und Lizenznehmer werden zuvor ermittelt.
Da wir an dieser Stelle die Liste der Sub-Chains erweitern, ist hier zum ersten Mal eine Überprüfung des Shortest Movement Constraint (SMC) notwendig.

move2:

$$\frac{[s] : [\gamma], a_1, ..., a_k, [t] : [\delta], b_1, ..., b_l}{[s] : [+c, \gamma], a_1, ..., a_k, [t] : [-c, \delta], b_1, ..., b_l}$$

Move2 wird wieder ähnlich wie merge3 umgesetzt. Es wird jedoch nicht an der ausgewählten Sub-Chain getrennt, sondern lediglich Lizenzgeber und Lizenznehmer hinzugefügt. Auch hier muss wieder das SMC überprüft werden.

leere Wörter

Den nun bestehenden Funktionen leere Wörter hinzuzufügen ist relativ einfach. Es müssten nur die Zerlegen-Funktionen für den Exponenten und die Sub-Chain-Liste so umgestaltet werden, dass auch leere Wörter verarbeitet werden können.
Dies ist jedoch aus Effizienzgründen unvorteilhaft. Der Parser sollte versuchen, so wenig leere Wörter wie möglich zu verwenden. Daher ergibt es Sinn, Varianten mit leeren Wörtern erst zu testen, wenn alle anderen Optionen fehlgeschlagen sind.
Dafür wurden bei der Implementierung fünf zusätzliche Funktionen eingefügt, welche die merge- und move-Funktionen explizit mit leeren Wörtern ermöglichen. Diese werden erst nach den ursprünglichen Funktionen ausgeführt.
Ein Problem entsteht bei den Abbruchfällen, der Verarbeitung von lexikalen Termen. Ein einzelnes Wort im Exponenten kann nun auch z.B. durch einen merge mit einem leeren Wort entstanden sein. Daher darf nun nicht mehr nach der Überprüfung des Lexikons abgebrochen werden, die übrigen Ableitungsregeln müssen auch überprüft werden. Dies kann zu Problemen führen, weil damit effektiv der Abbruchfall wegfällt. Das Programm könnte also endlos weiterlaufen, eine häufige Problematik bei der Verarbeitung von leeren Wörtern.
Im nächsten Abschnitt wird eine mögliche Teillösung für dieses Problem erläutert.

Optimierung der Featureliste

Man betrachte die Featurelisten γ und δ in allen umgekehrten Ableitungsfunktionen. In allen Funktionen werden diese entweder belassen wie sie sind oder um ein neues Feature erweitert. Vor allem aber werden nie Features entfernt oder verändert, und wenn Features hinzugefügt werden, dann immer am Anfang der Liste. Dies geschieht solange, bis die Featureliste eines lexikalen Terms vollständig zusammengesetzt wurde. Das bedeutet, dass jede Featureliste, die im Verlauf des Parsens entsteht, das Ende einer im Lexikon auftretenden Featureliste sein muss.
Wir können also eine Funktion erstellen, die jedes mal, wenn eine Featureliste erweitert wird, überprüft, ob diese Featureliste im Lexikon auftaucht. Dies lässt sich sogar noch verbessern, da wir den zur Featureliste gehörenden Exponenten kennen. Demnach darf die Featureliste nur zu einem Wort aus diesem Exponenten oder einem leeren Wort gehören.
Damit lässt sich auch das oben angesprochene Problem beheben. Wenn Schleifen innerhalb der leeren Wörter ausgeschlossen werden können, dann ist durch die Be-

grenzung der Featureliste nur noch eine endliche Anzahl an Ausdrücken möglich; der Parser kann also nicht mehr endlos weiterlaufen.

6.3. Code-Beschreibung

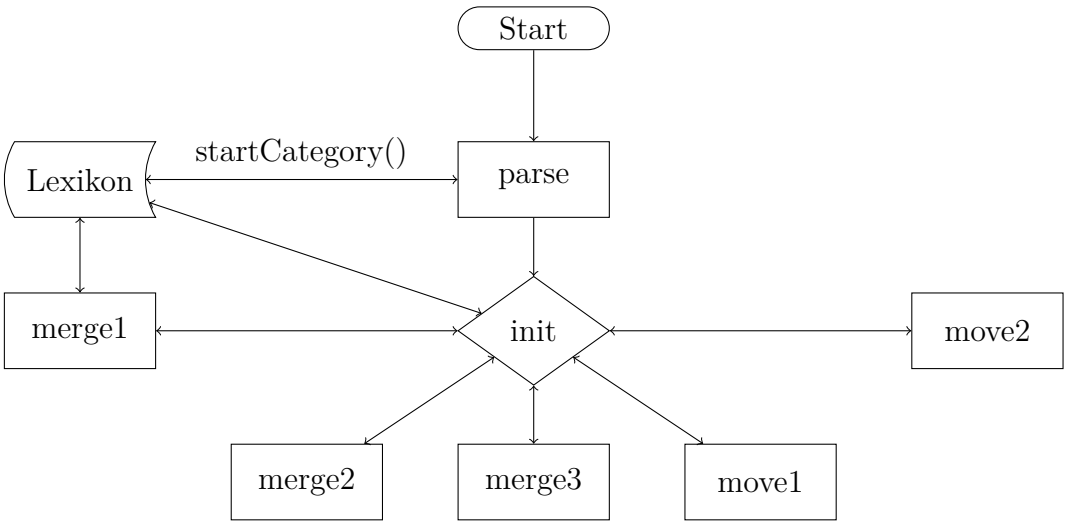

Abb. 6.1.: Ablaufdiagramm des Parsers

Das Programm beginnt mit der Funktion "parse". Diese erhält die Tokenfolge als Eingabe, ebenso wie die freie Variable "T". Diese dient zur späteren Ausgabe des Ableitungsbaumes.
”parse” liest die Startkategorie aus dem Lexikon aus und übergibt sie, zusammen mit der Tokenfolge, an die "init"-Funktion.
Die "init"-Funktion ist der wichtigste Teil des Programms, da er die hauptsächliche Rekursion auf der Eingabe darstellt. Sie bekommt den Ausdruck des Teilbaumes als Eingabe und versucht herauszufinden, welche merge- bzw. move-Funktion an dieser Stelle angewendet wurde. Sie erreicht dies dadurch, dass es insgesamt 12 verschiedene Instanzen gibt, um die unterschiedlichen Funktionen und ihre Sonderfälle zu testen.
Zunächst gibt es zwei mögliche Fälle, die keinen rekursiven Aufruf beinhalten und somit als Abbruchfälle fungieren. Der Erste benötigt eine Eingabe, bei der die Tokenfolge nur ein Element enthält und überprüft, ob dieses Wort mit seiner Featureliste

im Lexikon auftaucht. Ist dies der Fall, so wird T als Blattknoten initiiert, um rekursiv das Ende des Baumes zu erzeugen. Der Zweite arbeitet analog auf einer leeren Tokenfolge in der Eingabe und überprüft dementsprechend, falls vorhanden, ob die Featureliste zu einem leeren Wort aus dem Lexikon passt.
Die übrigen Fälle stellen die fünf Ableitungsregeln dar. Dafür wird der übergebene Ausdruck als Ergebnis betrachtet und alle Variationen probiert, die diesen erzeugen können. Die merge-Funktionen erzeugen dafür zwei neue Ausdrücke, die move-Funktionen einen. Diese werden nun rekursiv an die init-Funktion zurückgegeben. Dieser Vorgang wird solange wiederholt, bis alle Teilstränge beim Abbruchfall angekommen sind oder keine gültige Ableitung möglich ist.
Für jede Ableitungsregel gibt es zwei Funktionen. Die Zweite dient dabei jeweils dem Fall, dass eine der bearbeiteten Chains einen leeren Exponenten besitzt. Dies wurde explizit so umgesetzt, um Ableitungen ohne leere Wörter zu bevorzugen.

Die im Anhang aufgeführten Tabellen A.2 geben ein vollständige Dokumentation des implementierten Parsers an.
Zu beachten sei die Darstellung der Parameter. Ein + bedeutet, dass der entsprechende Parameter bereits initialisiert ist und an die Funktion übergeben wird. Ein − signalisiert eine freie Variable, welche durch die Funktion mit Werten belegt wird.

6.4. Ein Beispiel-Parsiervorgang

Folgend wird der Ablauf eines Parsierungsvorganges anhand eines Beispiels gezeigt. Dafür wurde das Lexikon ”Zahlen_avec_epsilon” A.4 und der Satz ”zweiundvierzig” ausgewählt. Genauere Erklärungen können der Dokumentation A.2 entnommen werden. Der Parser wird mit der Funktion ”parse(X,T)” aufgerufen. X entspricht hierbei unserer Tokenfolge, also [zwei,und,vier,zig]. T wird als Variable freigelassen, es dient der späteren Ausgabe des Ableitungsbaumes.
Parse ruft nun zuerst startCategory(D) auf. Darüber wird die Startkategorie des jeweiligen Lexikons initialisiert. Im Falle von ”Zahlen_avec_epsilon” ist das $c4$.
Wir haben nun also den kompletten Satz und die Startkategorie. Das entspricht

$$[zwei, und, vier, zig] : [c4]$$

dem finalen, akzeptierenden Zustand unserer Grammatik.
Die Tokenfolge und die Startkategorie werden nun in die Struktur eines Ausdrucks übertragen und an die init-Funktion übergeben. Der Ausdruck besteht hierfür aus einer Liste aus Tupeln, welche die einzelnen Chains darstellen. Da in einem akzeptierenden Zustand keine Sub-Chains mehr existieren dürfen, sieht der Ausdruck hier wie folgt aus:

$$[([zwei, und, vier, zig], [c4])]$$

Die init-Funktion testet nun der Reihe nach alle ihre Definitionen durch, bis eine möglich ist.
Begonnen wird mit den beiden Funktionen für die Überprüfung lexikaler Terme. Hier setzt die Funktionsdeklaration voraus, dass der Exponent ein oder kein Wort enthält und die Sub-Chains leer sind. Letzteres ist zwar der Fall, allerdings enthält der Exponent vier Wörter. Daher sind diese Definitionen nicht möglich.
Darauf folgt merge1 als unsere erste Ableitungsregel. Da der linke Teilbaum ein lexikaler Term sein muss, kann er hier nur ein Wort enthalten. Dafür wird der Exponent direkt beim Funktionsaufruf in head und tail unterteilt.

$$[zwei, und, vier, zig] \longrightarrow [zwei], [und, vier, zig]$$

Daraus werden nun zwei neue Ausdrücke erzeugt. Der Erste enthält den linken Exponenten und die bestehende Featureliste mit einem zusätzlichen Selektor.

$$[zwei] :: [= F, c4]$$

Der Selektor =F ist hierbei noch variabel und wird später mit dem Lexikon abgeglichen.
Der zweite Ausdruck enthält den verbleibenden Exponenten und eine neue Featureliste, bestehend aus der zum Selektor =F gehörenden Kategorie F.

$$[und, vier, zig] : [F]$$

Da der erste Ausdruck lexikal sein muss, wird nun direkt mit dem Lexikon abgeglichen, ob ein Term existiert, der =F mit einem gültigen Wert belegen kann.
Das ist hier nicht der Fall, da kein Eintrag im Lexikon existiert, der *zwei* als Exponenten und *c4* in der Featureliste hat.
Merge1 kann an dieser Stelle also abgebrochen werden.

Als nächstes folgt merge2. Hier kommt zuerst die Hilfsfunktion ”zerlegen1” zum Einsatz, welche den Exponenten in alle gültigen Zerlegungen aufteilt. Das sind hier:

- $[zwei], [und, vier, zig]$
- $[zwei, und], [vier, zig]$
- $[zwei, und, vier], [zig]$

Direkt danach folgt die Hilfsfunktion ”featureListe”. Diese überprüft nun, ob für die Wörter der jeweils rechten Featurelisten ein Eintrag im Lexikon existiert, dessen Featureliste auf [=F,c4] endet. Dies wird wieder scheitern.
Prolog nimmt hierfür das erste Ergebnis von ”zerlegen1”, erhält ein False bei ”featureListe” und springt dann automatisch zurück zu ”zerlegen1”, um ein neues Ergebnis zu suchen. Wenn alle Ergebnisse von ”zerlegen1” fehlgeschlagen sind, dann schlägt auch merge2 fehl und wir springen zurück zur init-Funktion.
Da die Hilfsfunktion ”featureListe” auch in den nächsten drei Funktionen fehlschlägt, landen wir schließlich bei merge1Leer. Diese lässt nun zum ersten Mal auch leere Wörter zu, welche an dieser Stelle benötigt werden. Sie beginnt mit der Hilfsfunktion ”zerlegen1Leer”, welche unseren Exponenten effektiv in die beiden Teile

$$[zwei, und, vier, zig] \longrightarrow [], [zwei, und, vier, zig]$$

und

$$[zwei, und, vier, zig] \longrightarrow [zwei, und, vier, zig], []$$

zerlegt. Bei der ersten Zerlegung prüfen wir nun also den Ausdruck

$$[] :: [= F, c4]$$

auf Einträge im Lexikon, wobei wir tatsächlich fündig werden, mit F = c3. Merge1Leer ruft nun init auf dem rechten Ausdruck auf und der Ableitungsbaum schaut bisher wie folgt aus:

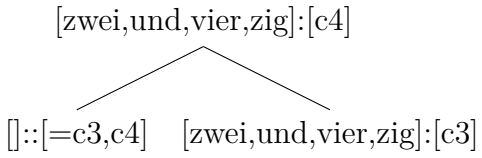

Abb. 6.2.: Ableitungsbaum Schritt 1

Die Auswertung des rechten Teilbaums läuft exakt wie der vorherige Abschnitt ab, da wieder erst merge1Leer eine gültige Ableitung liefert.

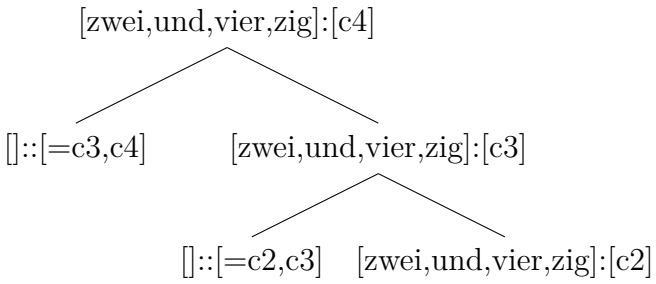

Abb. 6.3.: Ableitungsbaum Schritt 2

Im folgenden Verlauf wird nicht weiter auf fehlschlagende Funktionen eingegangen. Wenn die nächste erfolgreiche Funktion genannt wird, ist davon auszugehen, dass alle vorherigen Funktionen ein negatives Ergebnis aufwiesen. Dies wird oft an der featureListe-Funktion liegen, doch auch andere Operationen wie z.B. die Überprüfung des SMC können dazu führen. In einigen Fällen werden korrekte Ableitungen gefunden, doch in tieferen Rekursionsschritten sind keine weiteren Ableitungen möglich.

Die nächste gültige Ableitung ist ein move1Leer. Wir zerlegen hier den Exponenten in $[zwei, und, vier, zig]$ und [] und bestimmen den Lizenzgeber $+taus$ mit "featurelisteLeer". Die Funktion überprüft hier auch auf SMC und zerlegt die Liste der Sub-Chains, allerdings haben wir zum jetzigen Zeitpunkt noch keine, weshalb hier nichts passiert.

Wir erstellen hier aber nicht zwei neue Ausdrücke, sondern fügen den linken Teil als Sub-Chain in den rechten ein. Das Ergebnis sieht wie folgt aus:

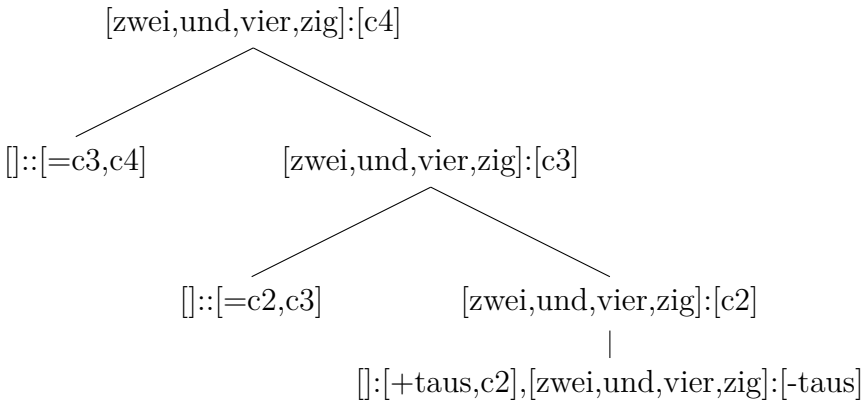

Abb. 6.4.: Ableitungsbaum Schritt 3

Da wir beim jetzigen Stand zum ersten Mal Sub-Chains im auszuwertenden Ausdruck haben, können wir nun merge3 und move2 in Betracht ziehen. Beide Funktionen setzen für eine erfolgreiche Ableitung nämlich die Existenz von Sub-Chains voraus.
In diesem Falle wenden wir merge3Leer an, was zu folgendem Baum führt:

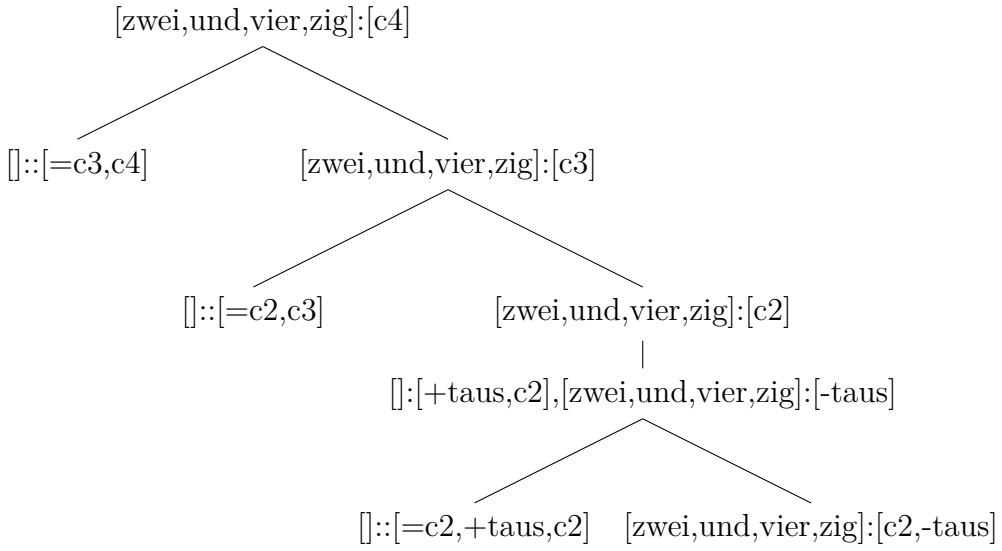

Abb. 6.5.: Ableitungsbaum Schritt 4

Im nächsten Schritt wird merge2 angewandt. Dies erzeugt in diesem Beispiel zum ersten mal die Situation, dass beide Teilbäume nicht lexikal sind und weiter abgeleitet werden müssen.

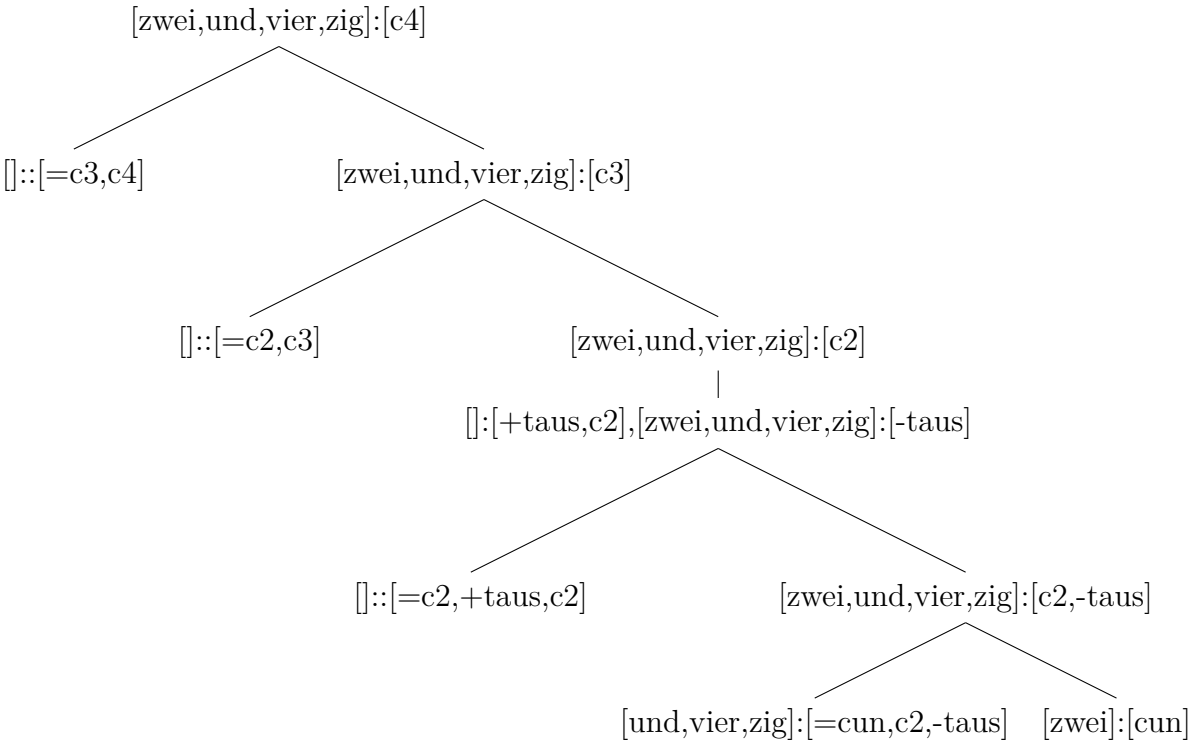

Abb. 6.6.: Ableitungsbaum Schritt 5

Da Prolog alle Bedingungen in der gegebenen Reihenfolge abarbeitet und der rekursive init-Aufruf zuerst auf dem linken Ausdruck durchgeführt wird, wird nun erst der komplette linke Teilbaum zusammengesetzt, bevor der rechte erstellt wird. Aufgrund der Übersichtlichkeit wurden hier alle weiteren Ableitungsschritte zusammengefasst.

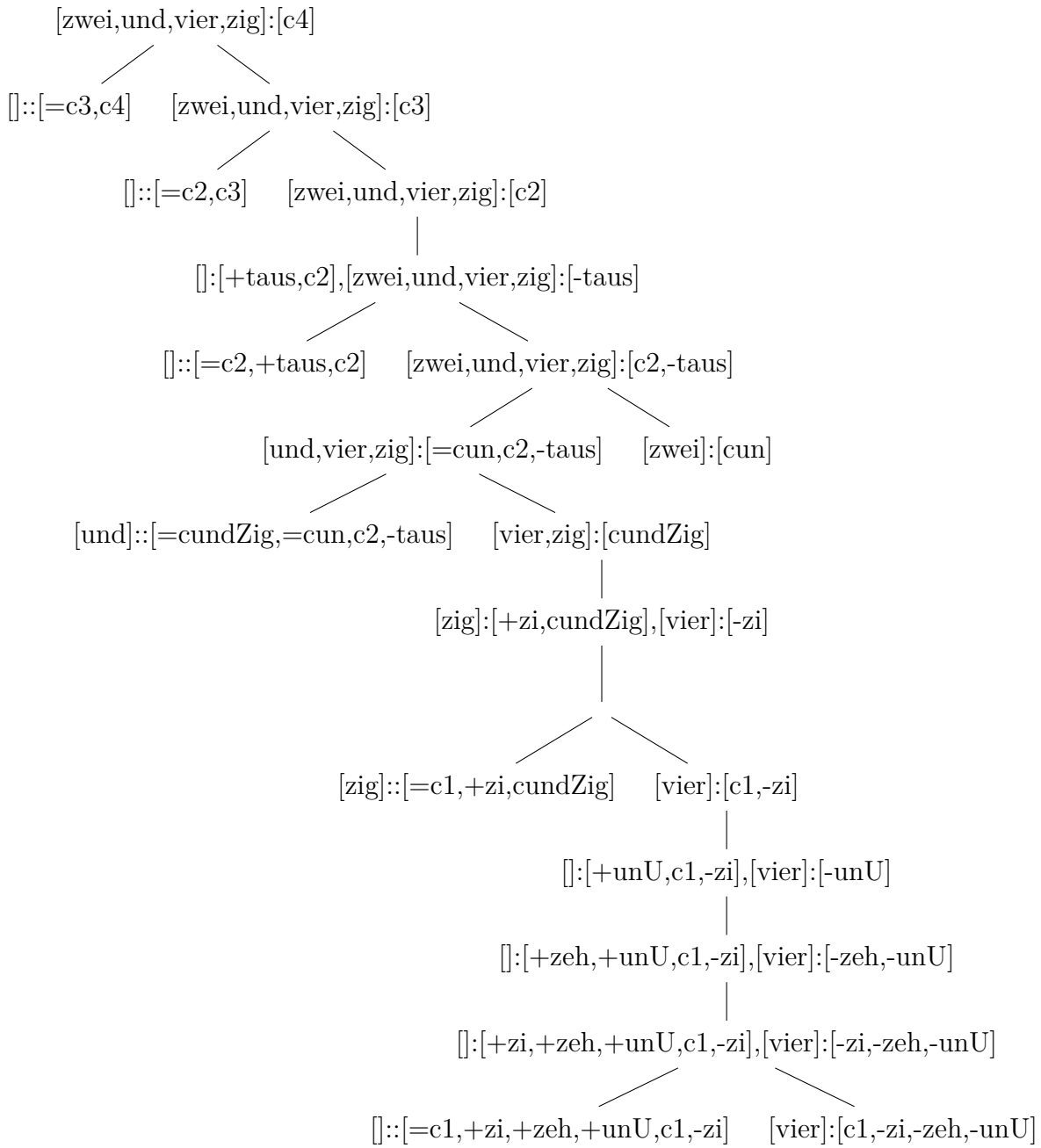

Abb. 6.7.: vollständige Ableitung des linken Teilbaums. Man beachte den Term [zwei]:[cun], welcher noch nicht fertig abgeleitet wurde.

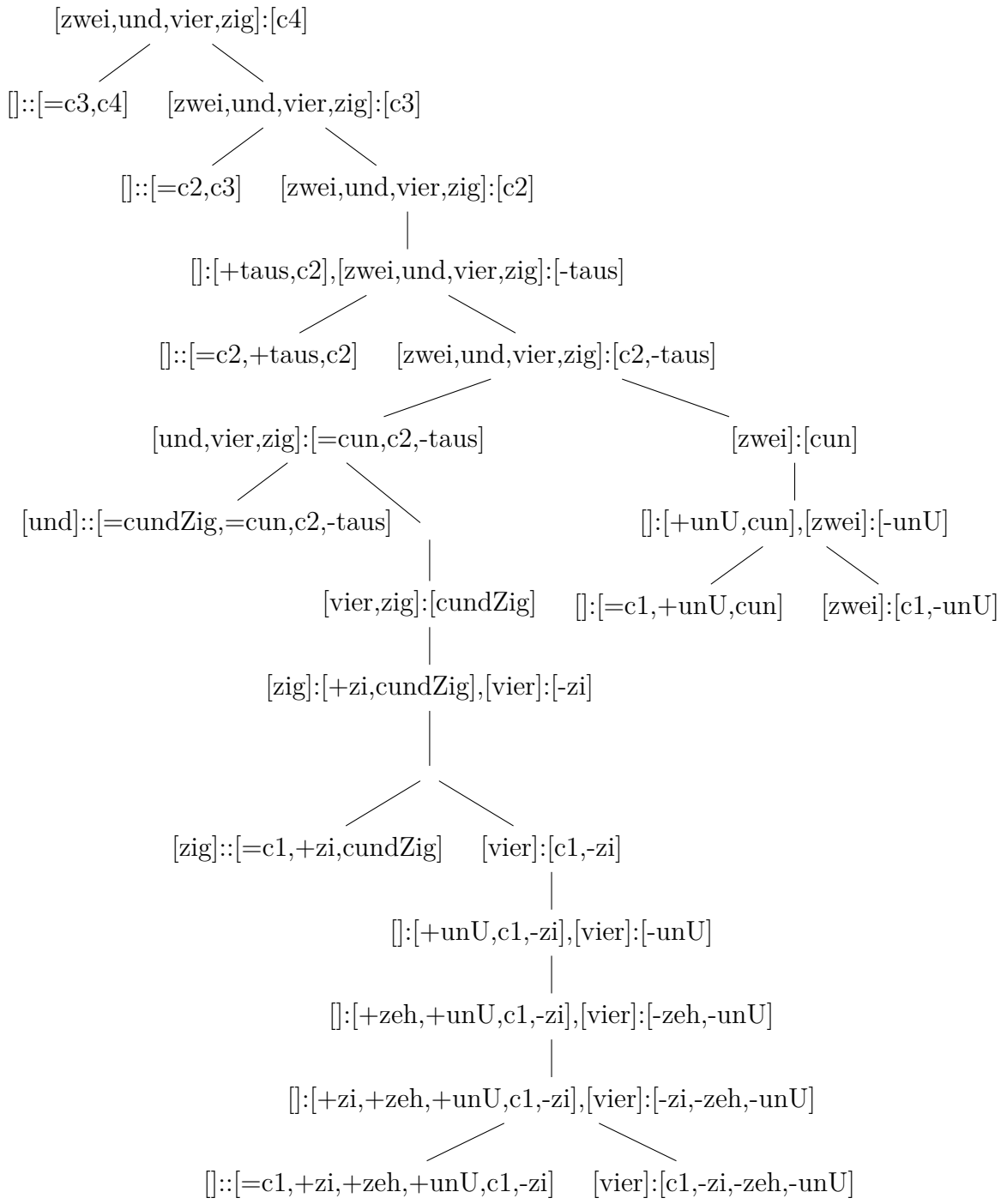

Abb. 6.8.: finaler Ableitungsbaum

6.5. Laufzeitanalyse

Das Testen der Laufzeit erwies sich als kompliziert, da der erste Programmdurchlauf stets mehr Zeit benötigt als alle Folgenden. Um Aussagen über die maximale Laufzeit zu tätigen war es also notwendig, das Programm für jede Eingabe neu zu starten. Es war daher nicht möglich, eine vollumfängliche Laufzeitanalyse durchzuführen.
Eingehende Hypothese für die Tests war die Annahme, dass die Laufzeit mit steigender Anzahl Wörter im eingegebenen Satz ebenfalls steigt. Die Tests wurden ebenfalls mit dem Lexikon ”Zahlen_avec_epsilon” gemacht. Alle getesteten Sätze wurden nach ihrer Anzahl Wörter klassifiziert und jeweils fünf mal getestet, um Schwankungen auszugleichen. Die längsten in diesem Lexikon möglichen Sätze hatten dabei eine Länge von dreizehn Wörtern.
Alle gemessenen Laufzeiten lagen zwischen 0,000 und 0,932 Sekunden. Es ist also davon auszugehen, dass für Lexika mit einer vergleichbaren Größe ähnlich effiziente Laufzeiten zu erwarten sind. Die Laufzeiten sind zufriedenstellend.
Die gemessenen Werte deuten auf ein exponentielles Wachstum nach der Anzahl Wörter hin (6.9).

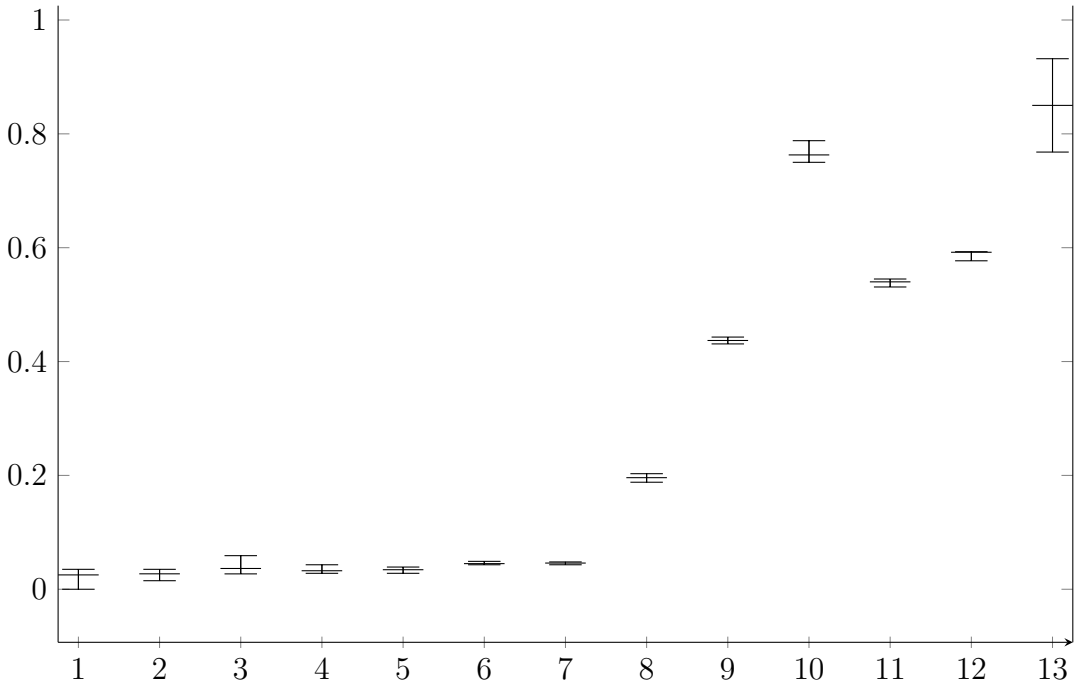

Abb. 6.9.: Laufzeitanalyse

7. Zusammenfassung und Ausblick

Ziel dieser Arbeit war die Implementierung eines Top-Down-Parsers für Minimalistische Grammatiken. Dieser kann nun verwendet werden, um neue Lexika effizienter zu erstellen und zu überprüfen. Dies ermöglicht ein tieferes Verständnis von Minimalistischen Grammatiken zu erhalten, um bessere Aussagen über ihre Fähigkeit, menschliche Sprache abzubilden, zu erstellen.
Dafür wurde aufbauend auf Arbeiten von vor allem Edward P. Stabler und Milos Stanojevic ein neuer Top-Down-Parser in Prolog erstellt. Diese Programmiersprache wurde hierbei explizit ausgewählt, da sich Prolog besonders gut für die Erstellung von Parsern eignet. Entstanden ist ein effizienter, übersichtlicher und gut lesbarer Parser, der alle geforderten Kriterien erfüllt:

- Die Implementierung aller Definitionen und Regeln von Minimalistischen Grammatiken wurde erfolgreich umgesetzt
- Die bereitgestellten Lexika wurden eingehend getestet und können problemlos verarbeitet werden
- Auch Lexika mit leeren Wörtern sind durch die Implementierung abbildbar

Darüber hinaus ist die Funktionalität für allgemeine Lexika ebenfalls vorhanden, diese konnten jedoch bisher nicht ausführlich getestet werden. Ein bereits bekanntes Problem stellen hier Schleifen innerhalb von leeren Wörtern dar, da diese dazu führen können, dass der Parser nicht terminiert.
Dieses Problem lässt sich möglicherweise durch eine Beschränkung der Rekursionstiefe oder eine Vorverarbeitung des Lexikons umgehen. Auch ein zusätzliches Programm zur Erkennung von Schleifen wurde hypothetisiert, jedoch ist an dieser Stelle weitere Forschung nötig.
Das Programm kann alle bisher überprüften Sätze in relativ kurzer Zeit ableiten. Trotzdem wurden bereits weitere Überlegungen zur Effizienzsteigerung gemacht.
Bei der Implementierung der Verarbeitung leerer Wörter wurde entschieden, die

Tests auf leere Wörter von nicht leeren Wörtern zu trennen, da dies nach anfänglichen Tests schneller erschien als andere Alternativen. Dies wurde nicht ausreichend getestet, weshalb andere Varianten effizienter sein könnten.
Besonders bei den Regeln für merge3 und move2 ist eine Separierung fragwürdig, da hier keine Exponenten verändert werden. Weitere Tests sind hier nötig.
Darüber hinaus kann die Reihenfolge der Funktionsdeklarationen in Prolog große Unterschiede machen. Durch die stark rekursive Natur von Prolog können Änderungen der Reihenfolge der Ableitungsregeln oder der Lexikoneinträge enormen Einfluss auf die Laufzeit haben. Dies kann mit weiteren Tests überprüft werden.
Weitere Verfahren zur Steigerung der Effizienz wie eine Vorverarbeitung des Lexikons oder die Zwischenspeicherung von Teilergebnissen wurden in dieser Arbeit nicht behandelt, bieten aber einen möglichen Ausblick auf weitere Forschung.

Literaturverzeichnis

Chomsky 1959

Chomsky, Noam: On certain formal properties of grammars. In: *Information and control* 2 (1959), Nr. 2, S. 137–167 3.3

Harkema 2005

Harkema, Henk: A recognizer for minimalist languages. In: *New Developments in Parsing Technology* (2005), S. 251–268 5.4

Puchala 2019

Puchala, Magdalena A.: Inkrementeller Parser für minimalistische Grammatiken. (2019), Oktober 5.4

Stabler 1996

Stabler, Edward: Derivational minimalism. In: *International conference on logical aspects of computational linguistics* Springer, 1996, S. 68–95 2

Stabler 2001

Stabler, Edward P.: *Minimalist grammars and recognition.* na, 2001 5.4

Stabler 2010

Stabler, Edward P.: Computational perspectives on minimalism. (2010), S. 616–641 3.3

Stabler 2013

Stabler, Edward P.: Two models of minimalist, incremental syntactic analysis. In: *Topics in cognitive science* 5 (2013), Nr. 3, S. 611–633 5.4

Stanojević 2016

Stanojević, Miloš: Minimalist Grammar Transition-Based Parsing. (2016), S. 273–290. ISBN 978–3–662–53825–8 5.4

Stanojević und Stabler 2018

Stanojević, Miloš ; Stabler, Edward P.: A Sound and Complete Left-Corner Parsing for Minimalist Grammars. (2018), S. 65–74 4.1, 5.4

Ulbricht 2024

Ulbricht, Leonie A.: Realisierung eines Bottom-Up-Parsers für Minimalistische Grammatiken. (2024), Februar 5.4

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-Dokumentation

Funktionsname	Parameter	Beschreibung
parse	+X : Tokenfolge −T : Ableitungsbaum	Gibt für eine Tokenfolge den zugehörigen Ableitungsbaum aus. Dafür ruft sie die init-Funktion mit der im Lexikon angegebenen Startkategorie auf.
zerlegen1	+X : Eingabeliste −L : linke Ausgabeliste −R : rechte Ausgabeliste	Zerlegt eine Liste in zwei Teile. Beide Teillisten müssen nichtleer sein. Erzeugt automatisch alle möglichen Varianten.
zerlegen2	+X : Eingabeliste −L : linke Ausgabeliste −R : rechte Ausgabeliste	Zerlegt eine Liste in zwei Teile. R == [] und L == [] sind erlaubt. Erzeugt automatisch alle möglichen Varianten.
zerlegenLeer	+X : Eingabeliste −L : linke Ausgabeliste −R : rechte Ausgabeliste	Zerlegt eine Liste in zwei Teile. Entweder R oder L oder beide müssen hierbei eine leere Liste sein. Erzeugt automatisch alle möglichen Varianten.
featureliste	+S : Exponent +D : Featureliste. Der head dieser Liste ist variabel. −D : Featureliste. Der head wurde mit einem Wert belegt.	Die übergebene Featureliste ist mit einem variablen head übergeben worden. Diese Wird nun mit allen Wörtern des Exponenten, ob es im Lexikon eine gültige Belegung für die freie Variable gibt. Falls ja werden alle verschiedenen Featurelisten zurückgegeben.
featurelisteLeer	+S : Exponent +D : Featureliste. −D : Featureliste.	Funktionalität wie bei Featureliste, mit zusätzlicher Überprüfung der leeren Wörter.
smc	+D : Feature +Y : Liste der Sub-Chains	Umsetzung des Shortest Movement Constraint. Gibt true zurück, wenn D kein head einer Featureliste in Y ist.

Tab. A.1.: Programm-Dokumentation Teil 1

Funktionsname	Parameter	Beschreibung
init	+[(S,F)|Y] : Ausdruck +S : Exponent +F Featureliste +Y Liste der Sub-Chains −T : Ableitungsbaum	Allgemeiner rekursiver Aufruf. Gibt Abletungsbaum T zurück, wenn [(S,F)|Y] ein gültiger Ausdruck ist. Wenn S nur ein oder kein Wort enthält, wird mit dem Lexikon abgeglichen. Sonst werden alle unten stehenden Funktionen initialisiert.
merge1	+S : Exponent +F Featureliste +Y Liste der Sub-Chains −T : Ableitungsbaum	Definition von merge1. Zerlegt S in head und tail und erstellt zwei Unterbäume. Überprüft den linken Teilbaum direkt mit dem Lexikon und ruft init auf dem rechten auf.
merge1Leer	+S : Exponent +F Featureliste +Y Liste der Sub-Chains −T : Ableitungsbaum	Definition von merge1 speziell für leere Wörter. Funktionsaufbau wie bei merge1. Einer der beiden Teilbäume hat einen leeren Exponenten.
merge2	+S : Exponent +F Featureliste +Y Liste der Sub-Chains −T : Ableitungsbaum	Definition von merge2. Zerlegt S und F in jeweils zwei Teile und erstellt zwei Unterbäume. Rekursiver init-Aufruf auf beide Teilbäume. Optimierung durch featureListe.
merge2Leer	+S : Exponent +F Featureliste +Y Liste der Sub-Chains −T : Ableitungsbaum	Definition von merge2 speziell für leere Wörter. Funktionsaufbau wie bei merge2. einer der beiden Teilbäume hat einen leeren Exponenten.

Tab. A.2.: Programm-Dokumentation Teil 2

merge3	+S : Exponent +F Featureliste +Y Liste der Sub-Chains −T : Ableitungsbaum	Definition von merge3. Zerlegt Y in zwei Teile und erstellt zwei Unterbäume. Der head vom rechten Y-Teil wird neue Head-Chain. Optimierung durch featureListe.
merge3Leer	+S : Exponent +F Featureliste +Y Liste der Sub-Chains −T : Ableitungsbaum	Definition von merge3 speziell für leere Wörter. Funktionsaufbau wie bei merge3. einer der beiden Teilbäume hat einen leeren Exponenten.
move1	+S : Exponent +F Featureliste +Y Liste der Sub-Chains −T : Ableitungsbaum	Definition von move1. Zerlegt S und Y, erstellt eine neue Sub-Chain und fügt diese an der Zerlegestelle von Y ein. init-Aufruf auf dem gesamten Ausdruck. Optimierung durch featureListe.
move1Leer	+S : Exponent +F Featureliste +Y Liste der Sub-Chains −T : Ableitungsbaum	Definition von move1 speziell für leere Wörter. Funktionsaufbau wie move1. Die head-Chain oder die neue Sub-Chain können einen leeren Exponenten haben.
move2	+S : Exponent +F Featureliste +Y Liste der Sub-Chains −T : Ableitungsbaum	Definition von move2. Zerlegt Y und fügt neues Feature bei head-Chain und ausgewählter Sub-Chain hinzu. Optimierung durch featureListe.
move2Leer	+S : Exponent +F Featureliste +Y Liste der Sub-Chains −T : Ableitungsbaum	Definition von move2 speziell für leere Wörter. Funktionsaufbau wie move2. Head-Chain oder ausgewählte Sub-Chain können leere Exponenten haben.

Tab. A.3.: Programm-Dokumentation Teil 3

A.3. Code

```
% Auflistung der verwendeten Lexika
% Hierbei sollte stets nur ein Lexikon aktiv sein, der Rest sollte auskommentiert bleiben.
%:- [maus].
:- ['Zahlen_avec_epsilon'].
%:-['Lexikon_test_full_sans'].
%:- ['German'].
%:- ['English'].


/*
* Nutzeraufruf.
*    Parsiert die eingegebene Tokelfolge X und gibt, wenn möglich, den
*    erhaltenen Ableitungsbaum zurück. Mehrere Ableitungsbäume sind möglich.
*
* @param X: Tokenfolge, der zu parsende Satz, in seine lexikalen Terme zerlegt
* @param T: Ableitungsbaum, das Ergebnis der Funktion.
*/
parse(X,T):- startCategory(D),init([(X,[D])],T),write("T= "),write(T).

/*
* Hilfsfunktionen zum Zerlegen von Listen.
*   Sie zerlegen die Liste X in zwei Teillisten L & R.
*      zerlegen1 gibt alle Zerlegungen von X aus, bei denen L & R nicht leer sind.
*      zerlegen2 gibt alle Zerlegungen von X aus, L & R dürfen leer sein.
*      zerlegenLeer gibt die Zerlegungen von X aus, bei denen mindestens eine Teilliste leer ist.
*   die 2. zerlegenLeer-Funktion spezifiziert als Eingabe [X|XS], um Redundanz zu vermeiden, wenn X leer ist.
*
* @param X: Eingabeliste
* @param L: linke Teilliste
* @param R: rechte Teilliste
*/
zerlegen1(X,L,R):-append(L,R,X),L\=[],R\=[].
zerlegen2(X,L,R):-append(L,R,X).
zerlegenLeer(X,L,R):- L=[],R=X.
zerlegenLeer([X|XS],L,R):- L=[X|XS],R=[].

/*
* Hilfsfunktion zum Auswählen möglicher Features.
*   D ist hierbei mit eine Variable als Head initialisiert.
*   Die Funktion bestimmt für alle Wörter aus S, ob eine Featureliste existiert, die auf D endet.
```

```
41 *   Ist dies der Fall, so werden alle verschiedenen Belegungen vom Head von D zurü
       ckgegeben.
42 *   Ist keine Belegung möglich, so wird false zurückgegeben.
43 *   featurelisteLeer erlaubt neben S auch leere Wörter aus dem Lexikon
44 *
45 * @param S: Liste mit Wörtern
46 * @param D: gewünschte Teil-Featureliste
47 * @param W: einzelnes Wort aus S
48 * @param F: Featureliste des Wortes W
49 */
50 featureliste_sub(S,D):- member(W,S),::([W],F),zerlegen2(F,_,D).
51 featureliste(S,D):- distinct(D,featureliste_sub(S,D)).
52
53 featureliste_subLeer(S,D):- member(W,S),::([W],F),zerlegen2(F,_,D).
54 featureliste_subLeer(_,D):- ::([],F),zerlegen2(F,_,D).
55 featurelisteLeer(S,D):- distinct(D,featureliste_subLeer(S,D)).
56
57 /*
58 * Hilfsfunktion zur Umsetzung des Shortest-Movement-Constraints.
59 *   Erhält ein Feature D und die Liste der Sub-Terme als Eingabe und gibt True zurück,
60 *   wenn D in keiner Featureliste der Sub-Terme dem Head endspricht.
61 *
62 * @param D: zu überprüfendes Feature
63 * @param [(_,[-F|_])|YS]: Liste der Sub-Terme. Nur der Head der Featureliste ist relevant
       .
64 */
65 smc(_,[]).
66 smc(D,[(_,[-F|_])|YS]):- D\=F, smc(D,YS).
67
68 /*
69 * Zentraler rekursiver Aufruf.
70 *   Erhält einen Term als Eingabe und überprüft, ob dieser als Ableitungsbaum erzeugt
       werden kann.
71 *   Dafür wird zunächst überpruft, ob der Term bereits ein lexikaler Term ist.
72 *   Danach werden die merge- und move-Funktionen rückwärts angewendet und die init-
       Funktion rekursiv auf
73 *   den entstehenden Teilbäumen ausgeführt.
74 *   Die Varianten mit leeren Wörtern wurden aus Performance-Gründen ausgelagert und
       werden erst am Ende überprüft.
75 *
76 * @param [(S,F)|Y]: Eingabe-Term mit
77 *   @param S: Tokenfolge
78 *   @param F: Featureliste
79 *   @param Y: Sub-Terme
80 * @param T: Ableitungsbaum
81 */
82 init([([],F)],T):- ::([],F),T=li([],F).
```

```
83 init([([X],F)],T):- ::([X],F),T=li([X],F).
84 init([(S,F)|Y],T):- merge1([(S,F)|Y],T).
85 init([(S,F)|Y],T):- merge2([(S,F)|Y],T).
86 init([(S,F)|Y],T):- merge3([(S,F)|Y],T).
87 init([(S,F)|Y],T):- move1([(S,F)|Y],T).
88 init([(S,F)|Y],T):- move2([(S,F)|Y],T).
89 init([(S,F)|Y],T):- merge1Leer([(S,F)|Y],T).
90 init([(S,F)|Y],T):- merge2Leer([(S,F)|Y],T).
91 init([(S,F)|Y],T):- merge3Leer([(S,F)|Y],T).
92 init([(S,F)|Y],T):- move1Leer([(S,F)|Y],T).
93 init([(S,F)|Y],T):- move2Leer([(S,F)|Y],T).
94
95 /*
96 * Umsetzung der merge- und move- Funktionen.
97 * Essentiell auch Teil des rekursiven init-Aufrufs, aber aus Übersichtlichskeitsgrünen
       ausgelagert.
98 * Jede Funktion setzt eine der Grundfunktionen invertiert um, die Versionen mit -Leer im
       Namen
99 * bilden die Funktionen speziell für leere Wörter ab.
100 *
101 * Die Parameter werden im Allgemeinen so wie in den anderen Funktionen verwendet
102 */
103 merge1([([X|XS],F)|Y],T):- XS\=[],
104     ::([X],[=D|F]),
105     init([(XS,[D])|Y],TR),T=tree([([X|XS],F)|Y],li([X],[=D|F]),TR).
106
107 merge1Leer([(S,F)|Y],T):- zerlegenLeer(S,L,R),
108     ::(L,[=D|F]),
109     init([(R,[D])|Y],TR),T=tree([(S,F)|Y],li(L,[=D|F]),TR).
110
111 merge2([(S,F)|Y],T):- zerlegen1(S,R,L),featureliste(L,[=D|F]),\+(::(L,[=D|F])),zerlegen2(
       Y,Y1,Y2),
112     init([(L,[=D|F])|Y1],TL),
113     init([(R,[D])|Y2],TR),T=tree([(S,F)|Y],TL,TR).
114
115 merge2Leer([(S,F)|Y],T):- zerlegenLeer(S,R,L),featurelisteLeer(L,[=D|F]),\+(::(L,[=D|F]))
       ,zerlegen2(Y,Y1,Y2),
116     init([(L,[=D|F])|Y1],TL),
117     init([(R,[D])|Y2],TR),T=tree([(S,F)|Y],TL,TR).
118
119 merge3([(S,F)|Y],T):- featureliste(S,[=D|F]),zerlegen2(Y,Y1,[(S2,F2)|Y2]),featureliste(S2
       ,[D|F2]),
120     init([(S,[=D|F])|Y1],TL),
121     init([(S2,[D|F2])|Y2],TR),T=tree([(S,F)|Y],TL,TR).
122
123 merge3Leer([(S,F)|Y],T):- featurelisteLeer(S,[=D|F]),zerlegen2(Y,Y1,[(S2,F2)|Y2]),
       featurelisteLeer(S2,[D|F2]),
```

```
124     init([(S,[=D|F])|Y1],TL),
125     init([(S2,[D|F2])|Y2],TR),T=tree([(S,F)|Y],TL,TR).
126
127 move1([(S,F)|Y],T):- zerlegen1(S,L,R),featureliste(R,[+D|F]),smc(D,Y),zerlegen2(Y,Y1,Y2),
128     append([[(R,[+D|F])],Y1,[(L,[-D])],Y2],X),init(X,TL),T=tree([(S,F)|Y],TL,epsilon).
129
130 move1Leer([(S,F)|Y],T):- zerlegenLeer(S,L,R),L\=[],featurelisteLeer(R,[+D|F]),smc(D,Y),
      zerlegen2(Y,Y1,Y2),
131     append([[(R,[+D|F])],Y1,[(L,[-D])],Y2],X),init(X,TL),T=tree([(S,F)|Y],TL,epsilon).
132
133 move2([(S,F)|Y],T):- zerlegen2(Y,Y1,[(S2,F2)|Y2]),featureliste(S,[+D|F]),featureliste(S2
      ,[-D|F2]),smc(D,Y1),smc(D,Y2),
134     append([[(S,[+D|F])],Y1,[(S2,[-D|F2])],Y2],X),init(X,TL),T=tree([(S,F)|Y],TL,epsilon)
          .
135
136 move2Leer([(S,F)|Y],T):- zerlegen2(Y,Y1,[(S2,F2)|Y2]),featurelisteLeer(S,[+D|F]),
      featurelisteLeer(S2,[-D|F2]),smc(D,Y1),smc(D,Y2),
137     append([[(S,[+D|F])],Y1,[(S2,[-D|F2])],Y2],X),init(X,TL),T=tree([(S,F)|Y],TL,epsilon)
          .
138
139
140
141 %+ smc testen
142 %+ move2 testen
143 %+ fehlerhafe Eingaben testen
144 %- leere Woerter
145 %- neue Lexika verwenden
146 %- eigene Lexika erstellen
```

A.4. Verwendete Lexika

```
1  :- op(500, xfy, ::). % infix predicate for lexical items
2  :- op(500, fx, =). % for selection features
3
4  startCategory(c4).
5  [vier] :: [c1].
6  [vier] :: [c1,-zeh].
7  [neun] :: [c1].
8  [neun] :: [c1,-zeh].
9  [fuenf] :: [c1].
10 [fuenf] :: [c1,-zeh].
11 [drei] :: [c1].
12 [drei] :: [c1,-zeh].
13 [acht] :: [c1].
```

```
[acht] :: [c1,-zeh].
[zwoelf] :: [c4].
[zwoelf] :: [c3].
[zwoelf] :: [c2].
[zwoelf] :: [c3,-taus].
[zwoelf] :: [c2,-tausU].
[zwei] :: [c4].
[zwei] :: [c3].
[zwei] :: [c3,-taus].
[zwei] :: [c2].
[zwei] :: [cun].
[zwei] :: [c1,-un].
[zwei] :: [c1].
[zwei] :: [c2,-tausU].
[zwei] :: [c1,-unU].
[zwanzig] :: [c4].
[zwanzig] :: [c3].
[zwanzig] :: [c2].
[zwanzig] :: [c3,-taus].
[zwanzig] :: [c2,-tausU].
[zig] :: [=c1,+zi,cundZIG].
[zig] :: [=c1,+zi,c4].
[zig] :: [=c1,+zi,c3].
[zig] :: [=c1,+zi,c2].
[zig] :: [=c1,+zi,c3,-taus].
[zig] :: [=c1,+zi,c2,-tausU].
[zehn] :: [=c1,+zeh,c4].
[zehn] :: [=c1,+zeh,c3].
[zehn] :: [=c1,+zeh,c2].
[zehn] :: [=c1,+zeh,c3,-taus].
[zehn] :: [=c1,+zeh,c2,-tausU].
[zehn] :: [c4].
[zehn] :: [c3].
[zehn] :: [c2].
[zehn] :: [c3,-taus].
[zehn] :: [c2,-tausU].
[vier] :: [c4].
[vier] :: [c3].
[vier] :: [c2].
[vier] :: [c1,-zi].
[vier] :: [c1,-zi,-zeh,-unU].
[undzwanzig] :: [=c1,+un,c4].
[undzwanzig] :: [=c1,+un,c3].
[undzwanzig] :: [=c1,+un,c2].
[undzwanzig] :: [=c1,+un,c3,-taus].
[undzwanzig] :: [=c1,+un,c2,-tausU].
[undsiebzig] :: [=c1,+un,c4].
```

```
[undsiebzig] :: [=c1,+un,c3].
[undsiebzig] :: [=c1,+un,c2].
[undsiebzig] :: [=c1,+un,c3,-taus].
[undsiebzig] :: [=c1,+un,c2,-tausU].
[undsechzig] :: [=c1,+un,c4].
[undsechzig] :: [=c1,+un,c3].
[undsechzig] :: [=c1,+un,c2].
[undsechzig] :: [=c1,+un,c3,-taus].
[undsechzig] :: [=c1,+un,c2,-tausU].
[und] :: [=cundZIG,=cun,c4].
[und] :: [=cundZIG,=cun,c3].
[und] :: [=cundZIG,=cun,c2].
[und] :: [=cundZIG,=cun,c3,-taus].
[und] :: [=cundZIG,=cun,c2,-tausU].
[tausend] :: [=c3,=c3,+taus,c4].
[tausend] :: [=c3,+taus,c4].
[ssig] :: [=c1,+ssi,cundZIG].
[ssig] :: [=c1,+ssi,c4].
[ssig] :: [=c1,+ssi,c3].
[ssig] :: [=c1,+ssi,c2].
[ssig] :: [=c1,+ssi,c3,-taus].
[ssig] :: [=c1,+ssi,c2,-tausU].
[siebzig] :: [c4].
[siebzig] :: [c3].
[siebzig] :: [c2].
[siebzig] :: [c3,-taus].
[siebzig] :: [c2,-tausU].
[siebzehn] :: [c4].
[siebzehn] :: [c3].
[siebzehn] :: [c2].
[siebzehn] :: [c3,-taus].
[siebzehn] :: [c2,-tausU].
[sieben] :: [c4].
[sieben] :: [c3].
[sieben] :: [c3,-taus].
[sieben] :: [c2].
[sieben] :: [cun].
[sieben] :: [c1,-un].
[sieben] :: [c1].
[sieben] :: [c2,-tausU].
[sieben] :: [c1,-unU].
[sechzig] :: [c4].
[sechzig] :: [c3].
[sechzig] :: [c2].
[sechzig] :: [c3,-taus].
[sechzig] :: [c2,-tausU].
[sechzehn] :: [c4].
```

```
[sechzehn] :: [c3].
[sechzehn] :: [c2].
[sechzehn] :: [c3,-taus].
[sechzehn] :: [c2,-tausU].
[sechs] :: [c4].
[sechs] :: [c3].
[sechs] :: [c3,-taus].
[sechs] :: [c2].
[sechs] :: [cun].
[sechs] :: [c1,-un].
[sechs] :: [c1].
[sechs] :: [c2,-tausU].
[sechs] :: [c1,-unU].
[neun] :: [c4].
[neun] :: [c3].
[neun] :: [c2].
[neun] :: [c1,-zi].
[neun] :: [c1,-zi,-zeh,-unU].
[hundert] :: [=c2,=cun,c4].
[hundert] :: [=c2,=cun,c3].
[hundert] :: [=c2,=cun,c3,-taus].
[hundert] :: [=c1,+un,c4].
[hundert] :: [=c1,+un,c3].
[hundert] :: [=c1,+un,c3,-taus].
[fuenf] :: [c4].
[fuenf] :: [c3].
[fuenf] :: [c2].
[fuenf] :: [c1,-zi].
[fuenf] :: [c1,-zi,-zeh,-unU].
[elf] :: [c4].
[elf] :: [c3].
[elf] :: [c2].
[elf] :: [c3,-taus].
[elf] :: [c2,-tausU].
[einundzwanzig] :: [c4].
[einundzwanzig] :: [c3].
[einundzwanzig] :: [c2].
[einundzwanzig] :: [c3,-taus].
[einundzwanzig] :: [c2,-tausU].
[einundsiebzig] :: [c4].
[einundsiebzig] :: [c3].
[einundsiebzig] :: [c2].
[einundsiebzig] :: [c3,-taus].
[einundsiebzig] :: [c2,-tausU].
[einundsechzig] :: [c4].
[einundsechzig] :: [c3].
[einundsechzig] :: [c2].
```

```
[einundsechzig] :: [c3,-taus].
[einundsechzig] :: [c2,-tausU].
[einund] :: [=cundZIG,c4].
[einund] :: [=cundZIG,c3].
[einund] :: [=cundZIG,c2].
[einund] :: [=cundZIG,c3,-taus].
[einund] :: [=cundZIG,c2,-tausU].
[eintausend] :: [=c3,c4].
[eintausend] :: [c4].
[eins] :: [c4].
[eins] :: [c3].
[eins] :: [c2].
[eins] :: [c1].
[einhundert] :: [=c2,c4].
[einhundert] :: [=c2,c3].
[einhundert] :: [=c2,c3,-taus].
[einhundert] :: [c4].
[einhundert] :: [c3].
[einhundert] :: [c3,-taus].
[drei] :: [c4].
[drei] :: [c3].
[drei] :: [c2].
[drei] :: [c1,-ssi].
[drei] :: [c1,-ssi,-zeh,-unU].
[acht] :: [c4].
[acht] :: [c3].
[acht] :: [c2].
[acht] :: [c1,-zi].
[acht] :: [c1,-zi,-zeh,-unU].
[acht] :: [cun].
[acht] :: [c1,-un].
[acht] :: [c3,-taus].
[acht] :: [c2,-tausU].
[drei] :: [cun].
[drei] :: [c1,-un].
[drei] :: [c3,-taus].
[drei] :: [c2,-tausU].
[fuenf] :: [cun].
[fuenf] :: [c1,-un].
[fuenf] :: [c3,-taus].
[fuenf] :: [c2,-tausU].
[neun] :: [cun].
[neun] :: [c1,-un].
[neun] :: [c3,-taus].
[neun] :: [c2,-tausU].
[vier] :: [cun].
[vier] :: [c1,-un].
```

```
202 [vier] :: [c3,-taus].
203 [vier] :: [c2,-tausU].
```

```
1  startCategory(c4).
2
3  :- op(900,xfx,::).
4  :- op(200,fy,=).
5
6  [zig] :: ([=c1, +zi, c2, -tausU]). % 10 * A %Hauptfunktionen
7  [zig] :: ([=c1, +zi, cundZIG]). % 10 * A
8  [und] :: ([ =cundZIG, =cun, c2, -tausU]). % A + B
9  [zehn] :: ([=c1, +zeh, c2, -tausU]). % A + 10
10 [hundert] :: ([=c2, =cun, c3, -taus]). % A + 100 * B
11 [hundert] :: ([=c1, +un, c3, -taus]). % 100 * A
12 [einhundert] :: ([=c2, c3, -taus]). % A + 100
13 [einhundert] :: ([c3, -taus]). % 100
14 [tausend] :: ([=c3, =c3,+taus, c4]). % A + 1000 * B
15 [tausend] :: ([=c3, +taus, c4]). % 1000 * A
16 [eintausend] :: ([=c3, c4]). % A + 1000
17 [eintausend] :: ([c4]). % 1000
18
19 [eins] :: ([c1]). % 1    %c1
20 [zwei] :: ([c1, -unU]). % 2
21 [drei] :: ([c1, -ssi, -zeh, -unU]). % 3
22 [vier] :: ([c1, -zi, -zeh, -unU]). % 4
23 [fuenf] :: ([c1, -zi, -zeh, -unU]). % 5
24 [sechs] :: ([c1, -unU]). % 6
25 [sieben] :: ([c1, -unU]). % 7
26 [acht] :: ([c1, -zi, -zeh, -unU]). % 8
27 [neun] :: ([c1, -zi, -zeh, -unU]). % 9
28 [zehn] :: ([c2, -tausU]). % 10  %diverse Ausnahmen
29 [elf] :: ([c2, -tausU]). % 11
30 [zwoelf] :: ([c2, -tausU]). % 12
31 [sechzehn] :: ([c2, -tausU]). % 16
32 [siebzehn] :: ([c2, -tausU]). % 17
33 [zwanzig] :: ([c2, -tausU]). % 20
34 [einundzwanzig] :: ([c2, -tausU]). % 21
35 [sechzig] :: ([c2, -tausU]). % 60
36 [einundsechzig] :: ([c2, -tausU]). % 61
37 [siebzig] :: ([c2, -tausU]). % 70
38 [einundsiebzig] :: ([c2, -tausU]). % 71
39 [undzwanzig] :: ([=c1, +un, c2, -tausU]). % A + 20
40 [ssig] :: ([=c1, +ssi, c2, -tausU]). % 0 * A + 30
41 [ssig] :: ([=c1, +ssi, cundZIG]). % 0 * A + 30
42 [einund] :: ([=cundZIG, c2, -tausU]). % A + 1
43 [undsechzig] :: ([=c1, +un, c2, -tausU]). % A + 60
44 [undsiebzig] :: ([=c1, +un, c2, -tausU]). % A + 70
```

```
45
46
47 [] :: ([=c1, +unU, cun]).
48 [] :: ([=c1, +unU,c1,-un]).
49 [] :: ([=c1, +zi, c1]). % epsilon
50 [] :: ([=c1, +zi, +zeh, +unU, c1, -zi]). % epsilon
51 [] :: ([=c1, +ssi, c1]). % epsilon
52 [] :: ([=c1, +ssi, +zeh, +unU, c1, -ssi]). % epsilon
53 [] :: ([=c1, +zeh, c1]). % epsilon
54 [] :: ([=c1, +zeh, +unU, c1, -zeh]). % epsilon
55 [] :: ([=c1, +unU, c1]). % epsilon
56 [] :: ([=c2, +tausU, c2]). % epsilon
57 [] :: ([=c3, +taus, c3]). % epsilon
58 [] :: ([=c3, c4]). % epsilon
59 [] :: ([=c2, +tausU, c3, -taus]). % epsilon
60 [] :: ([=c2, c3]). % epsilon
61 [] :: ([=c1, +unU, c2, -tausU]). % epsilon
62 [] :: ([=c1, c2]). % epsilon
```