BTU Cottbus–Senftenberg
Fakultät 1 MINT - Mathematik, Informatik, Physik, Elektro- und Informationstechnik
Lehrstuhl Kommunikationstechnik
Prof. Dr.-Ing. habil. Matthias Wolff

Brandenburgische Technische Universität Cottbus - Senftenberg

Inkrementeller Parser für minimalistische Grammatiken

– Bachelorarbeit –

Magdalena Agnieszka Puchała
Informatik

Cottbus, Oktober 2019

Betreuer: Dr. rer. nat. Peter beim Graben

Eidesstattliche Erklärung

Der Verfasser erklärt an Eides statt, dass er die vorliegende Arbeit selbständig, ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt hat. Die aus fremden Quellen (einschließlich elektronischer Quellen) direkt oder indirekt übernommenen Gedanken sind ausnahmslos als solche kenntlich gemacht. Die Arbeit ist in gleicher oder ähnlicher Form oder auszugsweise im Rahmen einer anderen Prüfung noch nicht vorgelegt worden.

Ort, Datum Unterschrift des Verfassers

Kurzfassung

In dieser Arbeit wird die Implementierung eines inkrementellen top-down Parsers für minimalistische Grammatiken nach dem Konzept von Stabler präsentiert. Seine Idee besteht darin, zu einer minimalistischen Grammatik eine äquivalente multiple kontextfreie Grammatik zu finden und dafür einen top-down Parser mit einer Prioritätswarteschlange zu erstellen. Die Umsetzung des Programms erfolgte in MATLAB. Jedoch hat Stabler die Erstellung der multiplen kontextfreien Regeln anhand nur eines Ableitungsbaumes gezeigt, was bei der Voraussetzung, dass der geparste Satz unbekannt ist und es potentiell unendlich viele Ableitungsbäume existieren, zu einer endlose Schleife geführt hat. Die Lösung dieses Problems erfolgte durch Begrenzung der Rekursionstiefe.

Abstract

This thesis presents an implementation of an incremental top-down parser for minimalist grammars based on Stabler's concept. His idea is to find an equivalent multiple context-free grammar to a minimalist grammar and to create a top-down parser with a priority queue for it. The program was implemented in MATLAB. However, Stabler has shown the creation of multiple context-free rules using only one derivation tree, which can lead to an endless loop since the parsed sentence is unknown and there are potentially infinite derivation trees. The solution to this problem was found by limiting the recursion depth.

Abbildungsverzeichnis

1 Kontextfreie Beispielgrammatik für einen Satz aus der natürlichen Sprache[1] . . . 5
2 Minimalistisches Lexikon von Stabler[2] . . . 8
3 Ableitungsbaum der MG von Stabler[2] . . . 9
4 Kontextfreie Beispielgrammatik . . . 10
5 Top-down Erzeugung eines Ableitungsbaumes . . . 10
6 Top-down Verarbeitung des Wortes 'aaabbb' . . . 11
7 Bottom-up Erzeugung eines Ableitungsbaumes . . . 12
8 Top-down Verarbeitung des Wortes 'aaabbb' . . . 13
9 Zahlengrammatik mit semantischen Ausdrücken[3] . . . 15
10 MKFG zum Satz 'Mary knows who John likes' . . . 16
11 Top-down Verarbeitung des Satzes 'Mary knows who John likes' . . . 17
12 Parserverlauf mit Semantikstapel . . . 17
13 Minimalistische Beispielgrammatik ohne semantische Ausdrücke . . . 18
14 Beispielbandinhalt . . . 18
15 Reihenfolge der Elternregeln beibehalten . . . 20
16 Reihenfolge der Elternregeln getauscht . . . 20
17 Multiple kontextfreie Beispielgrammatik . . . 21
18 Problematik der Operation merge-3 . . . 22
19 Vereinfachte MKFG ohne semantische Ausdrücke . . . 23
20 Vereinfachte MKFG mit semantischen Ausdrücken . . . 23
21 Parserverlauf des Satzes 'who likes John' . . . 29
22 Parserverlauf des Satzes 'zwei und vier zig' . . . 29

Inhaltsverzeichnis

Eidesstattliche Erklärung **i**

Kurzfassung **iii**

Abstract **iii**

Abbildungsverzeichnis **v**

1 Einführung **1**

2 Grundlagen **2**
2.1 Grammatiken und Sprachen 2
2.1.1 Kontextfreie Grammatik 5
2.1.2 Multiple kontextfreie Grammatik 6
2.1.3 Minimalistische Grammatik 7
2.2 Parser 9
2.2.1 Top-down Verarbeitung 10
2.2.2 Bottom-up Verarbeitung 12
2.3 Lambda-Kalkül 13

3 Inkrementeller Parser für minimalistische Grammatiken **16**
3.1 Das Konzept von Edward Stabler 16
3.2 Semantische Verarbeitung 17

4 Implementierung in MATLAB **18**
4.1 Grammatikumwandlung 19
4.2 Parsing 25
4.3 Programbeurteilung 30

5 Zusammenfassung **31**

Literatur **32**

1 Einführung

Seit Jahren probieren Linguisten natürliche Sprache zu formalisieren. Ein Ziel davon ist es ein kognitives System zu schaffen, das die menschliche Sprache versteht, interpretiert und anwendet. Natürliche Sprache zu formalisieren ist ein komplexes Problem, da sie die typische Domäne von Menschen ist – nirgendwo im Tierreich gibt es so ein entwickeltes Kommunikationssystem. Diese Fähigkeit ist deswegen nicht vergleichbar mit solchen Problemen wie z.B. die Bildverarbeitung oder die Bewegungssteuerung, die auch im Tierreich anzutreffen sind.[4]

Bereits in der Schule wird gelehrt, dass für korrekte Formulierungen in einer Sprache die Erkenntnis der Grammatik dieser Sprache notwendig ist. Auf der Suche nach der Formalisierung der natürlichen Sprache sind die formalen Grammatiken entstanden. Der große Verdienst dabei geht auf Noam Chomsky zurück, der Schöpfer der Chomsky Hierarchie. Er hat auch den Begriff *minimalistisches Programm* geprägt, dessen Formalisierung die minimalistischen Grammatiken sind. Diese arbeiten anhand eines minimalistischen Lexikons, d.h. eine Sammlung von Wörtern und sind viel versprechend im Bezug zur Verarbeitung der natürlichen Sprache.

Es ist nicht ausreichend die grammatischen Regeln zu kennen, es muss auch eine Möglichkeit geben zu entscheiden, ob ein gegebener Satz grammatisch korrekt ist oder nicht. Zu diesem Zweck wurde das Konzept des Parsers entworfen. Es wurden bereits verschiedene Konzepte an Parsern entwickelt, wovon zur Sprachverarbeitung ein inkrementeller Parser bevorzugt wird. Die Besonderheit dieser Parser besteht darin, dass sie die Struktur des Dokuments in Form seines Ableitungsbaumes aufbewahren. Dieser Baum wird benutzt um dieses Dokument nach Modifikationen zu aktualisieren.[5] Das bedeutet, dass nach eventuellen Änderungen der Inhalt nicht erneut verarbeitet werden muss. Solcher Vorteil kann durch Benutzung eines top-down Parsers erreicht werden.

Auf Grund der Spezifität der minimalistischen Grammatiken, ist die Erzeugung eines top-down Parsers dafür problematisch. Ein Konzept für solche Verarbeitung wurde von Stabler präsentiert.[2] Ziel dieser Arbeit ist es anhand dieses Konzeptes ein Programm zu schreiben, um zu prüfen, ob Stablers Idee realisierbar ist.

2 Grundlagen

2.1 Grammatiken und Sprachen

Obwohl jeder instinktiv weiß was eine Sprache ist, ist dies nicht so leicht zu definieren. Laut Noam Chomsky, ist die Sprache eine (endliche oder unendliche) Menge von Sätzen endlicher Länge, die aus einer endlichen Menge von Elementen konstruiert wurden.[6] Chomsky probierte den Aufbau natürlicher Sprache mittels formaler Logik und Begriffsbildung zu beschreiben – so ist die Theorie der formalen Sprachen entstanden.[1] Es sei wohlgemerkt, dass in formalen Sprachen ein Satz auch als ein Wort verstanden werden kann. Zum Beispiel 'Zweiundvierzig' ist ein Satz aus der Sprache, der durch die Zahlengrammatik[3] definiert wird, obwohl nach instinktiven Verständnis würde es als ein Wort betrachtet werden. Nachfolgend werden in dieser Arbeit die Begriffe ‚Wort' und ‚Satz' als Synonyme benutzt.

Bei der Definition einer Sprache ist es notwendig zuerst ein Alphabet zu definieren.

Ein Alphabet Σ ist eine endliche, nicht-leere Menge von Zeichen, wobei:

- ϵ das leere Wort (leere Folge von Zeichen) ist
- Σ^* Menge aller Wörter (d.h. aller endlichen Folgen von Zeichen) über Σ (inklusiv leeres Wort) ist.
- Σ^+ Menge aller nicht-leeren Wörter von Σ^* ist

Dabei ist eine formale Sprache L eine Teilmenge von Σ^* und ein Satz s (von L) eine Zeichenfolge $s \in$ L.[7] Jedoch ist die Information L $\in \Sigma^*$, nicht ausreichend, um zu bestimmen, welche Wörter genau zur Sprache gehören. Diese müssen also noch zusätzlich definiert werden, was auf die vier unten aufgezählten Arten erfolgen kann.[7] Als Beispiel ist ein Alphabet Σ = {a,b} gegeben und dazu definierte Sprache, der aus Wörter besteht, die am Anfang a's (zulässig auch leeres Wort) und am Ende b's (zulässig auch leeres Wort) haben.

1. Aufzählung aller gültigen Wörter, z.B. ϵ, a, ab, aaab, aabbb usw.
   Da es in einer Sprache unendlich viele Wörter geben kann, ist diese Methode meistens unpraktisch.
2. Mittels regulären Ausdrücke (Definition unten), z.B. L = {a*b*}.
3. Mittels eines Automaten.
4. **Mittels einer Grammatik**.

Als zweite Methode zur Sprachdefinition wird das Benutzen von regulären Ausdrücke vorgeschlagen. Für ein Alphabet Σ werden die reguläre Ausdrücke wie folgt definiert:

- Jedes Element des Alphabets Σ...
- Das leere Wort ϵ...
- Die leere Menge...

... ist ein regulärer Ausdruck.

Dazu sind a und b reguläre Ausdrücke, so ist auch:

- ihre Alternative: a|b
- ihre Verkettung: ab
- eine Wiederholung(Kleene-Stern): a* (alle Wörter, die nur als a bestehen und das leere Wort)

ein regulärer Ausdruck.[7]

Reguläre Ausdrücke können mit einem Automaten erkannt werden – und somit kann auch die Sprache durch einen Automaten definiert werden. Da die Automaten keinen wesentlichen Bestandteil dieser Arbeit bilden, werden diese nicht weiter betrachtet.

Eine formale Grammatik G ist ein Tupel G = (V,T,R,S). Dabei ist:

- **V** eine endliche Menge von Variablen (Nichtterminale), die meistens mit Gtoßbuchstaben bezeichnet werden.
- **T** eine endliche Menge von Terminalen, die meistens mit Kleinbuchstaben bezeichnet werden; es gilt V∩T = ∅.
- **R** eine endliche Menge von Produktionsregeln.
- **S** ein Startsymbol S∈V.

Eine Sprache L(G), die von G erzeugt wird, ist die Menge aller terminalen Wörter, die mit Hilfe der Produktionsregeln aus S erzeugt werden können.[8] Als Beispiel sei die Sprache L = {a*b*} gegeben, zu der folgende Grammatik formuliert werden kann: G = ({S,A,B},{a,b},R,S), wobei die Produktionsregeln wie folgt aussehen:

$S \rightarrow AB$
$S \rightarrow \epsilon$
$A \rightarrow aA$
$A \rightarrow \epsilon$
$B \rightarrow bB$
$B \rightarrow \epsilon$

Da es mehreren Ableitungsmöglichkeiten der einzelnen Nichtterminale gibt, kann die Form der Grammatik vereinfacht werden:

$S \rightarrow AB \mid \epsilon$
$A \rightarrow aA \mid \epsilon$
$B \rightarrow bB \mid \epsilon$

Mittels dieser Grammatik werden jetzt zwei Wörter auf Sprachzugehörigkeit geprüft: w = aabbb und v = aababb.

$S \to AB \to aAB \to aaAB \to aaB \to aabB \to aabbB \to aabbbB \to aabbb$

Das Wort w kann aus S abgeleitet werden und besteht nur aus Terminalen, womit $w \in L$.
$S \to AB \to aAB \to aaAB \to aaB \to aabB$

Da keine Regel existiert, die ein 'a' aus einem 'B' erzeugt, ist keine weitere Ableitung möglich. Das Endwort enthält ein Nichtterminal, also $v \notin L$.

Es sei angemerkt, dass die Terminale den regulären Ausdrücke entsprechen und mit Hilfe der Produktionsregel können die Alternative, Verkettung und Kleene-Stern beschrieben werden:

- **Alternative**

  $L = \{a,b\}$

  $S \to a|b$

- **Verkettung**

  $L = \{ab\}$

  $S \to ab$

- **Kleene-Stern**

  $L = \{a^*\}$

  $S \to aS \mid \epsilon$

In den weiteren Abschnitten dieser Arbeit wird die Grammatikdefinition nur auf den Produktionsregeln begrenzt mit Annahme, dass Kleinbuchstaben die Terminale und Großbuchstaben die Nichtterminale bezeichnen. Ausnahme bilden die minimalistischen Grammatiken, die anders definiert werden. Als Startsymbol wird immer S verwendet.

Chomsky hat in seinen Arbeiten[9,10] die Grammatiken in vier Gruppen eingeteilt, abhängig von Automatentyp und Beschränkungen der Produktionsregeln, die zur Erkennung bestimmter Grammatik geeignet sind. Diese Aufteilung ist als Chomsky-Hierarchie bekannt.

Es wurden folgende Typen unterscheidet:

- **Typ 0** – unbeschränkte Grammatiken

  Produktionsregeln haben Form von $P \to R$

  wobei P und R beliebig sind und $P \neq \epsilon$.

- **Typ 1** – kontextsensitive Grammatiken

  Produktionsregeln haben Form von $r_1 P r_2 \to r_1 R r_2$

  wobei P ein Nichtterminal ist, R, r_1 und r_2 beliebig sind. P und R treten jeweils in gleichem Kontext auf.

- **Typ 2** – kontextfreie Grammatiken

  Produktionsregeln haben Form von $P \to R$

  wobei P ein Nichtterminal und R beliebig ist.

- **Typ 3** – reguläre Grammatiken

  Produktionsregeln haben Form von P $\rightarrow$ rR oder P $\rightarrow$ R

  wobei P und R Nichtterminale sind und r ein Terminal ist.

Es gilt Typ 3 $\subseteq$ Typ 2 $\subseteq$ Typ 1 $\subseteq$ Typ 0.

In Abschnitt 2.1.1 wird die Gruppe der kontextfreien Grammatiken und ihre Bedeutung zur Analyse der natürlichen Sprachen näher besprochen. Anschließend werden die Multiple Kontextfreie Grammatiken (MKFG), die aus der Chomsky Hierarchie stammen, beschrieben. Zuletzt wird Aufbau und Bedeutung einer Minimalistischen Grammatik (MG) erklärt.

2.1.1 Kontextfreie Grammatik

Mit Hilfe der kontextfreien Grammatiken kann nicht nur eine künstliche Sprache wie L = $\{a^n b^n | \ n \in \mathbb{N}\}$ formuliert werden, sondern auch solche, die mit der natürlichen Sprache mehr zu tun hat. Ein Beispiel dazu zeigt eine Grammatik,[1] aus der der Satz 'der Hund beißt den Briefträger' erzeugt werden kann.

Regel 1: SATZ $\rightarrow$ SUBJEKT PRÄDIKAT OBJEKT
Regel 2: SUBJEKT $\rightarrow$ ARTIKEL SUBSTANTIV
Regel 3: OBJEKT $\rightarrow$ ARTIKEL SUBSTANTIV
Regel 4: PRÄDIKAT $\rightarrow$ VERB
Regel 5: SUBSTANTIV $\rightarrow$ Hund | Briefträger
Regel 6: VERB $\rightarrow$ beißt
Regel 7: ARTIKEL $\rightarrow$ den | der

Abbildung 1: Kontextfreie Beispielgrammatik für einen Satz aus der natürlichen Sprache[1]

Der Satz kann wie folgend abgeleitet werden:

SATZ $\rightarrow$ SUBJEKT PRÄDIKAT OBJEKT $\rightarrow$
ARTIKEL SUBSTANTIV PRÄDIKAT OBJEKT $\rightarrow$
der SUBSTANTIV PRÄDIKAT OBJEKT $\rightarrow$
der Hund PRÄDIKAT OBJEKT $\rightarrow$
der Hund VERB OBJEKT $\rightarrow$
der Hund beißt OBJEKT $\rightarrow$
der Hund beißt ARTIKEL SUBSTANTIV $\rightarrow$
der Hund beißt den SUBSTANTIV $\rightarrow$
der Hund beißt den Briefträger

Daraus folgt, dass es Sätze gibt, die kontextfrei dargestellt werden können. Dabei stellt sich die Frage, ob alle Sätze durch kontextfreie Grammatiken darstellbar sind. Die Antwort auf diese Frage lautet nein. Obwohl ein Satz mit verschachtelten Abhängigkeiten mittels einer kontextfreien Grammatik darstellbar ist, lässt sich ein mit überkreuzten Abhängigkeiten nicht kontextfrei formalisieren.[11] Nachfolgend ist jeweils ein Beispiel zu den beiden Abhängigkeitstypen angegeben.

Verschachtelte Abhängigkeiten:

Die Katze, die den Hund, der die Ratte biss, jagte, starb.

In solchem Satz lässt sich das Subjekt-Prädikat Verhältnis mittels 'aaabbb' beschreiben, was offensichtlich in $L = \{a^n b^n | \ n \in \mathbb{N}\}$, die kontextfrei ist, liegt.

Überkreuzte Abhängigkeiten werden am meisten mit Hilfe von Schweizerdeutsch illustriert:

Jan sagt, dass wir die Kinder dem Hans das Haus lassen helfen anstreichen.

Hier lässt sich das Subjekt-Prädikat Verhältnis nicht mehr mittels einer kontextfreien Sprache beschreiben. Dazu ist die Formulierung $L = \{a^n b^m c^n d^m | \ n, m \in \mathbb{N}\}$ notwendig, die auf keinen Fall kontextfrei ist.[11]

Damit ist klar, dass die natürlichen Sprachen nicht kontextfrei sind – dann wäre der nächste logische Schritt die Formulierung von natürlichen Sprachen in der Gruppe der kontext-sensitiven Grammatiken zu suchen. Nach Joshi[12] ist das aber unnötig und sogar ungewollt, da die Grammatik der natürlichen Sprache nur ein wenig strenger als die kontextfreie ist. Die gesuchten Grammatiken sind ein Untertyp von kontext-sensitive Grammatiken und wurden von Joshi *mildly context sensitiv grammars* (leicht kontextfreie Grammatiken) genannt. Eine davon ist die multiple kontextfreie Grammatik.

2.1.2 Multiple kontextfreie Grammatik

Im Gegenteil zu kontextfreien Grammatiken, dürfen die multiple kontextfreien Grammatiken (MKFG) mehrere Stringargumente erhalten. Die Ableitungsregeln der MKFG sind durch partielle Funktionen definiert, die die Konkatenation (Verkettung) der konstanten Strings mit der Bestandteile der Regelargumenten zulassen.[13]

Eine Beispielregel aus dieser Grammatik hat folgende Form:

$$B(x_1 a, x_2 b) \leftarrow A(x_1, x_2)$$

Das bedeutet, dass aus den zwei Argumenten, die der Regel A übergeben wurden, die Regel B erzeugt wird, wo das erste Argument von A mit einem 'a' und das zweite Argument mit einem 'b' konkateniert wird. Also, wenn $x_1 = b$ und $x_2 = a$, wird die Ableitung wie folgt aussehen:

$$B(ba, ab) \leftarrow A(b, a)$$

Weiterhin ist es möglich die Argumentenreihenfolge zu tauschen. Es wird angenommen, dass die Startregel wie folgt aussieht:

$$S(x_2 x_1) \leftarrow B(x_1, x_2)$$

Dann sieht die Ableitung wie folgt aus:

$$S(abba) \leftarrow B(ba,ab)$$

Bei diesem Formalismus sind die Terminale nicht mehr aus Kleinbuchstaben abgeleitet – wie bei früheren Grammatiken – hier sind die Stringargumente in den Klammern zu sehen. Ein Wort wird akzeptiert, wenn es ein einziger Stingargument des Startsymbols ist. Wird vorausgesetzt, dass S aus obigen Beispiel das Startsymbol ist, so würde das Wort 'abba' akzeptiert.

Es wurde schon erwähnt, dass es keine kontextfreie Grammatik gibt, die die Sprache $L = \{a^n b^m c^n d^m | \ n, m \in \mathbb{N}\}$ definieren kann. Dabei stellt sich die Frage, ob dies mit Hilfe einer MKFG möglich ist.

Es wird eine MKFG mit folgenden Regeln definiert:[11,13]

$S(x_1y_1x_2y_2) \leftarrow A(x_1,x_2)B(y_1,y_2)$
$B(y_1b,y_2d) \leftarrow B(y_1,y_2)$
$A(x_1a,x_2c) \leftarrow A(x_1,x_2)$
$B(\epsilon,\epsilon)$
$A(\epsilon,\epsilon)$

Diese Grammatik kann die Sprache $L = \{a^n b^m c^n d^m | \ n, m \in \mathbb{N}\}$ erkennen, was auf dem Beispiel von $n = 3$ und $m = 2$ gezeigt wird.

$S(aaabbcccdd) \leftarrow A(aaa,ccc)B(bb,dd) \leftarrow A(aa,cc)B(b,d)$
$\leftarrow A(a,c)B(\epsilon,\epsilon) \leftarrow A(\epsilon,\epsilon)B(\epsilon,\epsilon)$

Daraus folgt, dass die MKFG mehr geeignet zum Erkennung der natürlichen Sprache sind als die kontextfreien Grammatiken. Dieser Grammatiktyp ist ein wichtiger Bestandteil dieser Arbeit, da er die Programmierung eines inkrementellen Parsers für MG ermöglicht.

2.1.3 Minimalistische Grammatik

Zur weiteren Entwicklung der natürlichen Sprache hat Chomsky ein minimalistisches Programm entwickelt.[14] Dessen Formalisierung sind die minimalistischen Grammatiken (MG). Einer der Hauptforscher zu diesem Thema ist Edward Stabler. Er definiert eine MG als ein 4-Tupel G = (V,Cat,Lex,F), wobei:

- **V** – eine Menge der nicht-syntaktischen Elemente (Vokabular) ist.
- **Cat** – eine Menge der syntaktischen Elemente (Merkmale) ist.
- **Lex** – eine Menge der Ausdrücke gebildet aus V und Cat (Lexikon) ist.
- F – eine Menge der Erzeugungsfunktionen (*merge* und *move*) ist.[15]

Als Vokabular werden die Elemente genannt, die eine ähnliche Definition wie Terminale aus früher erwähnten Grammatiken besitzen. Dieser Teil der Regel wird als ein 'Exponent' bezeichnet. Nur Zeichenketten, die ausschließlich aus den Elementen aus V bestehen, können akzeptiert werden – jedoch nicht alle davon. Die zweite Bedingung ist von Cat abhängig.

Die Menge der Merkmalen ist in vier Untermengen geteilt: Kategorie, Selektoren, Lizensoren und Lizensierer. Kategorien sind mit Buchstaben bezeichnet und bestimmen Wortart oder Satzteil, z.B. n ist Nomen und v ist Verb. Es gibt auch eine Startkategorie. Nur die Zeichenketten, die ausschließlich aus Vokabular bestehen und die Startkategorie besitzen, werden akzeptiert. Selektoren haben Form des Zeichens '=' gefolgt von einer Kategorie, z.B. =n und =v. Wenn eine Regel einen Selektor =x besitzt, bedeutet dies, dass sie für eine *merge*-Operation eine Regel mit der Kategorie n braucht. Lizensoren werden mit einem '+' und einem Fall gebaut, z.B. +dat sei Dativ und +akk sei Akkusativ. Wenn eine Regel ein Lizensor +dat benutzt, bedeutet dies, dass sie für eine *move*-Operation eine Regel mit entsprechenden Lizensierer der Form -dat benötigt. Die Folgen von Merkmalen werden als Merkmalsliste bezeichnet.

Die Spezifität der MG besteht darin, dass sie keine Menge der Produktionsregeln besitzt. Die Anfangsregeln der Grammatik sind Lexemen aus dem Lexikon. Sie haben die Form wie in Abbildung 2 dargestellt.

(likes,::=d =d v)
(knows,::=c =d v)
(ϵ,::=v c)
(ϵ,::=v +wh c)
(Mary,::d)
(John,::d)
(who,::d -wh)

Abbildung 2: Minimalistisches Lexikon von Stabler[2]

Die dargestellten Regeln haben folgende Form: (Exponent,::Merkmalsliste). Mit '::' beginnen Merkmalslisten der Lexemen, also die Regeln, die im minimalistischen Lexikon definiert wurden. Mit ':' fangen Merkmalslisten, die mit *merge*- oder *move*-Operationen erstellt wurden, an.

Es gibt insgesamt fünf Erzeugungsfunktionen – drei *merge*- und zwei *move*-Operationen. Das Schema ihrer Funktionsweise wird nachfolgend dargestellt:

$$\frac{(v, ::= f\gamma) \qquad (w, \cdot f), \alpha_1, ..., \alpha_k}{(vw, : \gamma)} \text{ merge-1}$$

$$\frac{(v, := f\gamma), \alpha_1, ..., \alpha_k \qquad (w, \cdot f)\beta_1, ..., \beta_k}{(wv, : \gamma), \alpha_1, ..., \alpha_k, \beta_1, ..., \beta_k} \text{ merge-2}$$

$$\frac{(v, \cdot = f\gamma), \alpha_1, ..., \alpha_k \qquad (w, \cdot f\delta)\beta_1, ..., \beta_k}{(w, : \gamma), \alpha_1, ..., \alpha_k, (v, : \delta), \beta_1, ..., \beta_k} \text{ merge-3}$$

$$\frac{(w, : +f\gamma), \alpha_1, ..., \alpha_k, (v, : -f), \beta_1, ..., \beta_k}{(vw, : \gamma), \alpha_1, ..., \alpha_k, \beta_1, ..., \beta_k} \text{ move-1}$$

$$\frac{(w, : +f\gamma), \alpha_1, ..., \alpha_k, (v, : -f\delta), \beta_1, ..., \beta_k}{(w, : \gamma), \alpha_1, ..., \alpha_k, (v, : \delta), \beta_1, ..., \beta_k} \text{ move-2}$$

Dabei ist $\cdot \in \{:,::\}$. Dieses Schema ist so zu verstehen, dass von den Elementen im Zähler die Regel im Nenner mittels der genannten Operation erzeugt wird.

Nachfolgend werden in dieser Arbeit die Regeln aus dem Zähler als 'Elternregeln' bezeichnet.

Die Funktionsweise einer der Operationen wird anhand eines Beispiels gezeigt. Es werden die Regeln (likes,::=d =d v) und (who,::d -wh) betrachtet. Das erste Merkmal von 'likes' ist ein Selektor und das erste Merkmal von 'who' ist eine Kategorie, die zusammen passen. Das bedeutet, dass an den Regeln ein *merge* angewendet werden kann. Die Merkmalsliste der ersten Kategorie fängt mit '::' an, deshalb kann *merge*-2 ausgeschloßen werden. Da die zweite Regel eine Merkmalsliste besitzt, passt nur *merge*-3.

$$\frac{(likes, ::= d = dv) \qquad (who, :: d - wh)}{(likes, := dv), (who, : -wh)} \text{ merge-3}$$

Um die Sprachzugehörigkeit eines Satzes zu bestimmen, müssen die Bestandteile davon mittels der Erzeugungsfunktionen verarbeitet werden. Wenn am Ende der geprüfte Satz entsteht und die Startkategorie besitzt, bedeutet das, dass der Satz in der Sprache liegt. Am besten kann dieser Prozess mit Hilfe eines Ableitungsbaumes dargestellt werden. In Abbildung 3 wird anhand der Grammatik aus der Abbildung 2 der Ableitungsbaum des Satzes 'Mary knows who John likes' dargestellt. Dabei ist 'c' die Startkategorie.

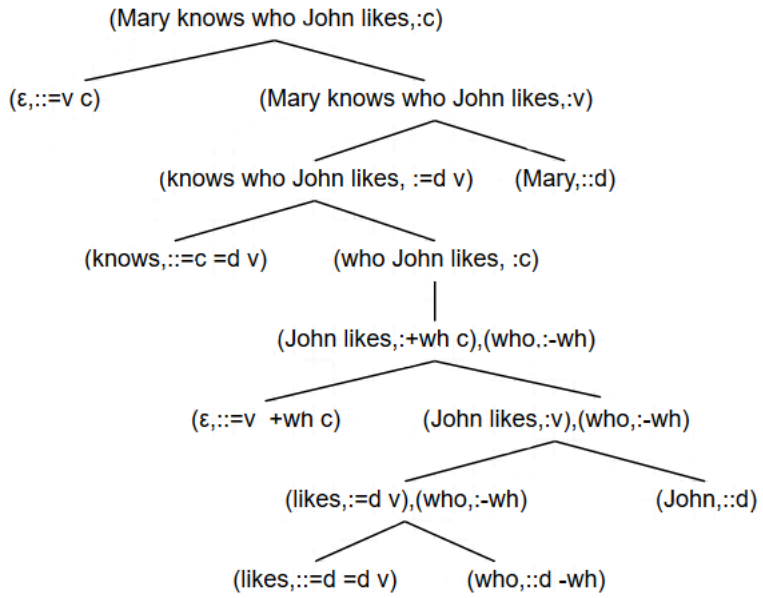

Abbildung 3: Ableitungsbaum der MG von Stabler[2]

Der gesuchte Satz konnte mittels dieser Grammatik erzeugt werden und besitzt die Startkategorie, also liegt er in der Sprache.

2.2 Parser

Ob ein Wort w zur Grammatik G gehört, kann mit Hilfe eines Parsers überprüft werden. Falls ja, wird der Parser ein Ableitungsbaum kreieren.[7] Es existieren verschiedene Typen von Parserverarbeitungen. Davon werden zwei detaillierter besprochen: top-down und bottom-up, die von gegensätzlichen Annahmen ausgehen. Die

Arbeitsweise dieser Parser werden anhand einer kontextfreien Beispielgrammatik (Abbildung 4) dargestellt. Die Grammatik definiert die Sprache $\mathrm{L} = \{\mathrm{a}^n\mathrm{b}^n \mid n \in \mathbb{N}_0\}$, die ein typisches Beispiel einer kontextfreien Sprache ist.

$$\mathrm{S} \rightarrow \mathrm{aSb} \mid \mathrm{ab} \mid \epsilon$$

Abbildung 4: Kontextfreie Beispielgrammatik

2.2.1 Top-down Verarbeitung

Die Idee einer top-down Analyse ist eine Verarbeitung vom Startsymbol (Wurzel) bis zu den Terminalen (Blätter).[1,7] Das bedeutet, dass der geparste Satz mittels Produktionsregeln aus dem Startsymbol abgeleitet wird. Der Satz wird mittels einer Linksableitung von links nach rechts erzeugt.[11] Das Bedeutet, dass ein Nichtterminal von der linken Seite der Regel mit ihrer rechten Seite ersetzt wird, z.B. wird ein 'S' mit 'aSb' ersetzt.

Die top-down Erzeugung eines Syntaxbaumes zu einem Wort w = 'aaabbb' (anhand der im vorherigen Abschnitt genannten Beispielgrammatik) wird in Abbildung 5 dargestellt.

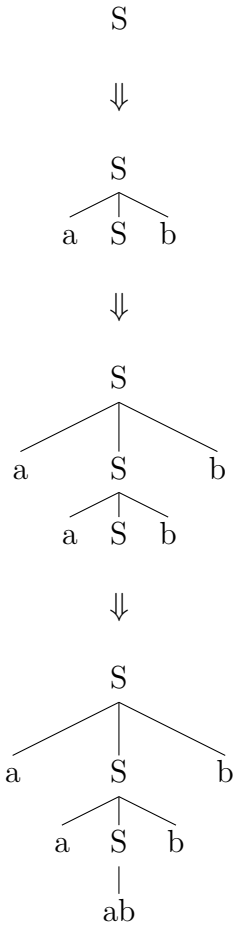

Abbildung 5: Top-down Erzeugung eines Ableitungsbaumes

Zur Durchführung braucht ein Parser einen Stapel und ein Band. Die Auswahl der entsprechenden Regel zur Ableitung erfolgt durch eine aus den zwei Funktionen *predict* und *match*, abhängig von den Stapel- und Bandinhalt.[7] Ihre Aufgaben sind:

- ***match*** – das erste Terminal aus dem Stapel wird mit dem ersten Element aus dem Band verglichen. Wenn der Vergleich erfolgreich ist, werden die ersten Elemente aus dem Band und Stapel gelöscht.

- ***predict*** – das erste Nichtterminal wird nach entsprechender Regel aus der Grammatik abgeleitet und damit auf dem Stapel ersetzt. Dies ist der Fall, wenn kein *match* möglich ist.

Das Wort wird akzeptiert (liegt in der durch Grammatik definierte Sprache), wenn der Stapel und das Band leer sind.

Die einzelnen Schritte zu der Parsererkennung des Wortes w ist in Abbildung 6 zu sehen.

Band	Stapel	Operation
S	aaabbb	Predict
aSb	aaabbb	Match
Sb	aabbb	Predict
aSbb	aabbb	Match
Sbb	abbb	Predict
abbb	abbb	Match
bbb	bbb	Match
bb	bb	Match
b	b	Match
ϵ	ϵ	Accept

Abbildung 6: Top-down Verarbeitung des Wortes 'aaabbb'

Im nächsten Abschnitt wird erklärt, warum MG zur bottom-up Verarbeitung geeignet sind. Zunächst stellt sich die Frage: warum sollte für MG ein top-down Parser gebaut werden?

Die top-down Verarbeitung entspricht ziemlich gut der menschlichen Sprachverarbeitung. Ein Hörer verarbeitet die Aussage des Sprechers schon während des Sprechens und probiert dessen Ende anhand der bekannten grammatischen (und semantischen) Regeln vorherzusehen (eng. predict). Dies ist kein bewusster Prozess, deswegen wird dies anhand eines Beispiels gezeigt. Man betrachtet eine unvollständige Aussage:

Leon hat an einem Wettbewerb...

Es fällt gleich auf, dass der Satz mit dem Wort 'teilgenommen' ergänzt werden soll. Obwohl es scheint, dass dies eine instinktive Entscheidung ist, wird zu dieser Aufgabe Vorwissen benötigt. Das Erste, was gewusst werden muss, ist welcher Teil des Satzes gesucht wird. Das Wort 'hat' ist ein Hilfsverb, daraus kann gefolgert werden, dass das gesuchte Ende des Satzes eine Form des Partizip II hat. Anschließend ist zu erkennen, dass das gesuchte Wort die feste Präposition 'an' besitzt, die mit Dativ verbunden ist. Das schließt zum Beispiel das Wort 'gewonnen' aus, da der Satz 'Leon hat an einem Wettbewerb gewonnen' nicht grammatisch korrekt ist. Schließlich wird anhand eigener Erfahrung entschieden, welches Wort am wahrscheinlichsten passt, da z.B. 'Leon hat an einen Wettbewerb gelitten' grammatisch korrekt ist, obwohl es keinen besonderen Sinn ergibt. Dieses Kriterium liegt jedoch

im Bereich der Semantik, also Bedeutung, die später im Abschnitt 2.3 besprochen wird.

Die Schlussfolgerung aus dieser Analyse ist, dass für die menschliche Sprachverarbeitung nicht der ganze Satz notwendig ist – sie findet schon während der Aussage statt und es wird probiert vorherzusehen, was gesagt wird.

2.2.2 Bottom-up Verarbeitung

Bei einer bottom-up Verarbeitung wird der Ableitungsbaum von unten nach oben gebaut, also von den Terminalen (Blättern) bis zu dem Startsymbol (Wurzel).[7] Der Satz wird von rechts nach links mit Hilfe der umgekehrten Rechtsableitung erzeugt.[11] Das bedeutet, dass die rechte Regelseite mit der linken Regelseite ersetzt wird, z.B. wird 'aSb' mit S ersetzt.

In der Abbildung 7 wird das Wort w = 'aaabbb' bottom-up verarbeitet.

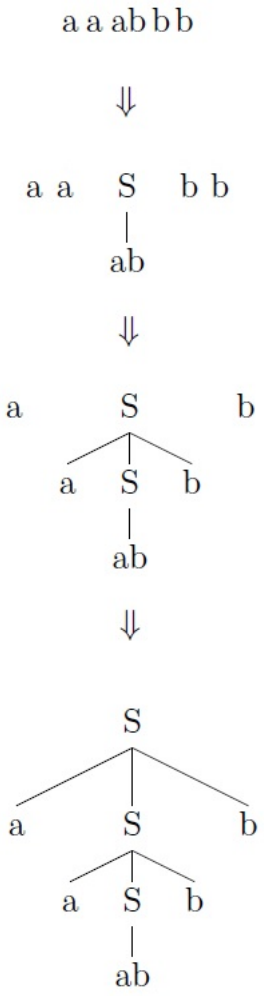

Abbildung 7: Bottom-up Erzeugung eines Ableitungsbaumes

Auch ein bottom-up Parser benötigt einen Stapel und ein Band zur Satzverarbeitung. Die Auswahl der entsprechenden Regel zur Ableitung erfolgt durch eine von zwei Funktionen: *shift* oder *reduce*, abhängig von Stapel- und Bandinhalt.[7] Ihre Aufgaben sind:

- ***reduce*** – eine Gruppe von Terminale und/oder Nichtterminale am Ende des Stapels wird mit der rechten Seite der Grammatikregeln verglichen, bis eine passende Regel gefunden wird. Dann wird diese Gruppe mit der linken Seite der gefundener Regel ersetzt.

- ***shift*** – erstes Zeichen aus dem Band wird an das Ende des Stapels verschoben. Dies kommt vor, wenn kein *reduce* möglich ist.

Das Wort wird akzeptiert (liegt in der Grammatik), wenn das Band leer ist und auf dem Stapel nur das Startsymbol liegt.

Die Parsererkennung des Wortes w ist in Abbildung 8 zu sehen.

Band	Stapel	Operation
ϵ	aaabbb	Shift
a	aabbb	Shift
aa	abbb	Shift
aaa	bbb	Shift
aaab	bb	Reduce
aaS	bb	Shift
aaSb	b	Reduce
aS	b	Shift
aSb	ϵ	Reduce
S	ϵ	Accept

Abbildung 8: Top-down Verarbeitung des Wortes 'aaabbb'

Es sei wohlgemerkt, dass, unabhängig davon welches Verfahren benutzt wird, der Ableitungsbaum und das Ergebnis (also die Information, ob der betrachtete Satz zur Sprache gehört) identisch werden.

Wie schon erwähnt ist für MG die bottom-up Verarbeitung geeignet. Grund dafür ist die Form dieser Grammatiken – es existieren keine explizite Ableitungsregeln. Wenn der Baum zum Satz 'Mary knows who John likes' (Abbildung 3) betrachtet wird, fällt auf, dass dieser bottom-up erzeugt wurde. Da am Anfang nur ein minimalistisches Lexikon existiert, muss z.B. ⟨0,=d v, -wh⟩(likes,who) aus ⟨1,=d =d v,⟩(likes) und ⟨1,=d -wh⟩(who) erstellt werden, bevor sie benutzt werden kann. Das gleiche gilt für das Startsymbol – zu Beginn ist es undefiniert, deswegen kann eine top-down Verarbeitung nicht stattfinden.

Um eine top-down Verarbeitung einer minimalistischen Grammatik zu ermöglichen, müssen zuerst explizite Regeln formuliert werden. Lösung dieses Problems wurde von Edward Stabler vorgeschlagen und wird im Abschnitt 3.1 beschrieben.

2.3 Lambda-Kalkül

Was bis jetzt betrachtet wurde, war eine syntaktische Analyse – also **wie** ein Satz geschrieben wird. Außerdem ist eine semantische Analyse möglich, also **was** gemeint ist. Dieser Unterschied wurde von Chomsky[6,9] mit einem berühmten Beispiel illustriert:

Colorless green ideas sleeps furiously.

Dieser Satz ist aus einer syntaktischen (grammatischen) Perspektive korrekt. Jedoch ist aus semantischer Betrachtung kein Sinn vorhanden.

Genauso sind die beiden nachfolgenden Sätze grammatisch korrekt, aber nur der Erste ist sinnvoll:[16]

The ape has an arm.
The arm has an ape.

Teil dieses Problems soll eine gut konstruierte MG lösen, da sie durch entsprechende Merkmalslisten die Anzahl der legalen Verbindungen verkleinert. Jedoch kann ein Parser nach einer Satzverarbeitung nur bestimmen, ob dieser Satz zur Sprache gehört oder nicht – die Bedeutung des Satzes ist dem Parser dabei nicht bekannt. Das Lambda-Kalkül kann hierbei hilfreich sein.

Sei C die Menge der Konstanten und X die Menge der Variablen. Die Menge Λ der Lambda-Ausdrücke ist definiert durch:[7]

Variablen	$x \in X \rightarrow x \in \Lambda$
Konstanten	$c \in C \rightarrow c \in \Lambda$
Applikationen	$E_1, E_2 \in \Lambda \rightarrow (E_1 E_2) \in \Lambda$
Abstraktion	$x \in X, E \in \Lambda \rightarrow (\lambda x.E) \in \Lambda$

Mit Hilfe dieser Ausdrücke können Funktionen definiert werde, z.B. besitzt eine Additionsfunktion die Form:

λx.λy. + x y

Jetzt wird diese Funktion auf die Zahlen 5 und 3 angewendet.

(λx.λy. + x y)(5)(3)	*x wird durch 5 ersetzt*
(λy. + 5 y)(3)	*y wird durch 3 ersetzt*
(+ 5 3)	*Funktionswert wird berechnet*
8	*Ergebnis*

Bei der Definition einer Funktion mittels Lambda-Abstraktion, ist es wichtig die Reihenfolge der Argumente beizubehalten. Beispiel dazu ist eine Subtraktionsfunktion:

(λx.λy. - x y) $\neq$ (λy.λx. - x y)

(λx.λy. - x y)(5)(3)		(λy.λx. - x y)(5)(3)
(λy. - 5 y)(3)		(λx. - x 5)(3)
(- 5 3)		(- 3 5)
2	$\neq$	-2

Mit den Lambda-Ausdrücken können nicht nur mathematische Operationen definiert werden. Zum Beispiel kann zum Satz:[11]

'Fritz liebt Lise.'

folgende Funktion definiert werden:

λx.λy. lieben x y
(λx.λy. lieben x y) (Fritz)(Lise)
(λy. lieben Fritz y) (Lise)
(lieben Fritz Lise)

Hier ist wieder die Reihenfolge der Argumente wichtig, weil von diesem Satz nicht gefolgert werden kann, ob Lise auch Fritz liebt.

(lieben Fritz Lise) $\neq$ (lieben Lise Fritz)

Nachfolgend wird die Nutzung der Lambda-Ausdrücke in Zusammenhang mit MG anhand der Zahlengrammatik[3] gezeigt (Abbildung 9). Betrachtet wird die Zeichenkette 'zweiundvierzig'. Die Bedeutung davon kann nach Definition der Dezimalzahlen als $(10^1 \cdot 4) + (10^0 \cdot 2)$ geschrieben werden. In der polnischen Notation wird dieser Ausdruck wie folgt formuliert: + ($\cdot$(4) (10^1)) ($\cdot$(2) (10^0)).[17] Mittels Lambda-Kalkül kann diese Zahl so geschrieben werden:

(λy.λx.+(y)(x)) (λx. *(10^1)(x)(4)) (2)
$\rightarrow$ (λy.λx.+(y)(x)) (λa. *(10^1)(a)(4)) (2)
$\rightarrow$ (λx.+(λa. *(10^1)(a)(4))(x)) (2)
$\rightarrow$ (+(λa. *(10^1)(a)(4))(2))
$\rightarrow$ (+(*(10^1)(4))(2))
$\rightsquigarrow$ 42

Diese Lambda-Ausdrücke können zu der MG hinzugefügt werden. Somit wird ein Lexem dieser Grammatik die Form (Exponent,::Merkmalliste,Lambda-Kalkül) besitzen, wie in der Zahlengrammatik.

(zwei,::d,2)
(vier,::d -k 4)
(zig,::=d +k d, λx. *(10^1)(x))
(und,::=d =d c, λy.λx.+(y)(x))
(ϵ,::=c d,)

Abbildung 9: Zahlengrammatik mit semantischen Ausdrücken[3]

Bei der Regelerstellung von *merge* und *move*-Operationen werden die semantischen Ausdrücke wie in den oberen Beispielen verarbeitet.

3 Inkrementeller Parser für minimalistische Grammatiken

3.1 Das Konzept von Edward Stabler

Das Konzept für einen top-down Parser für MG, auf dem diese Arbeit basiert, wurde von Edward Stabler im Paper *Top-down recognizers for MCFGs and MGs*[2] vorgeschlagen.

Eine Vormerkung, die gemacht werden muss ist, dass jede MG eine streng äquivalente MKFG besitzt. Da die MG zuerst nur aus einem minimalistischen Lexikon (mit einzelnen Wörtern) besteht, ist dafür die bottom-up Verarbeitung geeignet. Es gibt keine Startregel, von der das Parsen begonnen werden kann. Ausnahme bilden Lexemen mit einer Endkategorie. Es ist jedoch wünschenswert, dass die Sprachzugehörigkeit mit Hilfe der top-down Methode an ganzen Sätzen geprüft werden kann und nicht nur an einzelnen Wörtern. Eine Startregel muss zu Beginn aus dem minimalistischen Lexikon mittels *merge*- und *move*-Operationen abgeleitet werden. Die Lexemen und alle Regeln, die mittels der oberen Operationen erzeugt werden, bilden eine MKFG. Die multiplen kontextfreien Regeln, die zum Parsen eines Satzes benötigt werden, können aus dem Ableitungsbaum dieses Satzes ausgelesen werden. Die Regeln, die anhand des Ableitungsbaumes des Satzes 'Mary knows who John likes' (Abbildung 3) erstellt wurden, sind in Abbildung 10 zu sehen.

(1) $\langle : c\rangle(x_0x_1) \leftarrow \langle ::= v\, c\rangle(x_0) \quad \langle : v\rangle(x_1)$
(2) $\langle : v\rangle(x_1x_0) \leftarrow \langle := d\, v\rangle(x_0) \quad \langle :: d\rangle(x_1)$
(3) $\langle := d\, v\rangle(x_0x_1) \leftarrow \langle ::= c\ = d\, v\rangle(x_0) \quad \langle : c\rangle(x_1)$
(4) $\langle : c\rangle(x_1x_0) \leftarrow \langle : +wh\, c,\ -wh\rangle(x_0, x_1)$
(5) $\langle : +wh\, c,\ -wh\rangle(x_0x_1, x_2) \leftarrow \langle ::= v\ +wh\, c\rangle(x_0) \quad \langle : v,\ -wh\rangle(x_1, x_2)$
(6) $\langle : v,\ -wh\rangle(x_2x_0, x_1) \leftarrow \langle := d\, v, -wh\rangle(x_0, x_1) \quad \langle :: d\rangle(x_2)$
(7) $\langle := d\, v, -wh\rangle(x_0, x_1) \leftarrow \langle ::= d\ = d\, v\rangle(x_0) \quad \langle :: d\ -wh\rangle(x_1)$
(8) $\langle ::= d\ = d\, v\rangle(likes)$
(9) $\langle ::= c\ = d\, v\rangle(knows)$
(10) $\langle ::= v\, c\rangle(\epsilon)$
(11) $\langle ::= v\ + wh\, c\rangle(\epsilon)$
(12) $\langle :: d\rangle(Mary)$
(13) $\langle :: d\rangle(John)$
(14) $\langle :: d\ - wh\rangle(who)$

Abbildung 10: MKFG zum Satz 'Mary knows who John likes'

Die Regeln sind nun vorgegeben, jedoch muss noch einiges berücksichtigt werden, bevor die top-down Verarbeitung anfängt. Die Regeln der MKFG können mehrere String Argumente besitzen, die nicht immer in der gleichen Reihenfolge sind, in der sie übergeben wurden, wie z.B. in Regel 2. Um das zu beheben, schlägt Stabler die Benutzung einer Prioritätswarteschlange statt eines traditionellen Stapels vor. Auf den Warteschlangeninhalt kann bei jedem Parserverlauf zugegriffen werden, um deren Argumente in eine aufsteigende Ordnung zu sortieren. In der Abbildung 11 ist die Funktionsweise eines solchen Parsers anhand der obigen MKFG und des Satzes 'Mary knows who John likes' dargestellt.

Band	Warteschlange
Mary knows who John likes	$\langle : c \rangle(\epsilon)$
Mary knows who John likes	$\langle ::= v\, c \rangle(0)\langle : v \rangle(1)$
Mary knows who John likes	$\langle : v \rangle(1)$
Mary knows who John likes	$\langle := d\, v \rangle(11)\langle :: d \rangle(10)$
Mary knows who John likes	$\langle :: d \rangle(10)\langle := d\, v \rangle(11)$
knows who John likes	$\langle := d\, v \rangle(11)$
knows who John likes	$\langle ::= c\ = d\, v \rangle(110)\langle : c \rangle(111)$
who John likes	$\langle : c \rangle(111)$
who John likes	$\langle : +wh\, c,\ -wh \rangle(1111, 1110)$
who John likes	$\langle ::= v\ + wh\, c \rangle(11110)\langle : v,\ -wh \rangle(11111, 11102)$
who John likes	$\langle : v,\ -wh \rangle(11111, 11102)$
who John likes	$\langle := d\, v, -wh \rangle(111111, 111022)\langle :: d \rangle(111110)$
who John likes	$\langle ::= d\ = d\, v \rangle(111111)\langle :: d\ - wh \rangle(111022)\langle :: d \rangle(111110)$
who John likes	$\langle :: d\ - wh \rangle(111022)\langle :: d \rangle(111110)\langle ::= d\ = d\, v \rangle(111111)$
John likes	$\langle :: d \rangle(111110)\langle ::= d\ = d\, v \rangle(111111)$
likes	$\langle ::= d\ = d\, v \rangle(111111)$
ϵ	ϵ

Abbildung 11: Top-down Verarbeitung des Satzes 'Mary knows who John likes'

3.2 Semantische Verarbeitung

Wenn bei einer Grammatik semantische Ausdrücke mittels Lambda-Kalkül berücksichtigt werden, kann bei der Parserverarbeitung – außer dem Eingabeband und der Prioriätswarteschlange – ein Semantikstapel[18] erstellt werden. Darauf wird immer der semantische Ausdruck einer Regel, auf der gerade ein *match* erfolgte, gespeichert. Wenn der Satz akzeptiert wird, erfolgt die Auswertung des Stapelinhalts. Was mit dem Semantikstapel passiert, wird schematisch in der Abbildung 12 gezeigt. Der Parserverlauf wurde nur auf *match*-Operationen und Auswertung des Semantikstapels begrenzt, da sich sonst sein Inhalt nicht ändert.

Band	Semantikstapel
zwei und vier zig	ϵ
und vier zig	2
vier zig	$(\lambda y.\lambda x.+(y)(x))(2)$
zig	$(4)(\lambda y.\lambda x.+(y)(x))(2)$
ϵ	$(\lambda x.\ *(10^1)(x))(4)(\lambda y.\lambda x.+(y)(x))(2)$
ϵ	$+(*(10^1)(4))(2)$

Abbildung 12: Parserverlauf mit Semantikstapel

4 Implementierung in MATLAB

Das Programm auf Basis Stablers Konzept basiert hauptsächlich auf Strings. Es besteht aus mehreren Unterteilen, die in der Funktion *parse* der Datei *mainTDP* (mainTopDownParser) zusammengefasst sind. Ihre Wirkung wird anhand der Beispielgrammatik ähnlich der von Stabler (Abbildung 2) gezeigt.

Als Eingabe erhält die Funktion:

- **eine MG,**
  die als ein *cell Array* der Größe $n \times 1$ gespeichert ist, wobei n die Anzahl der Regeln angibt. Die Regeln sind vom Typ String. Das Programm wurde so eingerichtet, dass es die Grammatiken sowohl ohne, als auch mit semantischen Ausdrücke verarbeiten kann. In Abbildung 13 ist die Beispielgrammatik (basierend auf Stablersgrammatik) abgebildet.

```
mg = {'(likes,::=d =d v)';
      '(John,::d)';
      '(who,::d -wh)';
      '(,::=v +wh c)'
      };
```

Abbildung 13: Minimalistische Beispielgrammatik ohne semantische Ausdrücke

- **ein Band,**
  worauf der zu parsende Satz (wie in der Abbildung 14) gespeichert ist.

```
tape = 'who likes John';
```

Abbildung 14: Beispielbandinhalt

- **eine Startkategorie,**
  die nur zur Sprache gehörende Elemente besitzen. Sie ist als ein Char gespeichert, zum Beispiel wird die Kategorie Komplement als 'c' bezeichnet.

Am Anfang ruft *parse* die Funktion *mg2mcfg* auf. Sie ist für die Umwandlung der MG in eine MKFG verantwortlich. Danach wird, mit Hilfe der erhaltenen MKFG, der Satz auf dem Band durch *BTtopdown* geparst.

Nach diesen Operationen gibt die Funktion *parse* folgende Elemente aus:

- **eine vereinfachte MKFG,**
  die als ein *cell Array* der Größe $m \times 4$ oder $m \times 5$ (falls die Regeln semantische Ausdrücke beinhaltet) gespeichert ist, wobei m die Anzahl der Regeln ist. Sie wird genauer im Abschnitt 4.1 besprochen.

- **einen Parserverlauf,**
  der als ein *cell Array* der Größe $t \times 2$ oder $t \times 3$ gespeichert ist, wobei t Anzahl der Parseraktionen (*predict* bzw. *match*) angibt. In der ersten und zweiten Spalte sind entsprechend die Inhalte von Warteschlange und Band gespeichert.

Falls die Regeln auch semantische Ausdrücke besitzen, existiert zusätzlich eine dritte Spalte, die den aktuellen Inhalt des Semantikstapels anzeigt.

- **ein Stringergebnis,**
  das darüber informiert, ob der Satz aus dem Band in der von der Grammatik generierten Sprache liegt. Falls nicht, wird dies auch begründet.

4.1 Grammatikumwandlung

Der erste Schritt des Programms ist es, aus dem gegebenen minimalistischen Lexikon die Regeln einer MKFG zu erzeugen. Dafür ist die Funktion *mg2mcfg* verantwortlich. Ihre Eingaben sind gleich mit den Eingaben der Funktion *parse*, also:

- **eine MG**
- **ein Band**
- **eine Startkategorie**

Diese Parameter wurden bereits im Abschnitt 4 besprochen.

Es ist wohlgemerkt, dass die Regeln aus einem minimalistischen Lexikon rekursiv bottom-up erzeugt werden. Bei manchen Grammatiken wie die Beispielgrammatik (Abbildung 13) ist das unproblematisch. In anderen Fällen, wie z.B. Stablers Grammatik (Abblidung 2) oder Zahlengrammatik (Abbildung 9) können potentiell unendlich viele Regeln erzeugt werden. Dies hat zur Folge, dass eine endlose Schleife im Programm ausgeführt wird. Um dieses Problem zu lösen muss die Rekursionstiefe begrenzt werden. Dazu dient das Band, das der Funktion übergeben wurde. Die Anzahl der Wörter darauf wird mit der Exponentlänge der neu erzeugten Regel verglichen. Besitzt der Exponent mehrere Wörter, so wird die Regel verworfen, da sie nie beim Parsen des gegebenen Satzes auftreten kann. Zum Beispiel die Regel '(knows who likes Mary,:=d v)' wird zur Erkennung des Satzes 'John knows who Mary likes' relevant sein, aber es besteht keine Chance, dass sie zum Parsen des Satzes 'who likes John' benutzt wird – sie muss also nicht unnötig erzeugt und gespeichert werden. Solche Lösung ist nicht perfekt, weil bei einem idealen top-down Parsen sollte weder der Bandinhalt noch deren Satzlänge bekannt sein. Dadurch lässt sich das Problem der endlosen Schleife vermeiden ohne das Wörter auf dem Band genauer betrachten werden müssen – was schon einer bottom-up Verarbeitung entsprechen würde.

Um die multiplen kontextfreien Regeln zu erzeugen, wird eine Schleife angewendet, in der auf jede einzelne Regel und auf jede Regelnpaarkombination entsprechend die Funktionen *chooseMove* und *chooseMerge* angewendet werden. Ihre Aufgabe ist es zu prüfen, ob ein *move* bzw. *merge* möglich ist und wenn ja, eine neue Regel zu erzeugen, die bei weiteren Durchläufen berücksichtigt wird. Bevor aber eine neue Regel akzeptiert wird, wird sie der Funktion *isLegal* übergeben. Diese Funktion prüft, ob:

- der Regelexponent größer als die Wörteranzahl auf dem Band ist (Rekursionstiefebegrenzung)

- die betrachtete Regel zu *move* geeignet ist (also ob sie zwei Regelausdrücke getrennt durch ein Komma besitzt), falls sie ein Lizensierer als erstes Merkmal besitzt

- der semantische Ausdruck mit einem *lambda* beginnt, falls es sich um eine Endregel handelt (wenn semantische Ausdrücke vorhanden sind)

Falls die Antwort auf eine der oberen Fragen positiv ist, wird die Regel verworfen.

Eine Zusatzaufgabe dieser Funktion besteht darin zu kontrollieren, ob die Regel einen semantischen Ausdruck besitzt und ob sie eine Endregel ist. Zuerst wird nur die Zweite (Endregel) dieser Informationen gespeichert.

Bei jedem Durchlauf der Schleife, wird die aktuelle Regelanzahl mit der bekannten Regelanzahl aus dem vorherigen Schritt verglichen. Bleibt die Regelanzahl konstant, bedeutet dies, dass keine neue Regel entstehen kann – dann wird die Schleife abgebrochen.

Auf diese Art und Weise wird eine MKFG gebildet. Unabhängig davon, ob die Regeln semantische Ausdücke enthalten oder nicht, enthält das entstandene *cell Array* $m \times 4$ Dimension, wobei m die Regelanzahl angibt. In der ersten Spalte werden die neu entstandenen Regeln bzw. die Lexemen gespeichert. In der zweiten und dritten Spalte befinden sich die Elternregeln. Ihre Reihenfolge ist von der angewandten Operation abhängig. Im Fall, dass *merge*-1, *merge*-3 angewendet wird oder die erste Elternregel leeres Wort als Exponent besitzt, wird die Reihenfolge beibehalten (Abbildung 15).

$$\frac{\text{(likes,::=d =d v)} \qquad \text{(John,::d)}}{\text{(likes John,:=d v)}}\ \text{merge-1}$$

$$\Downarrow$$

$$\text{\{'(likes John,:=d v)'\}} \qquad \text{\{'(likes,::=d =d v)'\}} \qquad \text{\{'(John,::d)'\}}$$

Abbildung 15: Reihenfolge der Elternregeln beibehalten

Bei *merge*-2 werden die Plätze der beiden Regeln (Abbildung 16) getauscht, damit die Reihenfolge der Elternexponenten gleich mit der des neuen Exponenten ist. Dieser Tausch kann aber nur dann stattfinden, wenn keine von den beiden Elternregeln (somit auch die neue Regel) die „merge-3 Form“ besitzt. Damit ist eine Regel gemeint, bei der bereits die Operation *merge*-3 aufgetreten ist, aber noch keine *move*-Operation, z.B. '(likes,:=d v),(who,:-wh)'. Solche Fälle werden später bei der Grammatikvereinfachung betrachtet.

$$\frac{\text{(likes John,:=d v)} \qquad \text{(John,::d)}}{\text{(John likes John,:v)}}\ \text{merge-2}$$

$$\Downarrow$$

$$\text{\{'(John likes John,:v)'\}} \qquad \text{\{'(John,::d)'\}} \qquad \text{\{'(likes John,:=d v)'\}}$$

Abbildung 16: Reihenfolge der Elternregeln getauscht

Es kann auch passieren, dass eine (im Fall einer *move*-Operation) oder beide Spalten (bei Lexemen) leer sind. Vierte Spalte kann entweder eine '1' enthalten

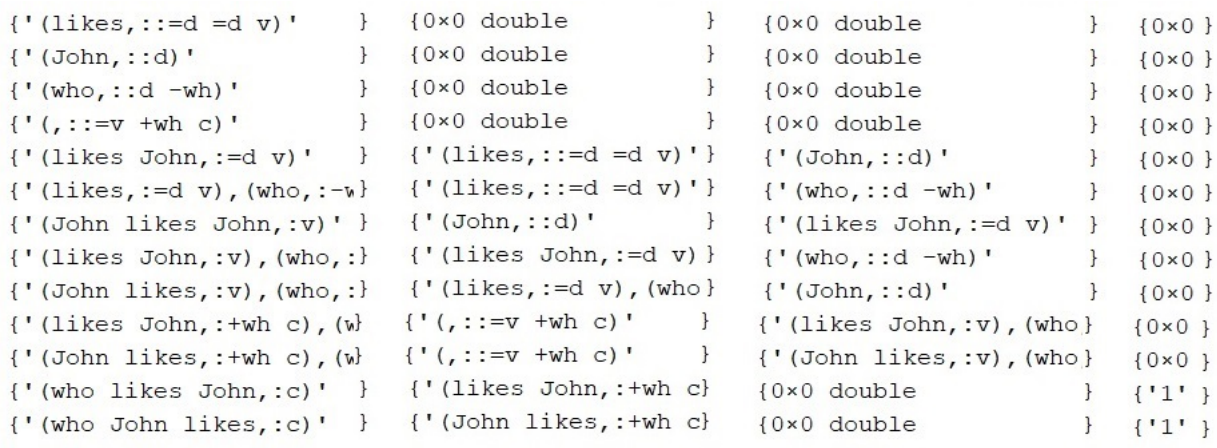

Abbildung 17: Multiple kontextfreie Beispielgrammatik

(falls Endregel) oder leer sein (sonst). Im Abbildung 17 wird das Ergebnis aus der Verarbeitung der Beispieleingaben aus dem Abschnitt 4 präsentiert.

Die entstandene Grammatik wäre jedoch ziemlich schwer zu verarbeiten. Wenn eine von den Regeln abgeleitet werden soll, muss zuerst ihre Position im Array gefunden werden. Dazu ist es notwendig die betrachtete Regel mit anderen Regeln zeichenweise zu vergleichen, bis die gleiche Regel gefunden wird. Aus diesem Grund muss die Grammatik vereinfacht werden. Dafür ist die durch die *mg2mcfg* ausgerufene Funktion *transform* verantwortlich, die nahezu die identische Eingabe wie *mg2mcfg* enthält. Nur statt einer MG erhält sie die bereits bestimmte MKFG. Auf dieser Basis erstellt die Funktion ein neues *cell Array* gleicher Größe oder mit einer zusätzlichen Spalte, falls semantische Ausdrücke enthalten sind.

Da alle möglichen Regeln bereits enthalten sind, ist es zu beachten, dass die Exponenten (bis auf Lexemen), Merkmalslisten und eventuelle semantische Ausdrücke zur Parserverarbeitung nicht relevant sind. Das bedeutet, dass die langen Regelnamen mit Symbolen ersetzt werden können, was die Suche bedeutend vereinfacht. Wichtig ist nur die Exponentenreihenfolge, die in der abgeleiteten Regel auftritt, vorzumerken.

Die Verarbeitung fängt mit den Lexemen an. Jedem Lexem wird ein kleiner Buchstabe zugeteilt, der im neuem *cell Array* auf gleichem Index liegt. Falls es mehr als 26 (Anzahl der Buchstaben im englischen Alphabet) Lexemen gibt, werden die neuen Regelnamen aus einigen kleinen Buchstaben verkettet, z.B. aa, ab, ac usw. Wie bereits erwähnt, sind bei den Lexemen die Exponenten wichtig – sie werden direkt daneben, in der zweiten Spalte gespeichert. Es wird zusätzlich durch die Funktion *isLegal* geprüft, ob die Lexemen die Startkategorie besitzen (falls ja, wird in der vierten Spalte eine '1' geschrieben) und ob semantische Ausdrücke vorhanden sind. Wenn semantische Ausdrücke vorhanden sind, wird die Dimension des Arrays um eine Spalte, wo die Ausdrücke gespeichert werden, erhöht.

Die Namen der abgeleiteten Regeln werden mit großen Buchstaben, bzw. eine Verkettung von großen Buchstaben (AA, AB, AC, usw.) ersetzt. Inhalt der vierten Spalte – also Information, ob betrachtete Regel die Startkategorie besitzt – wird aus dem alten Array kopiert. Falls in der Grammatik semantische Ausdrücke vorkommen, werden diese, genau wie im Fall der Lexemen, in der fünfte Spalte gespeichert. Die weitere Verarbeitung ist komplizierter, da die Exponentenreihenfolge

berücksichtigt werden muss. Ein Teil dieses Problems wurde während der Erstellung der MKFG gelöst, da die einfacheren Fälle von *merge*-2 bereits verarbeitet wurden (Abbildung 16). Die Fälle von *merge*-3 können zu Komplikationen führen, da eine Regel, auf der die *merge*-3 Operation bereits angewendet wurde, noch mit anderen Regeln mittels anderer *merge*-Operationen mehrmals kombiniert werden können, bevor es zu einer *move*-Operation kommt. Das ist eine wesentliche Schwierigkeit, denn abhängig davon, ob die *merge*-3 Regel erster oder zweiter Elternteil ist und ob es zur *merge*-1 oder *merge*-2 kommt, sind die Argumentenreihenfolgen verschieden. In Abbildung 18 gibt es vier Schemen zur Erstellung des Satzes 'John likes, who', die die Problematik solcher Fälle zeigen. Dabei ist nur der Erste der genannten Fälle bei der vorgegebenen Grammatik möglich – der Rest dient nur zu Beispielzwecken.

B[John likes, who] ← A[likes,who] a[John] merge-2
⇓
B[01, 2] ← A[1,2] a[0]

B[John likes, who] ← A[John,who] a[likes] merge-1
⇓
B[01, 2] ← A[0,2] a[1]

B[John likes, who] ← a[likes] A[John,who] merge-2
⇓
B[01, 2] ← a[1] A[0,2]

B[John likes, who] ← a[John] A[likes,who] merge-1
⇓
B[01, 2] ← a[0] A[1,2]

Abbildung 18: Problematik der Operation merge-3

Damit die Parserfunktion die korrekte Argumentenreihenfolge bestimmen kann, muss die vereinfachte Grammatik die Information, um welchen Fall es sich handelt, speichern. Dies wurde mittels Zahlen, die an den Regelsymbolen von der ersten Spalte in eckigen Klammern stehen, gemacht.

- **A[ ]** - A hat keine „merge-3 Form“.
- **A[0]** - A ist eine Regel von „merge-3 Form“, die entweder gerade durch eine merge-3 oder durch eine merge-1 an einer anderen Regel von „merge-3 Form“ entstanden ist.
- **A[1]** - A ist eine Regel von „merge-3 Form“, die durch eine merge-2 an einer anderen Regel von „merge-3 Form“ entstanden ist.

In der zweiten und dritten Spalte werden die Regel von „merge-3 Form“ mittels [0,1] markiert. Ansonsten kriegen die Regeln in der zweiten Spalte eine [0] und die in der dritten Spalte eine [1]. In der Abbildung 19 ist die Vereinfachung der Beispielgrammatik (Abbildung 13), die keine semantischen Ausdrücke besitzt, zu sehen. Ein

Beispiel von einer vereinfachten Grammatik mit semantischen Ausdrücken wird in der Abbildung 20 gezeigt. Sie wurde auf Basis der Zahlengrammatik (Abbildung 9) erstellt.

```
'a[]'     'likes'          []     []
'b[]'     'John'           []     []
'c[]'     'who'            []     []
'd[]'     ''               []     []
'A[]'     'a[0]'      'b[1]'      []
'B[0]'    'a[0]'      'c[1]'      []
'C[]'     'b[0]'      'A[1]'      []
'D[0]'    'A[0]'      'c[1]'      []
'E[1]'    'B[0,1]'    'b[1]'      []
'F[0]'    'd[0]'      'D[0,1]'    []
'G[0]'    'd[0]'      'E[0,1]'    []
'H[]'     'F[0,1]'         []    '1'
'I[]'     'G[0,1]'         []    '1'
```

Abbildung 19: Vereinfachte MKFG ohne semantische Ausdrücke

```
'a[]'     'zwei'           []    '1'    '(2)'
'b[]'     'vier'           []     []    '(4)'
'c[]'     'zig'            []     []    '(l x. *(10^1)(x))'
'd[]'     'und'            []     []    '(l y. l x. +(y)(x))'
'e[]'     ''               []     []                          []
'A[0]'    'c[0]'      'b[1]'     []    '(l x. *(10^1)(x)),(4)'
'B[]'     'd[0]'      'a[1]'     []    '(l x. +(2)(x))'
'C[0]'    'd[0]'      'b[1]'     []    '(l y. l x. +(y)(x)),...'
'D[]'     'A[0,1]'       []    '1'    '(*(10^1)(4))'
'E[]'     'a[0]'      'B[1]'     []    '(+(2)(2))'
'F[0]'    'B[0]'      'b[1]'     []    '(l x. +(2)(x)),(4)'
'G[1]'    'C[0,1]'    'a[1]'     []    '(l x. +(2)(x)),(4)'
'H[]'     'd[0]'      'D[1]'     []    '(l x. +(*(10^1)(4))(...'
'I[]'     'e[0]'      'E[1]'    '1'    '(+(2)(2))'
'J[]'     'D[0]'      'B[1]'     []    '(+(2)(*(10^1)(4)))'
'K[1]'    'C[0,1]'    'D[1]'     []    '(l x. +(*(10^1)(4))(...'
'L[]'     'd[0]'      'I[1]'     []    '(l x. +(+(2)(2))(x))'
'M[]'     'e[0]'      'J[1]'    '1'    '(+(2)(*(10^1)(4)))'
'N[1]'    'C[0,1]'    'I[1]'     []    '(l x. +(+(2)(2))(x))...'
'O[]'     'a[0]'      'H[1]'     []    '(+(*(10^1)(4))(2))'
'P[0]'    'H[0]'      'b[1]'     []    '(l x. +(*(10^1)(4))(...'
'Q[]'     'e[0]'      'O[1]'    '1'    '(+(*(10^1)(4))(2))'
'R[0]'    'L[0]'      'b[1]'     []    '(l x. +(+(2)(2))(x))...'
```

Abbildung 20: Vereinfachte MKFG mit semantischen Ausdrücken

Am Ende der *mg2mcfg* wird eine Liste erstellt, in der alle Endregeln gespeichert werden. Ihre Reihenfolge wird invertiert, damit die Wörter mit dem längsten Exponent am Anfang stehen – es besteht eine größere Wahrscheinlichkeit, dass eine Regel mit hohem Index die richtige ist.

Nach diesen Schritten gibt die Funktion folgendes aus:

- **die vereinfachte MKFG**

- **die Menge der Endregeln**
- **eine boolsche Variable**, die angibt, ob die Grammatik semantische Ausdrücke enthält

Mit diesen Daten kann die eigentliche Parserverarbeitung anfangen.

4.2 Parsing

Zweiter Teil des Programms ist für die Verarbeitung des gegebenen Satzes mit Hilfe der erstellten vereinfachten MKFG verantwortlich. Zu diesem Zweck wird die Funktion *BTtopdown* (Backtracing top down) ausgerufen. Als Eingabe erhält sie:

- **das Band**
- **die vereinfachte MKFG**
- **die Sammlung der potentiellen Endregeln** (also alle Regeln, die die Startkategorie besitzen)
- **eine boolsche Variable**, die angibt, ob die Grammatik semantische Ausdrücke enthält

Wie bereits im Abschnitt 2.2 erwähnt wurde, braucht ein top-down Parser zur Satzverarbeitung u.a. das Startsymbol (hier: Endregel), von dem er die Verarbeitung beginnen kann. Da die Regeln der MKFG rekursiv bottom-up erstellt wurden, ist nicht klar, welche Regel die Endregel ist. Es sei wohlgemerkt, dass jede Regel, die die Startkategorie besitzt, potentiell die Endregel sein kann. Aus diesem Grund hat die *mg2mcfg* eine Menge erzeugt, wo all diese Regeln gespeichert sind. Die Aufgabe der *BTtopdown* ist die Satzverarbeitung mit jeder potentiellen Endregel zu probieren, bis die Richtige gefunden wird oder alle Elemente erfolglos durchprobiert werden.

Die Funktion, die eigentlich zur Parserverarbeitung verantwortlich ist, ist *topdownparser*. Als Eingabe erhält sie:

- **das Band**
- **die vereinfachte MKFG**
- **ein Startsymbol** – potentielle Endregel
- **eine boolsche Variable**, die angibt, ob die Grammatik semantische Ausdrücke enthält

BTtopdown ruft diese Funktion, jeweils mit einem anderen Element der Menge der potentiellen Endregeln als Eingabeparameter, auf. Wenn die Verarbeitung erfolgreich beendet wird, bricht *BTtopdown* ihre Arbeit ab. Sie gibt das Stringergebnis (das informiert, dass der Satz in der Grammatik liegt) und den Parserverlauf aus. Falls der Satz aus keiner der Endregeln abgeleitet werden konnte, werden das Stringergebnis (der über den Grund des Misserfolges berichtet) und der Parserverlauf der zuletzt geprüften Endregel ausgegeben.

Nun wird die Funktionsweise der Funktion *topdownparser* besprochen. Am Anfang wird eine Warteschlange erstellt, auf der zuerst das Startsymbol gelegt wird. Anschließend wird vorgemerkt, wie viele Regeln es gibt, die aus den Lexemen entstanden sind.

Der nächste Teil des Programms wird solange ausgeführt, wie die Warteschlange nicht-leer ist. Es wird ein Parserverlaufarray erstellt, in dem die Inhalte der Warteschlange, des Eingabebands und des Semantiksstapels (falls semantische Ausdrücke vorhanden) bei jedem Schritt gespeichert werden. Das erste Symbol aus der Warteschlange und sein Argument – also die Zahlen aus den eckigen Klammern – werden

gespeichert. Beim ersten Durchlauf und wenn beim letzten Schleifenvorgang ein *match*-Operation stattfand, wird das erste Wort aus dem Band gespeichert. Sonst ist dieser Schritt nicht notwendig, da der Bandinhalt sich nicht ändert, solange nur *predict*-Operationen durchgeführt werden. Danach wird entschieden, welche Operation – *predict* oder *match* – ausgeführt wird, indem geprüft wird, ob das erste Warteschlangensymbol aus kleinen oder großen Buchstaben besteht. Im ersten Fall wird die *match* Operation gewählt.

Das Warteschlangensymbol wird unter den MKFG Regeln gesucht. Um unnötigen Aufwand zu sparen, wird die vorgemerkte Anzahl der Regeln, die aus den Lexemen entstanden sind, benutzt. So wird die Anzahl der untersuchten Regeln begrenzt. Wenn das Warteschlangensymbol gefunden wird, wird das dazu gehörende Wort mit dem ersten Wort aus dem Band verglichen. Stimmen sie nicht überein, wird die Funktion mit einer entsprechenden Fehlermeldung beendet. Ansonsten wird das erste Wort aus dem Band und das erste Warteschlangensymbol gelöscht. Falls das Warteschlangensymbol einem leeren Wort entspricht, dann wird bei einem *match* kein Wort aus dem Band gelöscht.

Falls das erste Warteschlangensymbol doch aus großen Buchstaben besteht, wird *predict* angewendet. Zuerst muss die entsprechende Regel in MKFG gefunden werden. Analog wie bei der Suche bei *match*, wird die durchgesuchte Regelanzahl begrenzt - die Regeln, die aus den Lexemen entstanden sind, werden nicht betrachtet. Wenn eine passende Regel gefunden wird, wird gleich geprüft, ob die eckigen Klammern dabei leer sind oder 0 bzw. 1 enthalten. Wie bereits im Abschnitt 4.1 erwähnt wurde, weisen die Zahlen darauf hin, ob die Regel eine „merge-3 Form“ hat und ob auf deren Elternregeln ein *merge*-2 angewendet wurde. Demnächst wird eine temporärer Warteschlange erzeugt, auf der das erste Warteschlangensymbol mit seiner Ableitung ersetzt wird. Danach müssen die Argumente angepasst werden, was von der Regelart bzw. angewandte *merge*- und *move*-Operationen abhängig ist. Nachfolgend werden die möglichen Handlungen beschrieben:

einfacher Fall

Wenn die eckige Klammer beim Warteschlangensymbol in MKFG keine Zahlen enthalten und es zwei Elternregeln gibt, hat die Funktion mit einem einfachen Fall zu tun. Der Argument des ersten Warteschlangensymbol muss vorgemerkt werden und bei Ableitungsregeln mit 0 (erste Regel) bzw. 1 (zweite Regel) wie auf unterem Schema konkateniert und gespeichert werden.

$$\mathrm{A}[01] \rightarrow \mathrm{B}[010]\mathrm{C}[011]$$

move-1

Wenn nur eine Elternregeln existiert, bedeutet das, dass darauf eine *move*-1 Operation ausgeführt wurde. Dabei ist anzumerken, dass was in diesem Fall passiert eigentlich eine umgekehrte *move*-1-Operation ist, weil die Elternregel aus seiner Kindregel wiederhergestellt wird. Das liegt daran, dass die MKFG Regeln bottom-up erstellt wurden und top-down parsiert werden. Das bedeutet, dass wenn eine Regel 'who likes John' wie folgend erstellt wurde:

$$\frac{\text{(likes John,:+wh c),(who,:-wh)}}{\text{(who likes John,:c)}}\ \text{move-1}$$

was sich wie folgend formulieren lässt:

$$\text{A[likes John,who]} \rightarrow \text{B[who likes John]}$$

muss sie beim *predict*-Schritt aus B nach A umgewandelt werden. Dabei muss das Argument von B gemerkt und entsprechend in A gespeichert werden. A soll zwei mit Komma getrennte Argumente besitzen, die Beide das B-Argument als Anfang enthalten. Beim ersten wird '1' und beim zweiten '0' hinzugeschrieben, da die Exponentenreihenfolge nach einer *move*-1-Operation ausgetauscht wird. Dieser Vorgang ist auf unterem Schema dargestellt.

$$\text{B[01]} \rightarrow \text{A[011,010]}$$

merge-3

Falls in den eckigen Klammern des Warteschlangensymbols in der MKFG '0' oder '1' steht, besteht der schwierigste Fall. Die Problematik der Verarbeitung von Regeln mit „merge-3 Form" wurde bereits in Abbildung 18 erläutert. Hier müssen noch einige Fälle unterschieden werden. Wie beim Fall von *move*-1 ist anzumerken, dass hier eigentlich eine Umkehrung von *merge*-3 stattfindet.

- **keine „merge-3 Form" bei den Elternegeln**

  Dieser Fall ist am einfachsten. Die betrachtete Regel hat zwei mit Komma getrennte Argumente, die den Elternsymbolen üebergeben werden. Dies wird anhand der Regel (likes,:=d v),(who,:-wh) gezeigt. Sie wird wie folgt erstellt:

  $$\frac{\text{(likes,::=d =d v)} \quad \text{(who,::d -wh)}}{\text{(likes,:=d v),(who,:-wh)}}\ \text{merge-3}$$

  Das lässt sich zu:

  $$\text{a[likes]b[who]} \rightarrow \text{A[likes,who]}$$

  vereinfachen. Was jetzt gemacht werden muss, ist A wieder in a und b aufzuteilen. Die Argumentenreihenfolge muss nicht getauscht werden. Die Aufteilung erfolgt im Schema:

  $$\text{A[01,11]} \rightarrow \text{a[01]b[11]}$$

  In diesem Fall muss das originale Argument weder mit '0' noch mit '1' konkateniert werden. Der Grund dafür ist die Tatsache, dass in diesem Fall kein Argument zweimal benutzen wird.

- **„merge-3 Form" bei der ersten Elternegel**

  Hier müssen noch zwei weitere Fälle unterschieden werden – ob auf den Elternregel ein *merge*-1 oder ein *merge*-2 ausgeführt wurde. Im Gegenteil zum Fall, wenn keine von den Elternregeln die „merge-3 Form" hat, müssen hier die Ursprungsargumente mit '0' oder '1' oder '2' konkateniert werden. Das ist dadurch bedingt, dass das erste Ursprungsargument in den erzeugten Regeln zwei mal vorkommen muss. Anhand der Beispiele in Abbildung 18, werden nachfolgend zwei Vorgehensschemen präsentiert.

Für den Fall, dass auf den Elternregeln ein *merge*-1 angewendet wird, findet folgende Argumentenanpassung statt:

$$\text{B}[01,\ 11] \rightarrow \text{A}[010{,}112]\,\text{a}[011]$$

Das Vorgehen im Fall der *merge*-2 erfolgt so:

$$\text{B}[01,\ 11] \rightarrow \text{A}[011{,}112]\,\text{a}[010]$$

- **„merge-3 Form“ bei der zweiten Elternegel**

  Dieser Fall wird analog zum vorherigen verarbeitet. Hier müssen auch zwei weitere Fälle unterschieden werden, die wieder anhand der Beispiele aus der Abbildung 18 erläutert werden.

  Wenn auf den Elternregeln ein *merge*-1 angewendet wurde, gibt es folgendes Vorgehensschema:

$$\text{B}[01,\ 11] \rightarrow \text{a}[010]\text{A}[011{,}112]$$

  Im Fall von *merge*-2 wird folgendes gemacht:

$$\text{B}[01,\ 11] \rightarrow \text{a}[011]\text{A}[010{,}112]$$

Nach dem erfolgreichen Anpassen von Argumenten, wird die temporäre Warteschlange, auf dem das Ergebnis des *predict* gespeichert wurde mit dem restlichen Inhalt der Warteschlange konkateniert. Das erste Warteschlangensymbol wird also mit seiner Ableitung ersetzt. Dann wird die Warteschlange der Funktion *sortArg* übergeben. Deren Aufgabe ist es die Symbole auf der Warteschlange so zu sortieren, damit ihre Argumente aufsteigend geordnet sind. Also wird z.B die Eingabe:

$$\text{a}[111]\text{c}[022]\text{b}[110]$$

so umgeordnet:

$$\text{c}[022]\text{b}[110]\text{a}[111]$$

Das Sortieren ist das Letzte, was bei einem Schleifendurchlauf gemacht wird.

Alle oben beschriebenen Schritte werden wiederholt, bis die Warteschlange geleert wird. Falls es vorkommt, dass inzwischen kein Wort auf dem Band vorhanden ist und die Warteschlange noch nicht leer ist, so wird die Funktion mit entsprechender Fehlermeldung beendet. Einzige Möglichkeit, dass leeres Band und nicht leere Warteschlange einen korrekten Satz repräsentieren, ist der Fall, wenn in der Warteschlange Symbole der leeren Wörter enthalten sind. Solche Situation wurde aber bereits bei der Grammatikumwandlung verhindert. Wenn ein Lexem mit einem

leeren Wort zur Elternregel wird, wird es immer in der zweiten Spalte gespeichert. Das bedeutet, dass auch, wenn Regel ähnlich wie diese:

$$\frac{\text{(example sentence,:=d c)} \quad \text{(,::d)}}{\text{(example sentence,:c)}} \text{ merge-2}$$

erzeugt wird, wird die Elternregelnreihenfolge im MKFG Array getauscht, also wie folgt gespeichert:

{'(example sentence,:c)'} {'(,::d)'} {'(example sentence,:=d c)'}

Damit ist gesichert, dass das letzte Symbol in der Warteschlange nicht das leere Wort repräsentiert.

Wenn die Schleife problemlos verlassen wird – also Warteschlange geleert wurde – wird noch das Parserverlaufarray aktualisiert. In Abbildung 21 ist ein Parserverlauf des erfolgreich verarbeiteten Satzes 'who likes John' dargestellt.

```
'H[]'              'who likes John'
'F[1,0]'           'who likes John'
'd[10]D[11,02]'    'who likes John'
'D[11,02]'         'who likes John'
'c[02]A[11]'       'who likes John'
'A[11]'            'likes John'
'a[110]b[111]'     'likes John'
'b[111]'           'John'
''                 ''
```

Abbildung 21: Parserverlauf des Satzes 'who likes John'

Falls ein Semantikstapel vorhanden ist, erfolgt ein zusätlicher Eintrag, in dem der semantische Ausdruck auf dem Stapel ausgewertet wird. In Abbildung 22 ist ein Parserverlauf des erfolgreich verarbeiteten Satzes 'zwei und vier zig', bei welchem semantische Ausdrücke vorkommen, dargestellt.

```
'Q[]'              'zwei und vier zig'    ''
'e[0]O[1]'         'zwei und vier zig'    ''
'O[1]'             'zwei und vier zig'    ''
'a[10]H[11]'       'zwei und vier zig'    ''
'H[11]'            'und vier zig'         '(2)[10]'
'd[110]D[111]'     'und vier zig'         '(2)[10]'
'D[111]'           'vier zig'             '(l y. l x. +(y)(x))[110] (2)[10]'
'A[1111,1110]'     'vier zig'             '(l y. l x. +(y)(x))[110] (2)[10]'
'b[1110]c[1111]'   'vier zig'             '(l y. l x. +(y)(x))[110] (2)[10]'
'c[1111]'          'zig'                  '(4)[1110] (l y. l x. +(y)(x))[110] ...'
''                 ''                     '(l x. *(10^1)(x))[1111] (4)[1110] ...'
''                 ''                     '(+(*(10^1)(4))(2))'
```

Abbildung 22: Parserverlauf des Satzes 'zwei und vier zig'

Zum Schluss wird geprüft, ob das Band auch leer ist. Falls ja, bedeutet das, dass der Satz in der Grammatik liegt. Sonst wird die entsprechende Fehlermeldung ausgegeben.

4.3 Programbeurteilung

Das im vorherigen Abschnitt beschriebene Programm ist geeignet zum top-down Parsen von MG. Sein Vorteil ist die Fähigkeit zum Parsen der Grammatik sowohl mit, als auch ohne semantische Ausdrücke. Jedoch ist das der erste Versuch zur Realisierung des Parserskonzeptes von Stabler, so hat es noch Schwächen, die verbessert werden können. Wie in Abschnitt 4.1 schon erwähnt wurde, ist zum Bilden der MKFG das Band erforderlich. Das soll eigentlich bei einer top-down Verarbeitung nicht passieren, dient aber zur Begrenzung der Rekursionstiefe bei der Regelerstellung. Was noch nicht optimal ist, ist der große Aufwand – zum Parsen eines Satzes muss immer die MKFG aus der MG erstellt werden (außerdem wird der Satz bei dieser Operation de facto bottom-up geparst). Das könnte bei großen Grammatiken problematische sein, vor allem, wenn mehrere Sätze mit einer Grammatik geparst werden sollen. Der Grund dafür ist, dass das Programm so eingerichtet wurde, damit es alles – von der Grammatikerstellung bis zum Parsen – macht, um zu testen, ob die Implementierung nach Stablers Konzept möglich ist. Falls es weiter benutzt werden sollte, ist es empfehlenswert, das Programm in einen MKFG-erstellungsteil und einen Parserteil zu unterteilen, vor allem, wenn es sich nur mit einer Grammatik beschäftigen sollte. Die MKFG könnte mit einer gewählten Rekursionstiefebegrenzung aus einer MG erstellt werden, die danach in einer Datenbank gespeichert würde. Dann müsste der Parser nicht bei jedem Verlauf die Funktion zur Erstellung von MKFG neu aufrufen, sondern einfach auf die Datenbank zugreifen.

Das Programm wurde auf wenigen Grammatiken getestet. Beim Verarbeiten einer Grammatik müssen viele verschiedene Fälle berücksichtigt werden, die nicht immer vorherzusehen wird. Zum Beispiel die Problematik der *merge*-3, die in Abbildung 18 dargestellt wird, tritt zwar in Stablers- und Beispielgrammatik auf, aber in der Zahlengrammatik nicht mehr. Deswegen ist es möglich, dass einige Situationen, die auftreten können, nicht berücksichtigt wurden.

Außerdem wird die Korrektheit der übergebenen MG nicht geprüft. Das bedeutet, dass der Benutzer selbst dafür sorgen muss, dass die Grammatik eine richtige Form besitzt, sonst funktioniert das Programm nicht. Zum Beispiel, wenn bei der Regel:

(likes,::=d =d v)

eine Klammer vergessen würde:

(likes,::=d =d v

was vielleicht auf dem ersten Blick nicht auffällt, würde einen Fehler erzeugen. Falls das Programm so verändert würde, dass die MKFG nur einmal erzeugt und in einer Datenbank gespeichert wird, ist dieses Problem weniger merklich.

5 Zusammenfassung

Linguisten probieren seit Jahren eine Grammatik natürlicher Sprache zu formalisieren. Demzufolge sind die formalen Grammatiken entstanden, wobei die Berühmtesten hat Noam Chomsky in der Chomsky Hierarchie definiert. Eine davon ist die kontextfreie Grammatik, die zur Formalisierung mancher Sätze benötigt werden kann, jedoch nicht zu allem. Da eine Grammatik, die nur ein wenig strenger als die kontextfreie Grammatiken ist, gebraucht wird, wurden die *mildly context sensitiv grammars* (leicht kontextfreie Grammatiken) erschafft. Ein Beispiel dafür ist eine multiple kontextfreie Grammatik, die zur Simulation natürlicher Sprache ziemlich gut geeignet ist. Anhand des minimalistischen Programms von Chomsky wurden minimalistische Grammatiken definiert, die aus einem minimalistischen Lexikon und zwei Operationen: *merge* und *move* bestehen. Diese Grammatiken können Verhältnisse zwischen Wörter gut definieren. Es ist bekannt, dass jede minimalistische Grammatik eine äquivalente multiple kontextfreie Grammatik besitzt. Erstellung von multiplen kontextfreien Regeln aus einer minimalistischen Grammatik erfolgt bottom-up.

Zur Erkennung, ob ein Satz in einer Grammatik liegt, werden Parser benutzt. Es gibt mehrere Parsertypen, wobei die am meisten erwünschte ist ein top-down Parser, da er am nähesten an der menschliches Satzverarbeitung liegt.

Da die Grammatiken nur den syntaktischen Aufbau des Satzes bestimmen, ist nicht klar welche Bedeutung die erzeugte Zeichenkette besitzt. Die Semantik kann mit Hilfe der Lambda-Ausdrücke beschrieben werden.

Es wurde von Edward Stabler vorgeschlagen aus einer minimalistischen Grammatik eine multiple kontextfreie Grammatik zu erzeugen und mit ihrer Hilfe ein Satz top-down zu parsen. Dabei sollte der Parser statt einem Stapel eine Prioritätswarteschlange besitzen, auf der die Stringargumente nach Bedarf sortiert werden können. Dazu wird die Nutzung eines Semantikstapels vorgeschlagen, der die Bedeutung der einzelnen Regeln mittels Lambda-Ausdrücke speichern kann.

Das Programm anhand des obigen Konzepts wurde in MATLAB geschrieben und realisiert eine top-down Verarbeitung eines Satzes zu einer minimalistischen Grammatik. Dabei muss jedoch eine Rekursionstiefenbegrenzung angesetzt werden, weil die multiplen kontextfreien Regeln bottom-up erzeugt werden. Dazu wird die Anzahl der Wörter auf dem Band benötigt, was in einem top-down Model nicht erwünscht ist. Das Programm kann zur weiteren Entwicklung des Themas benutzt werden, da es fähig ist sowohl die Grammatikumwandlung als auch das top-down Parsen einer multiplen kontextfreien Grammatik zu realisieren. Beide Teile funktionieren unabhängig voneinander und können nach Bedarf auch separat benutzt und entwickelt werden.

Literatur

[1] Kopp H. *Compilerbau Grundlagen, Methoden, Werkzeuge.* Hanser, München, 1988.

[2] Stabler E. Top-down recognizers for MCFGs and MGs. In *Proceedings of the 2nd Workshop on Cognitive Modeling and Computational Linguistics.*, Association for Computational Linguistics, pages 39 – 48, Portland, Oregon, 2011.

[3] beim Graben P., Meyer W., Römer R., and Wolff M. Bidirektionale Utterance-Meaning-Transducer für Zahlworte durch kompositionale minimalistische Grammatiken. In P. Birkholz and S. Stone, editors, *Tagungsband der 30. Konferenz lektronische Sprachsignalverarbeitung (ESSV)*, volume 91 of *Studientexte zur Sprachkommunikation*, pages 76 – 82, Dresden, 2019. TU-Dresden Press.

[4] Hemforth B. *Kognitives Parsing: Repräsentation und Verarbeitung sprachlichen Wissens.* Infix, 1993, Sankt Augustin.

[5] Graham S. Wagner T. Efficient and Flexible Incremental Parsing. *ACM Transactions on Programming Languages and Systems 20(5)*, 2000.

[6] Chomsky N. *Syntactic structures.* Mouton & Co., 1957.

[7] Hofstedt P. Vorlesung Compilerbau, Sommersemester 2018. Brandenburgische Technische Universitat Cottbus-Senftenberg.

[8] Erk K. and Priese L. *Theoretische Informatik.* Springer, Berlin, 2000.

[9] Chomsky N. Three models for the description of language. *IRE Transactions on Information Theory (2)*, 1956.

[10] Chomsky N. On certain formal properties of grammars. *Information and Control 2*, pages 137–167, 1959.

[11] beim Graben P. Vorlesung Kognitive Systeme II, Sommersemester 2018. Brandenburgische Technische Universitat Cottbus-Senftenberg.

[12] Joshi A. Tree Adjoining Grammars: How Much Context-Sensitivity Is Required to Provide Reasonable Structural Descriptions? In Dowty David, Karttunen Lauri, and Zwicky Arnold, editors, *Natural Language Parsing*, pages 206–250. Cambridge University Press, 1985.

[13] Götzmann D. Multiple Context-Free Grammars, November 2017.

[14] Chomsky N. A minimalist program for linguistic theory. In *MIT occasional papers in linguistics no. 1.* MIT Working Papers in Linguistics, 1993.

[15] Stabler E. Derivational minimalism. In *Logical Aspects of Computational Linguistics*, Nancy, France, 1996. Springer.

[16] Stabler E. *The logical approach to syntax.* MIT Press, 1992.

[17] beim Graben P., Römer R., Meyer W., Huber M., and Wolff M. Reinforcement Learning of Minimalist Numeral Grammars, 2019.

[18] beim Graben P., Meyer W., Römer R., and Wolff M. Bidirektionale Utterance-Meaning-Transducer für Zahlworte durch kompositionale minimalistische Grammatiken.