Effizientes Parsen und Generieren mit Baumadjunktionsgrammatiken

–Automatenmodelle–

Eine Ausarbeitung zu einem Proseminarvortrag von
Rainer Siedle, 29. Januar 1998

Inhaltsverzeichnis

Einleitung **1**

Phänomene der deutschen Sprache **2**

- Die Satzstruktur..... 2
- Topikalisierung..... 2
- Scrambling..... 3
- Extraposition..... 3
- Long−Distance Scrambling..... 4
- Linguistische Analyse..... 4

TAGs und Automaten **9**

- BEPDA..... 9
- Transformation von TAG nach BEPDA nach der ’leichten−Methode’..... 11
- Beweisidee: L(BEPDA) = L(TAG)..... 12
- V−TAGs..... 12
- {}−BEPDA..... 13
- Transformation von V−TAG nach {}−BEPDA nach der ’leichten−Methode’..... 15

Bestimmung der Verarbeitungslast **17**

- Ein Beispielautomat für BEPDA..... 17
- Metriken für die Bestimmung der Verarbeitungslast..... 20
- Ein Beispielautomat für {}−BEPDA..... 21
- Bestimmung der Verarbeitungslast..... 24

Abkürzungsverzeichnis **26**

Literaturverzeichnis **26**

Einleitung

Wir behaupten für TAGs ein entsprechendes äquivalentes Automatenmodell gefunden zu haben. Aber kann man bei entsprechend vorgegebenen Grammatiken gesprochener Sprachen tatsächlich von einer psycho−linguistisch korrekten Repräsentation bzw. Verarbeitung durch den Automaten sprechen? Kann man anhand des Modells Phänomene der Sprache nachvollziehen? Kann man damit herausfinden, welche Sätze von einem Sprecher der Sprache akzeptiert werden, welche weniger akzeptierbar sind und welche ausgeschlossen werden?

Diese Ausarbeitung soll Antworten auf all diese Fragen oder zumindest Ansätze dafür geben. Damit wir überhaupt eine Grundlage haben, müssen wir uns zuerst eine Sprache auswählen (Dieser Vorgang wird meistens deterministisch durch die Mutter bestimmt, daher unsere Muttersprache deutsch) und uns etwas näher mit ihr beschäftigen. Wir werden uns ein paar Phänomene und Strukturdaten zur deutschen Sprache ansehen und anhand eines Satzbeispiels überlegen, welche Variationen in welchem Ausmaße noch verständlich sind und warum das so ist. Danach werfen wir nochmals einen Blick auf den BEPDA [1], das Automatenmodell mit dem wir die TAGs modellieren wollen. Wir lernen eine erweiterte Form der TAGs kennen und das entsprechend erweiterte Automatenmodell {}−BEPDA. Dabei sehen wir uns noch eine einfache Methode an, wie man (V−)TAGs in äquivalente ({}−)BEPDAs umformen kann. Diese Methode wird die Eigenschaften der TAG direkt an den Automaten weitergeben, d.h. der Automat verhält sich linguistisch genauso wie die Grammatik. Eine Eigenschaft, die wir dann im letzten Teil ausnutzen werden, wenn wir uns unter Zuhilfenahme geeigneter Metriken den Versuch wagen, die Qualität von generierten Sätzen zu messen und mit der Akzeptanz bei Native−Speakern zu vergleichen.

1 Zu BEPDAs siehe auch die Ausarbeitung »Detetministisches Parsen« von Jens Haase.

Phänomene der deutschen Sprache

Die Satzstruktur

Das Verb kann an unterschiedlichen Stellen stehen:

- In Aussagesätzen ohne Komplementierer [2] steht das finite [3] Verb an zweiter Position:

  (1) Der Lehrer gibt den Kindern dieses Buch .

- Frage– und Befehlssätze beginnen mit dem finiten Verb:

  (2) Gibt der Lehrer den Kindern dieses Buch?

- Sätze mit Komplementierer enden mit dem finiten Verb:

  (3) Ich glaube, daß der Lehrer den Kindern dieses Buch gibt.

- Wenn das gebeugte Verb in einem finiten Hauptsatz ein Hilfsverb ist, endet der Satz mit dem infiniten Hauptverb, das finite Hilfsverb steht an zweiter Position:

  (4) Der Lehrer hat den Kindern dieses Buch gegeben.

Wir können die ersten beiden Fälle zusammenfassen, wenn wir im zweiten Fall, eine 'leere Einheit' an den Satzanfang stellen. Ein Satz ohne Komplementierer ist somit zerlegbar in Vorfeld (VF, der Satzteil vor dem finiten Verb) und Mittelfeld (MF, Satzteile zwischen dem finiten und dem infiniten Verb bzw. Satzende):

(5a) VF finites Verb MF (infinites Verb)

Enthält der Satz einen Komplementierer, kann man entsprechend aufteilen:

(5b) Komp MF (infinites Verb) finites Verb

Die grundlegende Aufteilung von Sätzen in Vorfeld und Mittelfeld gibt uns zwei Möglichkeiten, die Wortstellung zu wechseln. Erstens können wir unterschiedliche Elemente ins Vorfeld bringen und zweitens können wir auch die Reihenfolge der Komponenten im Mittelfeld ändern.

Topikalisierung (Vorfeldbesetzung)

Dies entspricht dem Wechsel des Elementes im Vorfeld:

(7a) Der Lehrer hat den Kindern dieses Buch gegeben.

(7b) Dieses Buch hat der Lehrer den Kindern gegeben.

---

2 Komplementierer (engl. Complementizer) bezeichnen eine kleine Menge grammatischer Elemente wie nebensatzeinleitende Konjunktionen (im Dt. u.a. »daß«, »ob«, »um (... zu)«, »weil«), die die spezifische Funktion von eingebetteten satzmäßigen Strukturen (= Komplemente) anzeigen

3 Finite Formen sind Verbformen, die Person und Numerus angeben und die grammat. Merkmale von Person, Numerus, Tempus und Modus tragen, z.B. ich *lebe*, du *lebst*, er *lebt*

(7c) Gestern hat der Lehrer den Kindern dieses Buch gegeben.

(7d) * Gestern der Lehrer hat den Kindern dieses Buch gegeben.

(7e) * Hat der Lehrer den Kindern dieses Buch gegeben.

(7a) Standard, Subjekt im VF. (7b) Direktes Objekt wurde ins VF gezogen, Subjekt verbleibt im MF. (7c) Eine Zeitangabe besetzt das VF, somit gehen alle Argumente von »geben« ins MF. (7d) und (7e) werden ausgesondert, weil das VF unbesetzt bzw. übervölkert ist.

Scrambling (Durcheinandermischen)

Die Bezeichnung für die Permutation im Mittelfeld.

(8) ...daß [eine hiesige Firma] [meinem Onkel][die Möbel][vor drei Tagen][ohne Voranmeldung] zugestellt hat.

Jede Permutation der fünf Elemente im MF (5! = 120) ist grammatikalisch korrekt. Es ist also möglich, beim Durcheinandermischen mehrere Elemente gleichzeitig zu beteiligen.

Extraposition (Herausstellung)

Ein Teilsatz wird herausgestellt:

(9a) Max hat dem Schneider [PRO [4] die Brücke zu reparieren] versprochen.

(9b) Max hat dem Schneider versprochen, [PRO die Brücke zu reparieren].

(9c) Weil Max dem Schneider versprochen hat, [PRO die Brücke zu reparieren], ...

Wenn der eingebettete Satz ein finiter mit Komplementierer ist, ist die Variante mit Extraposition die bevorzugte Variante:

(10a) ? [5] Max hat dem Schneider [daß er die Brücke reparieren wird] versprochen.

(10b) Max hat dem Schneider versprochen, [daß er die Brücke reparieren wird].

(10c) Weil Max dem Schneider versprochen hat, [daß er die Brücke reparieren wird], ...

4 PRO steht als Platzhalter, 'leere NP' (NP = Nominalphrase)

5 Dieses »?« bedeutet, daß der dahinterstehende Satz grammatikalisch fragwürdig ist.

Long−Distance Scrambling

Bei eingebetteten Sätzen können unter bestimmten Umständen Satzteile mit dem einbettenden Satz durchmischt werden, wobei sie eine Spur an ihrem Ursprung hinterlassen:

(11a) ... daß niemand [PRO den Kühlschrank zu reparieren] versprochen hat.

(11b) ... daß [den Kühlschrank] $_i$ niemand [t $_i$ zu reparieren] versprochen hat.

Dabei gibt es allerdings Einschränkungen:

(12a) Peter hat versprochen, daß er den Kühlschrank reparieren wird.

(12b) * Peter hat den Kühlschrank versprochen, daß er reparieren wird.

(12c) Peter hat gesagt, er wird den Kühlschrank reparieren.

(12d) * Peter hat den Kühlschrank gesagt, er wird reparieren.

Scrambling im MF ist relativ freizügig. Es gibt keine Beschränkung in der Anzahl der Teilsatzgrenzen, die überschritten werden dürfen.

(13a) ... daß [den Kühlschrank] $_i$ niemand [[t $_i$ zu reparieren] zu versuchen] versprochen hat.

(13b) ... daß [den Kühlschrank] $_i$ niemand [[[t $_i$ zu reparieren] zu versuchen] zu versprechen] bereit ist

Gleichzeitig können auch mehrere Bestandteile im selben Satz umgestellt werden.

(14) ... daß [dem Kunden] $_i$ [den Kühlschrank] $_j$ bisher noch niemand t $_i$ [[t $_j$ zu reparieren] zu versuchen] versprochen hat

Obwohl Scrambling prinzipiell auf zweierlei Weise unbegrenzt ist (bei der Überwindung von Satzgrenzen und bei der Anzahl der betroffenen Satzbestandteile), werden die dadurch generierten Variationen nicht alle gleichwertig akzeptiert. Bestimmte Variationen erscheinen beispielsweise wegen semantischen Einflußfaktoren besser als andere.

Linguistische Analyse

(15) Weil [niemand] [das Fahrrad] [zu reparieren] [zu versuchen] [verspricht].

Satz (15) hat 5 Satzbausteine, die prinzipiell unbeschränkt permutierbar sind. Die Anzahl von 5! = 120 Permutationen können wir um den Faktor 2*2 reduzieren, wenn wir die beiden NPs darauf beschränken, den entsprechenden Verben voranzugehen. Die verbleibenden 30 Permutationen sind in Tabelle 1 aufgeführt.

Keine Extraposition nach 'verspricht'		
(i)	Weil niemand das Fahrrad zu reparieren zu versuchen verspricht.	ok
(ii)	Weil das Fahrrad niemand zu reparieren zu versuchen verspricht.	?
(iii)	Weil niemand zu versuchen, das Fahrrad zu reparieren, verspricht.	?
(iv)	Weil niemand das Fahrrad zu versuchen zu reparieren verspricht.	?
(v)	Weil das Fahrrad niemand zu versuchen zu reparieren verspricht.	?
(vi)	Weil das Fahrrad zu reparieren niemand zu versuchen verspricht.	ok
(vii)	Weil das Fahrrad zu reparieren zu versuchen niemand verspricht.	?
(viii)	Weil zu versuchen, das Fahrrad zu reparieren, niemand verspricht.	?
(ix)	Weil zu versuchen das Fahrrad niemand zu reparieren verspricht.	*
(x)	Weil das Fahrrad zu versuchen niemand zu reparieren verspricht.	*
(xi)	Weil zu versuchen niemand das Fahrrad zu reparieren verspricht.	*
(xii)	Weil das Fahrrad zu versuchen zu reparieren niemand verspricht	?

Extraposition der 'versuchen'–Phrase		
(xiii)	Weil niemand verspricht, das Fahrrad zu reparieren zu versuchen.	ok
(xiv)	Weil niemand das Fahrrad verspricht zu reparieren zu versuchen.	?
(xv)	Weil das Fahrrad niemand verspricht zu reparieren zu versuchen.	?
(xvi)	Weil niemand verspricht, zu versuchen, das Fahrrad zu reparieren.	ok
(xvii)	Weil niemand verspricht, das Fahrrad zu versuchen zu reparieren.	?
(xviii)	Weil niemand das Fahrrad verspricht zu versuchen zu reparieren.	ok
(xix)	Weil das Fahrrad niemand verspricht zu versuchen zu reparieren.	?
(xx)	Weil niemand das Fahrrad zu reparieren verspricht zu versuchen.	?
(xxi)	Weil das Fahrrad zu reparieren niemand verspricht zu versuchen.	?
(xxii)	Weil das Fahrrad niemand zu reparieren verspricht zu versuchen.	?

Ausschließliche Extraposition der 'reparieren'–Phrase		
(xxiii)	Weil niemand zu versuchen verspricht, das Fahrrad zu reparieren.	?
(xxiv)	Weil niemand zu versuchen das Fahrrad verspricht zu reparieren.	*
(xxv)	Weil niemand das Fahrrad zu versuchen verspricht zu reparieren.	*?
(xxvi)	Weil das Fahrrad niemand zu versuchen verspricht zu reparieren.	*?
(xxvii)	Weil zu versuchen niemand das Fahrrad verspricht zu reparieren.	*
(xxviii)	Weil zu versuchen niemand verspricht das Fahrrad zu reparieren.	*?
(xxix)	Weil zu versuchen das Fahrrad niemand verspricht zu reparieren.	*
(xxx)	Weil das Fahrrad zu versuchen niemand verspricht zu reparieren.	*?

Tabelle 1 [6]: Die Rohdaten

6 In der letzten Spalte wird jeweils die Bewertung von Native Speakern angegeben, dabei bedeutet »ok« : gut, »?«: fragwürdig, »*?«: besonders fragwürdig und »*«: nicht akzeptabel.

Mit den zuvor eingeführten Begriffen ist es möglich, die Herleitung der Variationen zu beschreiben. Verblüffenderweise können alle Sätze, die gut oder fragwürdig bewertet werden, mit dem Modell beschrieben werden und alle Sätze, die nicht akzeptabel sind, scheiden aus. Wir werden aus jedem Bereich eine Beispielherleitung im Detail ansehen.

Satz (xxii) aus Tabelle 1 als Beispiel einer fragwürdigen Variante:

Wir starten mit der Eingabe...

(16a) Weil niemand [PRO [PRO das Fahrrad zu reparieren] zu versuchen] verspricht.

Zuerst führen wir ein Long–Distance Scrambling der NP »das Fahrrad« an die Anfangsposition des einbettenden Satzes durch...

(16b) Weil [das Fahrrad] $_i$ niemand [PRO [PRO t $_i$ zu reparieren] zu versuchen] verspricht.

Dann wird der Rest der »reparieren«–Phrase ohne Textänderung in den einbettenden Satz gemischt (Scrambling)...

(16c) Weil [das Fahrrad] $_i$ niemand [PRO t_i zu reparieren] $_j$ [PRO t_j zu versuchen] verspricht.

Schließlich wird die verbleibende »versuchen«–Phrase durch Extraposition herausgestellt und man erhält (xxii)...

(16d) ? Weil das Fahrrad niemand zu reparieren verspricht zu versuchen.

Satz (xxiv) aus Tabelle 1 als Beispiel einer nicht akzeptablen Variante:

(17a) Weil niemand [PRO [PRO das Fahrrad zu reparieren] zu versuchen] verspricht.

Wir könnten zuerst das NP per Long–Distance Scrambling ohne Änderung des Textes verlagern...

(17b) Weil niemand [das Fahrrad] $_i$ [PRO [PRO t $_i$ zu reparieren] zu versuchen] verspricht.

Danach den Rest der »reparieren«–Phrase herausstellen...

(17c) Weil niemand [das Fahrrad] $_i$ [PRO t $_j$ zu versuchen] verspricht [PRO t $_i$ zu reparieren] $_j$.

Nun könnten wir eigentlich, um zum angedeuteten Ziel (17d) zu kommen, via Scrambling die »versuchen«–Phrase an den gewünschten Ort bringen, aber in diesem

Fall wird mindestens eine Spur verlorengehen. Denn die »[t_j zu versuchen]«–Phrase muß von der »[t_i zu reparieren] $_j$«–Phrase c–kommandiert [7] werden, welche wiederum von der Nominalphrase »[das Fahrrad] $_i$« c–kommandiert werden muß. Das ist unmöglich, denn »das Fahrrad« muß von der »versuchen«–Phrase c–kommandiert werden, da es im String zwischen der »versuchen«–Phrase und dem Hauptverb liegt.

(17d) * Weil niemand zu versuchen das Fahrrad verspricht zu reparieren.

Die Analyse aller Sätze ist in Tabelle 2 kurz zusammengefaßt.

Keine Extraposition nach 'verspricht'		
(i)	Keine Umstellung	ok
(ii)	Scrambling von NP	?
(iii)	Extraposition von »reparieren«	?
(iv)	Scrambling von NP (SV) + Extraposition von »reparieren«	?
(v)	Scrambling von NP + Extraposition von »reparieren«	?
(vi)	Scrambling von »reparieren«	ok
(vii)	Scrambling von »versuchen«	?
(viii)	Scrambling von »versuchen« + Extraposition von »reparieren«	?
(ix)	**Ausgesondert:** unverbundene Spur in [t $_i$ zu versuchen]	*
(x)	**Ausgesondert:** unverbundene Spur in [t $_i$ zu versuchen]	*
(xi)	**Ausgesondert:** unverbundene Spur in [t $_i$ zu versuchen]	*
(xii)	Scrambling von NP + Extraposition von »reparieren« + Scrambling von »versuchen«	?

Extraposition der 'versuchen'–Phrase		
(xiii)	Extraposition von »versuchen«	ok
(xiv)	Scrambling von NP (SV) + Extraposition von »versuchen«	?
(xv)	Scrambling von NP + Extraposition von »versuchen«	?
(xvi)	Extraposition von »versuchen« + Extraposition von »reparieren«	ok
(xvii)	Extraposition von »versuchen« + Scrambling von NP (SV) + Extraposition von »reparieren«	?
(xviii)	Scrambling von NP (SV) + Extraposition von »versuchen« + Extraposition von »reparieren«	ok
(xix)	Scrambling von NP + Extraposition von »versuchen« + Extraposition von »reparieren«	?
(xx)	Scrambling von »reparieren« (SV) + Extraposition von »versuchen«	?
(xxi)	Scrambling von »reparieren« + Extraposition von »versuchen«	?

7 c–kommandieren (von engl. constituent–command) erklärt sich folgendermaßen: Knoten n c–kommandiert Knoten m gdw. der Knoten, der n unmittelbar dominiert auch m dominiert, aber n es nicht tut. (dominieren auf syntaktischer Ebene, d.h. in der Verarbeitungshierarchie eine Stufe höher)

Extraposition der 'versuchen'–Phrase		
(xxii)	Scrambling von NP + Scrambling von »reparieren (SV) + Extraposition von »versuchen«	?

Ausschließliche Extraposition der 'reparieren'–Phrase		
(xxiii)	Scrambling von »reparieren« (SV) + Extraposition von »reparieren«	?
(xxiv)	**Ausgesondert:** unverbundene Spur in [t $_i$ zu versuchen]	*
(xxv)	Scrambling von NP (SV) + Scrambling von »reparieren« (SV) + Extraposition von »reparieren«	*?
(xxvi)	Scrambling von NP + Scrambling von »reparieren« (SV) + Extraposition von »reparieren«	*?
(xxvii)	**Ausgesondert:** unverbundene Spur in [t $_i$ zu versuchen]	*
(xxviii)	Scrambling von »versuchen« + Scrambling von »reparieren« (SV) + Extraposition von »reparieren«	*?
(xxix)	**Ausgesondert:** unverbundene Spur in [t $_i$ zu versuchen]	*
(xxx)	Scrambling von NP + Scrambling von »reparieren« (SV) + Extraposition von »reparieren« + Scrambling von »versuchen«	*?

Tabelle 2 [8]: Linguistische Analyse

Nun gibt es noch ein Problem: Vier Sätze in Tabelle 1 wurden mit »*?« deklariert, können aber analytisch mit Scrambling und Extraposition erfaßt werden. Eine einfache Erklärung hierfür gibt es nicht und soll in diesem Rahmen auch nicht angegangen werden [9].

8 Beim Scrambling steht NP immer für »das Fahrrad«. »SV« ist die Abkürzung von String–Vacuously und heißt, daß nur strukturelle, keine sichtbare Textänderung passiert. Andere Herleitungen sind evtl. möglich, insbesondere andere Reihenfolgen in der Anwendung der Operatoren.

9 Wer will, kann sich natürlich die Seiten 20ff des bearbeiteten Papiers ansehen. Dort zeichnen sich Ansätze einer Erklärung ab.

BEPDA

Im Rahmen des Proseminars sind wir schon einmal auf den BEPDA gestoßen. Bevor wir uns näher mit ihm beschäftigen, hier erst noch eine kleine Wiederholung.

Der BEPDA ist eine Erweiterung des PDA. Anstatt eines einfachen Stacks besitzt er einen Stack von Stacks und kann prinzipiell mit zwei Arten von Schritten beschrieben werden:

- UNWRAP:

  Endliche evtl. leere Folgen von Stacks direkt über und direkt unter einem bestimmten Stack (der zum neuen obersten Stack wird) werden entfernt. Zusätzlich wird eine endliche evtl. leere Folge von Stacksymbolen von dem neuen obersten Stack heruntergenommen und durch ein einzelnes neues Stacksymbol ersetzt.

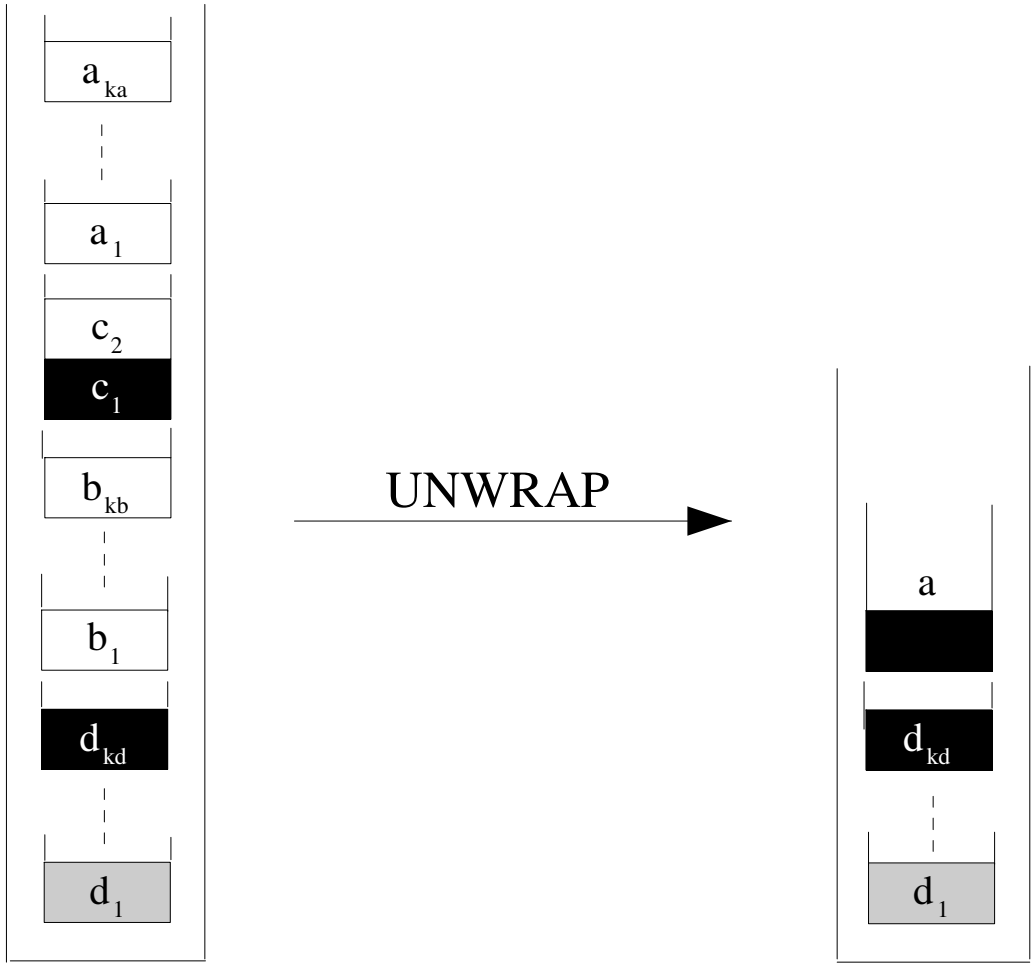


Abbildung 1: UNWRAP

- PUSH−NEW−STACK:

  Ein leerer Stack wird oben auf den Stack von Stacks gelegt.

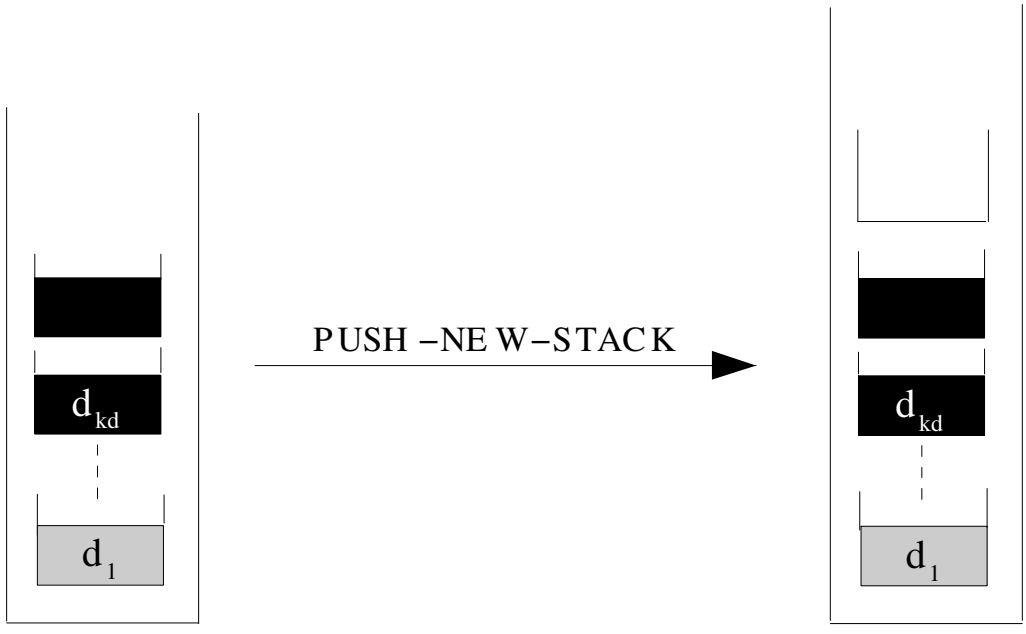


Abbildung 2: PUSH−NEW−STACK

Um es später einfacher zu haben, führen wir ein paar weitere Schrittypen ein, die als Spezialfall oder Kombination der beiden Grundtypen definiert sind. Damit wird der Umgang mit dem Automatenmodell lediglich erleichtert, ohne ein neues Automatenmodell einzuführen.

- PUSH:

  Ein einzelnes Stacksymbol wird ganz oben auf dem obersten Stack abgelegt. Dies ist ein einfaches UNWRAP, bei dem nichts entfernt wird. Ein PUSH, welches mehrere Symbole gleichzeitig auf den obersten Stack schreibt ist als Kombination mehrerer elementaren PUSHs möglich.

- POP:

  Eine Folge von Stacksymbolen wird oben vom obersten Stack entfernt, ohne ein neues Element aufzulegen. Simulierbar durch ein einfaches UNWRAP, welches die zu entfernenden Stacksymbole plus einem weiteren Stacksymbol vom obersten Stack entfernt und das zusätzliche Stacksymbol wieder auflegt. Dabei werden keine Stacks entfernt.

- NEW−PUSH:

  Ein einzelnes Stacksymbol wird in einem neuen Stack ganz oben im Stack von Stacks abgelegt. Also haben wir eine simple Hintereinanderreihung von PUSH−NEW−STACK und PUSH.

  SHIFT:

  Wenn wir dabei ein Eingabesymbol verbrauchen, nennen wir den entsprechenden Schritt einen SHIFT.

- IF Z ...:

  Wir können ein einfaches Konditional einbauen, indem wir das oberste Element des obersten Stacks wegnehmen und wieder auflegen (mittels UNWRAP) und den Automaten in einen entsprechenden Zwischenzustand (also die Bedingung) versetzen.

  Bsp.: IF Z PUSH−NEW−STACK führt die PUSH−NEW−STACK Aktion nur aus, wenn ganz zuoberst ein Z aufliegt.

Transformation von TAG nach BEPDA nach der 'leichten Methode'

Um eine direkte Umwandlung von TAG nach BEPDA hinzubekommen, benötigen wir entsprechende Überführungsfunktionen für unseren Automaten. Unnötige Formalismen können wir uns wieder ersparen, indem wir Metaregeln verwenden.

Die Schreibweise $A^{(T,a,z)}$ kennzeichnet jeden Knoten eindeutig: T steht für den Initial− oder Hilfsbaum indem der Knoten ist, a gibt die Adresse in Punktnotation [10] an und z nimmt einen der Werte »t« = 'Top' oder »b« = 'Bottom' an (jeder Knoten wird in 'Top' und 'Bottom' aufgespalten, obwohl es nur bei Knoten an denen Adjunktion stattfindet wirklich nötig ist, denn die betreffenden Knoten 'gehören' sozusagen zu zwei Bäumen).

- *Terminal Shift:*

  Führe für jeden äußeren Knoten eines Baumes, der mit einem Terminalsymbol bezeichnet ist ein SHIFT dieses Symbols aus.

- *Reduce:*

  Sei n ein Knoten in T, der mit A bezeichnet ist und Adresse a hat. Seien seine Töchter n_1, ..., n_n derart, daß n_i mit X_i gelabelt ist (X_i entweder Terminal oder Nichtterminal). Wir nehmen an, daß die k−te Tochter den Fußknoten dominiert (bzw. k = 1, falls T ein initialer Baum ist). Führe für jeden solchen Knoten n folgenden UNWRAP−Schritt aus:

  Der (n−k+1)−te Stack wird festgehalten. Entferne die (k−1) Stacks $[X_1^{(T,a.1,t)}$ ... $[X_{(k-1)}^{(T,a.(k-1),t)}$ darunter, die (n−k) Stacks $[X_{(k+1)}^{(T,a.(k+1),t)}$ ... $[X_n^{(T,a.n,t)}$ darüber und das obenauf befindliche Symbol $X_k^{(T,a.k,t)}$. Lege anschließend $A^{(T,a,b)}$ auf den fixierten Stack.

- ε−*Reduce:*

  Sei n ein Knoten in T, der mit A bezeichnet ist, Adresse a hat und einen einzelnen Tochterknoten mit Label »ε« besitzt. Führe ein NEW−PUSH aus, wobei $[A^{(T,a,b)}$ oben auf den Stack von Stacks gelegt wird.

- *Adjoin:*

  Für jeden Knoten n in T, der mit A bezeichnet ist und Adresse a hat, wo eine

10 Der Wurzelknoten hat Adresse e, wenn n1, ..., nk die Tochterknoten von Knoten n mit der Adresse a sind, dann haben sie die Adresse a.1, ..., a.k.

Adjunktion von Baum T' (dessen Fußknoten n' mit Adresse a' ist) stattfinden kann, füge einen PUSH des Fußknotens des adjungierten Baumes hinzu. (also: IF $A^{(T,a,b)}$ PUSH $A^{(T',a',b')}$). Die Aktion wird nicht dazugenommen, wenn der Knoten mit der Null−Adjoining (NA) Constraint oder der Selective−Adjoining (SA) Constraint, wobei T' nicht enthalten ist, markiert ist.

- *Unadjoin:*

  Für jeden Wurzelknoten n mit Label A in Hilfsbaum T tue folgendes:

  POP $A^{(T,e,t)}$

  Der Effekt ist, daß der Automat jegliches Wissen um den adjungierten Baum verliert.

- *Noadjoin:*

  Für alle Knoten n mit Label A an Adresse a, die nicht mit der Obligatory−Adjoining (OA) Constraint versehen sind, füge einen UNWRAP−Schritt hinzu, der das oberste Stacksymbol austauscht: aus dem Label des ('Bottom'−)Knotens wird das Label des ('Top'−)Knotens.

  $A^{(T,a,t)}$ –> $A^{(T,a,b)}$

Dieses Verfahren generiert für jede TAG einen nichtdeterministischen Automaten, dessen Überführungsfunktion zusammen mit der Stackbelegung unmittelbar in Bezug zur Quellgrammatik steht. Bei linguistisch motivierter Grammatik, ist es möglich die Verarbeitung durch den Automaten linguistisch zu beschreiben.

Beweisidee: L(BEPDA) = L(TAG)

Der Beweis, daß jede TAG einen äquivalenten BEPDA besitzt, basiert auf Induktion über die Anzahl der Adjunktionen in der Ableitung bzw. der Tiefe der Stacks im Stack von Stacks.

Um zu zeigen, daß für jeden BEPDA M eine äquivalente TAG G existiert, konstruiert man den zu M dualen EPDA [11] M'. Man kann per Induktion über die Anzahl der Verarbeitungsschritte zeigen, daß jede Verarbeitung durch M, der umgekehrten Verarbeitung in M' entspricht. Damit folgt die Äquivalenz als Reduktion auf die bekannte Äquivalenz zwischen EPDA und TAG (Vijay−Shanker, 1987).

V−TAGs

TAGs können mit fünf Angaben definiert werden: man benötigt eine Menge V_N von Nichtterminalsymbolen, eine Menge V_T von Terminalsymbolen, ein Startsymbol S, sowie Mengen V_I und V_A von Initial− bzw. Auxiliarbäumen.

V−TAGs sind eine erweiterte Form von TAGs. V_I und V_A sind hier Vektoren (d.h. Mengen von Mengen) von Bäumen. Während die Mengen in V_I genau einen

11 Zur Erinnerung der EPDA ist das Gegenstück zum BEPDA und arbeitet 'Top−Down'.

Initialbaum und beliebig viele andere Bäume enthalten dürfen, dürfen die Mengen in V_a nur Auxiliarbäume enthalten. Ein weiterer Unterschied ist, daß die Vektorien in V_I und V_A mit sogenannten Dominance Links versehen sind. Damit wird eine zweistellige Relation über den Knoten eines Vektors definiert, die angibt, von welchen Knoten n (Fußknoten eines Auxiliarbaumes) ein Knoten m (beliebiger Knoten) dominiert werden kann.

Bei der Ableitung in V–TAGs ist zu beachten, daß pro Ableitungsschritt für jeden Vektor von Bäumen dieselbe Anzahl von Bäumen aus diesem Vektor verwendet werden muß und daß am Ende der Ableitung die Dominance Links der Grammatik eine Teilmenge der Dominance Relation (reflexiv–transitive Hülle) des abgeleiteten Baumes sind. Die Ableitung funktioniert nach folgendem Schema: Zunächst wird ein einzelner Vektor aus V_I ausgewählt, dessen Initialbaum als Initialbaum für die Ableitung dient. Nach und nach können zusätzliche Vektoren aus V_A ausgewählt werden und die darin enthaltenen Bäume adjungiert werden. Es gibt keine weitere Einschränkung außer der, daß sobald ein Baum eines Vektors adjungiert wird, alle Bäume dieses Vektors irgendwann im Verlauf der Ableitung adjungiert werden müssen (diese Einschränkung gilt auch für den Startervektor aus V_I).[12]

Wir können bei den Dominance Links ausschließen, daß n und m im selben Baum sind, denn wenn n nicht der Fußknoten ist, dann ist der Link unerfüllbar, anderenfalls trivial. Dominance Links verbinden in der Regel also zwei unterschiedliche Bäume. Zwei Bäume sind dann durch Dominance Links zusammengeschlossen, wenn es sozusagen einen Pfad von Dominance Links zwischen diesen Bäumen gibt. Wenn der Knoten n den Knoten m per Dominance Link dominiert, dann sprechen wir bei n von einem passiven Dominanzbedürfnis an den Link und bei m von einem aktiven. Ist nur einer der beiden beteiligten Bäume adjungiert, dann bleibt das Dominanzbedürfnis unerfüllt. Die Menge aller unerfüllten aktiven Dominanzbedürfnisse eines Knotens/einer Ableitung wird mit $\top$ annotiert, die Menge der unerfüllten passiven mit $\bot$.

{}–BEPDA

Ein Automatenmodell, welches äquivalent zu V–TAGs sein soll, muß dazu in der Lage sein, mit den Dominance Links richtig umgehen zu können. Diese können den Fußknoten eines Baumes mit einem inneren Knoten eines anderen Baumes verbinden. Deshalb müssen wir in einer Bottom–Up Ableitung aufzeichnen können, daß einem Knoten der einen anderen via Dominance Link verbindet der verbundene Baum adjungiert werden muß.

Um dies zu erreichen, erweitern wir die Stacksprache des BEPDA, indem wir zu jedem Stack im Stack von Stacks eine Indexmenge hinzufügen. Beim UNWRAP werden die entsprechenden Indexmengen der betroffenen Stacks auf den obersten Stack gegeben, was seine Entsprechung darin findet, Indexe durch den abgeleiteten Baum nach oben

12 Im Verlauf des letzten Kapitels werden wir ein (sehr einfaches, fast triviales) Beispiel einer Ableitung sehen.

zu reichen. Indexmengen werden nie kopiert (der {}−BEPDA ist deshalb linear). Das löschen eines Indexes leitet die Adjunktion des entsprechenden Baumes ein. Da die Indizes in Mengen gruppiert werden, können sie in beliebiger Reihenfolge gelöscht werden, wie es eben auch keine Bestimmung über die Adjunktionsreihenfolge von Bäumen in V−TAG gibt.

Die Übergangsbewegungen müssen demnach angepaßt werden:

- UNWRAP:

  Die Indexmenge des neuen obersten Stacks ist die Vereinigung aller Indexmengen der betroffenen Stacks (inklusive des fixierten Stacks, welcher zum obersten Stack wird). Die Indizes werden sozusagen gesammelt.

- PUSH−NEW−STACK:

  Mit dem neuen Stack wird eine leere Indexmenge angelegt.

Ohne Änderung der formalen Eigenschaften könnten wir die Definition so modifizieren, daß Indexmengen nicht mit Stacks, sondern mit einzelnen Stacksymbolen verknüpft werden. Alle Operationen arbeiten dann nur mit der Indexmenge des obersten Stacksymbols des obersten Stacks, wobei es jedoch erlaubt ist, die Indexsymbole beliebig unter den Stacksymbolen eines Stacks auszutauschen.

Wir können hier ebenfalls die Makros definieren, die wir bereits vom einfachen BEDA kennen:

- PUSH:

  Ein einzelnes Stacksymbol wird ganz oben auf dem obersten Stack abgelegt. Zusätzlich können weitere Indexsymbole zur Indexmenge des obersten Stacks hinzugefügt werden.

- POP:

  Eine Folge von Stacksymbolen wird oben vom obersten Stack entfernt, ohne ein neues Element aufzulegen. Gleichzeitig kann eine Teilmenge der Indizes des obersten Stacks ersatzlos gelöscht werden.

- NEW−PUSH:

  Ein einzelnes Stacksymbol wird in einem neuen Stack mit einer neuen evtl. leeren Indexmenge ganz oben im Stack von Stacks abgelegt.

  SHIFT:

  Wenn wir dabei ein Eingabesymbol verbrauchen, nennen wir den entsprechenden Schritt einen SHIFT.

- IF Z ...:

  Hier gibt es auf diesem Abstraktionsniveau keinerlei Änderung zum Konditional des einfachen BEPDA.

Transformation von V−TAG nach {}−BEPDA nach der 'leichten Methode'

Die Transformation verläuft wie im 'TAG −> BEPDA'−Fall, wobei natürlich die Dominance Links beachtet werden müssen. Ein Indexsymbol repräsentiert jeweils eindeutig einen Dominance Link. An dieser Stelle sehen wir auch einen Nutzen der Zweiteilung der Knoten in Bottomknoten und Topknoten: Wenn die aktiven Dominanzbedürfnisse mit der Bottomversion und die passiven mit der Topversion verknüpft werden, ist es möglich Dominanzbedürfnisse zu erfüllen, die entstehen wenn Dominance Link und Adjunktion in einem Knoten zusammenfallen.

Es folgt die modifizierte Beschreibung der Übergangsfunktion:

- *Terminal Shift:*

  Führe für jeden äußeren Knoten eines Baumes, der mit einem Terminalsymbol bezeichnet ist ein SHIFT dieses Symbols aus.

- *Reduce:*

  Sei n ein Knoten in T, der mit A bezeichnet ist und Adresse a hat. Seien seine Töchter n_1, ..., n_n derart, daß n_i mit X_i gelabelt ist (X_i entweder Terminal oder Nichtterminal). Wir nehmen an, daß die k−te Tochter den Fußknoten dominiert (bzw. k = 1, falls T ein initialer Baum ist). Führe für jeden solchen Knoten n folgenden UNWRAP−Schritt aus:

  Der (n−k+1)−te Stack wird festgehalten. Entferne die (k−1) Stacks $[X_1^{(T,a.1,t)} \ldots [X_{(k-1)}^{(T,a.(k-1),t)}$ darunter, die (n−k) Stacks $[X_{(k+1)}^{(T,a.(k+1),t)} \ldots [X_n^{(T,a.n,t)}$ darüber und das obenauf befindliche Symbol $X_k^{(T,a.k,t)}$. Lege anschließend $A^{(T,a,b)}$ auf den fixierten Stack. Die Indexmenge des fixierten Stacks wird die Vereinigung von $\top(n)$ mit den Indexmengen aller n_i, abzüglich der Symbole aus $\bot(n)$.

- ε−*Reduce:*

  Sei n ein Knoten in T, der mit A bezeichnet ist, Adresse a hat und einen einzelnen Tochterknoten mit Label »ε« besitzt. Führe ein NEW−PUSH aus, wobei $[A^{(T,a,b)}$ und $\top(n)$ oben auf den Stack von Stacks gelegt werden.

- *Adjoin:*

  Für jeden Knoten n in T, der mit A bezeichnet ist und Adresse a hat, wo eine Adjunktion von Baum T' (dessen Fußknoten n' mit Adresse a' ist) stattfinden kann, füge einen PUSH des Fußknotens des adjungierten Baumes hinzu. Die Indexmenge des obersten Stacks wird um $\top(n')$ erweitert. (also: IF $A^{(T,a,b)}$ PUSH $(A^{(T',a',b')}, \top(n'))$). Die Aktion wird nicht dazugenommen, wenn der Knoten mit der Null−Adjoining (NA) Constraint oder der Selective−Adjoining (SA) Constraint, wobei T' nicht enthalten ist, markiert ist.

- *Unadjoin:*

  Für jeden Wurzelknoten n mit Label A in Hilfsbaum T tue folgendes:

POP $A^{(T,e,t)}, \perp(n)$

Der Effekt ist, daß der Automat jegliches Wissen um den adjungierten Baum verliert.

- *Noadjoin:*

  Für alle Knoten n mit Label A an Adresse a, die nicht mit der Obligatory−Adjoining (OA) Constraint versehen sind, füge einen UNWRAP−Schritt hinzu, der das oberste Stacksymbol austauscht: aus dem Label des Bottomknotens wird das Label des Topknotens.

  $A^{(T,a,t)} \rightarrow A^{(T,a,b)}$

Betrachten wir die Variante, bei der die Indexmengen den einzelnen Stacksymbolen zugeordnet werden, gibt es zwei weitere Übergangsbewegungen:

- *Raise:*

  Ein Symbol einer Indexmenge wird im Stack unter den Stacksymbolen nach oben verschoben.

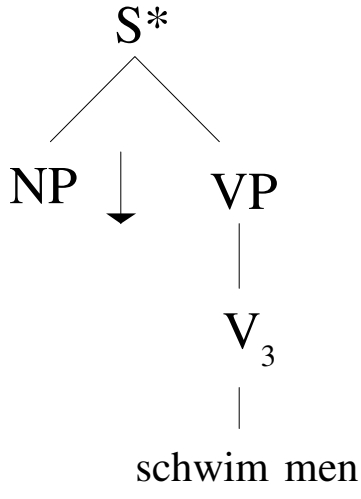
- *Lower:*

  Ein Symbol einer Indexmenge wird im Stack unter den Stacksymbolen nach unten verschoben.

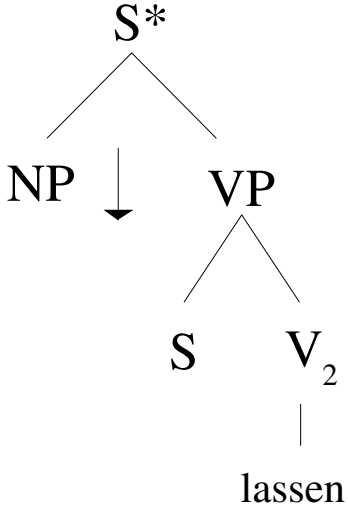
Ein Beispielautomat für BEPDA

Wir sehen uns zunächst einmal die Konstruktion eines BEPDAs aus einer Beispiel–TAG an. Gegeben sei das Fragment der deutschen Grammatik aus Abbildung 3. Einbettende Phrasen, werden am Wurzelknoten S in die eingebettenden Phrasen adjungiert. Die Grammatik kann center–embedded Sätze der folgenden Art erzeugen:

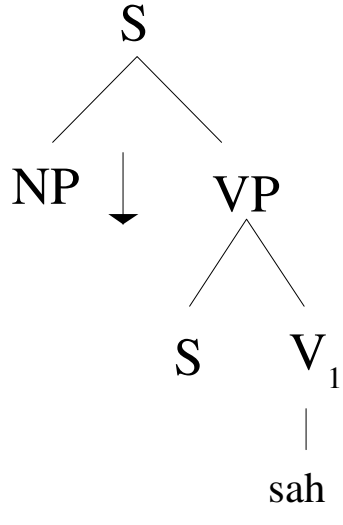
(18) ... daß Peter Maria die Kinder schwimmen lassen sah.



Initialbaum I



Hilfsbaum A



Hilfsbaum B

Abbildung 3: Fragment der deutschen Grammatik

Satz (18) wurde abgeleitet, indem zuerst die Nominalargumente in die »NP«–Knoten der Bäume I, A und B eingesetzt wurden. Dann wurde der Hilfsbaum A am Wurzelknoten von I in den Initialbaum I adjungiert und anschließend der Hilfsbaum B am Wurzelknoten des bereits abgeleiteten Baumes (also dem Wurzelknoten von Baum A) adjungiert. Abbildung 4 zeigt das Resultat.

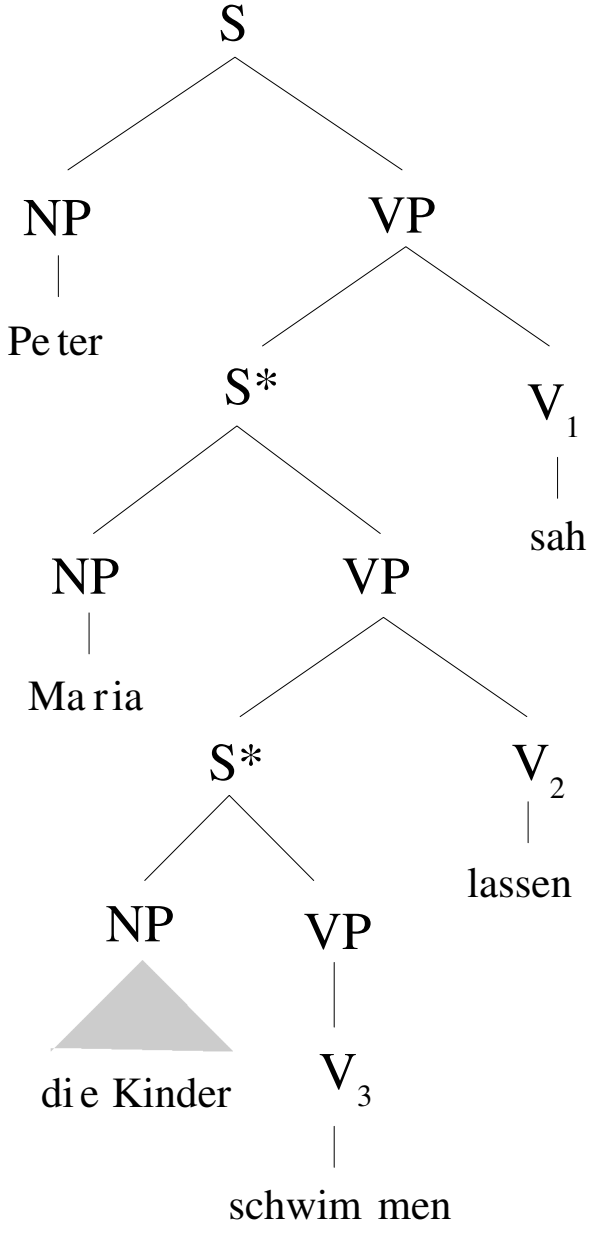


Abbildung 4: abgeleiteter Baum

Baum I ergibt folgende BEPDA–Regeln (19):

1. *Terminal Shift:* »die Kinder« und »schwimmen« kommen auf den Kellerspeicher
2. *Reduce:* »schwimmen« wird zu $V_I\{NP\}$
3. *Reduce:* [NP, $V_I\{NP\}$ wird zu $V_I\{\}$

Die dritte Regel besagt z.B., daß der zweitoberste Stack »[NP« entfernt wird und »V_I{VP}« auf dem obersten Stack durch »V_I{}«ersetzt wird. Wir nehmen weiterhin an, daß wir eine Syntax der NPs haben, die uns erlaubt, Instantiierungen zu erkennen (d.h. beispielsweise »die Kinder« durch »NP« zu ersetzen). Bisher haben wir die Adjunktion vernachlässigt, das müssen wir natürlich noch als weitere Regel berücksichtigen (20):

4. *Adjoin:* Baum A oder B kann am Wuzelknoten adjungiert werden.

Analog können wir auch den auxiliaren Baum B in Regeln erfassen (21):

1. *Terminal Shift:* »Peter« und »sah« kommen auf den Kellerspeicher
2. *Reduce:* »sah« wird zu V_B{NP}
3. *Reduce:* [NP, V_B{NP} wird zu V_B{}
4. *Adjoin:* Baum A oder B kann am Wurzelknoten adjungiert werden.
5. *Unadjoin:* am Wurzelknoten kann eine Adjunktion aufgelöst werden.

BEPDAs sind per Definition nichtdeterministische Automaten, was nicht sehr reizvoll für ein Modell menschlicher Sprachverarbeitung ist. Hauptsächlich kommt der Nichtdeterminismus bei der Adjunktion vor, wo der Automat erraten muß, daß ein bestimmter Baum adjungiert wurde, bevor irgendwelche Operationen in diesem Baum zur Anwendung kommen können. Wir nehmen deshalb an, daß der Automat die *Adjoin* –Regel nur dann verwendet, wenn er bereits weiß, daß er später eine *Reduce* –Regel in diesem Baum gebrauchen kann. Entsprechend kombinieren wir die beiden Bewegungen zu einer *Adjoin–Reduce* –Bewegung. In unserem Beispiel ist es möglich, mit einem Lookahead von einem Zeichen zu erkennen, ob das Verb mit der *Reduce* –Regel verarbeitet werden kann.

Bisher haben wir noch keine Reihenfolge dafür festgelegt, in welcher Ordnung Operationen angewendet werden sollen, die bei bestimmter Konfiguration von Eingabesymbol und Stack gleichzeitig ausgeführt werden können. Wir versuchen möglichst weit mit der Verarbeitung voranzuschreiten, bevor der ganze Satz gelesen wird, daher scheint folgende Reihenfolge sinnvoll (22):

1. Führe alle möglichen *reduce* , ε*–reduce* und *adjoin–reduce* zuerst aus.
2. Dann kommen alle *unadjoin* .
3. Wenn keine anderen Schritte möglich sind, dann mache einen *terminal–shift.*

Das Problem, das zwei Schritte derselben Kategorie zum gleichen Zeitpunkt ausgeführt werden können, soll uns hier nicht beschäftigen.

Unser Automat arbeitet demnach Satz (18) wie folgt ab:

Move	Rule	Store	Input Read	Input Scanned
0	0			
1	1	$[N_1$	N_1	N_1
2	1	$[N_1\ [N_2$	N_2	N_2
3	1	$[N_1\ [N_2\ [N_3$	N_3	N_3
4a	2	$[N_1\ [N_2\ [N_3\ [V_3\{N_3\}$	V_3	V_3
4b	3	$[N_1\ [N_2\ [V_3\{\}$	–	V_2
5a	2	$[N_1\ [N_2\ [V_3\{\}\ [V_2\{N_2\}$	V_2	V_2
5b	4	$[N_1\ [N_2\ [V_3\{\}\ V_2\{N_2\}$	–	V_1
5c	3	$[N_1\ [V_3\{\}\ V_2\{\}$	–	V_1
6a	2	$[N_1\ [V_3\{\}\ V_2\{\}\ [V_1\{N_1\}$	V_1	V_1
6b	4	$[N_1\ [V_3\{\}\ V_2\{\}\ V_1\{N_1\}$	–	–
6c	3	$[V_3\{\}\ V_2\{\}\ V_1\{\}$	–	–
6d	5	$[V_3\{\}\ V_2\{\}$	–	–
6e	5	$[V_3\{\}$	–	–
6f		[S		

Tabelle 3: Verarbeitungsprozeß zu (18)

Metriken für die Bestimmung der Verarbeitungslast

Um die Leistung eines Automaten zu messen, müssen wir eine Metrik mit dem Lauf verbinden. Joshi (1990) schlägt zwei Metriken vor. Eine einfache, die die maximale Anzahl von Elementen mißt, die während der Verarbeitung im Speicher stehen und eine komplexere, die zusätzlich die Verweildauer der Elemente im Speicher berücksichtigt. Wir verwenden den zweiten Ansatz in etwas abgewandelter Form. Die Grundidee ist die, daß bei jedem Verarbeitungsschritt gezählt wird, wieviele Elemente im Kellerspeicher liegen (jedes lexikalischeItem hat den Wert 1) und zum Schluß alle Einzelergebnisse zusammenaddiert werden. Je nach dem, wie man den Begriff Verarbeitungsschritt auffaßt, erhält man unterschiedliche Metriken. Eine Möglichkeit ist es, die primitiven Schritte zwischen zwei *terminal−shifts* zu einer Verarbeitungseinheit zusammenzufassen (in Tabelle 3 haben wir allen Schritten, die zu einer solchen Einheit gehören dieselbe Nummer gegeben, z.B. 3a – 3d). Die andere Möglichkeit ist es, jeden einzelnen Schritt unabhängig von seiner Komplexität als eigenständige Einheit anzusehen (3a – 3d sind demnach 4 Schritte). Wir bezeichnen die Metrik, die mit den zusammengesetzten Einheiten arbeitet, als »Metrik 1«, die andere als »Metrik 2«. Damit ist klar, daß die Ergebnisse, die mit Metrik 1 erzielt werden immer kleiner/gleich den Ergebnissen mit Metrik 2 sind.

Ein Beispielautomat für {}–BEPDA

Um alle Long–Distance Scramblings und Extrapositionen durchführen zu können, sind TAGs (bzw. BEPDAs) zu schwach; mit V–TAGs (bzw. {}–BEPDAs) ist es jedoch möglich. Damit man die Verarbeitungsleistung messen kann, die dazu benötigt wird, muß man also erstmal zum entsprechenden Automatenmodell greifen. (23) gibt einen Beispielsatz samt Phrasenstruktur wider.

(23) Weil [den Kühlschrank] $_i$ niemand [[t $_i$ zu reparieren] zu versuchen] verspricht.

Comp $_1$ N_3 N_1 V_3 V_2 V_1

Die NP »den Kühlschrank« wurde mit Long–Distance Scrambling aus der zutiefst eingebetteten *reparieren* –Phrase in die einbettende Phrase verschoben. Abbildung 5 liefert eine vollständige Grammatik, mit der diese Ableitung möglich ist. Die Pfeile zeigen dabei die Adjunktionen an.

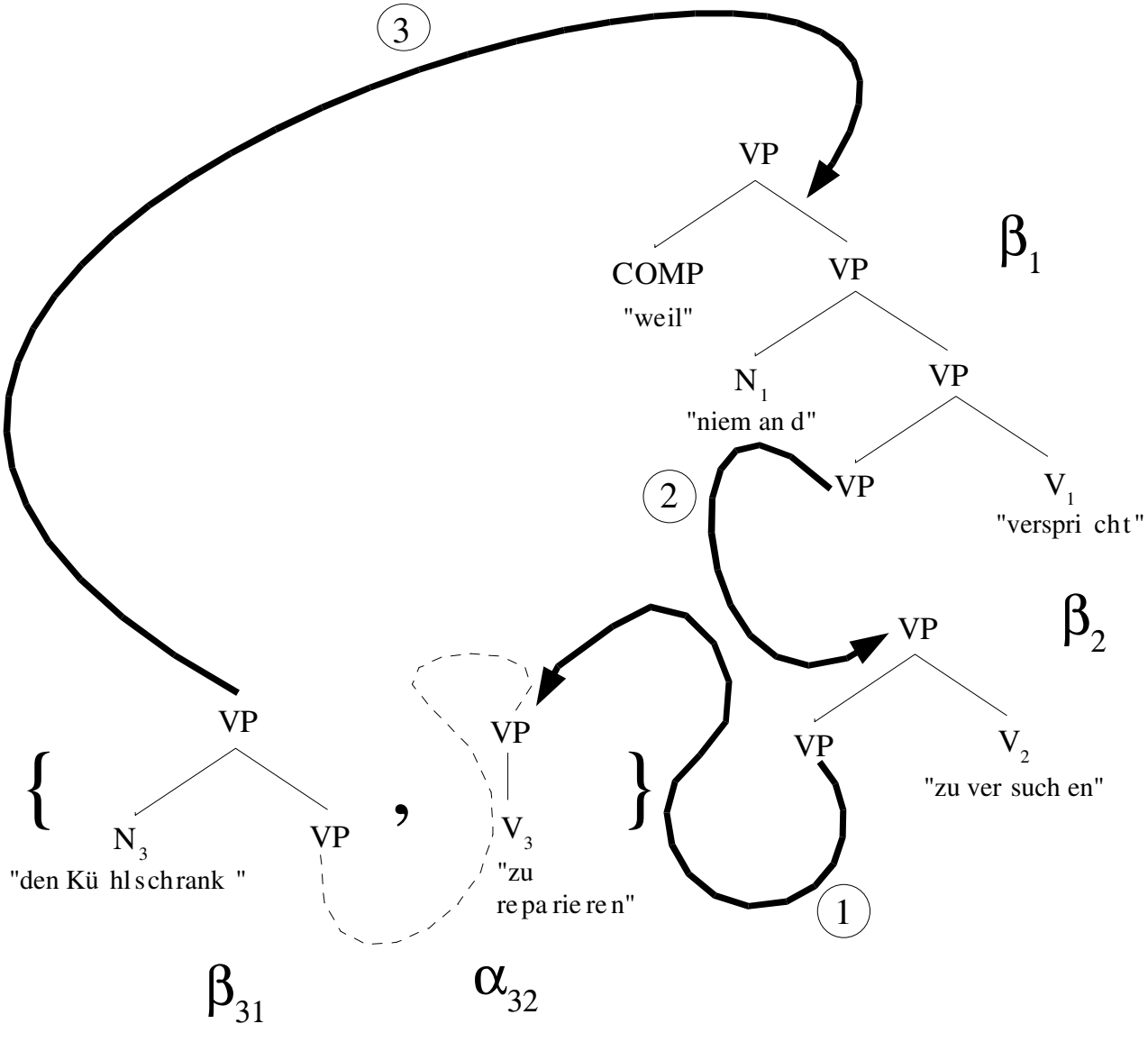


Abbildung 5: Beispielgrammatik mit Dominance Links

Der resultierende abgeleitete Baum ist in Abbildung 6 abgebildet.

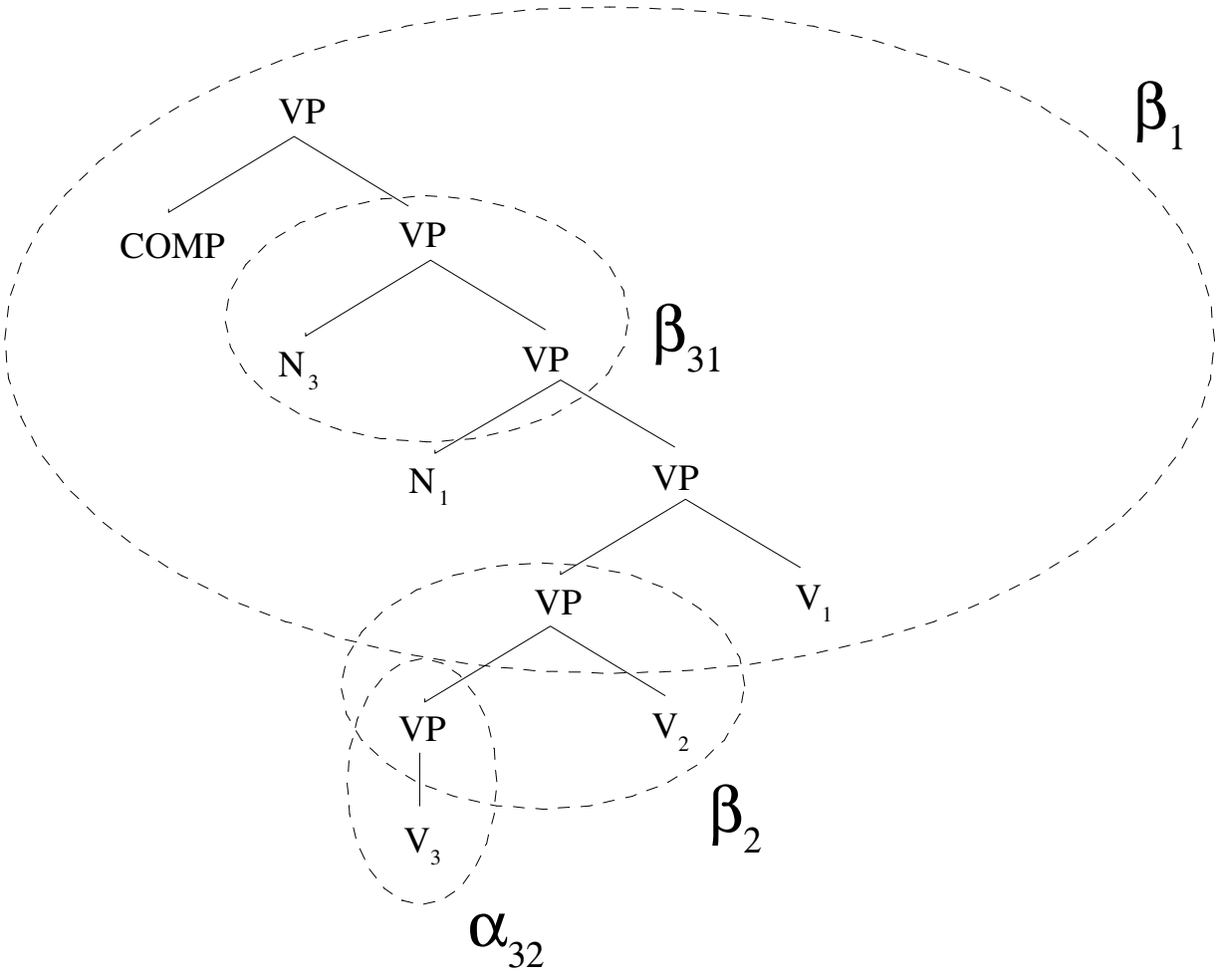


Abbildung 6: abgeleiteter Baum

Nun aber zu unserem Verarbeitungsmodell. Wir verwenden wie schon erwähnt einen {}−BEPDA. Dabei greifen wir auf die Variante zurück, die Indexmengen jedem einzelnen Stacksymbol und nicht nur den einzelnen Stacks zuordnet. Damit erhöhen wir die Ausdruckskraft, denn wir können so die Abtrennung von Nominalargumenten von ihren Verben simulieren. Damit die Verarbeitung von einer Variation mit Scrambling von der einer Variation ohne Scrambling bei Metrik 1 unterschieden werden kann, legen wir fest, daß ein neuer Schritt bezüglich Metrik 1 entweder mit *terminal−shift* oder mit *raise* eingeleitet wird. Letzterer kommt in der Verarbeitungsreihenfolge hinter allen Schrittypen außer *terminal−shift* .

Wir illustrieren die Funktionsweise des {}−BEPDA, indem wir zeigen, wie er mit Satz (23) umgeht. Dafür erstellen wir wiederum eine Schrittabelle, in der auch gleich die Metriken berechnet werden.

Step	Store	Metric 1 New Cum	Metric 2 New Cum
1a	[Comp $_1$	1 1	1 1
2a	[Comp $_1$ [N $_3$	2 3	2 3
3a	[Comp $_1$ [N $_3$ [N $_1$	3 6	3 6
4a	[Comp $_1$ [N $_3$ [N $_1$ [V $_3$ {N $_3$}	4 10	4 10
5a	[Comp $_1$ [N $_3$ [N $_1$ [V $_3$ {N $_3$} [V $_2$ {V $_3$}	5 15	5 15
5b	[Comp $_1$ [N $_3$ [N $_1$ [V $_3$ {N $_3$} V $_2$ {}	– 15	5 20
6a	[Comp $_1$ [N $_3$ [N $_1$ [V $_3$ {N $_3$} V $_2$ {} [V $_1$ {N $_1$ V $_2$}	6 21	6 26
6b	[Comp $_1$ [N $_3$ [N $_1$ [V $_3$ {N $_3$} V $_2$ {} V $_1$ {N $_1$}	– 21	6 32
6c	[Comp $_1$ [N $_3$ [V $_3$ {N $_3$} V $_2$ {} V $_1$ {}	– 21	6 38
6d	[Comp $_1$ [N $_3$ [V $_3$ {} V $_2$ {N $_3$} V $_1$ {}	6 27	6 44
6e	[Comp $_1$ [N $_3$ [V $_3$ {} V $_2$ {} V $_1$ {N $_3$}	6 33	6 50
6f	[Comp $_1$ [V $_3$ {} V $_2$ {} V $_1$ {}	– 33	6 56
6g	[V $_3$ {} V $_2$ {} V $_1$ {}	– 33	5 61
6h	[V $_3$ {} V $_2$ {}	– 33	3 64
6i	[V $_3$ {}	– 33	2 66
6j		– 33	0 66

Tabelle 4: Verarbeitungsprozeß zu (23)

Wenn wir diese Methode bei einigen anderen Sätzen aus Tabelle 1 anwenden, die nur Nominal–Scrambling und Extraposition gebrauchen, bekommen wir folgendes Ergebnis (24):

Nr	Satzstruktur	Metrik 1	Metrik 2	Bewertung
(xvi)	Comp $_1$ N $_1$ V $_1$ V $_2$ N $_3$ V $_3$	10	17	ok
(xiii)	Comp $_1$ N $_1$ V $_1$ N $_3$ V $_3$ V $_2$	12	24	ok
(i)	Comp $_1$ N $_1$ N $_3$ V $_3$ V $_2$ V $_1$	21	52	(ok)
(iv)	Comp $_1$ N $_1$ N $_3$ V $_2$ V $_3$ V $_1$	26	58	?
(ii)	Comp $_1$ N $_3$ N $_1$ V $_3$ V $_2$ V $_1$	33	66	?

Tabelle 5

Die Last bei Satz (xvi) fällt deshalb so gering aus, weil die Reihenfolge gerade so ist, daß die Phrasen frühzeitig vollständig sind und damit aus dem Speicher gelöscht

werden können. In (xiii) wurden die beiden am tiefsten eingebetteten Phrasen hinter die einbettende Phrase gestellt, jedoch so, daß die einbettende Phrase entfernt werden kann, bevor die Verarbeitung zu den eingebetteten Teilen kommt. Danach müssen jedoch alle Items auf dem Stack bleiben, bis der Satz komplett eingelesen ist. Satz (i), der vorgeschriebenermaßen akzeptierbar ist, ist die Variante mit Einbettung in der Mitte. Alle lexikalischen Items müssen im Speicher gehalten werden, bis der ganze Satz erfaßt ist. Dasselbe gilt für (iv) die Last ist aber noch größer, weil N_3 in die zweite Phrase gewandert ist (Long–Distance Scrambling). Schließlich hat (ii) die größte Last, weil hier Long–Distance Scrambling über zwei Phrasengrenzen hinweg stattgefunden hat. Die Rangfolge, die durch die Metriken bestimmt wird, entspricht der Reihenfolge, in der die Akzeptanz steigt. Zusätzlich entspricht die Rangordnung innerhalb der Bewertungsgruppen (»ok« und »?«) der Intuition der Muttersprachler. Allgemein gesprochen verbessert Extraposition einen Satz, während Long–Distance Scrambling ihn verschlechtert. Sämtliche Ergebnisse der Lauflastbestimmung sind in Tabelle 6 zusammengefaßt.

Bestimmung der Verarbeitungslast

Keine Extraposition nach 'verspricht'			
(i)	$Comp_1\ N_1\ N_3\ V_3\ V_2\ V_1$	ok	21 52
(ii)	$Comp_1\ N_3\ N_1\ V_3\ V_2\ V_1$	?	33 66
(iii)	$Comp_1\ N_1\ V_2\ N_3\ V_3\ V_1$	?	21 53
(iv)	$Comp_1\ N_1\ N_3\ V_2\ V_3\ V_1$	?	26 58
(v)	$Comp_1\ N_3\ N_1\ V_2\ V_3\ V_1$	?	33 66
(vi)	$Comp_1\ N_3\ V_3\ N_1\ V_2\ V_1$	ok	27 56
(vii)	$Comp_1\ N_3\ V_3\ V_2\ N_1\ V_1$	?	21 50
(viii)	$Comp_1\ V_2\ N_3\ V_3\ N_1\ V_1$	?	21 51
(ix)	$Comp_1\ V_2\ N_3\ N_1\ V_3\ V_1$	*	–
(x)	$Comp_1\ N_3\ V_2\ N_1\ V_3\ V_1$	*	–
(xi)	$Comp_1\ V_2\ N_1\ N_3\ V_3\ V_1$	*	–
(xii)	$Comp_1\ N_3\ V_2\ V_3\ N_1\ V_1$	?	25 55

Extraposition der 'versuchen'–Phrase			
(xiii)	$Comp_1\ N_1\ V_1\ N_3\ V_3\ V_2$	ok	12 24
(xiv)	$Comp_1\ N_1\ N_3\ V_1\ V_3\ V_2$	?	33 67
(xv)	$Comp_1\ N_3\ N_1\ V_1\ V_3\ V_2$	?	33 65

Extraposition der ’versuchen’–Phrase			
(xvi)	$Comp_1\ N_1\ V_1\ V_2\ N_3\ V_3$	ok	10 17
(xvii)	$Comp_1\ N_1\ V_1\ N_3\ V_2\ V_3$	?	15 28
(xviii)	$Comp_1\ N_1\ N_3\ V_1\ V_2\ V_3$	ok	39 64
(xix)	$Comp_1\ N_3\ N_1\ V_1\ V_2\ V_3$	?	39 62
(xx)	$Comp_1\ N_1\ N_3\ V_3\ V_1\ V_2$	?	27 57
(xxi)	$Comp_1\ N_3\ V_3\ N_1\ V_1\ V_2$	?	27 55
(xxii)	$Comp_1\ N_3\ N_1\ V_3\ V_1\ V_2$	?	39 65

Ausschließliche Extraposition der ’reparieren’–Phrase			
(xxiii)	$Comp_1\ N_1\ V_2\ V_1\ N_3\ V_3$	?	24 48
(xxiv)	$Comp_1\ N_1\ V_2\ N_3\ V_1\ V_3$	*	–
(xxv)	$Comp_1\ N_1\ N_3\ V_2\ V_1\ V_3$	*?	39 64
(xxvi)	$Comp_1\ N_3\ N_1\ V_2\ V_1\ V_3$	*?	39 63
(xxvii)	$Comp_1\ V_2\ N_1\ N_3\ V_1\ V_3$	*	–
(xxviii)	$Comp_1\ V_2\ N_1\ V_1\ N_3\ V_3$	*?	24 48
(xxix)	$Comp_1\ V_2\ N_3\ N_1\ V_1\ V_3$	*	–
(xxx)	$Comp_1\ N_3\ V_2\ N_1\ V_1\ V_3$	*?	39 63

Tabelle 6

In der Tabelle stehen alle Sätze, mit denen wir uns bereits zu Beginn beschäftigt haben in Phrasenstrukturdarstellung, mit Bewertung und Angabe der Verarbeitungslast unter den beiden vorgestellten Metriken. Grob eingeteilt würde man bei Metrik 2 Sätze mit Werten unter 40 akzeptabel (»ok«), mit Werten zwischen 41 und 60 als fragwürdig (»?«) und solche mit Werten darüber als besonders fragwürdig (»*?«) bezeichnen. Die Sätze mit »*« können nicht verarbeitet werden und erhalten dementsprechend auch keine metrische Wertzuteilung.

Es gibt jedoch einige Fälle, die aus dem Rahmen fallen, z.B. wird Satz (xxviii) schlechter bewertet, als es das Modell vorhersagt. Störender sind die Sätze bei denen es andersherum ist, nämlich diejenigen, bei denen die Vorhersage schlechter ausfällt als die Bewertung durch den Native Speaker. Ein generierender Sprachautomat würde dabei vermutlich gebräuchliche Formen unterdrücken und vielleicht weniger gute Sätze präsentieren. Es muß also noch gebastelt werden. [13]

13 Übersetzung von »Further research is required« und abschließende Beurteilung sowohl des Autors dieser Ausarbeitung als auch des Autors des zugrundeliegenden Papiers.

Abkürzungsverzeichnis

{}−BEPDA: Multiset−Indexed Bottom−Up Embedded Push−Down Automaton
BEPDA: Buttom−Up Embedded Push−Down Automaton
EPDA: Embedded Push−Down Automaton
PDA: Push−Down Automaton (Kellerautomat)
MF: Mittelfeld
NA: Null−Adjoining
NP: Nominalphrase
OA: Obligatory−Adjoining
SA: Selective−Adjoining
SV: String−Vacuously (ohne Textänderung)
TAG: Tree−Adjoining−Grammar (Baumadjunktionsgrammatik)
VP: Verbalphrase
VF: Vorfeld

Literaturverzeichnis

[1]: Automatenmodell (EPDA):
K. Vijay−Shanker, 1987,
»A Study of Tree Adjoining Grammars« (Dissertation)
[2]: Owen Rambow, 1994,
»Formal and Computational Models for Natural Language Syntax«
[3]: Zusätzliche Literatur:
Aravind K. Joshi, 1990,
»Processing Crossed and Nested Dependencies: an Automaton Perspektive on the Psycholinguistic Results«