Seminar
Effizientes Parsing und Generieren mit Baumadjunktionsgrammatiken (TAG)
CYK-basiertes Parsing

Sven Thiel

16. Dezember 1996

Zusammenfassung

In diesem Artikel befasse ich mich mit zwei Parsing-Algorithmen für Baumadjunktionsgrammatiken:

- Der Algorithmus von K. Vijay-Shanker
- Der Algorithmus von K. Harbusch

Beide Algorithmen haben gemeinsam, daß sie eine enge Beziehung zum Cocke-Younger-Kasami Algorithmus (s. [2]) zum Parsen von kontextfreien Grammatiken in Chomsky-Normalform haben.

In der Darstellung liegt mein Hauptaugenmerk auf dem ersten Algorithmus. Ich werde ausführlich die Arbeitsweise des Algorithmus besprechen sowie Aussagen über die Korrektheit und die Komplexität beweisen. Zentrales Ergebnis wird sein, daß das Wortproblem für Baumadjunktionsgrammatiken in Zeit $O(n^6)$ entscheidbar ist.

Bei der Besprechung des zweiten Algorithmus werde ich mich darauf beschränken, die wesentlichen Ideen zu erläutern.

1 Einleitende Definitionen und Bezeichnungen

In diesem Abschnitt stelle ich einige Definitionen und Bezeichnungen vor, die für die weiteren Ausführungen wichtig sind und teilweise über die im Zusammenhang mit Baumadjunktionsgrammatiken gebräuchlichen Begriffe hinausgehen. Wie üblich werde ich statt „Baumadjuntionsgrammatik" häufig die Abkürzung „TAG" des englischen Ausdruckes „Tree Adjoining Grammar" verwenden.

Die grundlegenden Definitionen wie TAG, TAG mit Constraints, usw. setze ich im folgenden voraus. Eine ausführliche Einführung in das Thema Baumadjunktionsgrammatiken findet man in [3]; die grundlegenden Definitionen werden in [5] und [1] behandelt.

Definition 1.1 (Knotenadresse) *Sei γ ein Baum. Dann sind die Adressen der Knoten von γ wie folgt rekursiv definiert:*

- *Die Wurzel von γ hat die Adresse ϵ.*
- *Sei n ein Knoten von γ, dessen Adresse bereits definiert sei, dann erhält der linke bzw. einzige Nachfolger die Adresse von n konkateniert mit „1" und der rechte Nachfolger, falls er existiert, die Adresse von n konkateniert mit „2".*

Definition 1.2 (Knotennummer) *Die elementaren Bäume einer Baumadjunktionsgrammatik seien durchnumeriert. Für einen Knoten n eines elementaren Baumes sei die Knotennummer von n dann die folgende Konkatenation:* $(Baumnummer\ von\ n) \cdot / \cdot (Knotenadresse\ von\ n)$.

Ein erläuterndes Beispiel für diese Definition ist in Abbildung 1 zu sehen.

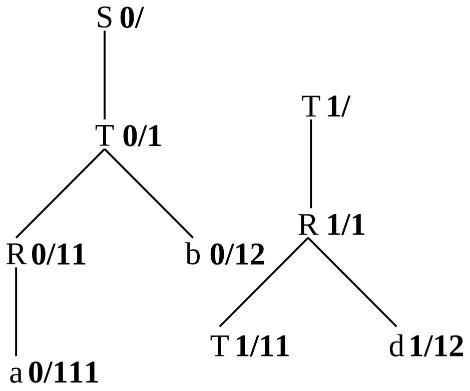

Abbildung 1: Knotennummern

Da mit dieser Definition eine Bijektion zwischen Knoten und Knotennummern gegeben ist, kann man die Knoten von elementaren Bäumen mit ihren Knotennummern identifizieren. Deshalb werde ich in Zukunft oft „der Knoten μ“ statt „der Knoten, der die Nummer μ trägt“ schreiben.

Definition 1.3 *Sei G eine Baumadjunktionsgrammatik. Sei β ein elementarer Baum von G und μ die Nummer eines Knotens in β. Dann sei die Menge $\mathcal{D}(\mu)$ wie folgt rekursiv definiert:*

- *Der von μ dominierte Unterbaum in β (bezeichnet mit β/μ) ist in $\mathcal{D}(\mu)$.*
- *Ist γ in $\mathcal{D}(\mu)$, dann sei auch jeder von γ mittels Adjunktion abgeleitete Baum in $\mathcal{D}(\mu)$.*

Abbildung 2 verdeutlicht die Bezeichnung β/μ.

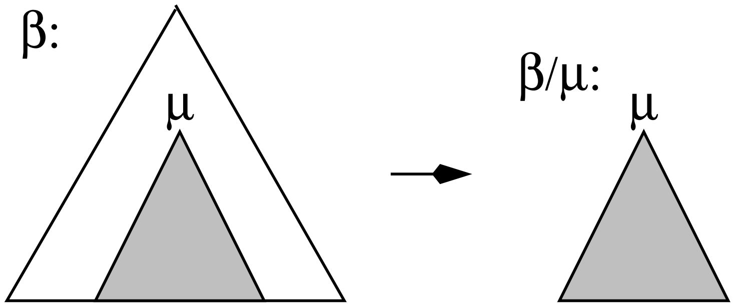

Abbildung 2: der Baum β/μ

Definition 1.4 (Zweiform) *Eine Baumadjunktionsgrammatik ist in Zweiform, wenn für jeden elementaren Baum gilt: Jeder Knoten dieses Baumes hat höchstens zwei Söhne.*

2 Der Parsing-Algorithmus von K. Vijay-Shanker

Das Original des im folgenden vorgestellten Algorithmus stammt von K. Vijay-Shanker (s. [4] od. [5]). Der dort beschriebene Algorithmus enthält jedoch mehere kleine Fehler. Ich werde hier eine Version vorstellen, in der alle Fehler, die ich finden konnte, korrigiert sind und die nun hoffentlich fehlerfrei ist.

Den Algorithmus kann man als eine Weiterentwicklung des bekannten CYK-Parsing-Algorithmus für kontextfreie Sprachen in Chomsky-Normalform ansehen (s. [2]).

Ähnlich wie dieser Algorithmus setzt der Algorithmus von Vijay-Shanker voraus, daß die zu parsende TAG gewissen Bedingungen genügt:

- Die Grammatik ist in Zweiform. (s. Def. 1.4)
- Jeder Baum produziert mindestens ein Terminalsymbol.
- Es sind keine Knoten mit der Beschriftung ϵ erlaubt.

Im CYK-Algorithmus wurde ein 2-dimensionales Feld mit Nichtterminalen als Einträgen benutzt. Ein Nichtterminal in einem bestimmten Feldelement bedeutete, daß man mit diesem Nichtterminal einen bestimmten Teilstring des Einagbewortes ableiten konnte.

Bei TAGs reicht ein 2-dimensionales Feld nicht aus, denn i.a. muß man 2 Teilstrings der Eingabe betrachten: Sieht man sich das Blattwort eines auxiliaren Baumes an, etwa $a_1 \cdots a_k \; X \; b_1 \cdots b_l$, so stellt man fest, daß es aus zwei Strings aus Terminalsymbolen besteht, die durch die Beschriftung des Fußknotens (X) getrennt werden. Es ist klar, daß diese beiden Strings im Laufe von Adjunktionen beliebig weit auseinander wandern können. Deshalb benutzt man zum Parsen von TAGs ein 4-dimensionales Feld; die ersten beiden Indizes beschreiben den ersten Teilstring, die letzten beiden beschreiben den zweiten Teilstring der Eingabe, der abgeleitet wird.

Legt man dem Algorithmus eine Eingabe $a_1 \cdots a_n$ vor, so baut er ein 4-dimensionales Feld $A[0..n, 0..n, 0..n, 0..n]$ auf, dessen Einträge die Nummern von Knoten von elementaren Bäumen sind (s. Def. 1.2).

Sei μ die Nummer eines Knotens in einem elementaren Baum, dann nimmt der Algorithmus μ genau dann in das Feldelement $A[i, j, k, l]$ ($i \leq j \leq k \leq l$) auf, wenn es einen Baum γ in $\mathcal{D}(\mu)$ (s. Def. 1.3) gibt, der gerade die beiden Teilstrings $a_{i+1} \cdots a_j$ und $a_{k+1} \cdots a_l$ ableitet.

Hierbei muß man zwei grundsätzliche Fälle unterscheiden:

1. μ gehört zu einem auxiliaren Baum und ist Vorgänger des Fußknotens. Dann umfaßt das Blattwort jedes Baumes aus $\mathcal{D}(\mu)$ zwei Strings aus Terminalsymbolen, die nicht zusammenhängen.

2. μ gehört zu einem initialen Baum, oder μ gehört zwar zu einem auxiliaren Baum, ist aber nicht Vorgänger des Fußknotens. Dann ist das Blattwort jedes Baumes aus $\mathcal{D}(\mu)$ ein zusammenhängender String von Terminalsymbolen.

Trifft auf μ der erste Fall zu, so wird der Algorithmus μ nur in Feldelemente aufnehmen, bei denen der zweite Index echt kleiner ist als der dritte. Trifft auf μ dagegen der zweite Fall zu, so wird der Algorithmus dies dadurch kennzeichnen, daß er den Knoten nur in Feldelemente aufnimmt, bei denen der zweite und der dritte Index gleich sind.

Diesen Zusammenhang verdeutlicht Abbildung 3.

Damit kann man das Feld, das der Algorithmus aufbauen soll, vollständig charakterisieren:

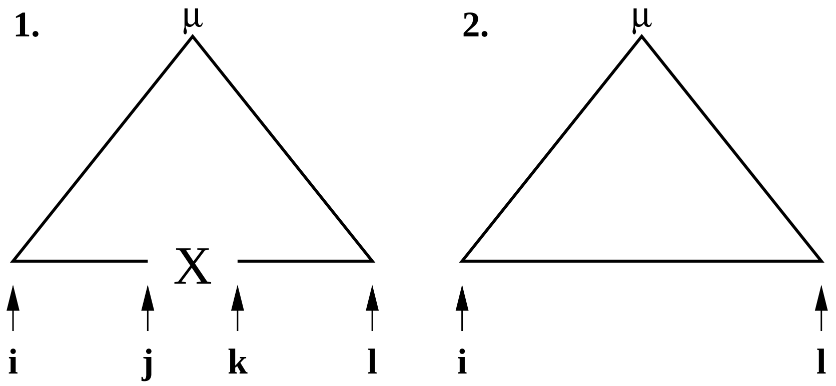

Abbildung 3:

Sei μ die Nummer eines Knoten in einem Elementarbaum, dessen Wurzel die Beschriftung X trägt:

1. Für i, j, k, l mit $0 \leq i \leq j < k \leq l \leq n$ gilt:
   $\mu \in A[i, j, k, l] \Longleftrightarrow \exists \gamma \in \mathcal{D}(\mu)$ mit Blattwort $a_{i+1} \cdots a_j\ X\ a_{k+1} \cdots a_l$
2. Für i, l mit $0 \leq i < l \leq n$ gilt:
   $\exists\, i \leq j \leq l : \mu \in A[i, j, j, l] \Longleftrightarrow \exists \gamma \in \mathcal{D}(\mu)$ mit Blattwort $a_{i+1} \cdots a_l$

Hierzu möchte ich noch ein paar Bemerkungen machen:

- In Punkt 2 wird nicht gefordert, daß μ in alle Feldelemente der Form $A[i, j, j, l]$ aufgenommen wird, sondern nur in mindestens eines davon.
- Gilt in Punkt 1 $i = j$ bzw. $k = l$, so ist $a_{i+1} \cdots a_j$ bzw. $a_{k+1} \cdots a_l$ das leere Wort ϵ.
- Man sollte sich die Feldindizes nicht als Zeiger auf Zeichen in der Eingabe vorstellen, sondern als Zeiger auf die Zwischenräume zwischen den Zeichen der Eingabe. Hat z.B. der Index i den Wert 5, so zeigt i auf den Zwischenraum zwischen a_5 und a_6. Das ist auch der Grund dafür, daß die Indizes von 0 bis n laufen.

2.1 Die Grundideen des Algorithmus

Analog zum CYK-Algorithmus besteht die Idee darin, eine gewisse Anzahl von Feldelementen zu initialisieren, so daß sie den Anforderungen von oben genügen, und dann aus bereits (korrekt) berechneten Feldelementen neue Elemente zu berechnen. Bei der Berechnung des neuen Feldelementes $A[i, j, k, l]$ können 5 Fälle auftreten:

- Case 1:
  Man weiß, daß der Knoten μ in $A[i, j, k, m]$ aus einem auxiliaren Baum stammt und Vorgänger des Fußknotens ist (d.h. $j < k$) und daß sein rechter Bruder ν in $A[m, p, p, l]$ ist. Dann sollte man ihren Vater λ in $A[i, j, k, l]$ aufnehmen. (s. Abb. 4)
- Case 2:
  Man weiß, daß der Knoten ν in $A[m, j, k, l]$ aus einem auxiliaren Baum stammt und Vorgänger des Fußknotens ist (d.h. $j < k$) und daß sein linker Bruder μ in $A[i, p, p, m]$ ist. Dann sollte man ihren Vater λ in $A[i, j, k, l]$ aufnehmen. (s. Abb. 5) Dieser Fall ist also symmetrisch zu Case 1.

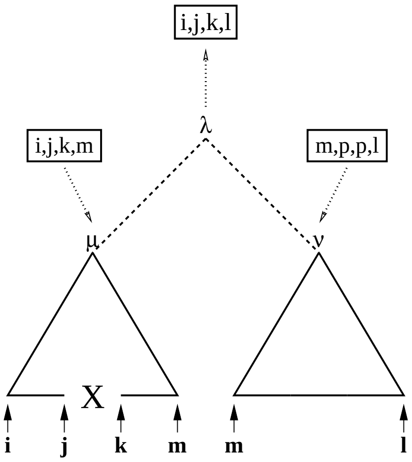

Abbildung 4: Case 1

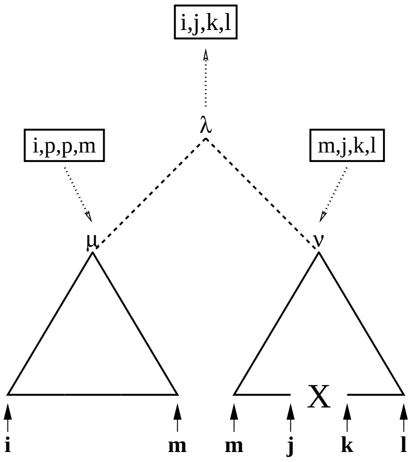

Abbildung 5: Case 2

- Case 3:
  Nun ist weder der Knoten μ in $A[i,j,j,m]$ noch sein rechter Bruder ν in $A[m,p,p,l]$ Vorgänger des Fußknotens, d.h. jeder leitet einen zusammenhängenden Teilstring der Eingabe ab und die beiden Teilstrings folgen unmittelbar aufeinander. Dann sollte man ihren Vater λ in $A[i,j,j,l]$ aufnehmen. (s. Abb. 6)

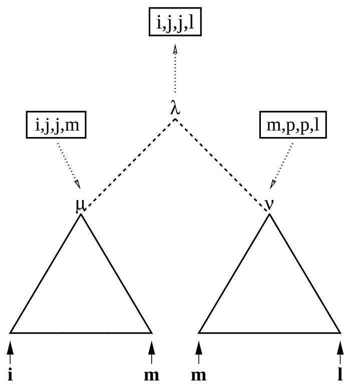

Abbildung 6: Case 3

- Case 4:
  In diesem Fall ist μ ein Knoten in $A[i,j,k,l]$, der keinen Bruder, aber einen Vater λ hat. Dann sollte man den Vater λ auch in $A[i,j,k,l]$ aufnehmen. Diesen Vorgang sollte man solange wiederholen, bis man keine neuen Knoten mehr erhält. (s. Abb. 7)

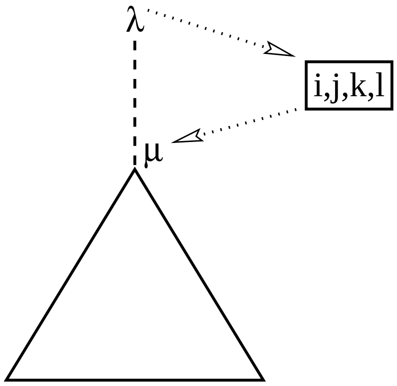

Abbildung 7: Case 4

- Case 5 (Adjunktion):
  In den bisherigen Fällen hat man immer nur Knoten aus einem einzigen elementaren Baum betrachtet. Mit Hilfe der Adjunktionsoperation auf den elemtaren Bäumen kann man bekanntlich verschiedene elementare Bäume kombinieren. Dem trägt dieser Fall Rechnung. Ist λ ein Knoten in $A[m,j,k,p]$, der die Beschriftung X trägt, und ist μ in $A[i,m,p,l]$ die Wurzel eines elementaren Baumes, die ebenfalls die Beschriftung X trägt, so sollte man λ in $A[i,j,k,l]$ aufnehmen. (s. Abb. 8)

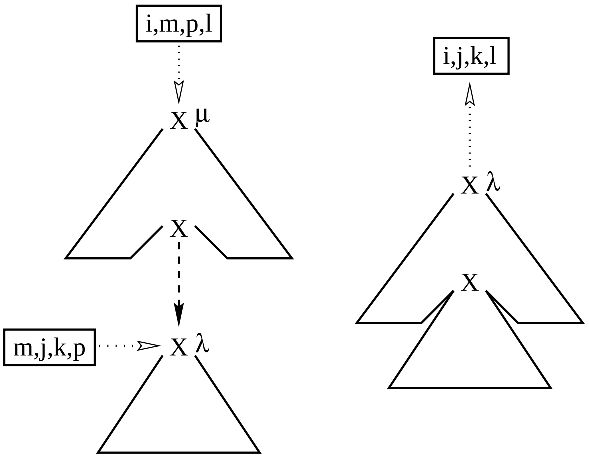

Abbildung 8: Case 5

Wie man sieht, greift man in Case 4 auf das aktuelle Feldelement zurück. Es wird sich herausstellen, daß dies auch in Case 5 nötig sein kann. Darauf werde ich in Abschnitt 2.6 genauer eingehen, wenn ich die Korrektheit des Algorithmus bespreche.

2.2 Die Initialisierung des Algorithmus

Wie in 2.1 erwähnt muß der Algorithmus gewisse Feldelemente initialisieren, so daß sie den auf S. 4 aufgestellten Bedingungen genügen.

Wenn man sich Punkt 1 der Bedingungen ansieht, so stellt man fest, daß man in ein Feldelement $A[i,i,j,j]$ (mit $0 \leq i < j \leq n$) alle Fußknotens von auxiliaren Bäumen schreiben sollte.

Punkt 2 der Bedingungen ergibt, daß man in das Feldelement A[i,i+1,i+1,i+1] (mit $0 \leq i < n$) genau die Blattknoten von elementaren Bäumen schreiben sollte, die mit dem Terminalsymbol a_{i+1} beschriftet sind, wobei $a_1 \cdots a_n$ die Eingabe sei. (Natürlich könnte man die Knoten genauso gut nach $A[i,i,i,i+1]$ schreiben.)

Eigentlich müßte man auch noch die Vorfahren der jeweiligen Knoten aufnehmen, wenn diese keine Brüder haben, d.h. man müßte Case 4 auf die von der Initialisierung betroffenen Feldelemente anwenden. Aber das wird sich in 2.6 als unnötig herausstellen.

Deshalb sieht die Initialisierung des Algorithmus folgendermaßen aus:

```
(1) for i=0 to n-1 step 1 do
(2)     Schreibe alle Blattknoten, die mit dem Terminalsymbol a_{i+1}
        beschriftet sind, nach A[i,i+1,i+1,i+1]
(3) for i=0 to n-1 step 1 do
(4)     for j=i+1 to n step 1 do
(5)         Schreibe alle Fußknoten von aux. Bäumen nach A[i,i,j,j]
```

2.3 Der Hauptteil des Algorithmus

Der Hauptteil des Algorithmus sieht wie folgt aus:

```
(6) for l=0 to n step 1 do
(7)     for i=l to 0 step -1 do
```

```
(8)         for j=i to l step 1 do
(9)             for k=l to j step -1 do
(10)                do Case 1
(11)                do Case 2
(12)                do Case 3
(13)                do Case 4; do Case 5
(14)                do Case 4
(15) Akzeptiere, falls die Wurzel eines intialen Baumes
     in A[0,j,j,n] ist, wobei 0 ≤ j ≤ n.
```

Wie man sieht, besteht der Hauptteil des Algorithmus darin, alle Feldelemente $A[i,j,k,l]$ (mit $0 \leq i \leq j \leq k \leq n$) systematisch durchzugehen und mit Hilfe der Prozeduren Case 1, ..., Case 5 zu berechnen. (Schritte (6) bis (14))

(Die Anordnung der „for"-Schleifen ist wesentlich für die Korrektheit des Algorithmus. (vgl. 2.6))

Schaut man sich nochmal Punkt 2 der Bedingungen von S. 4 an, so stellt man fest, daß die Vereinigung der Feldelemente $A[0,j,j,n]$ $(j = 0,\ldots,n)$ genau die Knoten umfaßt die den vollständigen Eingabestring ableiten können. Dies erklärt Schritt (15).

2.4 Genaue Diskussion von Case 1 bis Case 5

In Abschnitt 2.1 habe ich Case 1 bis Case 5 bereits vorgestellt, damit man sich ein Bild von der Arbeitsweise des Algorithmus machen kann. Ich habe eine Reihe von Feldindizes benutzt, ohne jedoch genau zu sagen, in welchen Schranken sich die Indizes bewegen müssen. Dies will ich jetzt nachholen. An die Diskussion der Schranken schließt sich eine Implementierung der jeweiligen Prozedur in Pseudocode an.

- Case 1:
  Aus Abbildung 4 ergibt sich, daß das Blattwort des rechten Teilbaumes mindestens ein Terminalsymbol umfassen muß, da das Blattwort ein zusammenhängender Teilstring des Eingabestrings ist und ϵ-Knoten nicht erlaubt sind. Das Blattwort des linken Teilbaumes muß dagegen kein Terminalsymbol umfassen, es kann durchaus einfach nur aus der Beschriftung (X) des Fußknotens in diesem Baum bestehen.

  Daraus ergibt sich folgende Ungleichungskette:

  $$i \leq j < k \leq m < l \quad \text{und} \quad m \leq p \leq l$$

  Pseudocode:

  ```
  (C1.1) if j=k then exit;
  (C1.2) for m=k to l-1 step 1 do
  (C1.3)     for p=m to l step 1 do
  (C1.4)         Falls der linke Sohn μ ∈ A[i,j,k,m]
                 und der rechte Sohn ν ∈ A[m,p,p,l] ist,
                 schreibe den Vater λ nach A[i,j,k,l].
  ```

- Case 2:
  Da Case 2 symmetrisch zu Case 1 ist (s. Abb. 5), beschränke ich mich darauf nur die Ungleichungskette und den Pseudocode anzugeben:

  $$i < m \leq j < k \leq l \quad \text{und} \quad i \leq p \leq m$$

Pseudocode:

```
(C2.1) if j=k then exit;
(C2.2) for m=j to i+1 step -1 do
(C2.3)     for p=m to i step -1 do
(C2.4)         Falls der linke Sohn μ ∈ A[i,p,p,m]
               und der rechte Sohn ν ∈ A[m,j,k,l] ist,
               schreibe den Vater λ nach A[i,j,k,l].
```

- Case 3:
  Wenn man sich Abbildung 6 ansieht, so stellt man fest, daß hier sowohl der linke als auch der rechte Teilbaum mindestens ein Terminalsymbol produzieren müssen. (Die Begründung hierfür ist analog wie in Case 1.)

  Daraus ergibt sich folgende Ungleichungskette:

  $$i < m < l \quad \text{und} \quad i \leq j \leq m \leq p \leq l$$

  Pseudocode:

```
(C3.1) if j<>k then exit;
(C3.2) for m=max(j,i+1) to l-1 step 1 do
(C3.3)     for p=m to l step 1 do
(C3.4)         Falls der linke Sohn μ ∈ A[i,j,j,m]
               und der rechte Sohn ν ∈ A[m,p,p,l] ist,
               schreibe den Vater λ nach A[i,j,k,l].
```

  (Der Schritt (C3.1) dient nur der Laufzeitoptimierung, man könnte ihn aber auch weglassen.)

- Case 4:
  Case 4 arbeitet nur auf dem aktuellen Feldelement, so daß die Angabe einer Ungleichungskette entfällt.

  Pseudocode:

```
(C4.1) repeat
(C4.2)     Hat der Knoten μ in A[i,j,k,l] keinen Bruder,
           aber einen Vater λ, dann schreibe λ nach A[i,j,k,l].
(C4.3) until Es wurden keine neuen Knoten hinzugefügt.
```

- Case 5 (Adjunktion):
  Der Baum, der adjungiert wird (s. Abb. 8), muß mindestens ein Terminalsymbol in seinem Blattwort haben; denn da μ die Wurzel eines auxiliaren Baumes ist, handelt es sich hier um einen vollständigen Baum (oder einen davon abgeleiteten Baum) und dieser muß mindestens ein Terminal produzieren. Dagegen muß der von λ dominierte Teilbaum nicht notwendigerweise ein Terminalsymbol in seinem Blattwort haben. Dies ist z.B. dann der Fall, wenn λ ein Fußknoten eines auxiliaren Baumes ist, oder wenn λ Vater eines Fußknotens ist, der keinen Bruder hat.

  Insgesamt ergibt sich folgende Ungleichungskette:

  $$i \leq m \leq j \leq k \leq p \leq l \quad \text{und} \quad (i < m \vee p < l)$$

Pseudocode:

```
(C5.1) for m=i to j step 1 do
(C5.2)     for p=k to l step 1 do
(C5.3)         Ist (i < m ∨ p < l) und μ ∈ A[i,m,p,l] die Wurzel
               eines aux. Baumes, die dieselbe Beschriftung wie
               λ ∈ A[m,j,k,p] trägt, dann schreibe λ nach A[i,j,k,l].
```

2.5 Zusammenspiel Case 4 – Case 5 – Case 4

Nach dieser Diskussion der Fälle möchte ich das Zusammenspiel der Prozeduren Case 4 und Case 5 an einem Beispiel erläutern. Dieses Beispiel wird gleichzeitig ein Beweis dafür sein, daß es nicht ausreicht, im Hauptteil des Algorithmus Case 4 nur einmal aufzurufen (s. Schritte (13) und (14) in 2.3). Das Beispiel wird durch Abbildung 9 verdeutlicht.

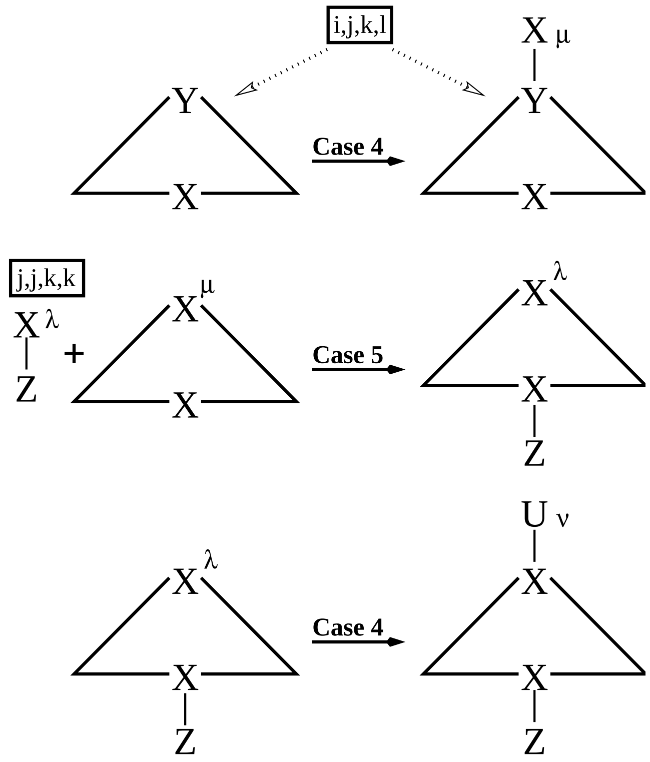

Abbildung 9: Zusammenspiel von Case 4 und Case 5

Ich betrachte eine TAG G, in der es einen auxiliaren Baum β gebe, dessen Fußknoten die Beschriftung X trage. In diesem Baum existiere ein Knoten mit der Beschriftung Y, der Vorgänger des Fußknotens sei. Dieser Knoten sei in dem aktuellen Feldelement $A[i,j,k,l]$ enthalten. Er könnte z.B. durch eine Anwendung Case 1 gerade aufgenommen worden sein. Der Knoten habe keinen Bruder, aber einen Vater, nämlich die Wurzel μ von β. Durch die erste Anwendung von Case 4 in Schritt (13) wird dann μ auch in $A[i,j,k,l]$ aufgenommen und steht dort für eventuelle Adjunktionen bereit.

Ich nehme weiter an, das in G ein Baum γ existiert, dessen Fußknoten die Beschriftung Z trägt. Dieser Knoten habe keinen Bruder, aber einen Vater λ, der mit X beschriftet sei. Dann ist dieser Knoten durch frühere Anwendungen von Case 4 in das Feldelement $A[j,j,k,k]$ aufgenommen worden. Nun kann in Case 5 (Schritt (13)) der Baum β in λ adjungiert werden, d.h. λ wird in $A[i,j,k,l]$ aufgenommen.

Geht man nun davon aus, daß λ in γ auch keinen Bruder, aber einen Vater mit der Beschriftung U hat, so kann dieser Knoten mit einer nochmaligen Anwendung von Case 4 (Schritt (14)) nach $A[i,j,k,l]$ geschrieben werden.

Rekapituliert man das Beispiel, so stellt man fest, daß die erste Anwendung von Case 4 eine Art „Adjunktionsvorbereitung" ist. Diese ist – wie sich in Abschnitt 2.6 herausstellen wird – nur nötig, wenn der Knoten, an dem die Adjunktion erfolgt, einen Unterbaum dominiert, der kein Terminalsymbol dominiert.

2.6 Die Korrektheit des Algorithmus

Ich komme nun auf die Bemerkung aus Abschnitt 2.2 zurück und werde beweisen, daß die Initialisierung des Algorithmus korrekt ist. Dies folgt daraus, daß auf die von der Initialisierung betroffenen Elemente Case 4 angewendet wird, bevor irgendeine andere Prozedur angewendet wird. Für Case 5 ist dies offensichtlich (vgl. Schritt (13) in 2.3). Für die Prozeduren Case 1, Case 2, Case 3 stellt man fest (s. 2.4), daß bei allen 3 Prozeduren für die Feldindizes $i < m < l$ gilt. Bei allen Feldelementen, auf die diese Prozeduren zurückgreifen, tritt der Index m entweder als erster oder als letzter Index auf. Sieht man sich die Anordnung der „for"-Schleifen im Hauptteil des Algorithmus an (speziell Schritt (6) und (7)), so bemerkt man, daß die Feldelemente, auf die die Prozeduren zurückgreifen, schon bearbeitet worden sind, insbesondere sind sie schon einer Anwendung von Case 4 unterzogen worden.

Dies liefert die Verankerung für einen Beweis folgender Invariante mittels Induktion:

> Nach dem Durchlauf der Schritte (10)-(14) mit Werten i_0, j_0, k_0, l_0 für die zugehörigen Laufvariablen gilt:
> Die Feldelemente $A[i,j,k,l]$ mit $l \leq l_0$, $i \geq j_0$, $j \leq j_0$, $k \geq k_0$ sind korrekt berechnet worden.

(Mit „korrekt berechnet" meine ich, daß die Anforderungen von S. 4 erfüllt sind.)

Beim Induktionsschritt kann man also davon ausgehen, daß vor dem Durchlauf der Schritte (10)-(14) alle Feldelemente $A[i,j,k,l]$, die der Algorithmus bis dahin bearbeitet hat, korrekt berechnet worden sind. Man muß zeigen, daß dann auch das aktuelle Feldelement $A[i_0,j_0,k_0,l_0]$ korrekt berechnet wird.

Da die Prozeduren Case 1, Case 2, Case 3 – wie oben gesehen – nur bereits berechnete Feldelemente benutzen, die nach Induktionsannahme korrekt sind, ist klar, daß diese Prozeduren richtig arbeiten und ein erneuter Aufruf dieser Prozeduren keine neuen Knoten hinzufügen würde.

Das Zusammenspiel von Case 5 und Case 4 ist jedoch etwas komplizierter, wie das Beispiel aus Abschnitt 2.5 vermuten läßt.

Schaut man sich nochmal die Ungleichungskette für Case 5 in Abschnitt 2.4 an, so ergibt sich, daß Case 5 nur auf bereits berechnete Feldelemente oder auf das aktuelle Feldelement zugreift, aber nie auf Elemente, auf die der Algorithmus noch nie im Hauptteil zugegriffen hat. (Dies ergibt sich im wesentlichen aus der Anordnung der „for"-Schleifen in den Schritten (6)-(9).) Damit ist gezeigt, daß Case 5 keine „falschen" Elemente zu $A[i_0,j_0,k_0,l_0]$ hinzunimmt.

Würde Case 5 nie auf das aktuelle Feldelement zurückgreifen, so gäbe es keine Probleme, denn dann wäre Case 4 die einzige Prozedur, die dieses Feldelement benutzt, und da sie als letztes aufgerufen wird (Schritt (14)), wäre dann klar, daß der Algorithmus korrekt arbeitet.

Nun ist das aber leider nicht so. Die Ungleichungskette für Case 5 liefert, daß die Prozedur genau dann auf das zu berechnende Feldelement zugreift, wenn $m = j$ und

$p = k$ gilt. Dann ist die Wurzel μ des zu adjungierenden Baumes in $A[i_0, j_0, k_0, l_0]$ und der Knoten λ, das „Adjunktionsziel“, in $A[j_0, j_0, k_0, k_0]$.

Das bedeutet aber, daß der Knoten λ einen Baum dominiert, der kein Terminalsymbol produziert. Nach der Anwendung von Case 5, also nach der Adjunktion, produziert der nun von λ domierte Baum mindestens ein Terminalsymbol, denn der adjungierte Baum mußte mindestens ein Terminalsymbol haben.

Außerdem kann weder λ noch ein Vorfahr von λ, der mittels Case 4 (in Schritt (14)) zu $A[j_0, j_0, k_0, k_0]$ hinzugefügt wird, Wurzel eines auxiliaren Baumes sein. Dies kann man sich folgendermaßen klarmachen:
Nimmt man an, λ ist doch Wurzel eines auxiliaren Baumes, so erhält man, wenn man die gerade ausgeführte Adjunktion rückgängig macht, einen auxiliaren Baum (oder davon abgeleiteten Baum), der kein Terminalsymbol produziert. Dies kann aber nicht sein.
Analog folgt, daß auch von Case 4 berechnete Vorfahren von λ nicht Wurzeln von auxiliaren Bäumen sein können.

Der Rest des Beweises ist nicht mehr schwierig. Man muß nur noch zeigen, daß nach dem Durchlaufen der Schritte (10)-(14) ein erneuter Aufruf von Case 5 keine neuen Knoten hinzufügen würde. Es ist klar, daß man nur den Fall $m = j$ und $p = k$ betrachten muß. (In allen anderen Fällen benutzt Case 5 das aktuelle Feldelement gar nicht.)

Da in $A[i_0, j_0, k_0, l_0]$ kein neuer Wurzelknoten und in $A[j_0, j_0, k_0, k_0]$ kein neues „Adjunktionsziel“ enstanden sein kann, sind keine neuen Adjunktionen möglich, so daß ein erneuter Aufruf von Case 5 keine neuen Knoten mehr hinzufügen würde.

Damit ist die Invariante gezeigt, und mit den Bemerkungen zu Schritt (15) in Abschnitt 2.3 folgt die Korrektheit des Algorithmus.

2.7 Die Komplexität des Algorithmus

Sei n die Länge der zu parsenden Eingabe. Die TAG wird im folgenden als fixiert angesehen.

Wie man sieht, ist die Speicherplatzkomplexität des Algorithmus $O(n^4)$, da im wesentlichen nur Platz für das 4-dimensionale Feld A verbraucht wird. Die maximale Anzahl der Einträge in einem Feldelement hängt nicht von der Eingabe ab, sondern von der TAG[1], unter obigen Vorraussetzungen ist dies also eine Konstante.

Wenn man die Zeitkomplexität des Algorithmus untersucht, so sieht man relativ leicht ein, daß die Schritte (1) und (2) Linearzeit und die Schritte (3) bis (5) quadratische Laufzeit benötigen (s. 2.2). Die Initialisierung ist also in qudratischer Zeit durchführbar.

Die Prozeduren Case 1, Case 2, Case 3 und Case 5 haben eine quadratische Laufzeit und die Prozedur Case 4 hat eine konstante Laufzeit, da die Grammatik als fest angesehen wird.

Ein Durchlauf der Schritte (10) bis (14) (s. 2.3) kostet also quadratische Zeit. Damit ist klar, das die Schritte (6) bis (14) eine Laufzeit in $O(n^6)$ brauchen. Offensichtlich kann Schritt (15) in Linearzeit durchgeführt werden.

Der gesamte Algorithmus hat also eine Laufzeit in $O(n^6)$. Damit ist folgender Satz bewiesen:

Satz 2.1 *Das Wortproblem für Baumadjunktionsgrammatiken ist in Zeit $O(n^6)$ entscheidbar (bei einem Speicherplatzverbrauch in $O(n^4)$), wobei n die Länge der zu parsenden Eingabe sei.*

[1] Genauer gesagt: Die maximale Anzahl der Einträge in einem Feldelement ist durch die Anzahl der Knoten in den elementaren Bäumen beschränkt.

2.8 Erweiterung des Algorithmus auf TAGs mit Constraints

In diesem Abschnitt möchte ich beschreiben, wie man den Algorithmus modifizieren muß, um damit auch TAGs mit Constraints parsen zu können. Anders als bei den Knotennummern können sich die Constraints, die zu einem bestimmten Knoten gehören im Laufe des Ableitungsvorganges ändern. Bei den Constraints handelt es sich offenbar um dynamische Informationen, die man mit in die Einträge des 4-dimensionalen Feldes $A[i,j,k,l]$ aufnehmen muß. Die Feldelemente sind also keine Knotennummern mehr, sondern Tupel der Form (μ, C). Hierbei ist μ die bekannte Knotennummer und C beschreibt die Constraints, die an dem Knoten vorliegen.

C selbst kann man wieder als ein Tupel (OA, SA) aufffassen. OA ist ein Flag, das angibt, ob an dem Knoten ein OA-Constraint (obligatorische Adjunktion) vorliegt; SA steht für das SA-Contraint (selektive Adjunktion), SA enthält Informationen darüber, welche auxiliaren Bäume an dem Knoten adjungiert werden können. (In einer konkreten Implementierung würde man hier z.B. die Nummern der Wurzeln der adjungierbaren Bäume speichern.)

Die Anforderungen an die Feldelemente (vgl. S. 4) ändern sich wie folgt:

> Sei (μ, C) ein Knoten-Constraints-Tupel, und die Wurzel des elementaren Baumes, aus dem μ stammt, trage die Beschriftung X:
>
> 1. Für i, j, k, l mit $0 \leq i \leq j < k \leq l \leq n$ gilt: $(\mu, C) \in A[i,j,k,l] \iff$ $\exists \gamma \in \mathcal{D}(\mu)$ mit Blattwort $a_{i+1} \cdots a_j \; X \; a_{k+1} \cdots a_l$, die Wurzel von γ hat das Constraint C und der außer der Wurzel hat kein Knoten von γ ein OA-Constraint.
> 2. Für i, l mit $0 \leq i < l \leq n$ gilt: $\exists\, i \leq j \leq l : (\mu, C) \in A[i,j,j,l] \iff$ $\exists \gamma \in \mathcal{D}(\mu)$ mit Blattwort $a_{i+1} \cdots a_l$, die Wurzel von γ hat das Constraint C und der außer der Wurzel hat kein Knoten von γ ein OA-Constraint.

Für die Initialisierung des Algorithmus (s. 2.2) ändert sich hierdurch nicht viel. Statt der jeweiligen Knotennummer nimmt man ein Knoten-Contraints-Tupel in ein Feldelement auf. In die Constraints-Komponente des Tupels schreibt man die Constraints, die der Knoten in seinem elementaren Baum besitzt.

Auch Case 1 bis Case 4 erfordern nur geringe Änderungen. Man muß darauf achten, daß keine Väter in ein Feldelement aufgenommen werden, wenn einer ihrer Söhne ein OA-Constraint trägt. Die Constraints der Väter liest man wie oben aus den elementaren Bäumen ab.

Etwas komplizierter ist Case 5. Wenn an einem Knoten λ eine Adjunktion eines Baumes mit Wurzel μ erfolgt, speichert der Algorithmus die Nummer λ im aktuellen Feldelement ab (und nicht etwa die Nummer μ) (s. 2.4). Schaut man sich die Definition der Adjunktion bei TAGs mit Constraints an, so stellt man fest, daß in oben beschriebenem Fall vor der Adjunktion überprüft werden muß, ob der Baum mit Wurzel μ überhaupt in λ adjungiert werden darf. Nach der Adjunktion muß λ die Constraints von μ übernehmen.

Dieser Sachverhalt ist in Abbildung 10 dargestellt. Man sieht, daß die Constraints (C_1) von λ überprüft und die Constraints (C_2) von μ gespeichert werden.

Im Hauptteil des Algorithmus (s. 2.3) muß man Schritt (15) dahingehend ändern, daß nur solche Wurzelknoten von initialen Knoten akzeptiert werden, die kein OA-Constraint haben.

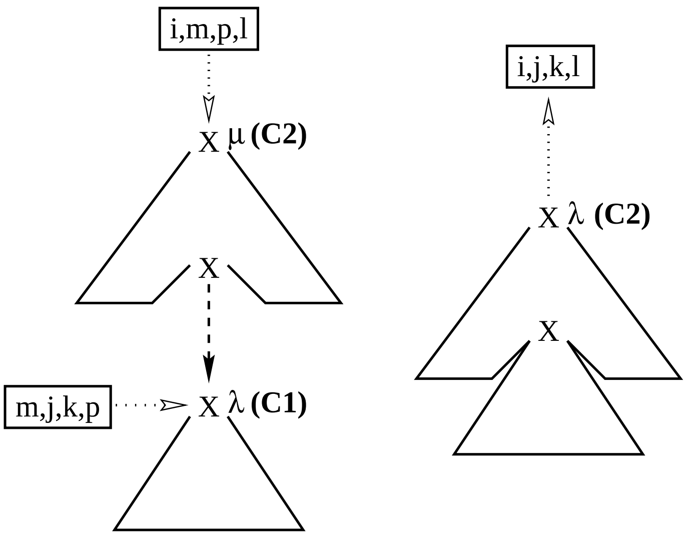

Abbildung 10: Case 5 unter Berücksichtigung von Constraints

2.9 Ein größeres Beispiel

In diesem Abschnitt erläutere ich die Arbeitsweise des Algorithmus anhand eines Beispieles. Dazu betrachte ich folgende Sprache:

$$L = \{a^n b^n e c^n d^n : n \geq 1\}$$

Eine Baumadjunktionsgrammatik G, die diese Sprache erzeugt ist in Abbildung 11 zu sehen. Meine weiteren Ausführungen werden sich auf diese Grammatik beziehen.

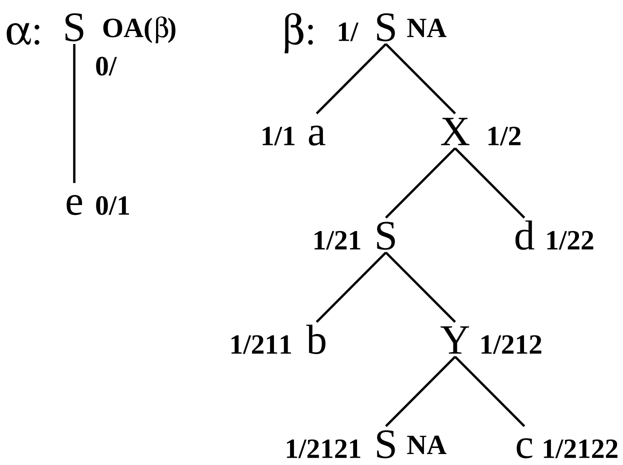

Abbildung 11: TAG zur Sprache L

Im Beispiel werde ich als Eingabe das Wort *aabbeccdd* betrachten. Der Baum, der dieses Wort ableitet, ist Abbildung 12 zu sehen.

Zunächst verfolge ich die Initialisierung des Algorithmus (s. 2.2). In den Schritten (1) und (2) werden die mit Terminalsymbolen beschrifteten Knoten in die entsprechenden Feldelemente geschrieben.

Bei der Eingabe $a_1a_2b_3b_4e_5c_6c_7d_8d_9$ bedeutet dies:

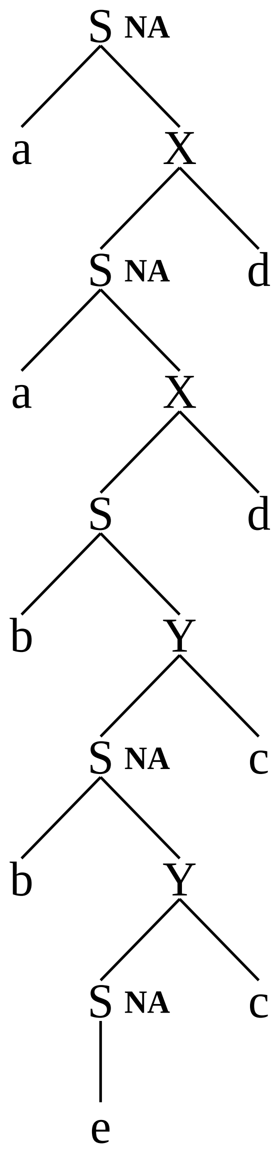

Abbildung 12: abgeleiteter Baum zur Eingabe *aabbeccdd*

- Der Knoten 1/1 (a) wird nach $A[0,1,1,1]$ und $A[1,2,2,2]$ geschrieben.
- Der Knoten 1/211 (b) wird nach $A[2,3,3,3]$ und $A[3,4,4,4]$ geschrieben.
- Der Knoten 0/1 (e) wird nach $A[4,5,5,5]$ geschrieben.
- Der Knoten 1/2122 (c) wird nach $A[5,6,6,6]$ und $A[6,7,7,7]$ geschrieben.
- Der Knoten 1/22 (d) wird nach $A[7,8,8,8]$ und $A[8,9,9,9]$ geschrieben.

Genau genommen wird ein Knoten-Constraints-Tupel gespeichert, aber da diese Knoten keine Constraints haben, habe ich dies der Einfachheit halber weggelassen. Auch in Zukunft werde ich den Constraints Teil des Tupels nur erwähnen, wenn er interessant ist.

Dies ist z.B. in den Schritten (3) bis (5) der Fall. Hier wird der Knoten 1/2121 zusammen mit seinem NA-Constraint in die Feldelemente $A[0,0,1,1] \dots A[8,8,9,9]$ geschrieben.

Dann beginnt der Hauptteil des Algorithmus (s. 2.3). Ich werde nicht alle Schritte des Algorithmus vorstellen, sondern nur die wichtigen.

Für $i = 0, \dots, 5$ wird der Knoten 1/212 nach $A[i,i,6,7]$ geschrieben (s. Abb. 13). Hierbei wird Case 1 angewendet. (Analog wird für $i = 0, \dots, 4$ das Feldelement $A[i,i,5,6]$ für c_6 aufgebaut.)

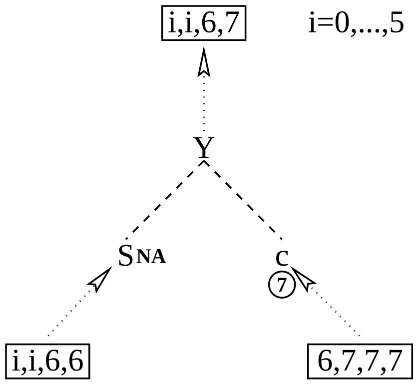

Abbildung 13: Anwendung von Case 1

Mittels Case 2 wird nun der Knoten 1/21 in $A[2,3,6,7]$ aufgenommen (s. Abb. 14). (Analog wird $A[3,4,5,6]$ für b_4 und c_6 berechnet.)

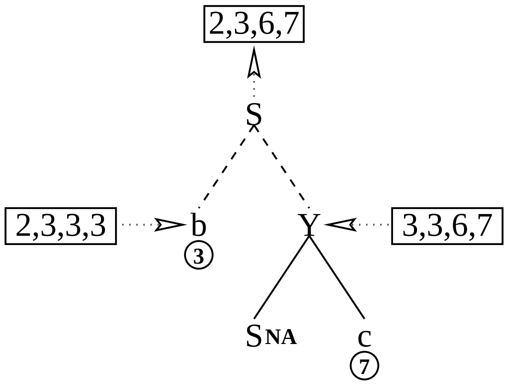

Abbildung 14: Anwendung von Case 2

Nach einer erneuten Anwendung von Case 1 und Case 2 hat der Algorithmus den ersten vollständigen Baum gefunden und kann den Knoten 1/, also die Wurzel von β, in $A[1, 3, 6, 8]$ aufnehmen. Man beachte, daß dieser Knoten ein NA-Constraint besitzt. Abbildung 15 stellt den Sachverhalt dar.

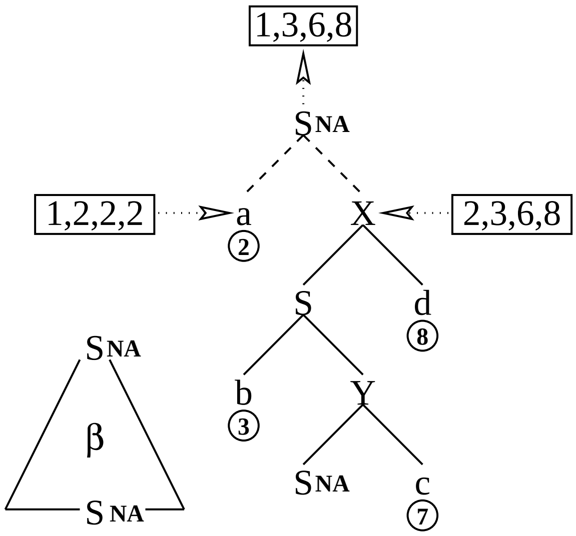

Abbildung 15: der erste vollständige Baum

Der soeben fertig gestellte Baum kann nun mittels Case 5 zu dem Teilbaum adjungiert werden, dessen Wurzel 1/21 sich in $A[3, 4, 5, 6]$ befindet. Die Wurzel des entstandenen Baumes wird nach $A[1, 4, 6, 8]$ geschrieben. Genauer gesagt wird der Knoten 1/21 – nun allerdings mit einen NA-Constraint versehen – dorthin geschrieben. Dies zeigt Abbildung 16.

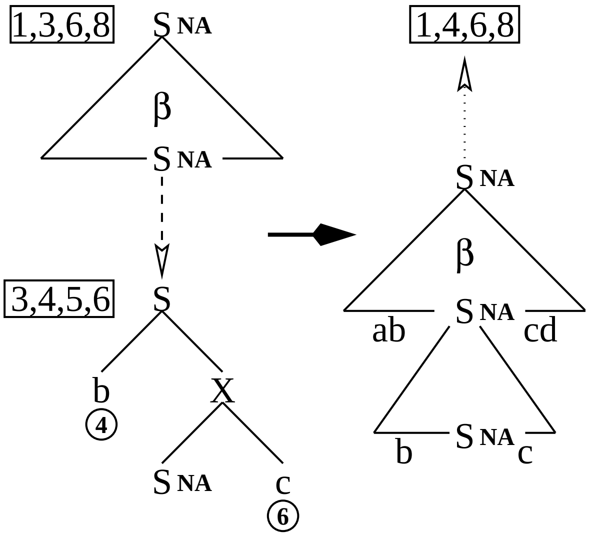

Abbildung 16: Anwendung von Case 5

Analog zu oben wird das Feldelement $A[0, 4, 5, 9]$ berechnet. Dieses enthält den Knoten 1/, also die Wurzel von β, versehen mit einem NA-Constraint. In einem letzten Schritt wird der von diesem Knoten dominierte Teilbaum, der die Teilstrings $a_1a_2b_3b_4$ und $c_6c_7d_8d_9$ der Eingabe abdeckt, zu dem Knoten 0/ in $A[4,5,5,5]$ adjungiert. Dadurch erhält man den fertigen Baum aus Abbildung 12. Wie man in Abbildung 17 sieht, wird hierbei das OA-Constraint des Knotens gelöscht und durch ein NA-Constraint ersetzt.

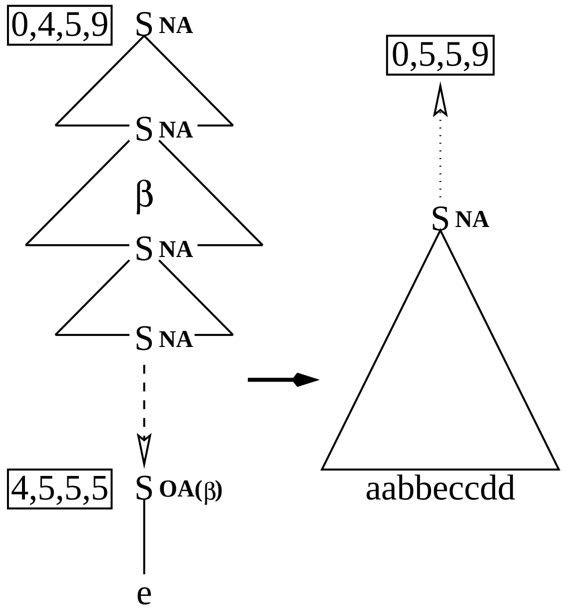

Abbildung 17: der letzte Schritt

Somit enthält $A[0,5,5,9]$ die Wurzel 0/ des initialen Baumes α, die kein OA-Constraint hat. Deshalb wird die Eingabe akzeptiert.

3 Der Algorithmus von K. Harbusch

Der Parsing-Algorithmus von Karin Harbusch in [1] hat wie der oben vorgestellte Algorithmus eine enge Verbindung zum CYK-Algorithmus für kontextfreie Sprachen. Beim Ansatz von Harbusch handelt es sich jedoch nicht um eine Weiterentwicklung des CYK-Algorithmus, sondern in ihrem Ansatz kommt eine etwas erweiterte Version dieses Algorithmus als eigenständiger Teilschritt vor.

Es stellt sich nun die Frage, wie man eine Baumadjunktionsgrammatik[2] so in eine kontextfreie Grammatik „umwandeln" kann, daß man den CYK-Algorithmus darauf ansetzen kann und verwertbare Ergebnisse erhält. Die Antwort auf diese Frage liefert

Definition 3.1 (Kontextfreier Kern (KFK))
Der kontextfreien Kern $K = (N_K, T_K, S_K, P_K)$ einer Baumadjunktionsgrammatik G ist folgendermaßen definiert:

- *Die Nichterminale (N_K) und die Terminale (T_K) sind dieselben wie bei G.*
- *Das Startsymbol (S_K) ist dasselbe wie bei G.*
- *Jede Beschriftung eines inneren Knotens in einem elementaren Baum von G ist linke Seite einer kontextfreien Regel in P_K und die Beschriftungen seiner*

[2]Alle im folgenden betrachteten Grammatiken sind grundsätzlich TAGs mit Constraints.

Söhne bilden gemäß ihrer Reihenfolge im Baum (von links nach rechts) die rechte Seite.

Betrachtet man die TAG in Abbildung 18, so ergeben sich für den kontextfreien Kern folgende Produktionen:

1:	S	$\longrightarrow$	X	Y
2:	X	$\longrightarrow$	a	
3:	Y	$\longrightarrow$	c	
4:	X	$\longrightarrow$	X	Z
5:	Z	$\longrightarrow$	b	

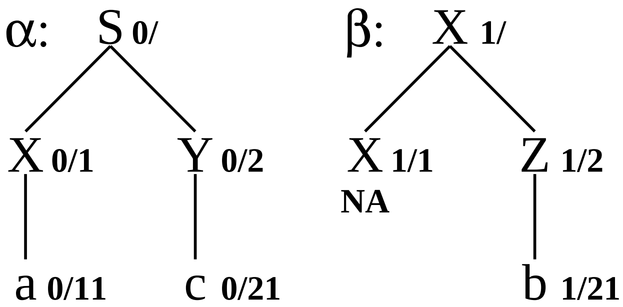

Abbildung 18: Beispiel-TAG

Um den KFK zum TAG-Parsen verwenden zu können, reicht es nicht aus, in die Regeln nur Informationen über die Beschriftungen von Knoten aus elementaren Bäumen aufzunehmen, sondern man muß sich auch die Knotennummern und die Contraints merken.

Den so erweiterten kontextfreien Kern bezeichne ich als „kontextfreien Kern mit Annotationen“. In obigem Beispiel würde sich folgender KFK mit Annotatinen ergeben:

1:	S (0/,-)	$\longrightarrow$	X (0/1,-)	Y (0/2,-)
2:	X (0/1,-)	$\longrightarrow$	a (0/11,-)	
3:	Y (0/2,-)	$\longrightarrow$	c (0/21,-)	
4:	X (1/,-)	$\longrightarrow$	X (1/1,NA)	Z (1/2,-)
5:	Z (1/2,-)	$\longrightarrow$	b (1/21,-)	

Ich möchte hierzu noch anmerken, daß es zu einer Regel aus dem KFK durchaus mehrere verschiedene Annotationen geben kann.

Der Begriff des KFK erlaubt es, eine neue Normalform für TAGs zu definieren, die eine Verschärfung der Zweiform (s. Def. 1.4) darstellt:

Definition 3.2 (Normalform) *Eine Baumadjunktionsgrammatik ist in Normalform, wenn ihr kontextfreier Kern in Chomsky-Normalform ist.*

Nun kann man die Bedingungen formulieren, die der Algorithmus von K. Harbusch an die zu parsende TAG stellt:[3]

- Die Grammatik ist in Normalform.
- Jeder Fußknoten eines auxiliaren Baumes trägt ein NA-Constraint.

[3] Zu jeder TAG gibt es eine äquivalente TAG mit Constraints, die diese Bedingungen erfüllt. (vgl. Satz 5 und 6 in [1])

3.1 Die Grundideen des Algorithmus

Grundlegend für den Parsing-Algorithmus von K. Harbusch ist die Beobachtung, daß für eine Baumadjunktionsgrammatik G und ihren kontextfreien Kern K gilt:

> Die von G erzeugte Sprache $L(G)$ ist eine Teilmenge der von K erzeugten Sprache $L(K)$.

Obige Aussage kann man auch anders formulieren: Falls ein Wort eine TAG-Ableitung in G besitzt, dann besitzt es erst recht eine kontextfreie Ableitung in K. Dies legt folgende Vorgehensweise zum Parsen einer Eingabe w nahe:

1. Wende den CYK-Algorithmus auf den KFK K und w an.
2. Sondere aus den kontextfreien Ableitungsbäumen diejenigen aus, die TAG-Ableitungen entsprechen.

Den ersten Schritt kann man ziemlich schnell durchführen, nämlich in $O(n^3)$; wenn es gelingt, den zweiten Schritt auch schnell auszuführen, so hat man insgesamt einen schnellen Algorithmus zum Parsen von TAGs gefunden.

Mit den Informationen, die der herkömmliche CYK-Algorithmus liefert, kann man den zweiten Schritt nicht effizient durchführen, man muß den Algorithmus so erweitern, daß er nicht nur auf dem KFK arbeitet, sondern auch die Annotationen sinnvoll verarbeitet.

3.2 Eine Erweiterung des CYK-Algorithmus

In diesem Abschnitt beschreibe ich, wie man CYK-Algorithmus erweitern muß, damit man ihn als Basis zum Parsen von TAGs einsetzen kann. Würde man den herkömmlichen Algorithmus auf den KFK der TAG aus Abbildung 18 und die Eingabe *abc* ansetzen, so erhielte man den Ableitungsbaum in Abbildung 19.

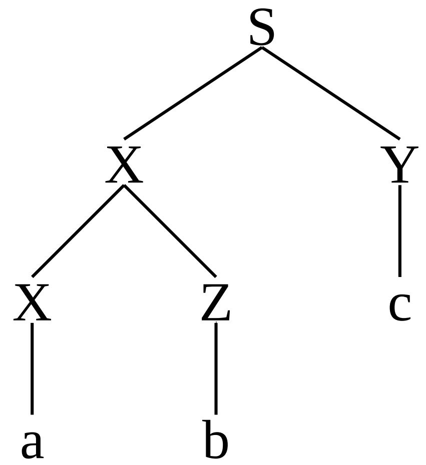

Abbildung 19: Ableitungsbaum ohne Berücksichtigung von Annotationen

Man erweitert den Algorithmus so, daß er den Ableitungsbaum mit Annotationen versieht. Dies geht folgendermaßen:

Wurde an einer Stelle im Baum die kontextfreie Regel $X\ (A_X) \longrightarrow Y\ (A_Y)\ Z\ (A_Z)$ verwendet, wobei (A_X), (A_Y) und (A_Z) die jeweiligen Annotationen sind, so soll der Algorithmus folgendes tun:

- An den Knoten wird neben der Beschriftung X auch noch die Annotation (A_X) geschrieben.
- An die linke aus dem Knoten ausgehende Kante wird die Annotation (A_Y) geschrieben.
- An die rechte aus dem Knoten ausgehende Kante wird die Annotation (A_Z) geschrieben.

Analog sollen Regeln der Form $X\ (A_X) \longrightarrow t\ (A_t)$ behandelt werden. Und schließlich soll an die mit Terminalsymbolen beschrifteten Knoten auch die Annotation, d.h. die Knotennummer geschrieben werden.

Wendet man den so modifizierten Algorithmus auf den KFK der TAG aus Abbildung 18 und die Eingabe *abc* an, so erhält man den Ableitungsbaum aus Abbildung 20.

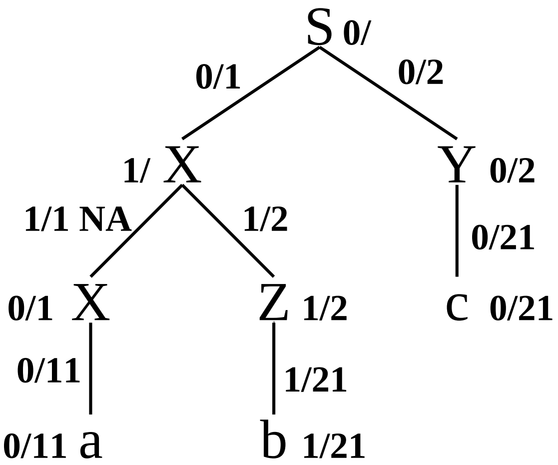

Abbildung 20: Ableitungsbaum mit Berücksichtigung von Annotationen

Dieser Ableitungsbaum repräsentiert, wie man sieht, auch eine gültige TAG-Ableitung des Wortes *abc*.

Ich werde nun erklären, wie ein Algorithmus die Zusatzinformationen im Baum nutzen kann, um festzustellen, ob es sich bei einem Ableitungsbaum bezüglich des KFK um eine TAG-Ableitung handelt.

Schaut man sich Abbildung 20 nochmals an, so stellt man fest, daß hier der initiale Baum α durch die Adjunktion des auxiliaren Baumes β im Knoten $0/1$ (X) „unterbrochen“ wurde, wenn man sich den Baum einmal als TAG-Ableitung vorstellt.

Dies kann man auch aus den Annotationen im Baum ablesen. Man sieht, daß die Kante, die in den Knoten $0/1$ einläuft, eine andere Annotion als der Knoten besitzt. Dagegen besitzen der Knoten $1/2$ (Z) und die einlaufende Kante dieselben Annotationen; gleiches gilt für den Knoten $1/21$ (b).

Der Baum β ist also nicht „unterbrochen“. Einen solchen Baum nennt man auch vollständig[4].

Man kann sich leicht klarmachen, daß der Baum, der entsteht, wenn man aus dem Ableitungsbaum zu einem KFK einen vollständigen Baum entfernt, genau dann eine TAG-Ableitung repräsentiert, wenn dies bei dem ursprünglichen Baum der Fall war.

[4]Eine formale Definition des Begriffes findet man in Def. 38 in [1].

Entfernt man also im Beispiel den vollständigen Baum β, so erhält man den initialen Baum α, der nun vollständig ist. Dies ist in Abbildung 21 dargestellt. Damit hat man gezeigt, daß die Eingabe *abc* tatsächlich zu der von der TAG G (s. Abb. 18) erzeugten Sprache gehört.

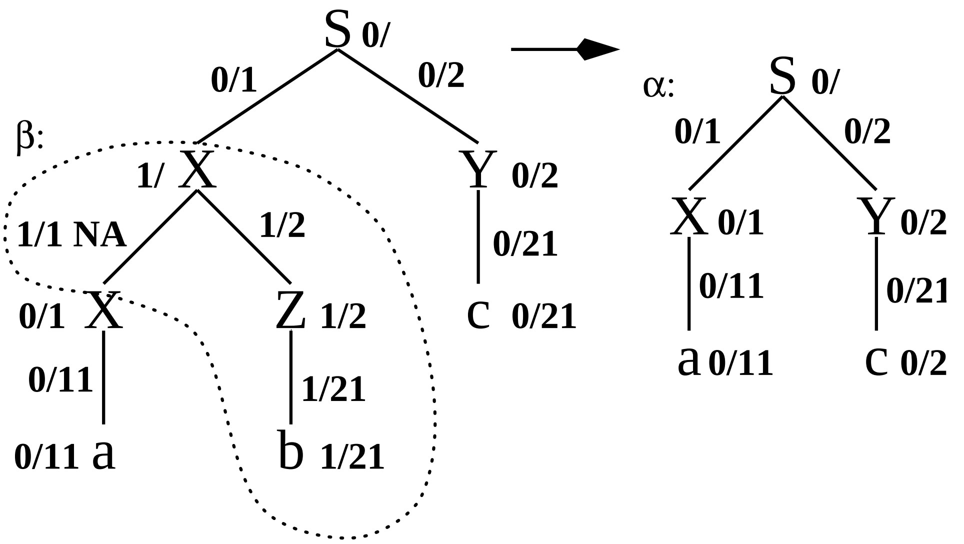

Abbildung 21: Entfernung eines vollständigen Baumes

3.3 Innerste Bäume und wie man sie findet

Die Überlegungen des vorigen Abschnittes liefern ein Verfahren, wie man bei einem gegebenen (erweiterten) Ableitungsbaum bezüglich des KFK einer TAG entscheiden kann, ob dieser Baum auch TAG-gemäß ableitbar ist:

Man sucht in dem Baum alle vollständigen Bäume und elimiert diese. Danach sucht man wieder alle vollständigen Bäume, eliminiert diese, usw.

Dies kann man auch mit Hilfe des Begriffes des „innersten Baumes i-ter Stufe“ beschreiben:

Definition 3.3 (innerster Baum i-ter Stufe)
Sei γ ein Baum, der mittels einer Baumadjunktionsgrammatik G ableitbar ist. Für $i \geq 1$ ist ein innerster Baum i-ter Stufe in γ wie folgt rekursiv definiert:

- *Jeder vollständige Baum in γ ist ein innerster Baum erster Stufe in γ.*
- *Sei $i > 1$ und γ enthalte mindestens einen innersten Baum (i-1)-ter Stufe. Weiter sei γ' der Baum, den man erhält, indem man in γ alle Bäume der Stufen $\leq$ i-1 entfernt.*
  Dann sind alle vollständigen Bäume in γ' innerste Bäume i-ter Stufe in γ.

Nun ergibt sich die Frage: Welche Stufe können innerste Bäume eines Baumes, der ein Wort der Länge n ableitet, höchstens haben?

Bei einer beliebigen TAG lautet die Antwort unendlich, denn wenn ein auxiliarer Baum kein Terminalsymbol produziert, kann man ihn beliebig oft adjungieren, ohne dadurch die Länge des abgeleiteten Wortes zu verändern[5].

[5]Das geht natürlich nur, wenn die Constraints das zulassen.

Bei TAGs in Normalform sieht die Antwort jedoch besser aus. Da der KFK in Chomsky-Normalform sein muß, folgt, daß jeder Baum mindestens ein Terminalsymbol produzieren muß. Einzige Ausnahme kann der sogenannte ϵ-Baum sein; dieser Baum entspricht der Regel $S \longrightarrow \epsilon$ und muß an seiner Wurzel das NA-Constraint tragen, so daß er für Adjunktionen nicht in Betracht kommt. Geht man also davon aus, daß jeder Baum mindestens ein Terminal produziert, so können bei einer Eingabe der Länge n höchstens n-1 Adjunktionen stattgefunden haben, da der initiale Baum ja auch mindestens ein Terminal liefert. Daraus ergibt sich, daß ein innerster Baum maximaler Stufe höchstens Stufe n haben kann. Der innerste Baum maximaler Stufe ist der ursprüngliche initiale Baum.

Hieraus ergeben sich Aussagen über die Laufzeit eines Algorithmus, der nach dem hier beschriebenen Prinzip arbeitet.

Der erweiterte CYK-Algorithmus generiert ein 2-dimensionales Feld, in dem alle möglichen Ableitungsbäume mit Annotationen zu der gegebenen Eingabe der Länge n repräsentiert sind. Dazu benötigt er eine Laufzeit in $\Omega(n^3)$.

Um die innersten Bäume einer Stufe i zu finden, nachdem alle innersten Bäume niedrigerer Stufe eliminiert worden sind, muß man mindestens einmal das Feld durchgehen, und es ist plausibel, daß man dies analog zum CYK-Algorithmus tun muß. Also braucht man hierfür auch Zeit $\Omega(n^3)$.

Insgesamt gibt es höchstens n Stufen, und man kann sich überlegen, daß es Beipiele mit genau n Stufen gibt. Deshalb kann man erwarten, daß die Laufzeit des worst-case in $\Omega(n^4)$ liegt.

3.4 Der Algorithmus

Der Algorithmus von K. Harbusch hat folgende Struktur:

```
(tag1) Behandlung des leeren Wortes
(tag2) Vergabe von eindeutigen Knotennummern
(tag3) Bestimmung des kontextfreien Kerns mit Annotationen der TAG
(tag4) erweiterte CYK-Analyse mit der Eingabe w (Länge n)
(tag5) Berechnung der innersten Bäume erster Stufe
(tag6) Iterative Berechnung der innersten Bäume der Stufe i (1 < i ≤ n)
       Abbruch, wenn initialer Baum gefunden wurde
(tag7) Ausgabe des Ergebnisses
```

Auf die Details der Implementierung möchte ich hier nicht eingehen, da dies den Rahmen dieses Artikels sprengen würde und verweise auf [1].

Literatur

[1] **K. Harbusch:** *Effiziente Strukturanalyse natürlicher Sprache*, Dissertation am Lehrstuhl Prof. Dr. W. Wahlster, an der Universität des Saarlandes, Saarbrücken, 1989.

[2] **J.E. Hopcroft, J.D. Ullman:** *Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie*, Addison-Wesley, Bonn, 1994.

[3] **A.K. Joshi:** *An Introduction to Tree Adjoining Grammars*, Technical Report MS-CIS-86-64, Department of Computer and Information Science, Moore School, University of Pennsylvania, Philadelphia, Pennsylvania, 1985.

[4] **K. Vijay-Shanker, A.K. Joshi:** *Some computational Properties of Tree Adjoining Grammars*, Proceedings of the 23^{rd} Annual Meeting of the Association for Computational Linguistics, Chicago, Illinois, 1985.

[5] **K. Vijay-Shanker:** *A Study of Tree Adjoining Grammars*, Dissertation Proposal, Department of Computer and Information Science, Moore School, University of Pennsylvania, Philadelphia, Pennsylvania, 1988.