Ein Earley-basierter Parsing Algorithmus für Baumadjunktionsgrammatiken

Ausarbeitung: Baré SAID

Seminar

Effizientes Parsing und Generieren mit

Baumadjunktionsgrammatiken (TAG)

im Wintersemester 1996/97

Prof. Wolfgang Wahlster, Dr. Tilman Becker

I N H A L T S V E R Z E I C H N I S

In diesem Seminar handelt es sich um einen praktischen Parsing-Algorithmus für TAGs, der von Yves Schabes entwickelt wurde. Dieser Algorithmus hat den Vorteil, daß seine Laufzeit sich im Durchschnittsfall von der Laufzeit im schlechtesten Fall unterscheidet. Dieser Laufzeitunterschied macht den Yves-Algorithmus zu einem besseren Algorithmus im Vergleich zu dem CKY-Algorithmus, dessen Laufzeiten immer gleich sind.
Die Laufzeit des CKY-Algorithmus beträgt im schlechtesten sowie auch im Durchschnittsfall $0(n^6)$ und das macht diesen Algorithmus unbrauchbar für die praktische Anwendung. Deshalb war es erforderlich, einen neuen Algorithmus zu entwickeln, dessen Laufzeit im Durchschnittsfall besser ist. Die GPE-Version[1)] des Algorithmus hat im schlechtesten Fall die Laufzeit $0(n^9)$, aber sein Durchschnittsfall ist viel besser, und man kann sogar bei bestimmten Grammatiken, wie zum Beispiel bei der Grammatik für die Sprache a^n b^n e c^n d^n, lineare Zeiten erreichen.
Abgesehen von dem Laufzeitverhalten der beiden Algorithmen ist die GPE-Version des Algorithmus praktischer als der CKY-Algorithmus, weil er im Gegensatz zu diesem keine Annahmen über die Grammatik macht. Der Baum muß z. B. nicht immer binär sein und muß auch nicht immer mindestens ein Blatt haben.

Schabes hat zwei Versionen von seinem Parser beschrieben. Die beiden Versionen sind Earley-Typen, d.h. sie sind *„left to right Bottom Up“*-Parser und benutzen *Top Down*-Informationen, indem sie *Predictions* (Vorhersagen) machen. Die zwei Parser unterscheiden sich in der Art, wie sie mit den *Top Down*-Informationen umgehen. Der erste Parser benutzt alle *Top Down*-Informationen und ist dadurch ein echter Earley-Typ, während der zweite Parser nur einen Teil der *Top Down*-Informationen benutzt.
In dieser Ausarbeitung werde ich den ersten Parser detailliert vorstellen und zeigen, wie man ihn leicht modifizieren kann, um *Constraints* und Substitutionen zu behandeln. Nachher werde ich seine Laufzeit- und Speicherplatz-Komplexität informal erklären. Dann werde ich auf den zweiten Parser eingehen und ihn kurz darstellen. Ich werde Unterschiede, sowie die Vorteile und Nachteile im Vergleich zum ersten Parser zeigen.

1) GPE bedeutet Gültige-Präfix-Eigenschaft und wird in einem späteren Kapitel erklärt.

1. DEFINITIONEN :

In diesem Abschnitt werde ich ein Konzept einführen, das in dem Algorithmus benutzt wird. Zuerst werde ich zeigen, wie gepunktete Regel (Produktionen) für die TAGs erweitert werden können. Dann werde ich eine Baumtraversierung einführen, die der Parser simulieren wird, um das Eingabewort zu *scannen* .

Gepunktetes Symbol :

Ein gepunktetes Symbol ist ein Symbol mit einem Punkt an einer seiner vier Ecken, d.h. das Symbol hat einen Punkt, entweder an seiner oberen rechten oder linken Ecke, oder an seiner unteren rechten oder linken Ecke.

${}^{\bullet}S,\ S^{\bullet},\ {}_{\bullet}S,\ S_{\bullet}$

Abbildung 1.

Gepunkteter Baum :

Ein gepunkteter Baum ist ein Baum mit genau einem einzigen gepunkteten Symbol .

Baumtraversierung :

Unter einer Baumtraversierung versteht man das Bewegen des Punktes im Baum auf eine bestimmte Weise, die dem Parser ermöglicht, das Eingabewort von links nach rechts zu *scannen* und dabei noch die Adjunktionen am Baum zu erkennen .
Das Baumtraversierung ist folgendermaßen definiert :
1) Wenn der Punkt an der Position la (*Left Above*) eines inneren Knotens steht (d.h. der Punkt hängt nicht an einem Blatt), dann verschiebt man den Punkt an die Position lb (*Left Bottom*).

${}^{\bullet}S \Rightarrow {}_{\bullet}S$

Abbildung 2.

2) Wenn der Punkt an der Position lb eines inneren Knotens steht, dann verschiebt man ihn an die Position la seines am weitesten links stehenden Sohnes.

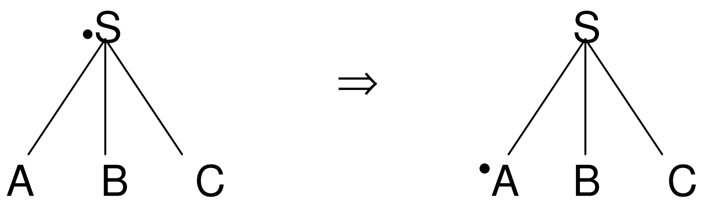


Abbildung 3.

3) Wenn der Punkt an der Position la eines Blattes steht, dann verschiebt man ihn an die Position ra (*Right Above*) desselben Blattes.

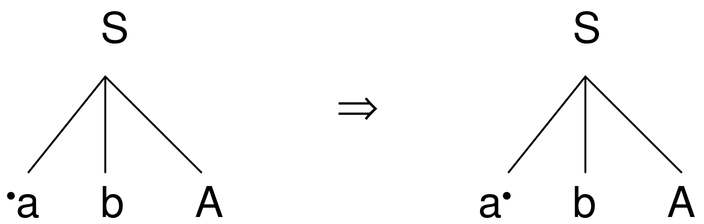


Abbildung 4.

4) Wenn der Punkt an der Position rb (*Right Bottom*) eines Knotens steht, dann verschiebt man den Punkt an die Position ra desselben Symbols.

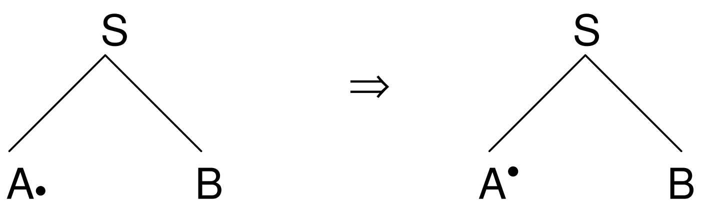
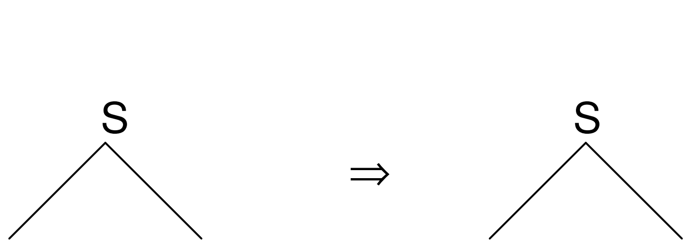


Abbildung 5.

5) Wenn der Punkt an der Position ra eines Knotens steht, dann muß man die zwei folgende Fällen betrachten :

a) Wenn der Knoten einen rechten Bruder hat, dann verschiebt man den Punkt an die Position la seines rechten Bruders.



$A^{\bullet}$ B A $^{\bullet}B$

Abbildung 6.a.

b) Wenn der Knoten keinen rechten Bruder hat, dann verschiebt man den Punkt an die Position rb seines Vaters.

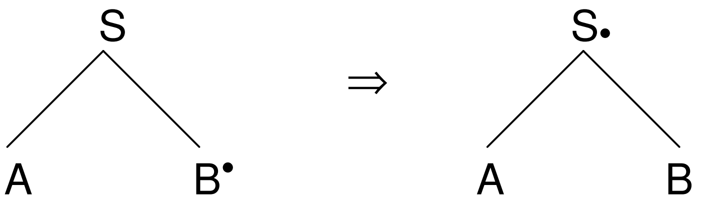


Abbildung 6.b.

Die oben definierte Baumtraversierung ermöglicht das *Scannen* der Blätter des Baumes und das gleichzeitige Erkennen der an dem Baum vorgenommenen Adjunktionen.

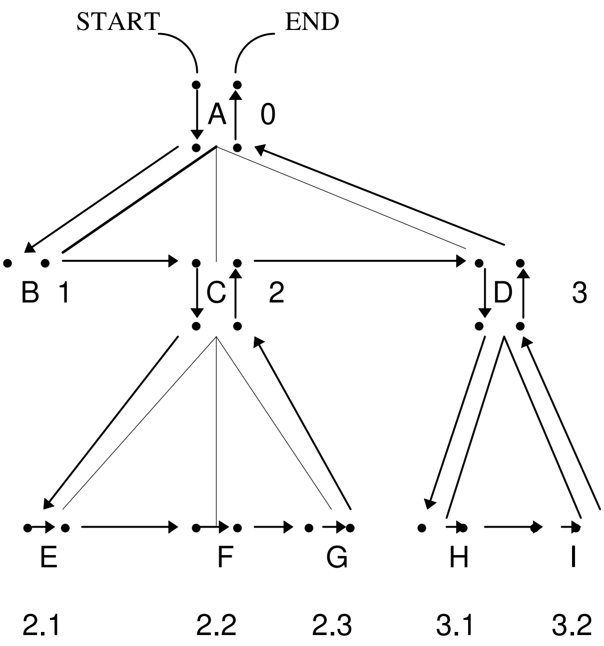


Abbildung 7.

2. EINFÜHRUNG DES ALGORITHMUS :

Zuerst werde ich die Strategie des Algorithmus informal darstellen und dann die einzelnen Schritte des Algorithmus detailliert und formal beschreiben.

2.1. Informale Beschreibung :

2.1.1. Die Datenstrukturen :

Der Algorithmus benutzt die *„left to right"*-Strategie. Bei dieser Strategie betrachtet man einen Zustand S als gepunkteten Baum mit gewissen Informationen über sich selbst.

Eine Zustandsmenge (S_i) ist eine Menge von Zuständen. Der Index i bezieht sich auf das Eingabewort a_1 a_2 ......a_i .....a_n .

Eine neue Zustandsmenge wird erzeugt, wenn ein neues Symbol des Eingabewortes erkannt wird.

Das Vorhandensein eines Zustandes in einer Zustandsmenge S_i bedeutet, daß das Eingabewort a_1 a_2 ...a_i gelesen wurde .

In der Zustandsmenge S_0 befinden sich alle Initialbäume mit dem Punkt an der Position la Ihrer Wurzel.

2.1.2. Die Grundidee des Algorithmus :

Der Algorithmus versucht die Adjunktionen zu erkennen, indem er für jeden Zustand der aktuellen Zustandsmenge einen der folgenden sieben Schritte durchführt: *Left Predictor*, *Left Completor, Right Predictor, Right Completor, Scanner, Move Dot Down, Move Dot Up.*
Der Algorithmus fängt mit einem Initialbaum an, bei welchem der Punkt an der Position la seiner Wurzel steht.
Wenn der Parser an die Position la eines Nichtterminalknoten kommt, dann *predictet* er, ob an diesem Knoten eine Adjunktion stattgefunden hat oder nicht .

Der Parser führt in diesem Fall seinen *Left Predictor* Prozeß durch. In diesem Prozeß betrachtet er zwei Annahmen gleichzeitig:

- Erste Annahme: eine Adjunktion hat stattgefunden.

In diesem Fall holt sich der Parser alle auxiliaren Bäume, die an diesem Knoten adjungiert werden können, und dann versucht er herauszufinden, welcher dieser Bäume adjungiert wurde, indem er den Baum weiter traversiert und Vergleiche mit dem Eingabewort macht.

- Zweite Annahme: Der Parser nimmt an, daß keine Adjunktion an dem Knoten stattgefunden hat.

In diesem Fall versucht er den Baum weiter zu traversieren, indem er an jedem Zustand (wir haben gesehen, daß ein Zustand einen gepunkteten Baum darstellt) einen seiner Prozesse durchführt.

Wenn der Parser an die linke Seite eines Fußknoten eines vorher *left predicteten* auxiliaren Baum kommt, dann weiß er, daß der Präfix des Eingabewortes mit der linken Seite des Baumes übereinstimmt und daß seine Vorhersage zur Hälfte richtig war (zur Hälfte, da wir nur die linke Seite erkannt haben). In diesem Fall geht der Parser zu dem Vaterbaum, wo die Adjunktion vermutet wurde und vervollständigt (*complete*) ihn, indem er neue Informationen in ihn hineinschreibt. Dieser Prozeß nennt sich *Left Completor .*

Wenn der Parser an die Position rb eines Nichtterminalknoten kommt, dann führt er einen der folgenden zwei Prozesse durch, je nachdem, ob an diesem Knoten eine Adjunktion stattgefunden hat oder nicht.

1. Prozeß: wenn eine Adjunktion stattgefunden hat, versucht er den rechten Teil des an dem Nichtterminalknoten adjungierten auxiliaren Baumes zu erkennen (später wird erklärt, was man darunter verstehen soll)

2. Prozeß: wenn an diesem Knoten keine Adjunktion stattgefunden hat, dann schiebt man einfach den Punkt an die Position ra.

Wir werden später sehen, daß man leicht feststellen kann, ob an dem Knoten eine Adjunktion stattgefunden hat oder nicht (durch * Marke).

Wenn der Parser an die Position ra einer Wurzel eines auxiliaren Baumes kommt, dann ist die Adjunktion komplett erkannt und der Parser versucht den Rest des Baumes, an dem der auxiliaren Baum adjungiert wurde, zu erkennen.

Der Parser benutzt die drei anderen Prozesse *Move Dot Down*, *Move Dot Up* und *Scanner*, um den Baum weiter zu traversieren und dabei das Eingabewort zu *scannen*.
An dieser Stelle werde ich keine weitere Beschreibung zu den drei letzten Prozessen machen, da sie leicht nachvollziehbar sind.

Wenn der Parser am Ende an die Position ra der Wurzel eines Initialbaumes kommt und das Eingabewort komplett *gescannt* hat, dann bedeutet dies, daß das Eingabewort zu der Sprache gehört. Sonst wird die Eingabe abgelehnt.

Nachdem wir jetzt intuitiv wissen, wie der Algorithmus abläuft, möchte ich die Bedeutung eines gepunkteten Baumes erläutern.

2.1.3. Die Bedeutung des gepunkteten Baumes :

Es gibt insgesamt acht Fälle: vier Fälle, wo der Punkt sich an einem Terminalsymbol befindet und vier Fälle, wo der Punkt sich an einem Nichtterminalsymbol befindet.

1) Ein Punkt an der Position la eines Terminalsymbols bedeutet, daß das Symbol darauf wartet, gelesen zu werden.

2) Ein Punkt an der Position lb eines Teminalsymbols wird nie erscheinen.

3) Ein Punkt an der Position ra eines Terminalsymbols bedeutet, daß das Symbol schon gelesen wurde.

4) Ein Punkt an der Position rb eines Terminalsymbols wird nie erscheinen.

5) Ein Punkt an der Position la eines Nichtterminalsymbols bedeutet, daß das Symbol noch nicht *left predicted* wurde. Entweder muß eine Adjunktion stattfinden, oder der Punkt muß an die Position lb verschoben werden.

6) Ein Punkt an der Position lb eines Nichtterminalsymbols bedeutet, daß ein *Left Predictor* Prozeß oder ein *Left Completor* Prozeß durchgeführt wurden.

7) Ein Punkt an der Position rb eines Nichtterminalsymbols bedeutet, daß das Symbol darauf wartet, *right predicted* zu werden .

8) Ein Punkt an der Position ra eines Nichtterminalsymbols bedeutet, daß das Symbol darauf wartet, *right completed* zu werden, oder daß ein *Right Completor* Prozeß schon durchgeführt wurde.

2.2. Formale Beschreibung :

Jetzt werde ich auf die formale und detaillierte Beschreibung des Algorithmus eingehen. Dazu benötigen wir einige Definitionen :

2.2.1. Definitionen :

Definition 1:

Ein Baum α wird als Funktion von Baumadressen nach Symbolen der Grammatik betrachtet. Wenn x eine gültige Adresse in α ist, dann ist $\alpha(x)$ das Symbol in der Adresse x des Baumes α.

Definition 2:

Linker Teil eines gepunkteten Baumes: der linke Teil eines gepunkteten Baumes ist die Menge der Knoten im Baum, die von beiden Seiten traversiert wurden, vereinigt mit der Menge der Knoten, die nur von ihrer linken Seite traversiert wurden.

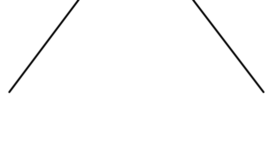
Definition 3:

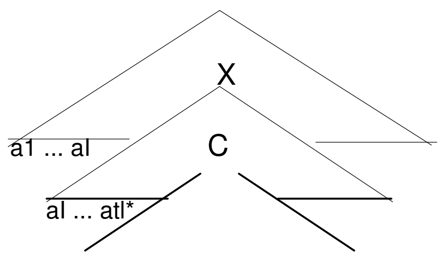
Ein Zustand S ist ein 10er -Tupel, [α , Punkt, *side* ,pos , l, fl, fr, *star*, tl*, bl*], wobei :

- α der Name des gepunkteten Baumes ist, der diesen Zustand beschreibt.
- *side* angibt, auf welche Seite des Symbols der Punkt sich befindet. *side* $\in$ {links, rechts}
- pos die Position des Punktes am Symbol angibt ; pos $\in$ {unten, oben} .
- L ein Index im Eingabewort ist, der angibt, wo die Ableitung von α begann.

z.B. : wenn L = 2, dann können wir daraus schließen, daß a_2 schon *gescannt* war, als die Ableitung von α begann und d.h. wiederum, daß man sich in der Zustandsmenge S_2 befand.

- Fl ein Index im Eingabewort ist, der dem Punkt gleich vor dem Fußknoten eines von α abgeleiteten Baumes entspricht.
- Fr ein Index im Eingabewort ist, der dem Punkt gleich nach dem Fußknoten des von α abgeleiteten Baumes entspricht, falls so ein abgeleiteter Baum existiert, d.h. das Paar fl, fr bedeutet, daß der Fußknoten das Wort a_{fl+1} ....a_{fr} subsumiert.
- *Star* die Adresse des tiefsten Knotens in α ist, an dem eine Adjunktion partiell erkannt wurde, d.h. die linke Seite des auxiliaren Baumes ist erkannt.
- tl* ein Index im Eingabewort ist, der dem mit einem Stern markierten Symbol entspricht.
- bl* ein Index im Eingabewort ist, der dem Punkt gleich vor dem Fußknoten eines Baumes, welcher an einem mit einem Stern markierten Symbol adjungiert wurde, entspricht.





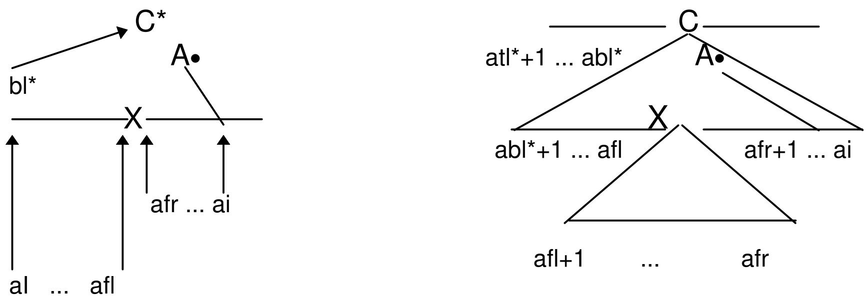


Abbildung 8.

Definition 4:

adjunk (α, Adresse) ist die Menge aller auxiliaren Bäume, die an dem Knoten α (Adresse) adjungiert werden können.

2.2.2. Die verschiedenen Prozesse des Algorithmus :

Nachdem wir jetzt die zum Verstehen des Algorithmus nötigen Begriffe definiert haben, werde ich die sieben Prozesse des Algorithmus einführen :

1) *Scanner* Prozeß: der *Scanner* Prozeß wird gestartet, wenn der Punkt an die Position la eines Terminalsymbols kommt. Der *Scanner scannt* das Eingabewort. Falls das gescannte Symbol dem gepunkteten Symbol entspricht, dann registriert das Programm, daß ein neues Symbol erkannt wurde, und der Punkt wird an die Position ra verschoben.

2.2.2.1 Der *Scanner* Prozeß :

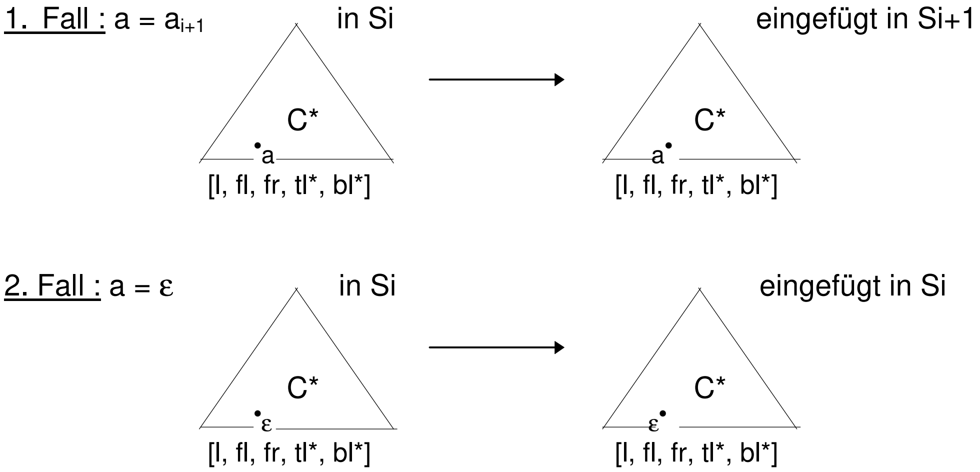
Angenommen, wir sind in der Zustandsmenge S_i. Der *Scanner* Prozeß wird für die folgenden Zustände durchgeführt. S = [α, Punkt*, left, above*, l, fl, fr, *star*, tl*, bl*], wobei α (Punkt) ein Terminalsymbol ist und α (Punkt) = a_{i+1} oder α (Punkt) das leere Wort ε ist.

<u>1. Fall : α (Punkt) = a_{i+1}</u> (da wir in der Zustandsmenge S_i sind, bedeutet dies, daß wir a_i schon *gescannt* und erkannt haben).

In diesem Fall fügt der S*canner* S = [α *,* Punkt*, right, above*, l, fl, fr, *star*, tl*, bl*], in die Zustandsmenge S_{i+1} ein.

<u>2. Fall : α (Punkt) = ε</u>

In diesem Fall fügt der *Scanner* [α, Punkt*, right, above*, l, fl, fr, *star,* tl*, bl*] in die Zustandsmenge S_i ein.



<u>Abbildung 9.</u>

<u>2.2.2.2. Der *Move Dot Down* Prozeß :</u>

Dieser Prozeß hilft bei dem Traversieren des gepunkteten Baumes.

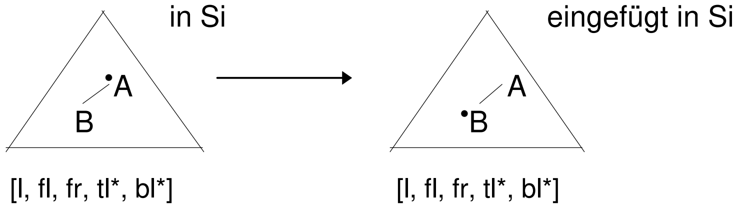
Der Move Dot Down wird gestartet, wenn der Punkt sich an der Position lb eines Nichtterminalsymboles befindet. Dieser Prozeß bewegt den Punkt an die Position la seines am weitesten links stehenden Sohnes.

Formale Beschreibung :

Der *Move Dot Down* Prozeß wird für die folgenden Zustände durchgeführt :

S= [α, Punkt*, left, below*, l, fl, fr, *star*, tl*, bl*], wobei der Knoten, an dem der Punkt sich befindet, seinen am weitesten links stehenden Sohn in der Adresse u hat.

Move Dot Down fügt S = [α, u, *left, above*, l, fl, fr, *star*, tl*, bl*] in die Zustandsmenge S_i ein.



<u>*Abbildung 10.*</u>

<u>2.2.2.3. Der *Move Dot Up* Prozeß :</u>

Der *Move Dot Up* Prozeß bewegt den Punkt von der Position ra des gepunkteten Knotens an die Position la seines rechten Bruders, wenn solch ein Bruder existiert. Sonst wird der Punkt an die Position rb seines Vaters bewegt.

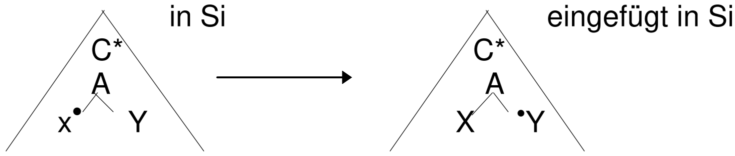
Formale Beschreibung :
Move Dot Up wird für die folgenden Zustände durchgeführt : S = [α, Punkt*, right*, *above*, l, fl, fr, *star*, tl*, bl*], wobei der gepunktete Knoten einen Vater hat.

Zum *Move Dot Up* Prozeß betrachten wir zwei Fälle :

<u>1. Fall:</u> wenn der gepunktete Knoten einen rechten Bruder in der Adresse r hat, dann wird der *Move Dot Up*-Prozeß den Zustand S = [α, r, *left, above,* l, fl, fr, *star*, tl*, bl*] in die Zustandsmenge S_i einfügen.

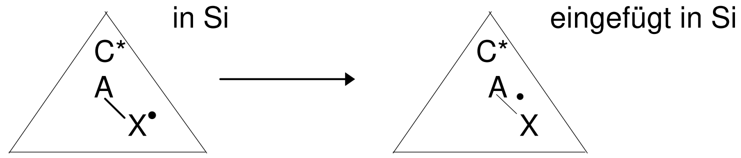
<u>2. Fall:</u> wenn der gepunktete Knoten selbst der am weitesten rechts stehende Sohn eines Vaters p ist, dann wird der Zustand S = [α, p, *right, below*, l, fl, fr, *star*, tl*, bl*] in die Zustandsmenge S_i eingefügt.

<u>1. Fall : X hat einen rechten Bruder</u>



[l, fl, fr, tl*, bl*] [l, fl, fr, tl*, bl*]

2. Fall : X ist der am weitesten rechts stehende Sohn



[l, fl, fr, tl*, bl*] [l, fl, fr, tl*, bl*]

Abbildung 11.

2.2.2.4. Der *Left Predictor* Prozeß :

Der *Left Predictor* Prozeß wird gestartet, wenn sich der Punkt an der Position la eines Nichtterminalsymbols befindet. Der Algorithmus verfolgt zwei Wege gleichzeitig :

Er nimmt an, daß eine Adjunktion an dem gepunkteten Symbol stattgefunden hat und versucht, den adjungierten Baum zu erkennen. Schritt 1.

Er betrachtet auch den Fall, daß keine Adjunktion stattgefunden hat. Schritt 2.

Formale Beschreibung der zwei Schritte :
Der *Left Predictor* Prozeß wird für die folgenden Zustände durchgeführt : S = [α, Punkt, *left, above*, l, fl, fr, *star*, tl*, bl*], wobei α (Punkt) ein Nichtterminalsymbol ist.

1. Schritt

Der *Left Predictor* fügt folgende Zustände in die Zustandsmenge S_i ein.
{ [β, 0, *left, above*, i, -, -, -, -, -] | $\beta \in$ {Adjunkt (α, Punkt)}}.

Erklärung :
Der *Left Predictor* muß für jeden Baum, der an dem Knoten adjungiert werden kann, einen Zustand kreieren, der den Baum beschreibt.

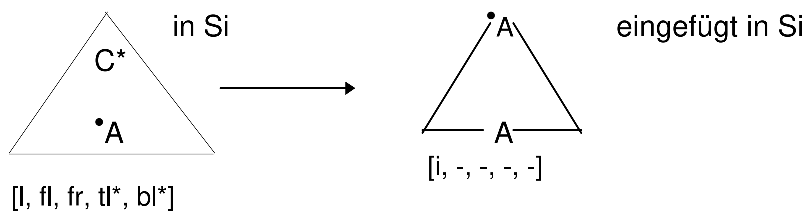
0 an der zweiten Stelle des Tupels weist darauf hin, daß sich der Punkt an der Wurzel des Baumes β befindet.

L = i : wir waren in der Zustandsmenge S_i, als wir den neuen Baum *left predicted* haben, d.h. wir haben bis a_i *gescannt* und erkannt und d.h. wiederum, daß die Ableitung des neuen Baumes nach i angefangen hat.

Der Rest des Tupels bezieht sich auf den neuen *left predicted* Baum .

Da wir aber gerade mit der Ableitung des Baumes angefangen haben, wissen wir nicht, was vor oder nach seinem Fußknoten steht.

Wir wissen auch nichts über den Baum, der von β abgeleitet werden wird, oder ob solch ein Baum überhaupt abgeleitet wird. Deshalb ist der Rest des Tupels unbeschränkt (siehe Abbildung 12).

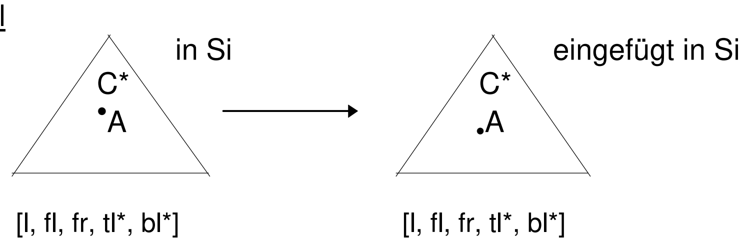


<u>*Abbildung 12.*</u>

<u>2. Schritt :</u>

- 1. Fall : der Punkt liegt nicht an einem Fußknoten. In diesem Fall wird der folgende Zustand S´= [α, Punkt*, left, below*, l, fl, fr, *star*, tl*, bl*] in die Zustandsmenge S_i eingefügt (siehe Abbildung 13).

<u>1. Fall</u>



<u>*Abbildung 13.*</u>

Da der Punkt nicht an einem Fußknoten liegt, wissen wir nichts Neues über die Werte fl, fr, *star*, tl*, bl*. Deshalb werden sie von S übernommen.

- 2. Fall: falls der Punkt an einem Fußknoten liegt, dann heißt das, daß der Fußknoten noch nicht traversiert wurde. Daraus folgt, daß fl und fr in S unbeschränkt sind.

Der Punkt ist jetzt an dem Fußknoten. Daraus folgt, daß fl jetzt definiert ist und zwar mit fl = i, während fr noch unverändert unbeschränkt ist.

In diesem Fall wird der Zustand S´= [α, Punkt*, left, below*, l, i, -, *star*, bl*, tl*] in die Zustandsmenge S_i eingefügt (siehe Abbildung 14).

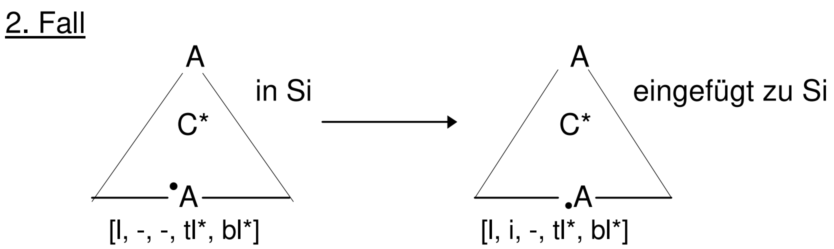


Abbildung 14.

2.2.2.5. Der *Left Completor* Prozeß :

Der *Left Completor* Prozeß wird gestartet, wenn der Punkt an den Fußknoten eines vorher *left predicteten* auxiliaren Baumes (α) ankommt.

Der Algorithmus betrachtet die Zustandsmenge, in der der Baum *left predicted* wurde und fügt dort neue Informationen zu allen Bäumen hinzu, an denen der *left predicted* Baum adjungiert werden könnte.

Formale Beschreibung :

Der *Left Completor* wird für die folgenden Zustände durchgeführt : S = [α, Punkt, *left*, *below*, l, i, -, *star*, tl*, bl*], wobei der Punkt an einem Fußknoten liegt.

Für alle S´ = [β, Punkt*', left, above*, l', fl', fr', *star*', tl*', bl*'] in S_l ,wobei $\alpha \in$ adjunk (β, Punkt')

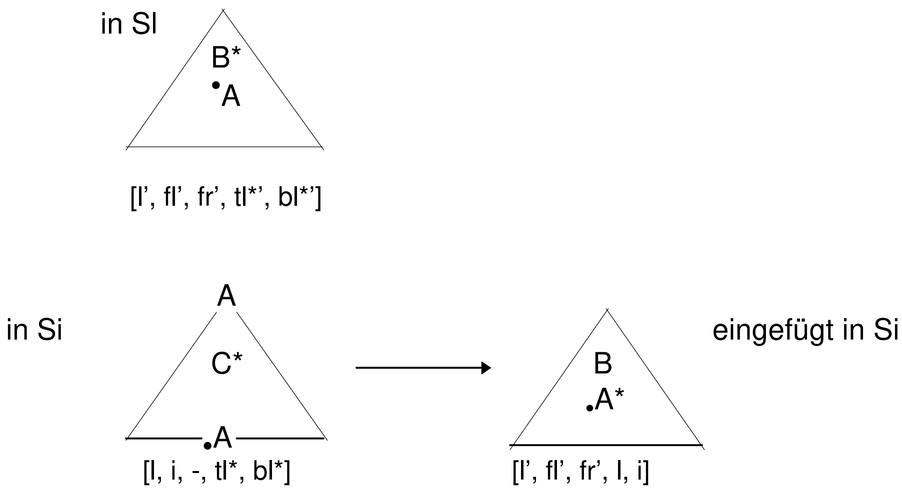
- 1. Fall: Punkt' liegt an dem Fußknoten von β.

In diesem Fall ist es offensichtlich, daß fl', fr' unbeschränkt sind.

Der *Completor* fügt den folgenden Zustand [β, Punkt*', left, below*, l', i, -, Punkt', l, i] in die Zustandsmenge S_i ein.

- 2. Fall: Punkt' liegt nicht am Fußknoten von β.

Der *Completor* fügt den folgenden Zustand [β, Punkt*', left, below*, l', fl', fr', Punkt', l, i] in die Zustandsmenge S_i ein (siehe Abbildung 15).



<u>*Abbildung 15.*</u>

Erklärungen: im Fall 1 wird fl' durch i ersetzt, weil wir jetzt wissen, was es vor dem Fußknoten gibt. Fr ist immer noch unbeschränkt. *star* = Punkt', da wir zuletzt den Baum, der an Punkt' adjungiert wurde, erkannt haben.

tl*, bl* beschreiben normalerweise die Wurzel und den Fußknoten des adjungierten auxiliaren Baumes.

In unserem Fall tl* = l ,da l zeigt, wo man anfing, den Baum abzuleiten (d.h. l = Wurzel des Baumes).

bl* = i , da wir am Fußknoten sind und bis jetzt nur bis a_i *gescannt* und erkannt haben.

2.2.2.6. Der *Right Predictor* Prozeß :

Der *Right Predictor* Prozeß wird gestartet, wenn der Punkt an der Position rb eines Symbols steht.

Wenn an diesem Symbol eine Adjunktion stattgefunden hat, dann versucht der Algorithmus, den rechten Teil des adjungierten Baumes zu erkennen. Falls aber keine Adjunktion am Symbol stattgefunden hat, dann wird der Punkt einfach an die Position ra des Symbols verschoben.

Man kann leicht feststellen, ob eine Adjunktion am Symbol stattgefunden hat oder nicht, indem man einfach die *Star*-Information liest.

Formale Beschreibung :

Der *Right Predictor* wird für die folgenden Zustände S = [α, Punkt, *right, below*, l, fl, fr, *star*, tl*, bl*] durchgeführt :

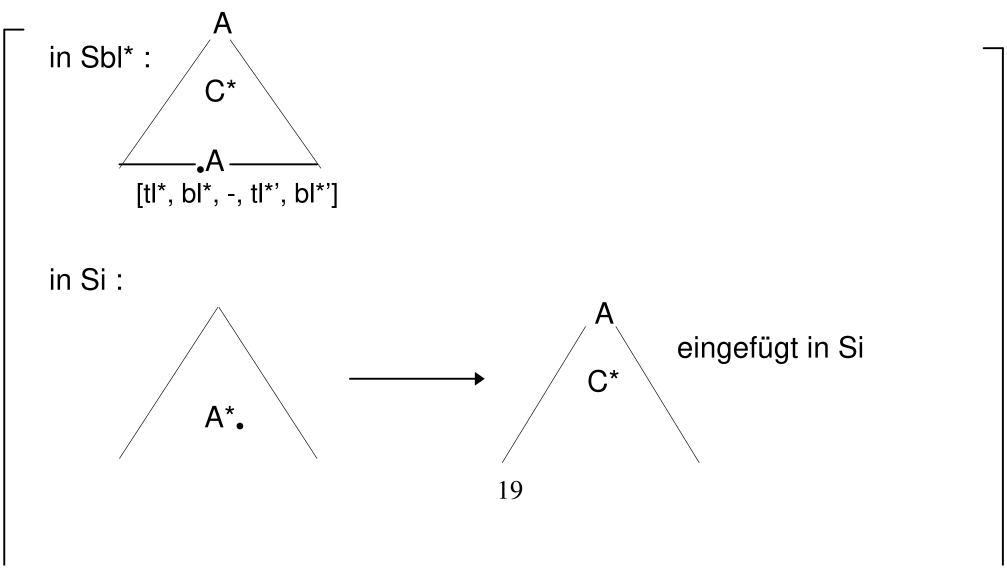
- 1. Fall : Punkt = *Star*

Für alle Zustände S' = [β, Punkt*', left, below*, tl*, bl*, -, *star'*, tl*', bl*'] in S_{bl*} ,wobei $\beta \in$ adjunk (α, Punkt), fügt der *Right Predictor* den Zustand [β, Punkt*', right, below*, tl*, bl*, i, *star'*, t l*', bl*'] in die Zustandsmenge S_i ein.

- 2. Fall : Punkt $\neq$ *Star*

Der *Right Predictor* fügt den Zustand [α, Punkt*, right, above*, l, fl, fr, *star*, tl*, bl*] in die Zustandsmenge S_i ein (siehe Abbildung 16).

1. Fall



2. Fall

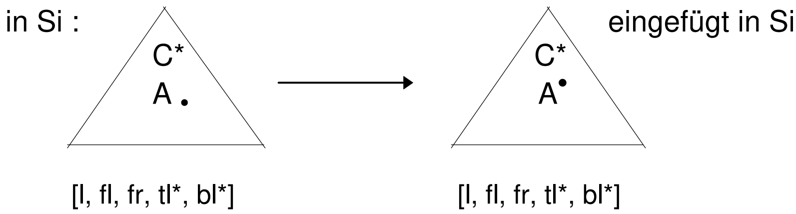


Abbildung 16.

Erklärungen :

- 1. Fall : wir gehen in die Zustandsmenge S_{bl^*} , da sich dort die Zustände befinden, die den linken Teil des Baumes beschreiben. Wir kopieren diese Zustände in die Zustandsmenge S_i und setzen dann den unbeschränkten Wert fr mit i gleich.
- 2. Fall : da man weiß, daß keine Adjunktion am Knoten stattgefunden hat, bleibt nur die Möglichkeit, den Punkt an die Position ra zu schieben.

2.2.2.7. Der *Right Completor* Prozeß :

Der *Right Completor* Prozeß wird gestartet, wenn der Punkt an der Position ra der Wurzel eines auxiliaren Baumes steht.

Das heißt, daß der adjungierte Baum jetzt komplett erkannt worden ist. In diesem Fall versucht der *Right Completor,* den Vaterbaum des gerade komplett erkannten adjungierten Baumes zu erkennen.

Formale Beschreibung :

Der *Right Completor* wird für den Zustand [α, 0, *right, above*, l, fl, fr, -, -,-] durchgeführt.

Für alle Zustände S_l = [β, Punkt *', left, above*, l', fl ', fr', *star'*, tl*', bl*] in S_l und für alle Zustände [β, Punkt*', right, below*, l', fl', fr', Punkt', l, fl] in S_{fr} , wobei $\alpha \in$ adjunk (β, Punkt'),fügt der *Right Completor* den Zustand [β, Punkt*', right, above*, l', fl', fr' , *star'*, tl*', bl*'] in die Zustandsmenge S_i ein, wobei f = f, falls f in S_l beschränkt ist. Ansonsten kann f irgend einen Wert haben (siehe Abbildung 17).

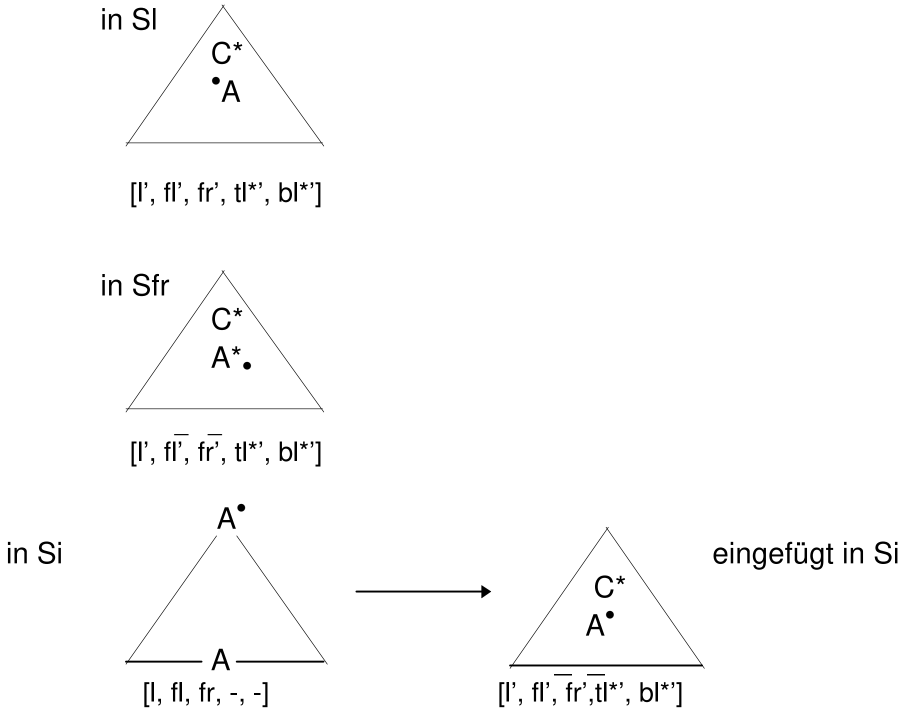


Abbildung 17.

Erklärung :

Der *Right Completor* wird für [α, 0, *right, above*, l, fl, fr, -, -, -] durchgeführt.

Punkt = 0, d.h der Punkt ist an der Wurzel.

Star ist unbeschränkt, da *Star* den am tiefsten stehenden adjungierten und erkannten von α abgeleiteten Baum markiert.

bl*, tl* sind unbeschränkt, da *Star* unbeschränkt ist.

Der *Completor* betrachtet die Zustände [β, Punkt *', left, above*, L', fl', fr'*, star* , tl*, bl*] in S_l , weil wir wissen, daß die erste *Prediction* des Baumes in S_l stattgefunden hat und zwar an eben diesen Bäumen, die durch die oben erwähnten Zustände beschrieben sind. Wir kopieren diese Zustände in die Zustandsmenge S_i und ändern dabei die Position des Punktes. Der Punkt wird an die Position ra verschoben.

Falls fl', fr' in S_l unbeschränkt sind, dann holt man die neuen Werte von den Zuständen [β, Punkt*', right, below*, l', fl', fr', Punkt', l, fl] in S_{fr} . Das heißt, daß sich der Algorithmus von S_l die Stern-Informationen und von S_{fr} die Werte von fl und fr holt.

2.2.3. Die GPE-Version des Algorithmus :

Seien G ein TAG und $a_1 \ldots a_n$ das Eingabewort.

Programm :

begin

S_0 = { [α, 0, *left, above*, 0, -, -, -, -, -] | α ist ein Initialbaum }

for *i :* = 0 to *n* do

> ***begin***
>
> Bearbeite alle Zustände der Zustandsmenge S_i ,indem du einen der folgenden sieben Prozesse für jeden Zustand S = [α, Punkt, *side*, pos, l, fl, fr, *star*, tl*, bl*] durchführst, bis kein neuer Zustand mehr eingefügt werden kann :
>
> 1. *Scanner*
> 2. *Move Dot Down*
> 3. *Move Dot Up*
> 4. *Left Predictor*
> 5. *Left Completor*
> 6. *Right Predictor*
> 7. *Right Completor*
>
> Falls S_{i+1} leer ist und i < n, dann *return* abgelehnt
>
> ***end***

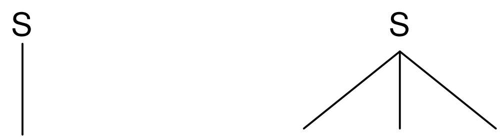
Wenn es in der Zustandsmenge S_n einen Zustand S = [α, 0, *right, above*, 0, –, –, –, –, –] gibt, so daß α ein Initialbaum ist, dann *return* akzeptiert.

end *.*

2.2.4. Beispiel :

In diesem Abschnitt gebe ich ein Beispiel, das darstellt, wie der Algorithmus funktioniert.

In diesem Beispiel benutzen wir die folgende Grammatik :



(α) (β)

e a S b

α ist Initialbaum und β ist auxiliaren Baum. Als Eingabewort nehmen wir das Wort aeb.

In dem Beispiel schreibe ich nach jedem Zustand den Namen des Prozesses, der für den Zustand durchgeführt werden kann und die Nummer der Zustände, die entstehen.

S_0 enthält folgende Zustände :

Wir fangen mit dem Initialzustand Zustand (0) = [α, 0, *left, above*, 0, -, -, -, -, -] an.

Zustand (0) = [α, 0, *left, above,* 0, -, -, -, -, -] *(Left Predictor)* $\Rightarrow$ (1), (2)

Zustand (1) = [β, 0, *left, above,* 0, -, -, -, -, -] *(Left Predictor)* $\Rightarrow$ (1), (3)

Zustand (2) = [α, 0, *left, below,* 0, -, -, -, -, -] *(Move Down)* $\Rightarrow$ (4)

Zustand (3) = [β, 0, *left, below,* 0, -, -, -, -, -] *(Move Down)* $\Rightarrow$ (5)

Zustand (4) = [α, e, *left, above,* 0, -, -, -, -, -] *(Scanner)* $\Rightarrow$ Stop

Zustand (5) = [β, a, *left, above,* 0, -, -, -, -, -] *(Scanner)* $\Rightarrow$ (6)

Zustand (6) wird in die Zustandsmenge S_1 eingefügt. S_1 :

Zustand (6) = [β, a, *right, above,* 0, 1, -, -, -, -] *(Move Dot Up)* $\Rightarrow$ (7)

Zustand (7) = [β, S, *left, above,* 0, 1, -, -, -, -] *(Left Predictor)* $\Rightarrow$ (8), (9)

(Annahme : eine Adjunktion hat stattgefunden)

Zustand (8) = [β, 0, *left, above,* 1, 1, -, -, -, -] *(Left Predictor)* $\Rightarrow$ (8), (10)

(Annahme : keine Adjunktion hat stattgefunden)

Zustand (9) = [β, S, *left, below*, 0, 1, -, -, -, -] *(Left Completor)* $\Rightarrow$ (10)

Zustand (10) = [α, S, *left, below,* 0, -, -, S, 0, 1] *(Move Dot Down)* $\Rightarrow$ (11)

Zustand (11) = [α, e, *left, above,* 0, -, -, S, 0, 1] *(Scanner)* $\Rightarrow$ (12)

Zustand (12) wird in die Zustandsmenge S_2 eingefügt. S_2 :

Zustand (12) = [α, e, *right, above,* 0, 2, -, S, 0, 1] *(Move Dot Up)* $\Rightarrow$ (13)

Zustand (13) = [α, S, *right, below,* 0, 2, -, S, 0, 1] *(Right Predictor)* $\Rightarrow$ (14)

Zustand (14) = [β, S, *right, below,* 0, 1, 2, -, -, -] *(Right Predictor)* $\Rightarrow$ (15)

Zustand (15) = [β, S, *right, above,* 0, 1, 2, -, -, -] *(Move Dot Up)* $\Rightarrow$ (16)

Zustand (16) = [β, b, *left, above,* 0, 1, 2, -, -, -] *(Scanner)* $\Rightarrow$ (17)

Zustand (17) wird in die Zustandsmenge S_3 eingefügt. S_3 :

Zustand (17) = [β, b, *right, above,* 0, 1, 3, -, -, -] *(Move Dot Up)* $\Rightarrow$ (18)

Zustand (18) = [β, S, *right, below,* 0, 1, 3, -, -, -] *(Right Predictor)* $\Rightarrow$ (19)

Zustand (19) = [β, S, *right, above,* 0, 1, 3, -, -, -] *(Right Completor)* $\Rightarrow$ (20)

Zustand (20) = [α, 0, *right, above,* 0, -, -, -, -, -] $\Rightarrow$ Stop (Eingabewort erkannt).

3. DIE BEHANDLUNG VON *CONSTRAINTS* DURCH MODIFIZIERUNG DES ALGORITHMUS :

Man kann für jeden Knoten eines elementaren Baumes eine der folgenden *Constraints* spezifizieren.

1) obligatorische Adjunktion (OA (T)), d.h., wenn an einem Knoten OA steht, dann muß man an diesem Knoten einen auxiliaren Baum aus der Menge T adjungieren.

2) selektive Adjunktion (SA (T)), d.h. an dem mit SA (T) versehenen Knoten können nur auxiliare Bäume aus der Menge T adjungiert werden.

3) Null-Adjunktion (NA = SA ($\emptyset$)), d.h. an dem Knoten, der mit NA versehen ist, darf kein auxiliarer Baum adjungiert werden.

Der vorherige Algorithmus kann leicht modifiziert werden, um solche *Constraints* zu behandeln. Zuerst muß die Funktion (α, Adresse) folgendermaßen modifiziert werden :

- Adjunk (α, Adresse) = $\emptyset$, falls NA an dem Knoten steht, der sich in der Adresse von α befindet.
- Adjunk (α, Adresse) = T , falls OA (T) an dem Knoten steht, der sich in der Adresse von α befindet.
- Adjunk (α, Adresse) = T , falls SA (T) an dem Knoten steht, der sich in der Adresse von α befindet.

- Adjunk (α, Adresse) = die Menge der auxiliaren Bäume, deren Wurzel mit dem gleichen Symbol bezeichnet sind, wie der Knoten an der Adresse in α, falls es an dem Knoten keine *Constraints* gibt.

Zweitens muß man folgende Prozesse leicht modifizieren :

- *Left Predictor* :

Für den *Left Predictor* betrachten wir drei Fälle:

- Fall 1: NA steht am *left predicted* Knoten. In diesem Fall darf der *Left Predictor* nur seinen zweiten Schritt durchführen.
- Fall 2: OA (T) steht am *left predicted* Knoten. In diesem Fall darf der *Left Predictor* nur seinen ersten Schritt durchführen.
- Fall 3: SA (T) steht am *left predicted* Knoten. In diesem Fall darf der *Left Predictor* nur die Bäume, die in der selektierten Menge A sind, *left predicten*.

- *Right Predictor* :

Der *Right Predictor* muß seinen ersten Schritt durchführen, wenn OA an dem *right predicted* Knoten steht.

Der *Right predictor* braucht nicht seinen ersten Schritt durchzuführen, wenn es NA *Constraint* an dem *right predicted* Knoten gibt.

Es ist klar, daß diese Änderungen die Arbeit unseres *Aktzeptators* erleichtern, denn dieser kann durch die *Constraints* auf viele parallele Schritte verzichten.

4. DIE BEHANDLUNG VON SUBSTITUTIONEN :

Substitutionen können auf einem Blattknoten durchgeführt werden.

Ein Substitutionsknoten wird mit einem Pfeil markiert. Falls ein Knoten mit einem Pfeil markiert ist, darf man dann an ihm keine Adjunktion durchführen .

Substitutionen sind immer obligatorisch, d.h. wenn ein Knoten mit einem Pfeil markiert ist, muß man dann den Knoten substituieren.

Ein Substitutionsknoten darf nur mit einem von einem Initialbaum abgeleiteten Baum, dessen Wurzel die gleiche Markierung hat wie der Substitutionsknoten, substituiert werden.

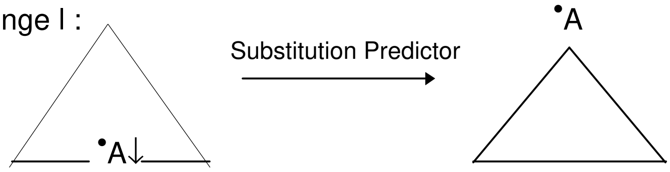
Der Parser, den wir vorher dargestellt haben, kann leicht modifiziert werden, um Substitutionen zu behandeln.
Der *Left Predictor* wird nur für Knoten benutzt, die nicht mit einem Pfeil markiert sind.
Alle anderen Prozesse verhalten sich wie oben beschrieben.
Wir werden jetzt zwei neue Prozesse durchführen und den Zustand S neu definieren. Zu einem Zustand S werden wir noch eine neue Komponente addieren. Die 11te Komponente wird mit sub besetzt, wobei sub ein boolscher Wert ist, der anzeigt, ob der Baum für eine Substitution *predicted* wurde oder nicht.

Substitution Predictor :
Angenommen, es gibt einen Punkt an der Position la eines Blattknotens A, das mit einem Pfeil markiert ist :

Der *Predictor* wird für diese Substitution alle Initialbäume, deren Wurzel A ist, *predicten* und dann versuchen, sie zu erkennen.

Substitution Completor :
Angenommen, der Initialbaum, den wir für die Substitution *predicted* haben, ist schon erkannt worden :
Der Algorithmus versucht, den Rest des Baumes, in dem wir ein Substitution *predicted* haben, zu erkennen (siehe Abbildung 18).

In Zustandsmenge I :



[-, -, -, -, subst?’]   [-, -, -, -, true]

In Zustandsmenge l :

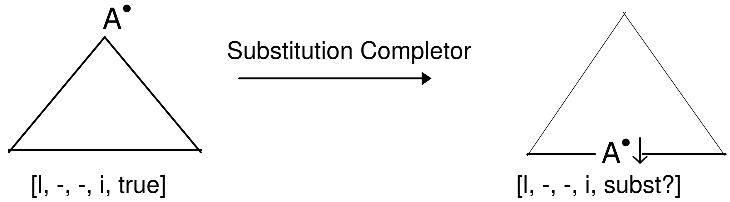


Abbildung 18.

Formale Beschreibung :

Substitution Predictor :

Der *Predictor* wird für den folgenden Zustand [α, Punkt, *left, above*, l, fl, fr, st, tl*, bl*,sub?] durchgeführt, wobei α (Punkt) ein Nichtterminal ist, das mit einem Pfeil markiert ist .

Der *Predictor* fügt folgende Zustände { [β, 0, *left, above*,l, -, -, *star*, tl*, bl*, *true*] | β ein Initialbaum ist und β (0) = α (Punkt) } in die Zustandsmenge S_i ein.

Substitution Completor :

Der *Completor* wird für die folgenden Zustände S = [α, 0, *right, above*, l, -, -, star, tl*, bl*, true] eingeführt.

Für alle Zustände S’ = [α’, Punkt’, *left, above*, l, -, -, star, tl*, bl*,Sub?] in S_l , wobei α’(Punkt’) mit einem Pfeil markiert ist und α’ (Punkt’) = α (0).

Der Algorithmus fügt die Zustände [α’, Punkt‘, *right, above*, l, -, -, Sub?] in die Zustandsmenge S_i ein.

Nachdem wir den Algorithmus komplett eingeführt haben und seine Erweiterung gezeigt haben, möchte ich auf die Komplexität dieses Algorithmus eingehen.

5. DIE KOMPLEXITÄTSANALYSE :

5.1 Die Laufzeit - Komplexität :

Unser Algorithmus hat lange Laufzeiten im schlechtesten Fall im Vergleich zu anderen *Parsing*-Algorithmen. Dafür behält er die sogenannte Gültige-Präfix-Eigenschaft bei.
Gültige-Präfix-Eigenschaft :
Parser, die die Gültige-Präfix-Eigenschaft beibehalten, garantieren, daß das schon *gescannte substring* ein gültiges Präfix der Sprache ist.

Formale Beschreibung :
Angenommen, wir haben das Eingabewort $a_1\ a_2 \ldots a_k\ a_{k+1} \ldots a_n$.
Wenn der Algorithmus einen Teil des Eingabewortes - sagen wir $a_1 \ldots a_k$ - gescannt und erkannt hat, dann garantiert der Algorithmus, daß es ein Suffix $b_1 \ldots b_m$ gibt, der nicht unbedingt ein Teil des Eingabewortes ist, so daß $a_1 \ldots a_k\ b_1 \ldots b_m$ ein Wort der Sprache ist.

Diese Eigenschaft verschlechtert die *worst case*-Laufzeit und verbessert die a*verage case*-Laufzeit.
Um die Gültige-Präfix-Eigenschaft beizubehalten, muß der Algorithmus alle *Top Down*-Informationen nutzen und bearbeiten. Das kostet den Algorithmus viel Zeit. Deshalb wird die *worst case*-Laufzeit des Algorithmus verschlechtert.
Im Gegensatz zu der *worst case*-Laufzeit wird die *average*-Laufzeit verbessert, denn der Algorithmus bemerkt sehr schnell, daß ein Eingabewort nicht zur Sprache gehört, ohne daß er das ganze Eingabewort gescannt hat. In diesem Fall hört der Algorithmus mit dem *scannen* auf und lehnt das Eingabewort ab.
Bevor wir jetzt mit der formalen Komplexitätsanalyse anfangen, müssen wir ein paar Ausdrücke, die wir in unserer Analyse brauchen, erklären.

$||G||_k = \sum |\gamma|^k$, wobei $|\gamma|$ die Anzahl der Knoten im Baum ist.

$\gamma \in I \cup A$

$||G||_0$ ist die Anzahl der Bäume in der Grammatik.

$||G||_1$ ist die Anzahl der Knoten in der Grammatik.

Behauptung :

Der vorher eingeführte Algorithmus braucht im worst case $O(||G||_0 \, ||G||_2 \, n^9)$ Zeit und $O(||G||_2 \, n^6)$ Speicherplatz.

Beweis :

Wir werden zuerst berechnen, wieviel Zeit der Parser im worst case braucht, um eine Zustandsmenge komplett abzuarbeiten.

Wir tun das, indem wir jeden der sieben Prozesse einzeln betrachten und berechnen, wie viele neue Zustände er in die Zustandsmenge einfügt. Wir müssen auch den Typ dieser Zustände betrachten, wobei man unter Typ eines Zustandes den Prozeß, der für den Zustand durchgeführt werden muß, versteht.

Das heißt, daß man sieben Typen { sc(i), mdd(i), mdu(i), lp(i),l c(i), rp(i),rc(i) } hat .

Berechnung der *worst case*-Laufzeit einer Zustandsmenge S_i :

1) *Scanner* :

Der *Scanner* wird für $O(||G||_2 \, i^5)$ Zustände durchgeführt.

$||G||_2$ kommt daher, daß Punkt und *star* relevant für diesen Prozeß sind.

i^5 kommt daher, daß l, fl, fr, bl*, tl*$\in$ [0 .....i] für diesen Prozeß relevant sind .

Der *Scanner* fügt zu jedem Zustand maximal einen Zustand vom Typ mdu(i) hinzu. Daraus folgt, daß der *Scanner* maximal $O(||G||_2 \, i^5 \, mdu(i))$ braucht.

2) *Move Dot Up* :

Dieser Prozeß wird für $O(||G||_2 \, i^5)$ Zustände durchgeführt (aus demselben Grund wie beim *Scanner*).

Zu jedem Zustand wird maximal ein Zustand vom Typ lp(i) oder rp(i) hinzugefügt. Daraus folgt, daß der *Move Dot Up* maximal $O(||G||_2 \, i^5 \max(lp(i), rp(i)))$ braucht.

3) *Move Dot Down* :

Wie bei *Move Dot Up* wird dieser Prozeß für maximal O($||G||_2$ i^5) Zustände durchgeführt. Zu jedem Zustand wird maximal ein Zustand vom Typ lp(i) oder vom Typ sc(i) hinzugefügt. Daraus folgt, daß dieser Prozeß maximal O($||G||_2$ i^5 max (lp(i), sc(i))) braucht.

4) *Left Predictor* :

Schritt 1 wird für maximal O($||G||_1$) Zustände durchgeführt, da die Zustände die *left predicted* sind, nur von α und der Position des Punktes abhängig sind, d.h. *Star* wird nicht berücksichtigt.

Bei diesem Schritt ist die Anzahl der neu eingefügten Zustände unabhängig von i, denn der *Left Predictor left predicted* eine Menge von Bäumen, die an dem Knoten adjungiert werden können. Daraus folgt, daß die Anzahl der neu eingefügten Zustände von der Anzahl der Bäume in der Grammatik abhängig ist.

Die neu eingefügten Zustände sind vom Typ lp(i). Daraus folgt, daß Schritt 1 maximal O ($||G||_0$ $||G||_1$ lp(i)) braucht.

Schritt 2: In dem Fall, in dem keine Adjunktion stattgefunden hat, haben wir zwei Subschritte.

1. Subschritt : der Punkt liegt nicht am Fußknoten.

Dieser Subschritt kann maximal für O($||G||_2$ i^5) Zustände durchgeführt werden und zu jedem dieser Zustände wird genau ein *Move Dot Down* Zustand hinzugefügt. Daraus folgt, daß dieser Subschritt maximal O($||G||_2$ i^5 mdd(i)) Zeit brauchen wird.

Erklärung :

Der Wert $||G||_2$ kommt daher, daß *star* und Punkt relevant sind.

Der Wert i^5 kommt daher, daß die Werte l, fl, fr, bl*, tl* alle relevant sind und variieren in dem Bereich [0....i].

2. Subschritt : der Punkt liegt an einem Fußknoten.

Dieser Subschritt kann maximal für O($||G||_1$ i^3) Zustände durchgeführt werden, und zu jedem Zustand wird ein neuer Zustand vom Typ lc(i) hinzugefügt. Daraus folgt, daß dieser Subschritt maximal O($||G||_1$ i^3 lc(i)) Zeit braucht.

Erklärung :

$||G||_1$ kommt daher, daß in diesem Subschritt nur Star relevant ist; der Punkt ist irrelevant.

Der Wert i^3 kommt daher, daß I, bl*, tl* relevant sind ; fr, fl in diesem Fall irrelevant, da sie unbeschränkt sind.

5) *Left Completor* :

Die Anzahl der Zustände, für die der *Left Completor* durchgeführt werden kann, hängt von der Anzahl der vorher *left predicted* Bäume und von I ab (I bezeichnet die Zustandsmenge, in welcher die *Left Prediction* stattgefunden hat). Daraus folgt, daß der *Left Completor* maximal für O($||G||_0$ i) Zustände durchgeführt wird. Zu jedem dieser Zustände wird maximal O ($||G||_1$ i^3) Zustände vom Typ mdd(i) oder vom Typ lc (i) hinzugefügt (wegen Star, fl, bl*,tl* ; fr = - wird nicht betrachtet). Daraus folgt, daß der *Left Completor* maximal O($||G||_1$ $||G||_0$ i^4 max (mdd(i),lc(i))) Zeit braucht.

6) *Right Predictor* :

- 1. Schritt: Punkt = *star*

Die Zustände, für die der 1. Schritt durchgeführt werden kann, sind von *star*, tl* , bl* abhängig. Daraus folgt, daß Schritt 1 maximal für O($||G||_1$ i^2) Zustände durchgeführt wird.

Zu jedem dieser Zustände wird maximal O($||G||_1$ i^2) vom Typ rp(i) hinzugefügt. Daraus folgt, daß Schritt 1 maximal O($||G||_1$ $||G||_1$ i^4) Zeit braucht.

- 2. Schritt: Punkt ≠ *star* .

Es ist leicht nachvollziehbar, daß dieser Schritt maximal O($||G||_2$ i^5 max (mdu(i), rc(i))) Zeit benötigt.

7) *Right Completor* :

Der *Right Completor* wird für maximal O($||G||_0$ i^3) Zustände durchgeführt, da *star*, bl*, tl*=-; Punkt = 0.

Zu jedem dieser Zustände wird maximal O($||G||_2$ i^5) neue Zustände vom Typ mdu (i) oder rc(i) hinzugefügt. Daraus folgt, daß der *Right Completor* maximal O($||G||_0$ $||G||_2$ i^8 max (mdu (i) ,rc(i))) braucht.

Beim Vergleich der einzelnen Zeiten der sieben Prozesse läßt sich schnell feststellen, daß der *Right Completor* Prozeß die schlechteste Laufzeit hat. Daraus folgt, daß der Algorithmus im schlechtesten Fall $O(||G||_0\ ||G||_2\ i^8\ \ max\ (mdu(i),rc(i)))$ Zeit braucht, um die Zustandsmenge S_i abzuarbeiten.

Analyse von mdu(i) , rc(i)

1) mdu(i)

mdu (i) bezeichnet die Zeit, die man braucht, um einen *Move Dot Up* Zustand einzufügen. Da aber das keine weitere Berechnung benötigt, ist mdu (i) konstant. Daraus folgt mdu (i)= O (1).

2) rc (i)

Da rc Zustand von keinem der anderen Prozesse abhängt, kann man ihn einfügen, ohne daß man weitere Berechnungen benötigt. Daraus folgt rc(i) = (1).

Aus der mdu, rc Analyse folgt, daß der Algorithmus $O(||G||_0\ ||G||_2\ i^8)$ Zeit in seinem schlechtesten Fall benötigt, um die Zustandsmenge S_i abzuarbeiten. Da wir n Zustandsmengen haben, braucht der Algorithmus $0(||G||_0\ \ ||G||_2\ \ i^9)$ Zeit im schlechtesten Fall, um das Eingabewort komplett abzuarbeiten.

5.2. Die Speicherplatz - Komplexität :

Um die Speicherplatz-Komplexität genau zu berechnen, müssen wir wissen, wie viele verschiedene Zustände eine Zustandsmenge S_i enthalten kann.

Da wir einen Zustand als 10-Tupel [α, Punkt*,* *side*, pos, l, fl, fr, *star*, bl*, tl*] definiert haben, ist es offensichtlich, daß eine Zustandsmenge S_i maximal $||G||_2\ \ i^5$ verschiedene Zustände haben kann. ($||G||_2$ wegen Punkt *,star* ; i^5 wegen l, fl, fr, bl*, tl*, die im Bereich [0.....i] variieren).

Da man eine konstante Anzahl von Speicherzellen braucht, um einen Zustand zu speichern. Daraus folgt, daß man $O(\ ||G||_2\ i^5)$ Speicherplatz braucht, um alle Zustände einer Zustandsmenge zu speichern, und da wir n Zustandsmengen haben. Daraus folgt, daß die Speicherplatz-Komplexität im schlechtesten Fall $O(||G||_2\ i^6)$ Speicherplatz beträgt.

6. DIE ZWEITE VERSION DES ALGORITHMUS :

In diesem Abschnitt werde ich nur kurz die 2. Version erwähnen und den Unterschied zu der ersten Version erläutern.
In dieser Version verzichtet man auf manche *Top Down*-Informationen, außerdem wird ein Zustand nur als 7-Tupel definiert. Die *Star*-Informationen fallen in dieser Version komplett weg.

Die üblichen sieben Prozesse des Schabes Algorithmus sind auch in dieser Version vertreten, und sie ähneln bis auf den *Left Completor* den Prozessen der ersten Version.
In dieser Version ist der *Left Completor* eher ein *Predictor* als *Completor,* denn wenn der Algorithmus den linken Teil eines auxiliaren Baumes α erkennt, *predicted* er alle Bäume, an denen α adjungiert werden kann, während in der ersten Version der Algorithmus die Zustandsmenge S_l betrachtet, in der man den Baum *left predicted* hat.
Aus diesem Grund wird die Gültige-Präfix-Eigenschaft in der neuen Version verletzt, denn man addiert nicht nur die Bäume, deren linker Teil mit dem Präfix des Eingabewortes übereinstimmt, sondern auch alle Bäume, an denen α adjungiert werden kann.
Durch der Verlust der Gültigen-Präfix-Eigenschaft verbessert sich die *worst case*-Laufzeit des Algorithmus und wir erhalten $0(n^6)$ Laufzeiten im schlechtesten Fall.
Deshalb wird man je nach Grammatik entscheiden, welche Version man benutzen soll. Das heißt, daß, wenn in manchen Grammatiken die Gültige-Präfix-Eigenschaft eine große Hilfe zu dem Parser bietet, man sich für die erste Version entscheiden soll. Ansonsten nimmt man die zweite Version.

ZUSAMMENFASSUNG

Wir haben zwei Parser für TAGs beschrieben, die auf der Earley-Strategie basieren. Der erste Parser hat im besten Fall lineare Laufzeit und $O(n^9)$ im schlechtesten Fall, aber dafür behält er die Gültige-Präfix-Eigenschaft bei. Der zweite Parser hat $O(n^6)$

Laufzeit im schlechtesten Fall und lineare Laufzeiten für eine große Menge von TAGs.

Wenn man die genaue Beschreibung des Algorithmus betrachtet, merkt man, daß der vorgeführte Algorithmus in Wirklichkeit ein Akzeptator und kein Parser ist.

Aber man kann den Algorithmus leicht modifizieren, um aus dem Akzeptator einen Parser zu machen. Dies erfolgt, indem man in die Zustände noch Zeiger (*Pointer*) einfügt. Diese Zeiger zeigen auf den Zustand, aus dem der aktuelle Zustand entstanden ist.

LITERATURVERZEICHNIS

Yves Schabes und Aravind K. Joshi : An Earley-Type Parsing Algorithm for Tree Adjoining Grammars, ACL'88, Buffalo, NY.

Yves Schabes : Mathematical and Computational Aspects of Lexicalized Grammars, Dissertation, University of Pennsylvania, PA, 1990 (erschienen als Technical Report: MS-CIS-90-48, LINC LAB179)

Earley J.: An efficient Context-Free Parsing Algorithm, Commun, 1970, ACM 13(2):94-102.