Diplomarbeit

Nejat Altug Anis und Muhammed Rasim Sayar

Implementierung von Algorithmen zur Grammatik-basierten Kompression von Second-Order Termen auf der Basis von Singleton Tree Grammars

Sommersemester 2009

Betreuer: Prof. Dr. Manfred Schmidt-Schauß
Arbeitsgruppe Künstliche Intelligenz und Softwaretechnologie
Institut: Fachbereich 12 – Informatik und Mathematik
Goethe–Universität Frankfurt am Main
Abgabetermin: 12. Juni 2009

Erklärung

Das vorliegende Dokument wurde gemäß der unteren Aufteilung in selbständiger Arbeit verfasst von Nejat Altug Anis.
Die Beschreibung der Datengewinnung wurde in Zusammenarbeit mit Muhammed Rasim Sayar angefertigt. Die folgende Tabelle dient der Übersicht.

Kapitel	N.A. Anis	M.R. Sayar
1 Einleitung	x	x
2 Grundlagen	x	
2.1	x	
2.2 bis 2.4		x
3	x	
4		x
5	x	x
5.1	x	x
5.2.1	x	
5.2.2	x	
5.2.3 bis 5.2.5		x
5.2.6 u. 5.2.7	x	
5.2.8 bis 5.2.10		x
5.2.11 u. 5.2.12	x	
5.3	x	x
6	x	x

Alle verwendeten Quellen und Hilfsmittel sind angegeben, nicht erlaubte Hilfsmittel wurden nicht verwendet.

Frankfurt, den 12. Juni 2009

Nejat Altug Anis

Erklärung

Das vorliegende Dokument wurde gemäß der unteren Aufteilung in selbständiger Arbeit verfasst von Muhammed Rasim Sayar.
Die Beschreibung der Datengewinnung wurde in Zusammenarbeit mit Nejat Altug Anis angefertigt. Die folgende Tabelle dient der Übersicht.

Kapitel	N.A. Anis	M.R. Sayar
1 Einleitung	x	x
2 Grundlagen	x	
2.1	x	
2.2 bis 2.4		x
3	x	
4		x
5	x	x
5.1	x	x
5.2.1	x	
5.2.2	x	
5.2.3 bis 5.2.5		x
5.2.6 u. 5.2.7	x	
5.2.8 bis 5.2.10		x
5.2.11 u. 5.2.12	x	
5.3	x	x
6	x	x

Alle verwendeten Quellen und Hilfsmittel sind angegeben, nicht erlaubte Hilfsmittel wurden nicht verwendet.

Frankfurt, den 12. Juni 2009

Muhammed Rasim Sayar

Danksagung

Wir möchten uns ganz herzlich bei Herrn Prof. Dr. Manfred Schmidt-Schauß und Dr. David Sabel bedanken, die uns die Möglichkeit gegeben haben, diese Diplomarbeit zu schreiben. Durch Ihre intensive Betreuung mit vielen Hinweisen, Ratschlägen, Anregungen und schnellen Antworten haben Sie uns tatkräftig unterstützt.
Des Weiteren bedanken wir uns bei Batuhan Güneysu, der uns mit vielen Vorschlägen weitergeholfen und uns immer unterstützt hat. Ein spezieller Dank gilt Marco Haibach, der uns für Rat und Tat zur Seite stand. Ausserdem möchten wir uns bei Igor Geier für die hilfreichen und nützlichen Anmerkungen, sowie Hilfestellungen, während dieser Diplomarbeit bedanken. Christine Lenhart gebührt ein spezieller Dank, die uns durch Anregungen zu einer Schönheitskorrektur verhalf. Bei Bert Besser und Sebastian Behr bedanken wir uns für das Korrekturlesen, das uns sehr geholfen hat. An dieser Stelle möchten wir uns auch bei allen Freunden bedanken, die uns während dieser Diplomarbeit nicht alleine gelassen haben. Von ganzem Herzen bedanke ich mich bei Tatiana, dass sie stets für mich da war und die mich in schweren Zeiten motiviert und mir neue Energie gegeben hat.

Nejat Altug Anis

Ich möchte mich ganz herzlich bei meiner Familie bedanken, die mir dieses Studium ermöglicht und während dieser Zeit mich unterstützt hat. Die Tatsache, dass sie in jeder Phase meines Lebens voll hinter mir stehen, gibt mir den nötigen Rückhalt, meinen Weg zu gehen.

Muhammed Rasim Sayar

Insbesondere gilt mein Dank meiner Familie und allen Bekannten. Vor allem meinen Eltern, die mit Liebe und Sorgfalt immer auf meiner Seite standen und mich stets motivierten niemals aufzugeben.

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

Terme und Termumformungen kennt Jeder aus der Mathematik. Auf diese Weise wird in allen mathematischen Fächern gerechnet. Dabei geht es meistens um die Gleichheit "wirklicher Größen". Die Informatik geht auf die syntaktische Seite der Gleichheit ein, nämlich den Prozess der Umformungen. Welche mathematischen Modelle stecken hinter all diesen Systemen von Rechenregeln? Dies führt zu den Konzepten abstrakter *Reduktionssysteme* und der *Termersetzungssysteme*. In der Tat erweist sich der abstrakte Zugang als kompliziert in den Fragen nach der Terminierung solcher Umformungen, nach der Eindeutigkeit des erzielten Ergebnisses und nach dem Zusammenhang zu der semantischen Sicht von Gleichheit, speziell nach deren Entscheidbarkeit. Ein wichtiger Begriff hierbei ist die *Unifikation*.

1.2 Problemstellung

Die *first-order Unifikation* ist ein Verfahren zur Überprüfung, ob zwei Terme (Gleichungen) ineinander überführbar sind. Dabei wird in den Termen nach Variablen gesucht, die eventuell auf ein Gegenstück abgebildet werden. Es dürfen hierbei nur Variablen auf Funktionen, Konstanten oder andere Variablen unifiziert werden. Bei zwei gegebenen Termen $s \doteq t$ wird nach einer Substitution σ für Variablen gesucht, die in beiden Termen s und t vorkommen, so dass $s\sigma = t\sigma$ gilt. Das Gleicheitszeichen in $s\sigma = t\sigma$ steht für die syntaktische Gleichheit, während das Zeichen in $s \doteq t$ für eine potentielle Gleichheit nach der Unifikation steht.

Beispiel 1.2.1 *Die Gleichung*

$$p(x_1, x_2, x_3) \doteq p(f(a, b), g(x_1, x_1), h(x_2, x_2))$$

hat folgende Lösung:

$$\{x_1 \mapsto f(a,b), x_2 \mapsto g(f(a,b),f(a,b)), x_3 \mapsto h(g(f(a,b),f(a,b)),g(f(a,b),f(a,b)))\}.$$

Das Ergebnis der Unifikation ist bis auf die Umbenennung der Variablen eindeutig und die resultierende Substitution wird als der *allgemeinste Unifikator* betrachtet.

Der naive Ansatz für einen Unifikationsalgorithmus wird in [1] erklärt. Hierbei werden die Terme s und t parallel von links nach rechts, Zeichen für Zeichen, gescannt. Wenn an einer Stelle eine Variable gefunden wird (sagen wir in s), überprüft man in dem anderen Term (in t), den dazu korrespondierenden Unterterm nach weiteren Vorkommen von Variablen. Falls hierbei eine Variable identisch mit der gefundenen Variable (aus s) ist, wird abgebrochen (*occurs-check*). Für den Fall $x \doteq x$ wird die leere Substitutionsmenge zurückgegeben. Ansonsten kann man die Variable auf den entsprechenden Unterterm abbilden. Schließlich werden alle Vorkommen der Variablen an allen anderen Stellen im Term (sowohl in s, als auch in t), mit dieser Abbildung ersetzt. Dieser Ansatz basiert auf einer nicht komprimierten Darstellung von Termen und wird im Abschnitt 2.1 behandelt.

Man kann die Unifikation auch auf einer komprimierten Darstellung von Termen durchführen. Dabei werden Terme durch Singleton-Tree-Grammatiken (STGs) dargestellt. Die grammatikbasierte Darstellung hat Anwendungen in verschiedenen Bereichen, wie etwa der Kompression von XML Datenstrukturen [2] und XPATH [3].

In der Informatik gibt es sehr viele Szenarien, in denen Bäume als Datenstrukturen von Computerprogrammen verwendet werden. Meistens ist es gut, den Baum in den Hauptspeicher zu legen, damit ein schnellerer Zugriff darauf möglich ist. Falls es sich hierbei um sehr grosse Bäume handelt, ist es wichtig eine platzsparende, effiziente Darstellung für die Bäume zu verwenden. Ein sehr populäres Beispiel für große Bäume sind XML-Dokumente. Sie bestehen aus einer sequentiellen Anordnung von geordneten (*ordered*[1]) Bäumen ohne Rang (*unranked*[2]). Die Ausführung einer Abfrage in einem XML-Dokument benötigt die Baumstruktur (zumindest einen Teil) im Hauptspeicher. Zur Abfrage von XML-Dokumenten gibt es spezielle Datenmodelle. Eines davon ist zum Beispiel DOM. Vergleichende Tests haben gezeigt, dass die Darstellung als DOM vier- bis fünfmal mehr Speicherplatz im Hauptspeicher benötigt, als die eigentliche XML-Datei. Eine effizientere Darstellung von

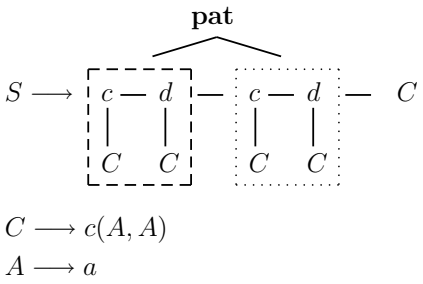
[1] Auf den Nachfolgern jedes Knotens gibt es eine Ordnung
[2] Die Anzahl der Nachfolger eines Knotens ist unbekannt

XML-Dateien als Datenmodelle sind binäre Bäume. In [2] geht es um die effiziente Darstellung von binären Bäumen.
Unter grammatikbasierter Baumkompression ist zu verstehen, dass eine möglichst kleine Grammatik für einen gegebenen Baum gesucht wird. Die Darstellung solch einer Grammatik kann wesentlich kleiner sein, als die Darstellung des zugehörigen Baumes. Die minimale reguläre Grammatik für einen Baum (*Baum-Grammatik*) t kann (amortisiert) in linearer Zeit berechnet werden. Die resultierende Grammatik ist isomorph zum minimalen DAG (*Directed Acyclic Graph*) von t. Eine Grammatik wird *straight-line* genannt, wenn jedes Nonterminal genau eine Produktion besitzt. Die minimale reguläre Baum-Grammatik von t, G_t, ist eine straight-line Grammatik.
Die Grammatik G_t ist die Eingabe für den Kompressionsalgorithmus *BPLEX*. Die Grundidee des Algorithmus ist es, gleiche Unterstrukturen im Baum festzustellen, die mehrmals auftreten, und für diese ein neues Nonterminal einzuführen. Diese Vorgehensweise wird *Multiplexing* genannt, weil mehrere Vorkommen eines Musters im Baum, das durch ein neues Nonterminal in der Grammatik repräsentiert wird, nur einmal in der Ausgabe des Algorithmus auftaucht. Die Reihenfolge, in der die Knoten, die den rechten Seiten der Eingabegrammatik entsprechen, gelesen werden, hängt davon ab, wie der dazugehörige Baum im *bottom-up* Verfahren gelesen wird. Deswegen heisst der Algorithmus *BPLEX (bottom-up multiplexing)*. Der ganze Algorithmus wird detailliert in [2] beschrieben. Zur Veranschaulichung betrachten wir ein Beispiel.

Beispiel 1.2.2 *Betrachten wir folgenden Baum,*

$$t = c(c(a,a), d(c(a,a), c(c(a,a), d(c(a,a), c(a,a))))),$$

der 18 Kanten besitzt (siehe Abbildung 1.1). Der minimale DAG, geschrieben als Baum-Grammatik, sieht wie folgt aus:



Dieser DAG ist die Eingabe für den BPLEX-Algorithmus, der versucht, eine entsprechende minimale kontextfreie Baum-Grammatik dafür zu finden. Dies wird gemacht, indem der Algorithmus die rechten Seiten der Produktionen

im bottom-up Verfahren durchläuft und währenddessen nach (nicht überlappenden) Mustern sucht, die mehrmals vorkommen. Im obigen Beispiel kommt das Pattern **pat** zweimal auf der rechten Seite der ersten Produktion vor. Ein Pattern p in einem Baum kann in geeigneter Weise durch einen Baum t_p mit den Blättern $y_1, \ldots, y_r$ dargestellt werden: Füge alle Nachfolger von jedem Knoten in p und alle dazugehörigen Kanten zu t_p hinzu und bezeichne jeden solchen $j-$ten Knoten (in preorder Traversierung) mit y_j. Im obigen Beispiel entsteht deshalb $t_{pat} = c(C, d(C, y_1))$. Dieser Baum wird die rechte Seite eines neuen Nonterminals B und somit wird die rechte Seite der ersten Produktion $B(B(C))$. Die resultierende minimale kontextfreie Baum-Grammatik sieht wie folgt aus:

$$
\begin{array}{lcl}
S & \longrightarrow & B(B(C)) \\
B(y_1) & \longrightarrow & c(C, d(C, y_1)) \\
C & \longrightarrow & c(A, A) \\
A & \longrightarrow & a
\end{array}
$$

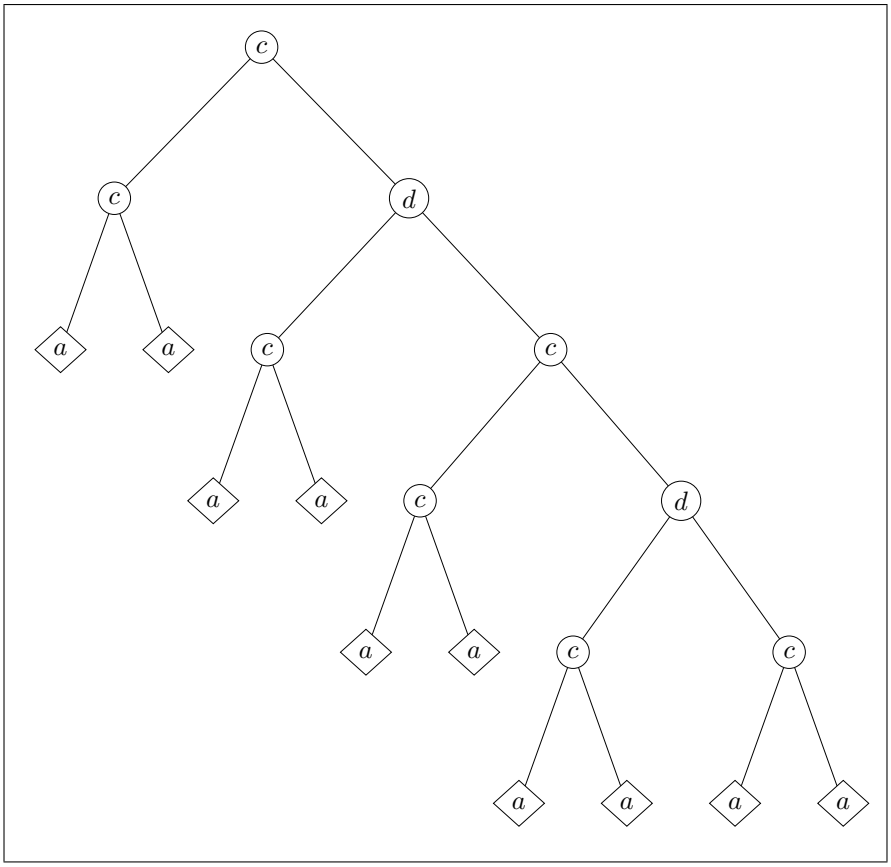


Abbildung 1.1: Baumdarstellung von t

In dieser Diplomarbeit wird die Unifikation, die in [13] beschrieben wird, in der funktionalen Programmiersprache Haskell implementiert. Dieser Unifikationsalgorithmus arbeitet auf den komprimierten Darstellungen von Termen (den STGs) und hat als Eingabe die STG-Grammatik und zwei Nonterminale, die gegebenenfalls unifiziert werden. Hierbei wird zuerst die binäre Suche auf der Singleton-Contextfree-Grammatik (SCFG) durchgeführt, um die erste unterschiedliche Stelle von den Termen, die durch die gegebenen Nonterminale gebildet werden, zu finden. Dabei geht die binäre Suche grob vereinfacht wie folgt vor:

1. Setze die Länge des kürzeren Terms von s und t als min. Wenn $min = 1$ brich ab.

2. Bilde von beiden Termen Präfixe s_1 und t_1 der Länge $m = \lfloor \frac{min}{2} \rfloor$.

3. Wenn $s_1 = t_1$, dann bilde Suffixe s_2 von s der Länge $|s_1| - m$ und t_2 von t der Länge $|t_1| - m$ und gehe mit s_2 und t_2 zu Schritt 1.

4. Wenn $s_1 \neq t_1$, gehe zu Schritt 1 und mache mit s_1 und t_1 weiter.

Beispiel 1.2.3 *Sei $s = f(a, b, x)$ und $t = f(b, b, c)$, wobei a, b und c Konstanten, f und g Funktionssymbole und x eine Variable ist. Die Längen von s und t sind $|s| = 4$ und $|t| = 4$. Es ist leicht zu sehen, dass sich die Terme (als erstes) an der zweiten Position unterscheiden. Nun gehen wir folgendermaßen vor:*

1. *$min = 4$ (da beide Terme die gleiche Länge besitzen).*

2. *$m = \lfloor \frac{min}{2} \rfloor = \lfloor \frac{4}{2} \rfloor = 2$. $s_1 = fa$ (ein Präfix von s der Länge 2) und $t_1 = fb$ (ein Präfix von t der Länge 2).*

3. *$s_1 = t_1$, trifft nicht zu.*

4. *$s_1 \neq t_1$, trifft zu.*

5. *$min = 2$ (da $|s_1| = |t_1| = 2$).*

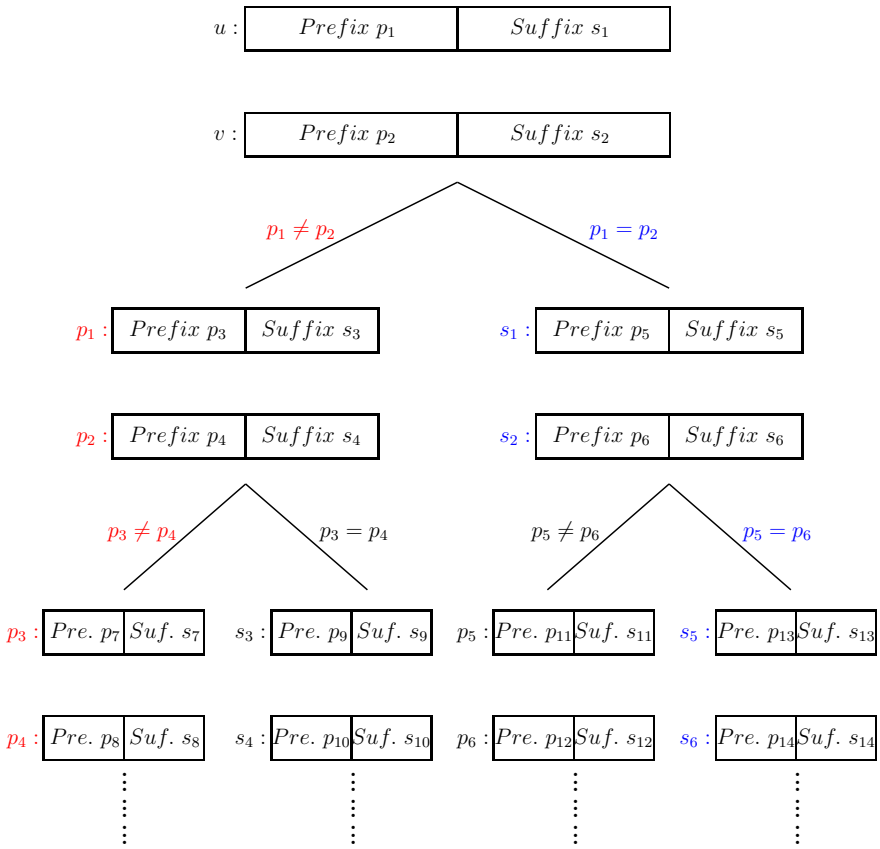
6. *$m = \lfloor \frac{min}{2} \rfloor = \lfloor \frac{2}{2} \rfloor = 1$. $s_3 = f$ (ein Präfix von s_1 der Länge 1) und $t_3 = f$ (ein Präfix von t_1 der Länge 1).*

7. *$s_3 = t_3$, trifft zu. $s_4 = a$ (ein Suffix von s_1 der Länge $|s_1| - m = 1$) und $t_4 = b$ (ein Suffix von t_1 der Länge $|t_1| - m = 1$).*

8. *$min = 1$ (da $|s_4| = |t_4| = 1$).*

Die Suche bricht ab und liefert die zweite Position von s und t als die erste unterschiedliche Stelle von Beiden.

Der Algorithmus beginnt immer mit dem Vergleich der Präfixe. Diese Vorgehensweise des Algorithmus gewährleistet, dass immer die erste unterschiedliche Stelle von den Termen gefunden wird. Die Eigenschaft der binären Suche ist leicht zu erkennen, da die Suche entweder im Suffix oder im Präfix eines Terms stattfindet, aber niemals in beiden gleichzeitig. Wegen dieser Eigenschaft wird im best-case nur die Hälfte eines Terms betrachtet (und somit gespart). Der worst-case Fall tritt auf, wenn die Terme sich an der letzten Stelle unterscheiden. Die folgende Baumdarstellung verdeutlicht die Arbeitsweise:



Der von der Wurzel an mit (rot) $\neq$ markierte Pfad im Baum repräsentiert den best-case, denn auf diesem Pfad werden die Suffixe s_1 von dem Term u und s_2 von dem Term v niemals betrachtet. Der von der Wurzel an mit (blau) $=$ markierte Pfad hingegen repräsentiert den worst-case. Auf diesem Pfad werden sowohl die Präfixe, als auch die Suffixe von u und v betrachtet, bis schließlich die ganzen Terme (bis auf das letzte Symbol) bearbeitet worden sind. Wenn nämlich die Präfixe der Terme gleich sind, bedeutet dies, dass die unterschiedliche Stelle nicht in den Präfixen liegt, sondern in den Suffixen der Terme. Deswegen betrachtet der Algorithmus in den

$p_i = p_j$ (i ungerade , $i \geq 1$ und $j = i + 1$) Fällen immer die dazugehörigen Suffixe und in den $p_i \neq p_j$ Fällen immer die Präfixe.

Die bei der binären Suche verwendete SCFG wird durch die preorder Traversierung der Singleton-Tree-Grammatik gebildet. Diese Form ermöglicht die binäre Suche auf den Termen. Im nächsten Schritt wird eine SCFG, nämlich die Positionsgrammatik gebildet, welche den Pfad zu den Untertermen an dieser unterschiedlichen Stelle angibt. Mit Hilfe dieser Positionsgrammatik untersucht man die jeweiligen Unterterme auf die Eigenschaft der first-order Unifikation. In der first-order Unifikation darf eine Variable entweder auf ein Konstantensymbol, eine andere Variable oder eine Funktion abgebildet werden. Falls diese Eigenschaften nicht zutreffen, bricht der Algorithmus ab. Andernfalls wird eine Substitution gefunden und der Algorithmus rekursiv auf die gleichen Nonterminale, mit der für diese Stelle erweiterten[3] STG, aufgerufen. Wenn beide Terme nach der Unifizierung syntaktisch gleich geworden sind, liefert der Algorithmus als Ergebnis die resultierende (neue) Singleton-Tree-Grammatik und die allgemeinste Substitution (den allgemeinsten Unifikator) für die Terme zurück. Falls die Terme schon von Anfang an, d.h ohne die Unifikation durchgeführt zu haben, gleich sind, wird als Ergebnis die eingegebene Singleton-Tree-Grammatik und die leere Menge ($\emptyset$) als allgemeinste Substitution zurückgeliefert.

Desweiteren wurde der Algorithmus von Plandowski implementiert. Der Algorithmus von Plandowski ist ein effizientes Werkzeug für das Wortproblem. Hierbei handelt es sich um die Frage, ob zwei Nonterminale einer Grammatik, die sich in Chomsky-Normalform befindet, das selbe Wort produzieren. Der naive Ansatz würde die Wörter der zu überprüfenden Nonterminale erzeugen, sie dann Zeichen für Zeichen von links nach rechts lesen, und vergleichen, ob sie an einer Stelle unterschiedlich sind. Falls ein Unterschied gefunden wird, liefert der Algorithmus False, sonst True zurück. Die Vorgehensweise von Plandowskis Algorithmus wird in Abschnitt 3 ausführlich behandelt. Ausserdem verwenden wir diesen Algorithmus auch als Hilfsmittel im Unifikationsalgorithmus, um zu überprüfen, ob die Präfixe, die während der Suche nach der unterschiedlichen Stelle zwischen den zu unifizierenden Termen gebildet werden, gleich sind.

Eine Erweiterung des Algorithmus von Plandowski auf Singleton-Tree-Grammatiken ist ebenfalls implementiert worden. Diesmal ist die Frage, ob zwei Nonterminale einer Singleton-Tree-Grammatik den gleichen Term erzeugen. Die Eingabe des Algorithmus ist die Singleton-Tree-Grammatik und zwei Nonterminale, deren Terme auf Gleichheit überprüft werden sollen. Falls die Terme, die durch die jeweiligen Nonterminale produziert werden, gleich sind, liefert der Algorithmus True, ansonsten False zurück.

[3] Die gefundene Substitution wurde auf die Grammatik angewendet

1.3 Gliederung der Diplomarbeit

In Kapitel 2 werden die grundlegenden Begriffe wie Unifikation, Singleton-Tree-Grammatik (STG), Singleton-Kontextfreie-Grammatik (SCFG) und die dafür relevanten Definitionen erläutert, die als Voraussetzung im weiteren Verlauf gebraucht werden. Wir werden hierbei auch noch auf einige Unifikationsalgorithmen eingehen, die sich nützlich erwiesen haben.

In Kapitel 3 wird Plandowskis Algorithmus vorgestellt. Das Wortproblem behandelt die Frage, ob zwei unterschiedliche Nonterminale einer SCFG (die in Chomsky-Normalform ist) das gleiche Wort bilden. Dabei betrachten wir die Vorgehensweise des Algorithmus genauer und gehen anschließend auf die Laufzeit ein.

Kapitel 4 beschreibt die Arbeitsweise des Unifikationsalgorithmus auf komprimierten Termen. Hierzu betrachten wir als erstes, wie die erste unterschiedliche Stelle zwischen zwei Termen mittels der *index*-Funktion gefunden werden kann. Diese Funktion arbeitet auf einer SCFG. Dazu werden Transformationsregeln verwendet, (diese sind in [13] beschrieben) wie eine STG in eine Preorder-SCFG umgewandelt werden kann. Nachdem die unterschiedliche Stelle gefunden worden ist, wird eine weitere SCFG, die sogenannte Positionsgrammatik, für beide Terme gebildet. Die genaue Vorgehensweise zur Bildung einer komprimierten Positionsgrammatik erläutern wir mit der Hilfe von weiteren Formalismen. Die Positionsgrammatik führt uns an die Position des Unterschieds in den Termen, während die index-Funktion nur die Stelle (also eine Zahl) liefert (Bsp.: $s = f(a), t = f(b)$. index liefert als Ergebnis 2, also die zweite Stelle, während die Positionsgrammatik zu a und b hinführt). Zum Abschluss des Kapitels stellen wir den Unifikationsalgorithmus vor, der den allgemeinsten Unifikator berechnet, falls es eine Lösung für das *Unifikations-Problem* gibt.

Kapitel 5 behandelt die Vorraussetzungen der Implementierung. Hierbei erläutern wir einige Eigenschaften der Programmiersprache Haskell, die während der Implementierung benutzt worden sind. Ausserdem gehen wir genauer auf die Implementierung der einzelnen Algorithmen ein.

Zum Schluss werden einige interessante Testfälle in Kapitel 6 durchgeführt. Hierbei sollen Tabellen und die daraus resultierenden Statistiken zum Verständnis der Laufzeit dienen.

Kapitel 2

Grundlagen

Eine Signatur ist eine Menge $\mathcal{F}$ zusammen mit einer Funktion $ar : \mathcal{F} \rightarrow \mathbb{N}$. Elemente der Menge $\mathcal{F}$ sind Funktionssymbole f und $ar(f)$ heisst die Stelligkeit von f. Funktionssymbole der Stelligkeit 0 werden als Konstanten bezeichnet. Sei $\mathcal{X}$ eine Menge von Variablen mit $\mathcal{F} \cap \mathcal{X} = \emptyset$. Die Menge $\mathcal{T}(\mathcal{F}, \mathcal{X})$ von Termen über $\mathcal{F}$ und $\mathcal{X}$ ist definiert durch $f(t_1, .., t_m) \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ wenn $f \in \mathcal{F},\ t_1, ..t_m \in (\mathcal{F}, \mathcal{X})$ und $ar(f) = m$. Die Länge eines Terms t ist die Anzahl der Variablen und Funktionen, die darin auftreten und wird durch $|t|$ bezeichnet. Die *Tiefe* (*depth*) T eines Terms t ist definiert als

$$T(t) = \begin{cases} 0, \text{ wenn } t \text{ eine Konstante oder Variable ist} \\ 1 + \max\{T(t_1), \ldots, T(t_m)\}, \text{ wenn } t = f(t_1, .., t_m). \end{cases}$$

Die *Position* eines Terms t ist eine Sequenz von natürlichen Zahlen p und q, die die Unterterme identifizieren. Die Länge einer Position ist definiert als $|p|$. Die Menge der Positionen $P(t)$ eines Terms t ist definiert als

$$P(t) = \begin{cases} \{\lambda\}, \text{ wenn } t \text{ eine Konstante oder Variable ist} \\ \{\lambda\} \cup \{1 \cdot p \mid p \in Pos(t_1)\} \cup, \ldots, \cup \{m \cdot p \mid p \in Pos(t_m)\}, \\ \text{wenn } t = f(t_1, .., t_m). \end{cases}$$

Hierbei ist λ die leere Sequenz, und $p \cdot q$ die Konkatenation von p und q. Wenn t ein Term und p eine Position ist, dann ist $t|_p$ der Unterterm von t an der Position p. Genauer wird es definiert als

$$t|_\lambda = t \text{ und } f(t_1, .., t_m)|_{i \cdot p} = t_i|_p.$$

Der Term $t[s]_p$ ist der gleiche Term t, mit dem Unterschied, dass der Unterterm $t|_p$ durch s ersetzt wird. Definitiongemäß ist

$$f(t_1, .., t_m)[s]_{i \cdot p} = f(t_1, .., t_{i-1}, t_i[s]_p, t_{i+1}, ..t_m).$$

Auf $Pos(t)$ wird eine partielle Ordnung $\leq$ definiert mit $p \leq q$, wenn p ein Präfix von q ist, d.h. $q = p{\cdot}p'$ mit $p' \in Pos(t)$. Hierbei bedeutet *disjoint* in der partiellen Ordnung weder $>$, noch $\leq$. Eine *Substitution* ist eine Abbildung $\sigma : \mathcal{X} \to \mathcal{T}(\mathcal{F}, \mathcal{X})$. Substitutionen können auf beliebige Terme $\sigma(f(t_1, .., t_m)) = f(\sigma(t_1), .., \sigma(t_m)))$ homomorph erweitert werden.
Kontexte sind Ausdrücke, die genau an einer Stelle ein *Loch* besitzen, welches im Folgenden mit $[\cdot]$ gekennzeichnet wird und in diesem Loch andere Kontexte oder Terme eingesetzt werden können. Kontexte, wie $C[\cdot]$ und $D[\cdot]$, werden in Großbuchstaben geschrieben. Wenn C und D Kontexte sind und s ein Term ist, repräsentiert $C[D[\cdot]]$ bzw. CD und $C[s]$ bzw. Cs den Term C ausser dem Loch, für welches D bzw. s eingesetzt worden sind. Man kann auch Kontexte als λ-Ausdrücke $\lambda x.C[x]$ schreiben, wobei die gebundene Variable im Ausdruck nur einmal vorkommt. Wenn $D_i = D_j D_k$, dann ist D_j ein Präfix und D_k ein Suffix von D_i und $i, j, k \in \mathbb{N}$. Mit $hp(C)$ (*hole path*) bezeichnen wir die Position von dem Loch im Kontext $C[\cdot]$ und die Länge bezeichen wir mit $|hp(C)|$.

2.1 Unifikation auf nicht komprimierten Termen

Wir möchten zuerst einen kurzen Rückblick auf die Geschichte der Unifikation machen ([8],[6]). In den Tagebüchern und Notizen von Emil Post gab es schon 1920 die ersten Anregungen auf das Konzept eines Unifikationsalgorithmus, der den allgemeinsten Repräsentanten im Gegensatz zu allen möglichen Instantiierungen berechnet. Die erste detaillierte Beschreibung eines Unifikationalgorithmus folgte 1930 in der Doktorarbeit von Jacques Herbrand. Dies war die erste Veröffentlichung eines Unifikationsalgorithmus und basierte auf dem später von Alberto Martelli und Ugo Montanari wiederentdeckten Verfahren, welches noch heute verwendet wird. Im Jahre 1962 gab es die erste Implementierung in den Bell Telephone Laboratories. Jim Guard und seine Forschungsgruppe in der Applied Logic Corporation begannen 1964 mit der Erforschung von *higher-order* Unifikationen. Der Name Unifikation und die erste formale Untersuchung dieses Begriffs wurde 1965 von J.A.Robinson eingeführt. Er verwendete die Unifikation als die grundlegende Operation seines Resolution-Prinzips. Robinson hat gezeigt, dass wenn zwei Terme unifizierbar sind, es immer einen allgemeinsten Unifikator (allgemeinste Substitution) gibt und hat einen Algorithmus entworfen, der den allgemeinsten Unifikator für zwei Terme berechnet [7]. Seine Arbeit war die einflussreichste Untersuchung auf dem Gebiet der Unifikation. Donald Knuth und Peter Bendix haben die Begriffe *Unifikation* und *allgemeinster Unifikator* 1967 neu eingeführt. Sie benutzten die Unifikation als Hilfsmittel zum Testen von lokaler Konfluenz bei Termersetzungssystemen. Gerard Huet und Claudio Lucchessi haben 1972 gezeigt, dass die high-order Unifikation unentscheidbar ist. Das Ergebnis hat Warren Goldfarb 1982 verschärft.

Gordon Plotkin hat 1972 gezeigt, wie gewisse Axiome, die die Gleichheit benutzen, in die Transformationsregeln eingebaut werden können (nämlich Resolution). Dies wird erreicht, indem die syntaktische Unifikation im Resolutionsschritt durch *equational unification* ([5]) ersetzt wird. Gerard Huet hat seine Forschungen 1976 weiterentwickelt. Dies war ein fundamentaler Beitrag im Bereich der first- und higher-order Unifikation. 1980 begannen die Unification Workshops (UNIF). Ab den 90'er Jahren haben sich die Anwendungsgebiete der Unifikation vermehrt. Schließlich hat Colin Stirling 2006 bewiesen, dass higher-order Unifikation entscheidbar ist, ein Problem, das es seit über 30 Jahren gab.

Wozu benötigt man Unifikation ?

- In Programmiersprachen zur Typberechnung.
- Für das Matching in Programmiersprachen, die auf pattern basieren (Mustererkennung).
- Für verschiedene Formalismen in der Computerlinguistik.
- Um eine Überschreibung in Termersetzungssystemen durchzuführen.
- etc.

Allgemein versucht die Unifikation zwei Terme syntaktisch gleich zu machen, indem gewisse Unterterme (Variablen) durch andere Terme ersetzt werden (das Lösen von Gleichungssystemen). Terme bestehen hierbei aus Funktionssymbolen (beispielsweise f, a und b, wobei f die Stelligkeit 2 und a, b die Stelligkeit 0 besitzen) und Variablen (zum Beispiel x und y). Das Unifikations-Problem für die Terme $s = f(a, x)$ und $t = f(y, b)$ kann man durch folgende Frage festlegen:

Ist es möglich die Variablen x in s und y in t durch Terme zu ersetzen, so dass die resultierenden Terme syntaktisch gleich werden?

Wenn wir in diesem Beispiel die Variable x durch b und die Variable y durch a substituieren, erhalten wir den unifizierten Term $f(a, b)$. Die Substitution wird durch $\sigma := \{x \mapsto b, y \mapsto a\}$ gekennzeichnet. Die Anwendung der Substitution auf einen Term wird als $s\sigma = f(a, b) = t\sigma$ geschrieben, also wird σ als Suffix an die Terme angehängt. Solch eine Substitution wird auch *Unifikator* von s und t genannt. Gleiche Variablen, die an unterschiedlichen Stellen in einem Term auftauchen, müssen immer durch denselben Term ersetzt werden.
Betrachten wir nun einige Beispiele, um die unterschiedlichen Aspekte des Problems zu sehen:

$$
\begin{array}{lcl}
f(x) & \doteq & f(a), \text{besitzt genau eine Substitution } \{x \mapsto a\} \\
x & \doteq & f(y), \text{besitzt mehrere Substitutionen: } \{x \mapsto f(y)\}, \\
 & & \{x \to f(a), y \mapsto a\}, \ldots \\
f(x) & \doteq & g(y), \text{besitzt keine Substitution} \\
x & \doteq & f(x), \text{besitzt keine Substitution}
\end{array}
$$

Diese Beispiele zeigen, dass es für eine Gleichung keine, eine oder mehrere Lösungen geben kann. Allerdings sind einige Lösungen besser als andere: $\{x \mapsto f(y)\}$ ist eine allgemeinere Substitution der Gleichung $x \doteq f(y)$, als $\{x \mapsto f(a), y \mapsto a\}$. Vielmehr ist $\{x \mapsto f(y)\}$ der allgemeinste Unifikator der Gleichung $x \doteq f(y)$. Alle anderen Substitutionen sind Instanzen des allgemeinsten Unifikators. Das bedeutet, dass alle Unifikatoren durch weitere Einsetzungen in den allgemeinsten Unifikator erzeugt werden können. Ein Unifikationsalgorithmus sollte nicht nur versuchen das Problem zu lösen, sondern, falls es eine Lösung für das Problem gibt, den allgemeinsten Unifikator berechnen.

Definition 2.1.1 *Eine Substitution σ ist allgemeiner als eine Substitution σ', falls es eine Substitution δ gibt, so dass $\sigma' = \sigma\delta$. In diesem Fall schreiben wir $\sigma \lesssim \sigma'$. Wir sagen σ' ist eine Instanz von σ.*

Wenn $\sigma := \{x \mapsto f(y)\}$ und $\sigma' := \{x \mapsto f(a), y \mapsto a\}$, dann ist σ' eine Instanz von σ ($\sigma \lesssim \sigma'$), weil $\sigma' = \sigma\delta$, mit $\delta := \{y \mapsto a\}$. Man kann leicht nachprüfen, dass $\sigma' = \sigma\delta$: $x\sigma' = f(a) = x\sigma\delta$, $y\sigma' = a = y\sigma\delta$ und $z\sigma' = z = z\sigma\delta$ für alle anderen Variablen z.

Lemma 2.1.2 *$\lesssim$ ist eine quasi Ordnungsrelation auf Substitutionen.*

Beweis. [4], Lemma 4.5.2 .

Definition 2.1.3 *Ein Unifikations-Problem ist eine endliche Menge von Gleichungen $\mathcal{S} = \{s_1 \doteq t_1, \ldots, s_n \doteq t_n\}$. Ein Unifikator (Lösung) von $\mathcal{S}$ ist eine Substitution σ, so dass $s_i\sigma = t_i\sigma$ für $i = 1, \ldots, n$. Die Menge aller Unifikatoren von $\mathcal{S}$ werden mit $\mathcal{U}(\mathcal{S})$ bezeichnet. $\mathcal{S}$ ist unifizierbar, wenn $\mathcal{U}(\mathcal{S}) \neq \emptyset$. Eine Substitution σ ist ein allgemeinster Unifikator von $\mathcal{S}$, wenn σ das kleinste Element in $\mathcal{U}(\mathcal{S})$ ist:*

1. $\sigma \in \mathcal{U}(\mathcal{S})$
2. $\forall \sigma' \in \mathcal{U}(\mathcal{S}), \sigma \lesssim \sigma'$.

Die Unifikation, die bis jetzt beschrieben worden ist, nennt man syntaktische Unifikation von first-order Termen. Syntaktisch bedeutet, dass die Terme syntaktisch gleich gemacht werden müssen und first-order bedeutet, dass keine higher-order Variablen erlaubt sind, d.h. Variablen für Funktionen.

Zum Beispiel können die Terme $s = f(x, a)$ und $t = g(a, x)$ durch first-order Unifikation nicht syntaktisch gleich gemacht werden. Aber die Terme $s = f(x, a)$ und $t = G(a, x)$ können durch higher-order Unifikation syntaktisch gleich gemacht werden, falls G eine higher-order Variable ist und durch f ersetzt wird.
Im Folgenden beschreiben wir einige Unifikationsalgorithmen, die alle den allgemeinsten Unifikator zurückliefern, falls die Terme unifizierbar sind. Wir werden mit einem naiven Algorithmus anfangen, dann den Algorithmus von Robinson anschauen und zum Schluss einen auf Transformationsregeln basierenden Algorithmus betrachten.
Bevor wir dies machen, möchten wir einige Notationen einführen, die im folgenden benutzt werden:

- $\mathcal{F}$ sei eine Menge von Funktionssymbolen mit fester Stelligkeit.
- $\mathcal{X}$ sei eine Menge von Variablen, so dass $\mathcal{F} \cap \mathcal{X} = \emptyset$.
- $t := x \mid f(t_1, \ldots, t_n)$, wobei $n \geq 0$, $x \in \mathcal{X}$ und $f \in \mathcal{F}$ mit $ar(f) = n$.

Konstanten sind Funktionssymbole mit Stelligkeit 0, wobei die Klammern weggelassen werden (a statt $a()$).

- x, y, z seien Variablen.
- a, b, c seien Konstanten.
- f, g, h seien beliebige Funktionen.
- s, t, r seien Terme.
- $\mathcal{T}(\mathcal{F}, \mathcal{X})$ sei die Menge von Termen über $\mathcal{F}$ und $\mathcal{X}$.

Ein *Grundterm* ist ein Term der keine Variablen enthält.

- $\mathcal{T}(\mathcal{F})$ sei die Menge von Grundtermen über $\mathcal{F}$.

Eine *Gleichung* ist ein Paar von Termen und wird als $s \doteq t$ geschrieben.

- *Vars*(t) sei die Menge von Variablen im Term t. Diese Notation wird auch für eine Menge von Termen, für Gleichungen und für eine Menge von Gleichungen benutzt.

Beispiel 2.1.4

- $f(x, g(x, a), y)$ *ist ein Term, wobei* f *eine Funktion mit der Stelligkeit 3,* g *eine Funktion mit der Stelligkeit 2,* a *eine Konstante und* x, y *Variablen sind.*

- $Vars(f(x, g(x, a), y)) = \{x, y\}$.
- $f(b, g(b, a), c)$ *ist ein Grundterm.*
- $Vars(f(b, g(b, a), c)) = \emptyset$.

Eine *Substitution* ist eine Abbildung von Variablen auf Terme.

Beispiel 2.1.5 *Eine Substitution wird als eine Menge von Bindungen geschrieben:*

$$\{x \mapsto f(a, b), y \mapsto z\}.$$

Diese Substitution bildet alle Variablen ausser x und y auf sich selbst ab.

- σ, η, ϑ und ρ seien beliebige Substitutionen.
- ϵ sei die Identitäts-Substitution.

Die Anwendung einer Substitution σ auf einen Term t wird folgendermaßen geschrieben:

$$t\sigma = \begin{cases} x\sigma, \text{ falls } t = x \\ f(t_1\sigma, \ldots, t_n\sigma), \text{ falls } t = f(t_1, \ldots, t_n). \end{cases}$$

Im zweiten Fall dieser Definition ist $n = 0$ erlaubt. Dann ist f eine Konstante und $f\sigma = f$.

Beispiel 2.1.6

- $\sigma = \{x \mapsto f(a, b), y \mapsto g(a)\}$.
- $t = f(x, g(f(x, f(y, z))))$.
- $t\sigma = f(f(a, b), g(f(f(a, b), f(g(a), z))))$.

Für eine Substitution σ ist die *Domain* die Menge von Variablen

$$Dom(\sigma) := \{x \mid x\sigma \neq x\},$$

die *Range* die Menge von Termen

$$Ran(\sigma) := \bigcup_{x \in Dom(\sigma)} \{x\sigma\}$$

und die *Variable Range* die Menge der Variablen, die in der *Range* vorkommen, also

$$VRan(\sigma) := Vars(Ran(\sigma)).$$

Beispiel 2.1.7

- $Dom(\{x \mapsto f(a,y), y \mapsto g(z)\}) = \{x, y\}$
- $Dom(\{x \mapsto f(a,b), y \mapsto g(c)\}) = \{x, y\}$
- $Dom(\epsilon) = \emptyset$
- $Ran(\{x \mapsto f(a,y), y \mapsto g(z)\}) = \{f(a,y), g(z)\}$
- $Ran(\{x \mapsto f(a,b), y \mapsto g(c)\}) = \{f(a,b), g(c)\}$
- $Ran(\epsilon) = \emptyset$
- $VRan(\{x \mapsto f(a,y), y \mapsto g(z)\}) = \{y, z\}$
- $VRan(\{x \mapsto f(a,b), y \mapsto g(c)\}) = \emptyset$
- $VRan(\epsilon) = \emptyset$

Die *Komposition* von zwei Substitutionen wird geschrieben als $\sigma\vartheta$ und ist definiert durch

$$t\sigma\vartheta = (t\sigma)\vartheta.$$

Ein Algorithmus zur Konstruktion der Komposition $\sigma\vartheta$ von zwei Substitutionen σ und ϑ ist wie folgt:

1. Wende ϑ auf jeden Term in $Range(\sigma)$ an um σ_1 zu erhalten.
2. Entferne aus ϑ alle Bindungen $x \mapsto t$ mit $x \in Dom(\sigma)$ um ϑ_1 zu erhalten.
3. Entferne aus σ_1 alle trivialen Bindungen $x \mapsto x$ um σ_2 zu erhalten.
4. Nimm die Vereinigung der Mengen von Bindungen σ_2 und ϑ_1.

Beispiel 2.1.8 *Für* $\sigma = \{x \mapsto f(y), y \mapsto z\}$, $\vartheta = \{x \mapsto a, y \mapsto b, z \mapsto y\}$ *gilt,*

- $\sigma_1 = \{x \mapsto f(y)\vartheta, y \mapsto z\vartheta\} = \{x \mapsto f(b), y \mapsto y\}$,
- $\vartheta_1 = \{z \mapsto y\}$,
- $\sigma_2 = \{x \mapsto f(b)\}$,
- $\sigma\vartheta = \{x \mapsto f(b), z \mapsto y\}$.

Die Komposition ist nicht kommutativ, wie man an

$$\vartheta\sigma = \{x \mapsto a, y \mapsto b\} \neq \sigma\vartheta$$

sieht. Eine Substitution σ heißt *idempotent*, falls $\sigma\sigma = \sigma$. Es ist leicht zu zeigen, dass dies der Fall ist, $\Leftrightarrow Dom(\sigma) \cap VRan(\sigma) = \emptyset$ gilt.

Beispiel 2.1.9 *Sei* $\sigma = \{x \mapsto f(z), y \mapsto z\}$ *und* $\vartheta = \{x \mapsto f(y), y \mapsto z\}$. *Dann ist* σ *idempotent und* ϑ *nicht idempotent,* $\vartheta\vartheta = \sigma \neq \vartheta$.

2.1.1 Ein naiver Unifikationsalgorithmus

Der wahrscheinlich einfachste Unifikationsalgorithmus sieht wie folgt aus:

Schreibe die zu unifizierenden Terme auf und setze an den Anfang von beiden jeweils einen Zeiger. Dann gehe folgende Schritte durch:

1. Bewege die Zeiger gleichzeitig Symbol für Symbol nach rechts bis beide Zeiger über das letzte Symbol der Terme hinüberlaufen (**success**), oder bis sie an zwei unterschiedlichen Symbolen angekommen sind.
2. Falls beide Symbole keine Variable sind, dann **fail**. Andernfalls ist eines von beiden eine Variable (sagen wir x) und das andere ist das erste Symbol eines Unterterms (sagen wir t):
   (a) Wenn x in t vorkommt, dann **fail**.
   (b) Andernfalls füge $x \mapsto t$ zur Lösung hinzu, ersetze jedes Vorkommen der Variable x durch t und kehre zu Schritt 1 zurück.

Dieser Algorithmus findet den Unterschied in den zu unifizierenden Termen und versucht dies zu reparieren, indem er die Variable an den Term bindet. Der Algorithmus bricht ab, wenn er an einer Stelle zwei unterschiedliche Funktionssymbole findet, oder der Term auf den die Variable gebunden wird, diese Variable beinhaltet. Auf die Korrektheit und Laufzeit wird in [5] eingegangen. Eine ausführliche Variante kann unter [1] betrachtet werden.

Beispiel 2.1.10 *Seien* $s = f(x, g(a), g(z))$ *und* $t = f(g(y), g(y), g(g(x)))$ *die zu unifizierenden Terme.*

1. Die Zeiger werden an die nullte Position gesetzt

$$\begin{array}{l} f(x, g(a), g(z)) \\ \uparrow \\ \\ f(g(y), g(y), g(g(x))) \\ \uparrow \end{array}$$

2. Die Zeiger werden an die erste Position bewegt und die Symbole verglichen

$$\begin{array}{l} f(x, g(a), g(z)) \\ \uparrow \\ \\ f(g(y), g(y), g(g(x))) \\ \uparrow \end{array}$$

3. Die Symbole x und g sind unterschiedlich

$$f(x, g(a), g(z))$$
$$\uparrow$$
$$f(g(y), g(y), g(g(x)))$$
$$\uparrow$$

4. Der Unterterm von g wird gefunden

$$f(x, g(a), g(z))$$
$$\uparrow$$
$$f(g(y), g(y), g(g(x)))$$
$$\uparrow$$

5. Alle Vorkommen x werden durch den Unterterm ersetzt

$$f(g(y), g(a), g(z))$$
$$\uparrow$$
$$f(g(y), g(y), g(g(g(y))))$$
$$\uparrow$$
$$\{x \mapsto g(y)\}$$

6.

$$f(g(y), g(a), g(z))$$
$$\uparrow$$
$$f(g(y), g(y), g(g(g(y))))$$
$$\uparrow$$
$$\{x \mapsto g(y)\}$$

7.

$$f(g(y), g(a), g(z))$$
$$\uparrow$$
$$f(g(y), g(y), g(g(g(y))))$$
$$\uparrow$$
$$\{x \mapsto g(y)\}$$

8.

$$f(g(a), g(a), g(z))$$
$$\uparrow$$
$$f(g(a), g(a), g(g(g(a))))$$
$$\uparrow$$
$$\{x \mapsto g(a), y \mapsto a\}$$

9.

$$f(g(a), g(a), g(z))$$
$$\uparrow$$
$$f(g(a), g(a), g(g(g(a))))$$
$$\uparrow$$
$$\{x \mapsto g(a), y \mapsto a\}$$

10.

$$f(g(a), g(a), g(z))$$
$$\uparrow$$
$$f(g(a), g(a), g(g(g(a))))$$
$$\uparrow$$
$$\{x \mapsto g(a), y \mapsto a\}$$

11.

$$f(g(a), g(a), g(z))$$
$$\uparrow$$
$$f(g(a), g(a), g(g(g(a))))$$
$$\uparrow$$
$$\{x \mapsto g(a), y \mapsto a\}$$

12.

$$f(g(a), g(a), g(g(g(a))))$$
$$\uparrow$$
$$f(g(a), g(a), g(g(g(a))))$$
$$\uparrow$$
$$\{x \mapsto g(a), y \mapsto a, z \mapsto g(g(a))\}$$

13.

$$f(g(a), g(a), g(g(g(a))))$$
$$\uparrow$$
$$f(g(a), g(a), g(g(g(a))))$$
$$\uparrow$$
$$\{x \mapsto g(a), y \mapsto a, z \mapsto g(g(a))\}$$

2.1.2 Der Unifikationsalgorithmus von Robinson

Wir erläutern den Algorithmus von Robinson mit der Hilfe von zwei Beispielen aus [6].

Beispiel 2.1.11 *Angenommen man möchte die Terme $s = f(x, g(a, z))$ und $t = f(g(a, y), x)$ unifizieren. Hierbei sind f und g Funktionen mit der Stelligkeit 2, a ein Konstantensymbol und x, y, z Variablen.*

Im ersten Schritt werden die Terme gleichzeitig von links nach rechts gelesen, bis der erste Unterschied vorkommt. Im obigen Beispiel kommt der erste Unterschied in der Variablen x in s und dem Funktionssymbol g in t vor. Die Stellen, an denen der Unterschied stattfindet, definieren die sogenannten Unterschiedsterme. In diesem Beispiel sind es x und $g(a, y)$. Um die Terme s und t unifizieren zu können, müssen diese Unterschiedsterme unifiziert werden. Es ist leicht zu sehen, dass dies durch die Substitution $\sigma_1 = \{x \mapsto g(a, y)\}$ gemacht werden kann.
Nun wird diese Substitution σ_1 auf die Terme s und t angewandt und die dadurch resultierenden Terme (welche $s\sigma_1 = f(g(a, y), g(a, z))$ *und* $t\sigma_1 = f(g(a, y), g(a, y))$ sind) werden von links nach rechts weitergelesen, bis nochmals ein Unterschied gefunden wird. Dieser Prozess muss so lange durchgeführt werden, bis die Terme unifiziert worden sind. Im obigen Beispiel ist die

nächste unterschiedliche Stelle z in s und y in t. Nachdem wir die Substitution $\sigma_2 = \{y \mapsto z\}$ auf $s\sigma_1$ und $t\sigma_1$ anwenden, erhalten wir den unifizierten Term $s\sigma_1\sigma_2 = f(g(a,z), g(a,z)) = t\sigma_1\sigma_2$. Die Komposition $\sigma := \sigma_1 \circ \sigma_2$ ist ein allgemeinster Unifikator von s und t.
Es ist klar, dass wir auch die Substitution $\{z \mapsto y\}$ statt $\sigma_2 = \{y \mapsto z\}$ benutzen könnten. Dies erklärt, warum allgemeinste Unifikatoren nur bis auf Umbenennung der Variablen eindeutig sind.
Während das obige Beispiel den Fall behandelt, dass zwei Terme unifizierbar sind, zeigt das nächste Beispiel alle möglichen Situationen, in denen zwei Terme nicht unifiziert werden können.

Beispiel 2.1.12
Angenommen man möchte als erstes die Terme $s = f(g(a,y),z)$ und $t = f(f(x,y),z)$ unifizieren.

Die erste unterschiedliche Stelle zwischen den beiden Termen ist das Funktionssymbol g in s und das zweite Symbol f in t. Dies bedeutet, dass die Unterschiedsterme $g(a,y)$ und $f(x,y)$ mit zwei unterschiedlichen Funktionssymbolen beginnen. Als Resultat folgt, dass die Unterschiedsterme nicht unifizierbar sind, womit auch die Terme s und t nicht unifizierbar sind. Dieser Grund für eine Nicht-Unifizierung wird *Failure Clash* genannt.
Nehmen wir jetzt an, dass wir die Terme $s = f(g(a,x),z)$ und $t = f(x,z)$ unifizieren möchten. Hier erhalten wir als Unterschiedsterme $g(a,x)$ und x, welche nicht unifiziert werden können, weil die Variable x in dem Term $g(a,x)$ enthalten ist. In der Tat wäre für jede Substitution σ die Länge des Terms $x\sigma$ immer kleiner als die Länge des Terms $g(a,x)\sigma = g(a,x\sigma)$. Dieser Grund für eine Nicht-Unifizierung wird *Occur-Check -Failure* genannt.
Das Folgende ist der von J.A.Robinson 1965 eingeführte Algorithmus und wurde vor allem in Computeralgebrasystemen (CAS) benutzt.

global σ: substitution; initialized to *id(ϵ)*
Unify (s: term, t: term);
begin
  if s is Variable ***then***
   $s = s\sigma$;
  if t is Variable ***then***
   $t = t\sigma$;
  if s is Variable and $s = t$ ***then***
    do nothing;
  else if $s = f(s_1, \ldots, s_n)$ and $t = g(t_1, \ldots, t_n)$ for $n, m \geq 0$ ***then***
    begin
      if $f = g$ ***then***
        for $i := 1, \ldots, n$ ***do***
          Unify (s_i, t_i)

else
exit with failure: Symbol clash;
end;
else if s is not a Variable ***then***
Unify (t, s)
else if s occurs in t ***then***
exit with failure: Occurs Check;
else
$\sigma = \sigma\{s \mapsto t\}$
end;

Beispiel 2.1.13 *Seien* $s = f(x, g(a), g(z))$ *und* $t = f(g(y), g(y), g(g(x)))$ *die zu unifizierenden Terme.*

1.Aufruf: $Unify(s, t), \sigma = \epsilon$.
s ist keine Variable. s hat die Form $f(s_1, \ldots, s_n)$ und t die Form $g(t_1, \ldots, t_n)$. Nun ist die Frage ob $f = g$? Dies trifft zu, denn $s = f(x, g(a), g(z))$ und $t = f(g(y), g(y), g(g(x)))$, also $f = f$.

2.Aufruf: $Unify(s_1 = x, t_1 = g(y)), \sigma = \epsilon$.
s_1 ist eine Variable, aber t_1 ist keine Variable. t_1 hat die Form $g(t_{1'}, \ldots, t_{n'})$, aber s_1 nicht. Also wird überprüft, ob s_1 in t_1 vorkommt. Nein, $s_1 = x$ kommt nicht in $t_1 = g(y)$ vor. Also erweitern wir σ mit $\{x \mapsto g(y)\}$.

3.Aufruf: $Unify(s_2 = g(a), t_2 = g(y)), \sigma = \{x \mapsto g(y)\}$
s_2 ist keine Variable. s_2 hat die Form $f(s_{1'}, \ldots, s_{n'})$ und t_2 die Form $g(t_{1'}, \ldots, t_{n'})$. Nun ist die Frage, ob $f = g$? Dies trifft zu, denn $s_2 = g(a)$ und $t_2 = g(y)$, also $g = g$.

4.Aufruf: $Unify(a, y), \sigma = \{x \mapsto g(y)\}$.
a ist keine Variable und beide haben nicht die Form $f(s_{1'}, \ldots, s_{n'})$.

5.Aufruf: $Unify(y, a), \sigma = \{x \mapsto g(y)\}$.
y ist eine Variable. a hat die Form $f(s_{1'}, \ldots, s_{n'})$ (da Konstanten per Definition Funktionen mit Stelligkeit 0 sind), aber y nicht. Also wird überprüft ob y in a vorkommt. Nein, dies trifft nicht zu. Wir erweitern σ mit $\{y \mapsto a\}$.

6.Aufruf: $Unify(s_3 = g(z), t_3 = g(g(x))), \sigma = \{x \mapsto g(a), y \mapsto a\}$.
s_3 hat die Form $f(s_{1'}, \ldots, s_{n'})$ und t_3 die Form $g(t_{1'}, \ldots, t_{n'})$. Ist $f = g$? Dies trifft zu, denn $s_3 = g(z)$ und $t_3 = g(g(x))$, womit $g = g$.

7.Aufruf: $Unify(z, g(x)), \sigma = \{x \mapsto g(a), y \mapsto a\}$.

z ist eine Variable. Nun muss überprüft werden, ob z im Term $g(x)$ vorkommt. Es ist leicht zu sehen, dass dies nicht der Fall ist. Jetzt erweitern wir σ mit $\{z \mapsto g(x)\}$.

Nun ist auch die äußerste **for**-Schleife beendet und der Algorithmus liefert als Resultat $\sigma := \{x \mapsto g(a), y \mapsto a, z \mapsto g(g(a))\}$, womit $s\sigma = f(g(a), g(a), g(g(g(a)))) = t\sigma$.

2.1.3 Eine auf Regeln basierende Methode

Nun stellen wir eine Methode vor, die mit Regeln arbeitet und durch die nichtdeterministische Anwendung dieser Regeln eine Lösung für ein Unifikationsproblem herleitet.

Definition 2.1.14 *Ein Unifikations-Problem $\mathcal{S} = \{x_1 \doteq t_1, \ldots, x_n \doteq t_n\}$ ist in gelöster Form, falls alle x_i paarweise disjunkte Variablen sind und keines davon in keinem der t_i vorkommt. In diesem Fall definieren wir:*

$$\vec{\mathcal{S}} = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}.$$

Lemma 2.1.15 *Wenn $\mathcal{S}$ in gelöster Form ist, dann gilt $\sigma = \sigma\vec{\mathcal{S}}$ für alle $\sigma \in \mathcal{U}(\mathcal{S})$.*

Beweis. [4], Lemma 4.6.2 .

Lemma 2.1.16 *Wenn $\mathcal{S}$ in gelöster Form ist, dann ist $\vec{\mathcal{S}}$ ein idempotenter allgemeinster Unifikator von $\mathcal{S}$.*

Beweis. [4], Lemma 4.6.3 .

Demzufolge wissen wir, wie wir aus einer gelösten Form eines Unifikations-Problems einen idempotenten allgemeinsten Unifikator extrahieren können. Um dorthin zu gelangen, verwenden wir folgende Transformationsregeln:

Delete:

$$\{t \doteq t\} \uplus \mathcal{S} \Longrightarrow \mathcal{S}$$

Decompose:

$$\{f(\overline{t_n}) \doteq f(\overline{u_n})\} \uplus \mathcal{S} \Longrightarrow \{t_1 \doteq u_1, \ldots, t_n \doteq u_n\} \cup \mathcal{S}$$

Orient:

$$\{t \doteq x\} \uplus \mathcal{S} \Longrightarrow \{x \doteq t\} \cup \mathcal{S}, \text{ falls } t \notin \mathcal{X}$$

Eliminate:

$$\{x \doteq t\} \uplus \mathcal{S} \Longrightarrow \{x \doteq t\} \cup \{x \mapsto t\}(\mathcal{S}), \text{ falls } x \in Vars(\mathcal{S}) \setminus Vars(t).$$

Die Anwendung einer Substitution auf $\mathcal{S}$ bedeutet, dass diese Substitution auf beide Seiten aller Gleichungen in $\mathcal{S}$ angewandt wird. Die Schreibweise $f(\overline{t_n})$ steht für $f(t_1, \ldots, t_n)$. Das Symbol $\uplus$ steht für die disjunkte Vereinigung. Dies erzwingt, dass die einzelnen Gleichungen auf der linken Seite von $\uplus$ aus der Menge von Gleichungen entfernt wird (obwohl die Operation **Eliminate** sie wieder einfügt).

Die einzelnen Regeln sind einfach zu verstehen:

Delete: löscht triviale Gleichungen.

Decompose: ersetzt Gleichungen zwischen Termen durch Gleichungen zwischen ihren Untertermen.

Orient: setzt die Variablen auf die linke Seite der Gleichung.

Eliminate: übergibt Lösungen und löscht dabei die gelöste Variable aus dem restlichen Teil des Problems.

Das Weglassen der Nebenbedingungen kann zu endlosen Schleifen führen. Zum Beispiel hat $x \in Vars(t)$ in der Regel **Eliminate** folgende Konsequenz:

$$\begin{array}{rcl} \{x \doteq f(x), \ldots x \ldots\} & \Longrightarrow & \{x \doteq f(x), \ldots f(x) \ldots\} \\ & \Longrightarrow & \{x \doteq f(x), \ldots f(f(x)) \ldots\} \Longrightarrow \ldots \ . \end{array}$$

Beispiel 2.1.17 *Der folgende Ablauf von Transformationen veranschaulicht die Arbeitweise der oben eingeführten Regeln:*

$$\begin{array}{lcl} \{x \doteq f(a), g(x,x) \doteq g(x,y)\} & \Longrightarrow & \textbf{\textit{Eliminate}} \\ \{x \doteq f(a), g(f(a), f(a)) \doteq g(f(a), y)\} & \Longrightarrow & \textbf{\textit{Decompose}} \\ \{x \doteq f(a), f(a) \doteq f(a), f(a) \doteq y\} & \Longrightarrow & \textbf{\textit{Delete}} \\ \{x \doteq f(a), f(a) \doteq y\} & \Longrightarrow & \textbf{\textit{Orient}} \\ \{x \doteq f(a), y \doteq f(a)\}. & & \end{array}$$

Zu bemerken ist, dass die Wahl der Regeln nichtdeterministisch geschieht. Wir hätten auch mit **Decompose** statt **Eliminate** beginnen können. Würde dies zu einer anderen gelösten Form führen? Die gelöste Form, die wir zum Schluss erhalten, liefert einen allgemeinsten Unifikator $\{x \mapsto f(a), y \mapsto f(a)\}$ für unser anfängliches Unifikations-Problem. Dies ist keineswegs ein Zufall. Wir behaupten, dass die folgende Funktion *Unify* einen allgemeinsten Unifikator berechnet, falls es einen gibt, und ansonsten fehlschlägt.

$$\begin{array}{rcl} Unify(\mathcal{S}) & = & \textbf{while } \text{es gibt ein } \mathcal{T}, \text{ so dass } \mathcal{S} \Longrightarrow \mathcal{T} \textbf{ do } \mathcal{S} := \mathcal{T}; \\ & & \textbf{if } \mathcal{S} \text{ ist in gelöster Form } \textbf{then return } \vec{\mathcal{S}} \textbf{ else fail.} \end{array}$$

Der Algorithmus *Unify* ist nichtdeterministisch:
Wenn es mehr als eine anwendbare Transformationsregel gibt, d.h. $\mathcal{S} \Longrightarrow \mathcal{T}_1$ und $\mathcal{S} \Longrightarrow \mathcal{T}_2$, sucht sich der Algorithmus eine beliebig aus. Die Terminierung von *Unify* hängt deswegen von der Terminierung von $\Longrightarrow$ ab. Letzteres ist nicht ganz trivial, da die Operation **Eliminate** das Unifikations-Problem vergrößern kann, wie in obigem Beispiel zu sehen ist.

Lemma 2.1.18 *Unify terminiert für alle Eingaben.*

Beweis. Wir folgen [4], Lemma 4.6.5. Eine Variable x wird gelöst genannt, wenn sie genau einmal in $\mathcal{S}$ vorkommt, und zwar auf der linken Seite einer Gleichung $x \doteq t$, wobei $x \notin Vars(t)$. Die Terminierung von $\Longrightarrow$ wird durch eine Messfunktion bewiesen, die ein Unifikations-Problem $\mathcal{S}$ auf ein Tripel (n_1, n_2, n_3) von natürlichen Zahlen abbildet, so dass

- n_1 die Anzahl der nicht gelösten Variablen in $\mathcal{S}$ ist,
- n_2 die Grösse von $\mathcal{S}$ ist, d.h. $\sum_{(s \doteq t) \in \mathcal{S}} (|s| + |t|)$, und
- n_3 die Anzahl der Gleichungen $t \doteq x$ in $\mathcal{S}$ ist.

Die folgende Tabelle zeigt, dass jeder Schritt die Tripel bezüglich der lexikographischen Ordnung kleiner werden lässt:

	n_1	n_2	n_3
Delete	$\geq$	$>$	
Decompose	$\geq$	$>$	
Orient	$\geq$	$=$	$>$
Eliminate	$>$		

Die Interpretation der Tabelle ist offensichtlich:
Eliminate reduziert n_1, welches durch keines der anderen Operationen erhöht werden kann. **Delete** und **Decompose** reduzieren n_2. **Orient** lässt n_2 unverändert, aber reduziert n_3.
Zum Beispiel werden die Transformationsregeln im Beispiel 2.1.17 abgebildet auf $(2, 9, 0) >_{lex} (1, 12, 0) >_{lex} (1, 10, 1) >_{lex} (1, 6, 1) >_{lex} (0, 6, 0)$.

■

Die Haupteigenschaft von $\Longrightarrow$ ist die Erhaltung der zu unifizierenden Mengen:

Lemma 2.1.19 *Wenn $\mathcal{S} \Longrightarrow \mathcal{T}$, dann $\mathcal{U}(\mathcal{S}) = \mathcal{U}(\mathcal{T})$.*

Beweis. [4], Lemma 4.6.6 .

Das folgende Lemma formuliert die Korrektheit der Funktion *Unify* und folgt direkt aus Lemma 2.1.19 und Lemma 2.1.16.

Lemma 2.1.20 *Falls Unify($\mathcal{S}$) eine Substitution σ zurückliefert, dann ist σ ein idempotenter allgemeinster Unifikator von $\mathcal{S}$.*

Der Beweis für die Vollständigkeit benötigt zwei fundamentale Eigenschaften von Termen:

Lemma 2.1.21 *Eine Gleichung der Form $f(\overline{s_m}) \doteq g(\overline{t_n})$, wobei $f \neq g$, hat keine Lösung.*

Beweis. [4], Lemma 4.6.8. Es gilt

$$(f(\overline{s_m}))\sigma = f(\overline{(s_m)\sigma}) \neq g(\overline{(t_n)\sigma}) = (g(\overline{t_n}))\sigma.$$

■

Lemma 2.1.22 *Eine Gleichung der Form $x \doteq t$, wobei $x \in Vars(t)$ und $x \neq t$, hat keine Lösung.*

Beweis. [4], Lemma 4.6.9. Wenn $x \neq t$, dann hat t die Form $f(\overline{t_n})$ mit $x \in Vars(t_i)$ für ein i. Deshalb können $\sigma(x)$ und $\sigma(t)$ nicht identisch sein, weil $|\sigma(x)| \leq |\sigma(t_i)| < |\sigma(t)|$.

■

Nun erhalten wir die Vollständigkeit von *Unify*, d.h. dass jedes lösbare System eine Lösung hat:

Lemma 2.1.23 *Falls $\mathcal{S}$ lösbar ist, dann bricht Unify($\mathcal{S}$) nicht ab (liefert also kein fail zurück).*

Beweis. [4], Lemma 4.6.10. Durch Lemma 2.1.19 genügt es zu zeigen, dass $\mathcal{S}$ in gelöster Form ist, falls $\mathcal{S}$ lösbar und bezüglich $\Longrightarrow$ in Normalform ist. $\mathcal{S}$ kann keine Gleichungen der Form $f(\dots) \doteq f(\dots)$ (wegen der Operation **Decompose**), $f(\dots) \doteq g(\dots)$ (wegen Lemma 2.1.21), $x \doteq x$ (wegen der Operation **Delete**) und $t \doteq x$ mit $t \notin \mathcal{X}$ (wegen der Operation **Orient**) beinhalten. Deshalb besitzen alle Gleichungen in $\mathcal{S}$ die Form $x \doteq t$ mit $x \notin Vars(t)$ (wegen Lemma 2.1.22). Wegen der Operation **Eliminate** kann x nicht mehrfach in $\mathcal{S}$ vorkommen. Deswegen ist $\mathcal{S}$ in gelöster Form.

■

Das folgende Theorem ist eine direkte Konsequenz der Korrektheit, Vollständigkeit und Terminierung von *Unify*.

Theorem 2.1.24 *Falls ein Unifikations-Problem $\mathcal{S}$ eine Lösung besitzt, dann besitzt es auch einen idempotenten allgemeinsten Unifikator.*

Herauszufinden, dass ein Unifikations-Problem unlösbar ist, kann mit den Regeln von $\Longrightarrow$ sehr lange dauern, da man zuerst die Normalform berechnen muss. Deswegen führen wir ein spezielles Unifikations-Problem $\bot$ ein, das keine Lösung besitzt und führen zusätzlich noch zwei neue Regeln ein:

Clash:

$$\{f(\overline{t_m}) \doteq g(\overline{u_n})\} \uplus \mathcal{S} \Longrightarrow \bot \text{ , falls } f \neq g,$$

Occurs-Check:

$$\{x \doteq t\} \uplus \mathcal{S} \Longrightarrow \bot \text{ , falls } x \in Vars(t) \text{ und } x \neq t,$$

deren Begründung aus dem Lemma 2.1.21 und Lemma 2.1.22 folgt. Nun ein kleines Beispiel, dass das Verhalten des Algorithmus mit den erweitertern Regeln zeigt:

$$\{f(x,x) \doteq f(y,g(y))\} \Longrightarrow \{x \doteq y, x \doteq g(y)\} \Longrightarrow \{x \doteq y, y \doteq g(y)\} \Longrightarrow \bot.$$

Da $\bot$ in nicht gelöster Form ist, bricht *Unify* bei Problemen ab, deren Normalform $\bot$ ist. Auch bei folgendem Haskell-Ausdruck tritt der *Occurs-Check*-Fehler auf:

```
(\xs -> case xs of y:ys -> y++ys)
```

Die bisherigen Betrachtungen lassen sich wie folgt zusammenfassen:

1. Das Ergebnis des Algorithmus, ist bis auf Umbenennung der Variablen, eindeutig.

2. Der Algorithmus ist total korrekt. Er terminiert und findet eine allgemeinste Lösung, sofern eine existiert.

Die Komplexität sowie der Speicherverbrauch von *Unify* sind exponentiell.

Beispiel 2.1.25 *Es ist leicht zu sehen, dass das Unifikations-Problem*

$$\{x_1 \doteq f(x_0,x_0), x_2 \doteq f(x_1,x_1), \ldots, x_n \doteq f(x_{n-1},x_{n-1})\}$$

den idempotenten allgemeinsten Unifikator hat:

$$\{x_1 \rightarrow f(x_0,x_0), x_2 \rightarrow f(x_1,x_1), \ldots, x_n \rightarrow f(x_{n-1},x_{n-1})\}.$$

Dieser bildet jedes x_i auf einen vollständigen binären Baum der Höhe i ab. Deshalb ist die Größe von jedem allgemeinsten Unifikator für dieses Beispiel exponentiell in der Größe der Eingabe.
Das obige Unifikations-Problem kann man auch dadurch erhalten, indem

man zwei Terme unifiziert, die jeweils nur aus Variablen und einer Funktion f der Stelligkeit 2 bestehen:

$$\begin{array}{lcllll} s_n(x) & = & f(x_1, & f(x_2, & f(\ldots, x_n &)\ldots)), \\ t_n(x) & = & f(f(x_0, x_0), & f(f(x_1, x_1), & f(\ldots, f(x_{n-1}, x_{n-1}) &)\ldots)). \end{array}$$

Die Namen der x_i auf den rechten Seiten hängen von den Namen der x auf den linken Seiten der Gleichung ab. Dies kann in Beispielen relevant sein, in denen die Terme s_n und t_n unterschiedliche Variablennamen x und y besitzen. Den allgemeinsten Unifikator zu berechnen benötigt bei diesem Beispiel, wegen den Kopien, die gemacht werden, exponentiellen Speicher. Wenn aber die Implementierung im Gegensatz zu Bäumen, auf Graphen mit *sharing* basiert, benötigt man nur linearen Speicherplatz. Diese Implementierungstechnik kann in [4], Abschnitt 4.8, genauer betrachtet werden.

Im Wesentlichen ist *Unify* der gleiche Algorithmus wie der Unifikationsalgorithmus von Herbrand [11]. Die Formulierung des Unifikations-Problems durch diesen Prozess der nichtdeterministischen Anwendung der Regeln wurde 1982 durch Martelli und Montanari [12] eingeführt und hat eine Anwendung als Darstellung für Unifikationsalgorithmen (zum Beispiel [9],[10]).

2.2 Kontextfreie Grammatiken

In diesem Abschnitt werden die kontextfreien Grammatiken und die Anwendungen in der Informatik erläutert. Wir folgen den Ausführungen aus [14]. Die kontextfreien Sprachen sind von großer praktischer Bedeutung, vor Allem bei der Definition von Programmiersprachen, bei der Formalisierung der Syntaxanalyse, beim Vereinfachen der Übersetzung von Programmiersprachen und in anderen Prozessen, bei denen Zeichenketten verarbeitet werden. Zum Beispiel sind kontextfreie Grammatiken nützlich zur Beschreibung korrekt geklammerter arithmetischer Ausdrücke und der Block-Struktur in Programmiersprachen (d.h. korrekte Klammerung der *begins* und *ends*). Keiner dieser Aspekte von Programmiersprachen kann durch reguläre Ausdrücke dargestellt werden. Eine *kontextfreie Grammatik* ist eine endliche Menge von Variablen (auch Nonterminale genannt), von denen jede eine Sprache repräsentiert. Die Wörter, die durch Nonterminale dargestellt werden, werden rekursiv durch die anderen Nonterminale und Terminale beschrieben. Die Regeln, die die Nonterminale untereinander verknüpfen, werden *Produktionen* genannt. Ein Wort aus der Grammatik wird abgeleitet, indem man durch die gegebenen Produktionen die Nonterminale auf den rechten Seiten ersetzt bis man auf Terminale trifft, deren Konkatenation die Zeichenketten (Wörter) bilden.

Definition 2.2.1 [1] *Eine kontextfreie Grammatik ist ein 4-Tupel* $\mathcal{G} = (\mathcal{V}, \mathcal{T}, \mathcal{P}, \mathcal{S})$*, wobei* $\mathcal{V}$ *und* $\mathcal{T}$ *endliche Mengen von Nonterminalen bzw. Terminalen*

[1] Siehe [14]

sind. Wir nehmen an, dass $\mathcal{V}$ und $\mathcal{T}$ disjunkt sind. $\mathcal{P}$ ist eine endliche Menge von Produktionen der Form $A \to \alpha$, wobei A Nonterminal und α eine Zeichenkette von Symbole aus $(\mathcal{V} \cup \mathcal{T})^*$ ist. $\mathcal{S}$ ist das Nonterminal, das als Startsymbol bezeichnet wird.

2.2.1 Ableitungen

Damit man ein Wort aus der Grammatik ableitet, brauchen wir zwei Relationen $\underset{G}{\Longrightarrow}$ und $\overset{*}{\underset{G}{\Longrightarrow}}$ zwischen Zeichenketten in $(\mathcal{V} \cup \mathcal{T})^*$. Wenn $A \to \beta$ eine Produktion aus $\mathcal{P}$ ist, und α und γ Zeichenketten in $(\mathcal{V} \cup \mathcal{T})^*$ sind, dann gilt $\alpha A \gamma \underset{G}{\Longrightarrow} \alpha\beta\gamma$. Durch die Produktion $A \to \beta$ kann man aus $\alpha A \gamma$ direkt in einem Ableitungsschritt $\alpha\beta\gamma$ erzeugen. Die reflexive und transitive Hülle von $\underset{G}{\Longrightarrow}$ ist $\overset{*}{\underset{G}{\Longrightarrow}}$. Seien $\alpha_1, \alpha_2, .., \alpha_m$, $m \geq 1$, Zeichenketten aus $(\mathcal{V} \cup \mathcal{T})^*$, und

$$\alpha_1 \underset{G}{\Longrightarrow} \alpha_2, \alpha_2 \underset{G}{\Longrightarrow} \alpha_3, ..., \alpha_{m-1} \underset{G}{\Longrightarrow} \alpha_m.$$

Dann gilt:

$$\alpha_1 \overset{*}{\underset{G}{\Longrightarrow}} \alpha_m,$$

beziehungsweise ist α_m in der Grammatik aus α_1 ableitbar.
Die von $\mathcal{G}$ erzeugte Sprache sei bezeichnet mit $L(\mathcal{G}) = \{w | w \in \mathcal{T}^* \text{ und } \mathcal{S} \overset{*}{\underset{G}{\Longrightarrow}} w\}$. Also ist $w \in L(\mathcal{G})$, wenn Folgendes gilt:

1. Die Zeichenkette besteht nur aus Terminalen.
2. Die Zeichenkette kann aus $\mathcal{S}$ abgeleitet werden.

Wir nennen L eine *kontextfreie Sprache*, wenn sie $L(\mathcal{G})$ für eine kontextfreie Grammatik $\mathcal{G}$ ist.

Beispiel 2.2.2 *Sei die Grammatik $\mathcal{G} = (\mathcal{V}, \mathcal{T}, \mathcal{P}, \mathcal{S})$, wobei $\mathcal{V} = \{S, F, A, B, K_1, K_2\}$, $\mathcal{T} = \{$(,),a,b,f,x$\}$, $\mathcal{S} = S$ und*

$$\begin{array}{lll} \mathcal{P} = & S \to FK_1ASBK_2 & B \to b \\ & S \to x & K_1 \to (\\ & F \to f & K_2 \to) \\ & A \to a & \end{array}$$

Die Ableitung eines Wortes aus $\mathcal{G}$ sieht dann folgendermaßen aus:

$$S \Rightarrow FK_1ASBK_2 \Rightarrow fK_1ASBK_2 \overset{*}{\Rightarrow}$$
$$f(aSb) \Rightarrow f(aFK_1ASBK_2b) \overset{*}{\Rightarrow} f(af(aSb)b) \Rightarrow f(af(axb)b)$$

Diese Grammatik erzeugt eine beliebig tief verschachtelte Funktion mit drei Argumenten, wobei die Verschachtelung jeweils im zweiten Argumenten stattfindet.

2.2.2 Ableitungsbäume

Ableitungen als Bäume darzustellen dient der Übersichtlichkeit. Diese graphische Darstellung, die man als Ableitungsbaum bezeichnet, gibt den Wörtern einer Sprache eine Struktur, die für Anwendungen, wie die Kompilierung von Programmiersprachen nützlich ist. In einer formalen Defnition sei $\mathcal{G} = (\mathcal{V}, \mathcal{T}, \mathcal{P}, \mathcal{S})$ eine kontextfreie Grammatik. Ein Baum ist ein Ableitungsbaum für $\mathcal{G}$, wenn folgendes gilt:

1. Jeder Knoten hat ein Symbol aus $\mathcal{V} \cup \mathcal{T} \cup \{\epsilon\}$ als Markierung.
2. Die Markierung der Wurzel ist $\mathcal{S}$.
3. Wenn ein innerer Knoten die Markierung A hat, dann muss $A \in \mathcal{V}$ gelten.
4. Wenn n die Markierung A hat und die Knoten $n_1, n_2, .., n_k$, dessen Söhne von links mit den Markierungen $X_1, X_2, ..X_k$ versehen sind, dann muss
$$A \to X_1 X_2 \dots X_k$$
eine Produktion in $\mathcal{P}$ sein.
5. Wenn der Knoten n die Markierung ϵ hat, dann ist n ein Blatt und der einzige Sohn seines Vaters.

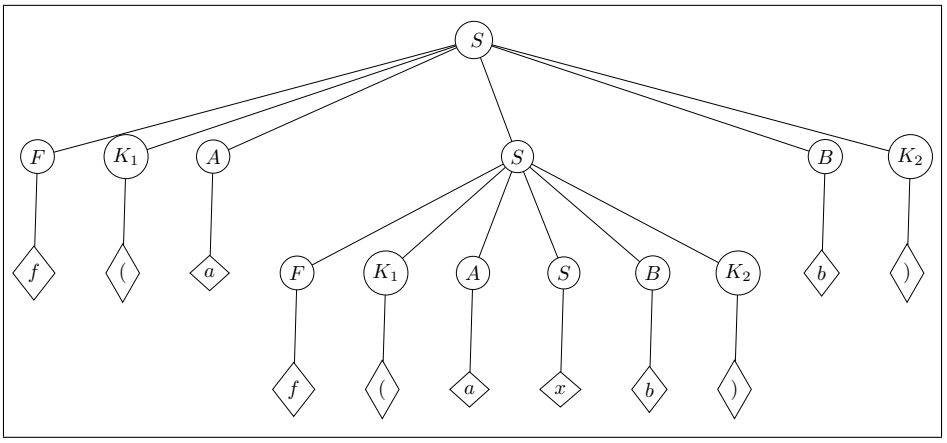


Abbildung 2.1: Ableitungsbaum

2.3 Chomsky Normalform

Es gibt verschiedene Methoden zur Einschränkung des Formats von Produktionen, ohne dass die Fähigkeit zur Erzeugung von Sprachen eingeschränkt wird. Wenn L eine nichtleere kontextfreie Sprache ist, so kann sie durch eine kontextfreie Grammatik $\mathcal{G}$ mit folgenden Eigenschaften erzeugt werden :

1. Jede Variable (Nonterminal) und jedes Terminal von $\mathcal{G}$ erscheint in der Ableitung eines beliebigen Wortes in L.
2. Es gibt keine Produktion der Form $A \to B$, wenn A und B Variablen sind.

Ausserdem sind, wenn ϵ (das leere Wort) nicht in L ist, keine Produktionen der Form $A \to \epsilon$ nötig. Wenn dies der Fall ist, dann kann man verlangen, dass jede Produktion von $\mathcal{G}$ die Form $A \to BC$ oder $A \to a$ hat, wobei A, B und C beliebige Variablen und a ein beliebiges Terminal ist. Diese spezielle Form wird *Chomsky-Normalform* genannt ([14]).

Sei $\mathcal{G} = (\mathcal{V}, \mathcal{T}, \mathcal{P}, \mathcal{S})$ eine Grammatik. Ein Symbol X heisst *nützlich*, wenn es eine Ableitung $S \stackrel{*}{\Rightarrow} \alpha X \beta \stackrel{*}{\Rightarrow} w$ für ein α, β und w gibt, wobei $w \in T^*$. Andernfalls heisst X nutzlos. Es gibt zwei Aspekte der Nützlichkeit :

1. Eine Terminalzeichenkette (Wort) muss aus X ableitbar sein.
2. X muss in einer Zeichenkette auftauchen, die aus $\mathcal{S}$ ableitbar ist.

Das Verfahren zur Entfernung von nutzlosen Nonterminalen aus einer Grammatik ist im Beweis von Lemma 4.1 in [14] zu sehen.
Diese beiden Bedingungen sind jedoch nicht hinreichend, um zu garantieren, dass X nützlich ist, denn X könnte nur in Produktionen als Nonterminal auftauchen, aus denen keine Terminalzeichenkette abgeleitet werden kann.

Beispiel 2.3.1

$$\begin{array}{lll} \mathcal{P} = & S \to AB & A \to a \\ & B \to CDX & C \to c \\ & D \to D & X \to x \end{array}$$

Hier sieht man, dass X sowohl eine Terminalzeichenkette ableitet ($X \to x$) , als auch in einer Zeichenkette auftaucht, die aus $\mathcal{S}$ ableitbar ist ($S \stackrel{*}{\Rightarrow} acDX$). Aber man sieht, dass in dieser Produktion das Nonterminal D verhindert, dass eine Zeichenkette nur aus Terminalen ensteht. Somit ist X, obwohl es die beiden Aspekte der Nützlichkeit erfüllt, ein nutzloses Nonterminal in der Grammatik.

Satz 2.3.2 *Jede nichtleere kontextfreie Sprache wird durch eine kontextfreie Grammatik ohne nutzlose Symbole erzeugt*

Beweis. [14], Satz 4.2.

In kontextfreien Grammatiken kann es unter anderem Produktionen der Form $A \to \epsilon$ geben, die als ϵ-Produktionen bezeichnet werden. Wenn das ϵ zu einer Sprache $L(\mathcal{G})$ gehört, kann diese ϵ-Produktion aus der Grammatik $\mathcal{G}$ nicht entfernt werden, falls allerdings ϵ nicht in der Sprache $L(\mathcal{G})$ liegt, ist

dies der Fall. Das Vorgehen beruht auf der Bestimmung der Nonterminalen A, deren Ableitungen die Form $A \stackrel{*}{\Rightarrow} \epsilon$ haben. Ist dies der Fall, wird A *nullierbar* genannt. Nachdem diese Produktionen gefunden worden sind, geht man wie folgt vor:

Jede Produktion der Form $B \rightarrow X_1, X_2, ..., X_n$ wird folgendermaßen umgewandelt:

1. Alle nullierbaren X_i sind aus der rechten Seite herauszustreichen,
2. Wenn die Produktion nach Ausführung von Schritt 1 die Form $B \rightarrow \epsilon$ hat, so muss auch diese Produktion aus der Grammatik entfernt werden.

Folgender Satz ([14], Satz 4.3) sagt aus, dass aus jeder Grammatik $\mathcal{G}$ die ϵ-Produktionen enthält und eine Sprache erzeugt, in der ϵ als Wort nicht vorhanden ist, eine Grammatik $\mathcal{G}'$ gemacht werden kann, die die gleiche Sprache erzeugt und keine ϵ-Produktionen enthält.

Satz 2.3.3 *Wenn $L = L(\mathcal{G})$ für eine kontextfreie Grammatik $\mathcal{G} = (\mathcal{V}, \mathcal{T}, \mathcal{P}, \mathcal{S})$ gilt, dann ist $L \setminus \{\epsilon\}$ die Sprache $L(\mathcal{G}')$ für eine kontextfreie Grammatik $\mathcal{G}'$ ohne nutzlose Symbole und ϵ-Produktionen.*

Beweis. [14], Satz 4.3.

Produktionen der Form $A \rightarrow B$ werden λ-*Produktionen* genannt.

Satz 2.3.4 *Jede kontextfreie Sprache ohne ϵ ist durch eine Grammatik definiert, die weder nutzlose Symbole, noch ϵ-Produktionen, noch λ-Produktionen enthält.*

Beweis. [14], Satz 4.4 .

Damit man eine kontextfreie Grammatik in Chomsky-Normalform bringen kann, muss man zuerst alle Vorkommen von ϵ und alle nutzlosen Symbole aus der Grammatik entfernen.

Der folgende Satz besagt, dass jede kontextfreie Grammatik eine äquivalente Grammatik besitzt, deren Produktionen bestimmte Eigenschaften erfüllen.

Satz 2.3.5 *(Chomsky-Normalform) Jede kontextfreie Sprache ohne ϵ wird von einer Grammatik erzeugt, in der alle Produktionen von der Form $A \rightarrow BC$ oder $A \rightarrow a$ sind. Hierbei sind A, B, $C \in \mathcal{V}$ und a ist ein Terminalsymbol.*

Beweis. Wir folgen [14], Satz 4.5. Sei $\mathcal{G}$ eine kontextfreie Grammatik, welche eine Sprache erzeugt, die nicht ϵ enthält. Nach Satz 2.3.4 können wir

eine äquivalente Grammatik $\mathcal{G}_1 = (\mathcal{V}, \mathcal{T}, \mathcal{P}, \mathcal{S})$ finden, so dass $\mathcal{P}$ keine λ-Produktionen und keine ϵ-Produktionen enthält. Wenn also eine Produktion nur ein einzelnes Symbol auf der rechten Seite hat, ist dieses Symbol ein Terminal, und die Produktion ist bereits in der gewünschten Form. Nun betrachten wir Produktionen aus $\mathcal{P}$ von der Form:

$$A \to X_1 X_2 \dots X_m \text{ mit } m \geq 2.$$

Ist X_i ein Terminal, so führen wir eine neue Variable (Nonterminal) C_a und eine Produktion $C_a \to a$ (die eine erlaubte Form besitzt) ein. Dann ersetzen wir X_i durch C_a. Die neue Variablenmenge sei $\mathcal{V}'$, die neue Menge von Produktionen $\mathcal{P}'$. Betrachte die Grammatik $\mathcal{G}_2 = (\mathcal{V}', \mathcal{T}, \mathcal{P}', \mathcal{S})$ [2]. Wenn $\alpha \underset{G_1}{\Longrightarrow} \beta$ gilt, dann gilt auch $\alpha \underset{\mathcal{G}_2}{\Longrightarrow} \beta$ und folglich gilt $L(\mathcal{G}_1) \subseteq L(\mathcal{G}_2)$. Wir zeigen nun durch Induktion über die Anzahl der Schritte in einer Ableitung, dass $A \overset{*}{\underset{\mathcal{G}_1}{\Longrightarrow}} w$ gilt, wenn $A \overset{*}{\underset{\mathcal{G}_2}{\Longrightarrow}} w$ für $A \in \mathcal{V}$ und $w \in \mathcal{T}^*$ gilt. Das Ergebnis ist trivial für aus einem Schritt bestehende Ableitungen. Gelte das Ergebnis nun für Ableitungen von bis zu k Schritten. Sei $A \overset{*}{\underset{\mathcal{G}_2}{\Longrightarrow}} w$ eine Ableitung mit $k+1$ Schritten. Der erste Schritt muss von der Form $A \to B_1 B_2 \dots B_m$ mit $m \geq 2$ sein. Wir können $w = w_1 w_2 \dots w_m$ schreiben, wobei $B_i \overset{*}{\underset{\mathcal{G}_2}{\Longrightarrow}} w_i$ für $1 \leq i \leq m$ gilt. Wenn B_i das zu einem Terminalzeichen a_i gehörende C_{a_i} ist, dann muss $w_i = a_i$ sein. Gemäß der Konstruktion von $\mathcal{P}'$ gibt es eine Produktion $A \to X_1 X_2 \dots X_m$ aus $\mathcal{P}$, wobei $X_i = B_i$, falls $B_i \in \mathcal{V}$, und $X_i = a_i$, falls $B_i \in \mathcal{V}' \setminus \mathcal{V}$ ist. Für die $B_i \in \mathcal{V}$ wissen wir, dass die Ableitung $B_i \overset{*}{\underset{\mathcal{G}_1}{\Longrightarrow}} w_i$ nicht mehr als k Schritte benötigt, so dass nach Induktionsannahme $X_i \overset{*}{\underset{\mathcal{G}_1}{\Longrightarrow}} w_i$ gilt. Demnach gilt $A \overset{*}{\underset{\mathcal{G}_1}{\Longrightarrow}} w$. Wir haben jetzt das Zwischenergebnis bewiesen, dass jede kontextfreie Sprache von einer Grammatik erzeugt werden kann, in der jede Produktion entweder von der Form $A \to a$ oder $A \to B_1 B_2 \dots B_m$ mit $m \geq 2$ ist. Hier sind $A, B_1, B_2, \dots, B_m$ Variablen, und a ist ein Terminal. Wir betrachten nun eine solche Grammatik $\mathcal{G}_2 = (\mathcal{V}', \mathcal{T}, \mathcal{P}', \mathcal{S})$: Wir modifizieren $\mathcal{G}_2$, indem wir einige Symbole zu $\mathcal{V}'$ hinzufügen und einige Produktionen aus $\mathcal{P}'$ ersetzen. Für jede Produktion $A \to B_1 B_2 \dots B_m$ mit $m \geq 3$ erzeugen wir neue Variablen $D_1, D_2, \dots, D_{m-2}$ und ersetzen $A \to B_1 B_2 \dots B_m$ durch die Menge der Produktionen

$$\{A \to B_1 D_1, D_1 \to B_2 D_2, \dots, D_{m-3} \to B_{m-2} D_{m-2}, D_{m-2} \to B_{m-1} B_m\}.$$

Sei $\mathcal{V}''$ die neue Variablenmenge, $\mathcal{P}''$ die neue Menge von Produktionen und $\mathcal{G}_3 = (\mathcal{V}'', \mathcal{T}, \mathcal{P}'', \mathcal{S})$. $\mathcal{G}_3$ ist in Chomsky-Normalform. Es ist klar, dass $A \overset{*}{\underset{\mathcal{G}_3}{\Longrightarrow}} w$ gilt, falls $A \overset{*}{\underset{\mathcal{G}_2}{\Longrightarrow}} w$ gilt und somit gilt $L(\mathcal{G}_2) \subseteq L(\mathcal{G}_3)$. Doch es gilt ebenfalls $L(\mathcal{G}_3) \subseteq L(\mathcal{G}_2)$, was im Wesentlichen genauso gezeigt werden kann, wie $L(\mathcal{G}_2) \subseteq L(\mathcal{G}_1)$.

[2] G_2 ist noch nicht in Chomsky-Normalform

■

Beispiel 2.3.6 *Betrachten wir die Grammatik* $\mathcal{G} = (\{S, A, B, C, D, E, K\}, \{a, e, f, k\}, \mathcal{P}, \mathcal{S})$*, die folgende Produktionen hat:*

$$\begin{array}{lll} \mathcal{P} = & S \to ABC & A \to a \\ & A \to \epsilon & B \to DE \\ & C \to KEE & D \to fK \\ & E \to e & K \to k. \end{array}$$

Dazu wollen wir eine äquivalente Grammatik in Chomsky-Normalform bilden.

Zunächst betrachten wir die Produktionen, die schon die richtige Form besitzen. Diese sind $B \to DE$, $E \to e$, $A \to a$ und $K \to k$. In der Grammatik gibt es eine ϵ-Produktion, $A \to \epsilon$, die wir entfernen müssen (da $\epsilon \notin L(\mathcal{G})$). Wenn wir diese Produktion entfernen, müssen wir in die Grammatik eine neue Produktion einfügen, damit sich die Sprache, die von $\mathcal{G}$ erzeugt wird, nicht verändert. Der Grund dafür ist, dass die Sprache sowohl Wörter enthält, die mit dem Terminalsymbol f, als auch Wörter, die mit dem Terminalsymbol a beginnen. Dies kann man daran erkennen, da das Nonterminal A zwei Produktionen besitzt, nämlich:

1. $A \to \epsilon$ und
2. $A \to a$.

Die erste Produktion ermöglicht es Wörter zu bilden, die mit dem Terminal f anfangen, womit folgende Produktion entsteht:

$S \to BC$ (die Ableitungsschritte dazu sind: $S \Rightarrow ABC \Rightarrow \epsilon BC = BC$).

Die zweite Produktion ermöglicht es Wörter zu bilden, die mit einem Terminal a anfangen, womit folgende Produktion entsteht:

$S \to ABC$ (die Ableitungsschritte dazu sind: $S \Rightarrow ABC \Rightarrow aBC \ldots$).

Wenn man die Produktion $A \to \epsilon$ einfach entfernen würde, könnte man keine Wörter mehr bilden, die mit einem Terminal f beginnen. Um dies zu vermeiden und das richtige Ergebnis beizubehalten, muss die Produktion $S \to BC$ hinzugefügt werden. Jetzt kann die ϵ-Produktion entfernt werden, ohne die Sprache zu verändern. Die Grammatik $\mathcal{G}'$ sieht nach dem Entfernen der ϵ-Produktion wie folgt aus:

$$\mathcal{G}' = (\{S, A, B, C, D, E, K\}, \{a, e, f, k\}, \mathcal{P}', \mathcal{S})$$

mit

$$\begin{array}{lll} \mathcal{P}' = & S \to ABC & A \to a \\ & S \to BC & B \to DE \\ & C \to KEE & D \to fK \\ & E \to e & K \to k. \end{array}$$

Im nächsten Schritt ersetzen wir die Terminalsymbole auf den rechten Seiten mit den entsprechenden Nonterminalen, die diese Terminalsymbole erzeugen. Hierbei werden Produktionen der Form $A \to a$ nicht betrachtet. Falls Terminalsymbole existieren, die von keinem Nonterminal erzeugt werden, muss ein neues Nonterminal eingeführt werden, dass dieses Terminalsymbol produziert. In diesem Beispiel trifft dies nur auf die Produktion $D \to fK$ zu. Das Terminalsymbol f hat kein zugehöriges Nonterminal. Deswegen erzeugen wir ein neues Nonterminal F und fügen die Produktion $F \to f$ zu $\mathcal{P}'$ hinzu. Nun können wir das Terminal f durch das Nonterminal F ersetzen. Die resultierende Grammatik $\mathcal{G}''$ sieht wie folgt aus:

$$\mathcal{G}'' = (\{S, A, B, C, D, E, K, F\}, \{a, e, f, k\}, \mathcal{P}'', \mathcal{S})$$

mit

$$\begin{array}{lll} \mathcal{P}'' = & S \to ABC & A \to a \\ & S \to BC & B \to DE \\ & C \to KEE & D \to FK \\ & E \to e & K \to k \\ & F \to f. & \end{array}$$

Die einzigen Produktionen, die nicht in Chomsky-Normalform sind, haben die Form

$$A \to B_1 B_2 \dots B_m \text{ mit } m \geq 3.$$

Die Produktionen, um die es sich im Beispiel handelt sind $S \to ABC$ und $C \to KEE$. Hierfür benutzen wir die Regeln aus dem Beweis von Satz 2.3.5. Wir erzeugen zwei neue Produktionen N_1 und N_2, die wie folgt aussehen:

1. $N_1 \to BC$ und
2. $N_2 \to EE$.

Diese Produktionen fügen wir zu $\mathcal{P}'$ hinzu. Nun setzen wir das N_1 und N_2 an den entsprechenden Stellen ein und erhalten die endgültige Grammatik

$$\mathcal{G}''' = (\{S, A, B, C, D, E, K, F, N_1, N_2\}, \{a, e, f, k\}, \mathcal{P}''', \mathcal{S})$$

mit

$$\begin{array}{lll} \mathcal{P}''' = & S \to AN_1 & A \to a \\ & S \to BC & B \to DE \\ & C \to KN_2 & D \to FK \\ & E \to e & K \to k \\ & F \to f & N_1 \to BC \\ & N_2 \to EE & . \end{array}$$

2.4 Singleton Contextfree Grammar

Wir betrachten eine spezielle Klasse von kontextfreien Grammatiken. Die Sprachen von Nonterminalen für SCFGs sind immer eindeutig, weil sie nur eine Produktion besitzen. Jedes abgeleitete Wort w ist wohldefiniert. In diesem Abschnitt folgen wir den Definitionen aus [15].

Definition 2.4.1 *Eine Singleton-Kontextfreie-Grammatik (SCFG) $\mathcal{G}$ ist ein 3-Tupel $(\mathcal{N}, \mathcal{T}, \mathcal{R})$, wobei $\mathcal{N}$ die Menge der Nonterminale, $\mathcal{T}$ die Menge der Terminalsymbole und $\mathcal{R}$ die Menge der Produktionen der Form*

$$X \to \alpha, \text{ wobei } X \in \mathcal{N} \text{ und } \alpha \in (\mathcal{N} \cup \mathcal{T})^*$$

ist. Die Mengen $\mathcal{N}$ und $\mathcal{T}$ müssen disjunkt sein. Für alle Nonterminale $X \in \mathcal{N}$ gilt:

$$|\{X \to \alpha \in \mathcal{R} \mid \alpha \in (\mathcal{N} \cup \mathcal{T})^*\}| = 1.$$

D.h., dass es für jedes Nonterminal $X \in \mathcal{N}$ exakt eine Produktion gibt. Die SCFG enthält keine Rekursion. Formaler bedeutet dies, dass die transitive Hülle $>_G^+$, die von allen Nonterminalen $X >_G Y$ erzeugt wird, falls $X \to \alpha_1 Y \alpha_2$, terminieren muss. Das durch ein Nonterminal $X \in \mathcal{N}$ aus $\mathcal{G}$ erzeugte Wort wird als $w_{\mathcal{G},X}$ geschrieben. Wenn $\mathcal{G}$ eindeutig ist, werden wir auch w_X schreiben. Mit dieser Bezeichnung wird das Wort aus $\mathcal{T}^*$ gekennzeichnet, das durch die Hintereinanderausführung der Regeln aus $\mathcal{R}$, startend von X, erhalten wird.

Manche Autoren setzen voraus, dass die Singleton-Kontextfreie-Grammatik in Chomsky-Normalform ist. In der obigen Definition ist diese Einschränkung nicht vorhanden. Aber es ist klar, dass man jede SCFG in die Chomsky-Normalform transformieren kann (Satz 2.3.5).

Definition 2.4.2 *Die Größe einer Singleton-Kontextfreien-Grammatik ist die Anzahl der Produktionen, die sie beinhaltet, und wird als $|\mathcal{G}|$ bezeichnet. Die Tiefe T eines Nonterminals X ist definiert als*

$$T(X) = 1 + \max(T(X_1), T(X_2)), \text{ falls die Regel } X \to X_1 X_2 \text{ ist.}$$

Die Tiefe $T(\mathcal{G})$ einer Grammatik $\mathcal{G} = (\mathcal{N}, \mathcal{T}, \mathcal{R})$ ist definiert als

$$T(\mathcal{G}) = \max(T(X_1), T(X_2), \ldots, T(X_m)), \text{ wobei } X_1, X_2, \ldots, X_m \text{ alle Nonterminale aus } \mathcal{N} \text{ sind.}$$

Wir definieren einige Operationen auf Singleton-Kontextfreien-Grammatiken.

Definition 2.4.3 *Die Erweiterung einer SCFG $\mathcal{G} = (\mathcal{T}, \mathcal{N}, \mathcal{R})$ ist eine SCFG $\mathcal{G}' = (\mathcal{T}', \mathcal{N}', \mathcal{R}')$ mit $\mathcal{T} \subseteq \mathcal{T}'$, $\mathcal{N} \subseteq \mathcal{N}'$ und $\mathcal{R} \subseteq \mathcal{R}'$.*

Wir notieren drei wohlbekannte Lemmata:

Lemma 2.4.4 *Gegeben sei eine Singleton-Kontextfreie-Grammatik $\mathcal{G}$, ein Nonterminal N und eine Zahl n, mit $0 < n < |w_N|$. Eine erweiterte Grammatik $\mathcal{G}'$ kann in polynomieller Zeit, in Abhängigkeit von der Größe der Grammatik $|\mathcal{G}|$, aus $\mathcal{G}$ erzeugt werden, so dass $\mathcal{G}'$ ein Nonterminal N' enthält, wobei $|N'| = n$, und $w_{N'}$ ist ein Suffix (oder Präfix) von w_N. Ferner gilt $|\mathcal{G}'| \leq |\mathcal{G}| + \mathrm{T}(\mathcal{G})$ und $T(\mathcal{G}') = T(\mathcal{G})$.*

Beweis. [15], Lemma 3.6 .

Lemma 2.4.5 *Gegeben sei eine Singleton-Kontextfreie-Grammatik $\mathcal{G}$, ein Nonterminal N und eine Zahl n, mit $n > 0$. Eine erweiterte Grammatik $\mathcal{G}'$ kann in polynomieller Zeit, in Abhängigkeit von der Größe der Grammatik $|\mathcal{G}|$ und $\log(n)$, aus $\mathcal{G}$ erzeugt werden, so dass $\mathcal{G}'$ ein Nonterminal N' enthält und $w_{N'} = w_{N^n}$ gilt. Ferner gilt $|\mathcal{G}'| \leq |\mathcal{G}| + 2 \cdot \lceil \log(n) \rceil$ und $T(\mathcal{G}') \leq T(\mathcal{G}) + \lceil \log(n) \rceil$.*

Beweis. [15], Lemma 3.7 .

Lemma 2.4.6 *Gegeben sei eine Singleton-Kontextfreie-Grammatik $\mathcal{G}$ und Nonterminale $N_1, \ldots, N_n$, mit $n > 0$. Eine erweiterte Grammatik $\mathcal{G}'$ kann in polynomieller Zeit, in Abhängigkeit von der Größe der Grammatik $|\mathcal{G}|$ und n, aus $\mathcal{G}$ erzeugt werden, so dass $\mathcal{G}'$ ein Nonterminal N' enthält und $w_{N'} = w_{N_1} \ldots w_{N_n}$ gilt. Ferner gilt $|\mathcal{G}'| \leq |\mathcal{G}| + n - 1$ und $T(\mathcal{G}') = T(\mathcal{G}) + \lceil \log(n) \rceil$.*

Beweis. [15], Lemma 3.8 .

Beispiel 2.4.7 *Sei $\mathcal{G} = (\mathcal{N}, \mathcal{T}, \mathcal{R})$ eine Singleton-Kontextfreie-Grammatik, mit $\mathcal{N} = \{A_1, \ldots, A_7\}$, $\mathcal{T} = \{p, f, h, g, a, b, x_1, x_2\}$ und sei die Menge der Produktionen $\mathcal{R}$ wie folgt,*

$$\begin{aligned} \mathcal{R} = \quad & A_1 \to pfA_2A_3 && A_2 \to ab \\ & A_3 \to gA_4A_4A_5A_7 && A_4 \to x_1 \\ & A_5 \to hA_6 && A_6 \to A_7 \\ & A_7 \to x_2. \end{aligned}$$

Mit dieser Grammatik kann man zum Beispiel das Wort aus Beispiel 1.2.1 erzeugen. Der Ableitungsbaum hierzu ist in Abbildung 2.2 zu sehen.

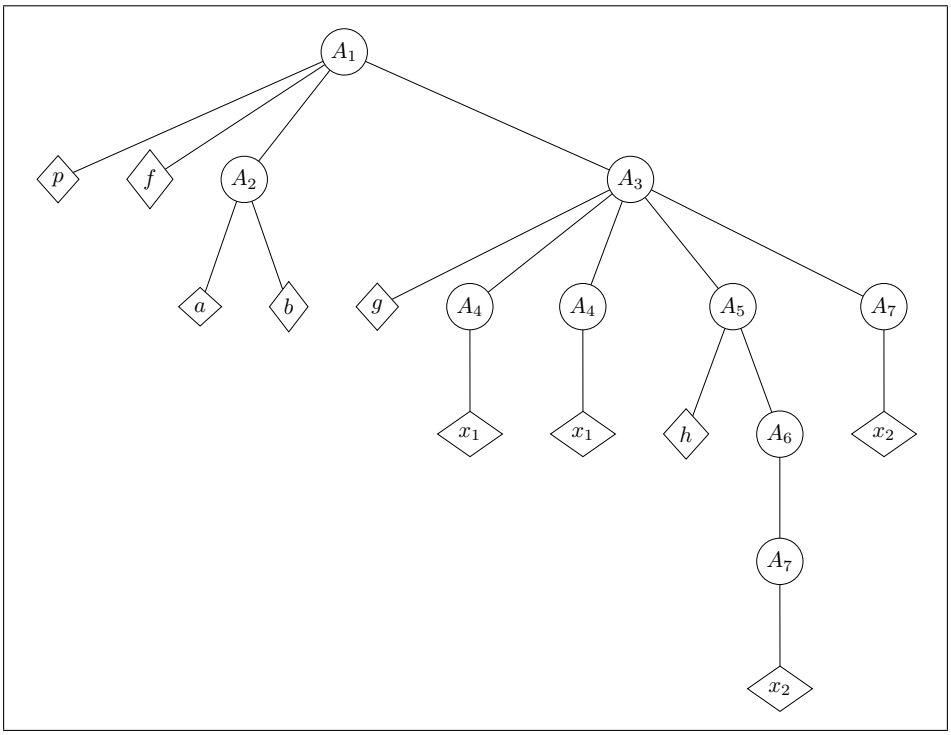


Abbildung 2.2: Ableitungsbaum SCFG

Beispiel 2.4.8 *Sei $\mathcal{G} = (\mathcal{N}, \mathcal{T}, \mathcal{R})$ eine Singleton-Kontextfreie-Grammatik, mit $\mathcal{N} = \{A_1, \ldots, A_5\}$, $\mathcal{T} = \{f, h, a, x_1\}$ und sei die Menge der Produktionen $\mathcal{R}$ wie folgt,*

$$\begin{aligned} \mathcal{R} = \quad & A_1 \to A_2 A_3 \quad A_3 \to f A_2 A_6 \\ & A_2 \to a \qquad A_4 \to x_1 \\ & A_5 \to A_4 A_6 \quad A_6 \to h A_3 \\ & A_3 \to a. \end{aligned}$$

Dieses Beispiel ist keine Singleton-Kontextfreie-Grammatik:

1. Das Nonterminal A_3 hat zwei Produktionen.
2. In der Grammatik ist eine Rekursion zwischen den Nonterminalen A_3 und A_6 vorhanden.

Singleton Tree Grammar

Singleton-Baum-Grammatiken (engl. Singleton-Tree-Grammar, STG) sind eine Erweiterung der Singleton-Kontextfreien-Grammatiken. Es werden Sprachen betrachtet, die von Bäumen mit Rang (ranked trees) erzeugt werden und die Ausdrucksstärke von Singleton-Kontextfreien-Grammatiken wird erweitert, indem Kontexte in der Grammatik zugelassen werden.

Definition 2.4.9 *Eine Singleton-Baum-Grammatik (STG) ist ein 4-Tupel* $\mathcal{G} = (\mathcal{TN}, \mathcal{CN}, \Sigma, \mathcal{R})$*, wobei* $\mathcal{TN}$ *die Menge der Baum/Tree-Nonterminale der Stelligkeit 0,* $\mathcal{CN}$ *die Menge der Kontext-Nonterminale der Stelligkeit 1, und* Σ *eine Signatur von Funktionssymbolen (den Terminalen) ist, so dass die Mengen* $\mathcal{TN}$*,* $\mathcal{CN}$ *und* Σ *paarweise disjunkt sind. Die Menge aller Nonterminale* $\mathcal{N}$ *ist definiert als*

$$\mathcal{N} := \mathcal{TN} \cup \mathcal{CN}.$$

Die Produktionen in $\mathcal{R}$ *haben folgende Form:*

- $A \to f(A_1, \ldots, A_m)$*, wobei* $A, A_i \in \mathcal{TN}$ *und* $f \in \Sigma$*, mit* $ar(f) = m$*.*
- $A \to C_1 A_2$*, wobei* $A, A_2 \in \mathcal{TN}$ *und* $C_1 \in \mathcal{CN}$*.*
- $C \to [\cdot]$*, wobei* $C \in \mathcal{CN}$*.*
- $C \to C_1 C_2$*, wobei* $C_i \in \mathcal{CN}$*.*
- $C \to f(A_1, \ldots, A_{i-1}, C_i, A_{i+1}, \ldots, A_m)$*, wobei* $A_1, \ldots, A_{i-1}, A_{i+1}, \ldots, A_m \in \mathcal{TN}, C_i \in \mathcal{CN}$ *und* $f \in \Sigma$ *mit* $ar(f) = m$*.*
- $A \to A_1$*, (*λ*-Produktion), wobei* $A, A_1 \in \mathcal{TN}$*.*
- $A \to x$*, wobei* $A \in \mathcal{TN}$ *und* $x \in \Sigma$*.*

Seien N_1 *und* N_2 *zwei Nonterminale. Falls* $N_1 \to t$ *und* N_2 *in* t *vorkommt, dann gilt* $N_1 >_{\mathcal{G}} N_2$*. Die STG enthält keine Rekursion. Formaler bedeutet dies, dass die transitive Hülle von* $>_G^+$ *terminieren muss. Des Weiteren existiert für jedes Nonterminal* $N \in \mathcal{N}$ *exakt eine Produktion, dessen linke Seite das* N *enthält. Gegeben sei ein Term* t *in dem Nonterminale vorkommen. Die Ableitung von* t *in der Grammatik* $\mathcal{G}$ *ist eine iterative Prozedur, indem jedes Nonterminal durch die entsprechende rechte Seite seiner Produktion ersetzt wird. Das Ergebnis wird als* $w_{\mathcal{G},t}$ *bezeichnet. Der durch ein Nonterminal* $N \in \mathcal{N}$ *aus* $\mathcal{G}$ *erzeugte Term wird als* $w_{\mathcal{G},N}$ *geschrieben. Wenn* $\mathcal{G}$ *eindeutig ist, so werden wir auch* w_N *schreiben.*

Zu beachten ist, dass hier Σ statt $\mathcal{F}$ (Kapitel 2) verwendet wird, um sowohl first-order Variablen, als auch Konstanten benutzen zu können.

Definition 2.4.10 *Die Größe einer Singleton-Baum-Grammatik ist die Summe aller Größen der Produktionen die sie beinhaltet, und wird als* $|\mathcal{G}|$ *bezeichnet, wobei die Größe einer Produktion* $X \to \alpha$ *definiert ist als*

$$|X| := 1 + |\alpha|.$$

Die Tiefe $T(N)$ *eines Nonterminals* N *in der Grammatik* $\mathcal{G}$ *ist rekursiv definiert als*

$$T(N) := 1 + \max\{(T(N')) \mid N' \text{ ist ein Nonterminal in } \alpha \text{ und } N \to \alpha \in \mathcal{G}\},$$ *wobei das Maximum über eine leere Menge 0 gesetzt wird.*

Nun definieren wir einige Operationen auf Singleton-Baum-Grammatiken.

Definition 2.4.11 *Die Erweiterung einer Singleton-Baum-Grammatik* $\mathcal{G} = (\mathcal{TN}, \mathcal{CN}, \Sigma, \mathcal{R})$ *ist eine Singleton-Baum-Grammatik* $\mathcal{TN}', \mathcal{CN}', \Sigma', \mathcal{R}'$ *mit* $\mathcal{TN} \subseteq \mathcal{TN}'$, $\mathcal{CN} \subseteq \mathcal{CN}'$,$\Sigma \subseteq \Sigma'$ *und* $\mathcal{R} \subseteq \mathcal{R}'$.

Lemma 2.4.12 *Gegeben sei eine Singleton-Baum-Grammatik* $\mathcal{G}$ *und ein Nonterminal* D *aus* $\mathcal{G}$. *Dann kann* $head(w_D)$[3] *polynomiell in* $|\mathcal{G}|$ *ermittelt werden.*

Beweis. [15], Lemma 5.1 .

Korollar 2.4.13 *Gegeben sei eine Singleton-Baum-Grammatik* $\mathcal{G}$ *und zwei Nonterminale* D_1 *und* D_2 *aus* $\mathcal{G}$. *Dann kann polynomiell in* $|\mathcal{G}|$ *ermittelt werden, ob* $head(w_{D_1}) = head(w_{D_2})$.

Definition 2.4.14 *Gegeben sei eine Singleton-Baum-Grammatik* $\mathcal{G}$ *und eine Singleton-Kontextfreie-Grammatik* $\mathcal{G}'$. *Des Weiteren seien die Terminalsymbole positive Integer und sei* $\mathcal{CN}_\mathcal{G} \rightarrow \mathcal{N}'$ *eine (partielle) injektive Abbildung, so dass für jedes Kontext-Nonterminal* $C \mapsto C_S$ *gilt. Wenn* $hp(w_{\mathcal{G},C}) = w_{\mathcal{G}',C_S}$, *für alle* $C \in \mathcal{CN}_\mathcal{G}$, *dann ist* $\mathcal{G}'$ *eine Positionsgrammatik von* $\mathcal{G}$.

Hier ist zu beachten, dass C_S die Kontexte in der Positionsgrammatik sind. Also wird jedes Kontext-Nonterminal aus der Singleton-Baum-Grammatik auf ein Nonterminal aus der Positionsgrammatik abgebildet.

Definition 2.4.15 *Gegeben sei eine Singleton-Baum-Grammatik* $\mathcal{G}$. *Wir definieren eine Singleton-Kontextfreie-Grammatik* $\mathcal{G}_{p\mathcal{G}}$, *dessen Terminalsymbole positive Integer sind, wie folgt:*

Für jedes Kontext-Nonterminal C *aus* $\mathcal{G}$ *existiert ein eindeutiges Nonterminal* C' *in* $\mathcal{G}_{p\mathcal{G}}$.

Die Regeln dazu sind:

1. $C' := i$, *wenn* $C \rightarrow f(A_1, \ldots, A_{i-1}, C_i, A_{i+1}, \ldots, A_m)$ *eine Produktion in* $\mathcal{R}_\mathcal{G}$ *ist.*
2. $C' := C'_1 C'_2$, *wenn* $C \rightarrow C_1 C_2$ *eine Produktion in* $\mathcal{R}_\mathcal{G}$ *ist.*

Lemma 2.4.16 *Gegeben sei eine Singleton-Baum-Grammatik* $\mathcal{G}$. *Dann kann* $\mathcal{G}_{p\mathcal{G}}$ *in polynomieller Zeit konstruiert werden. Ferner ist* $\mathcal{G}_{p\mathcal{G}}$ *eine Positionsgrammatik.*

Beweis. [15], Lemma 5.5 .

[3] Das erste Symbol von w_D

Lemma 2.4.17 *Gegeben sei eine Singleton-Baum-Grammatik $\mathcal{G}$ und eine Positionsgrammatik $\mathcal{G}_\mathcal{S}$. Für ein Nonterminal D aus $\mathcal{G}$ und ein Nonterminal N aus $\mathcal{G}_\mathcal{S}$, kann $w_N \in Pos(w_D)$ in polynomieller Zeit, in Abhängigkeit von $|\mathcal{G}|$ und $|\mathcal{G}_\mathcal{S}|$, überprüft werden.*

Beweis. [15], Lemma 5.6 .

Die Positionsgrammatik wird in späteren Kapiteln genauer besprochen.

Beispiel 2.4.18 *Sei $\mathcal{G} = (\mathcal{TN}, \mathcal{CN}, \Sigma, \mathcal{R})$ eine Singleton-Baum-Grammatik, mit $\mathcal{TN} = \{A_1, \ldots, A_9\}$, $\mathcal{CN} = \{C_1, C_2\}$, $\Sigma = \{p, f, h, g, a, b, x_1, x_2\}$ und sei die Menge der Produktionen $\mathcal{R}$ wie folgt*

$$\begin{array}{lll} \mathcal{R} = & A_1 \to pA_2A_3A_4 & A_2 \to fA_5A_6 \\ & A_5 \to a & A_6 \to b \\ & A_3 \to gA_7A_7 & A_7 \to x_1 \\ & A_4 \to C_1A_9 & C_1 \to hC_2A_8 \\ & C_2 \to [\cdot] & A_8 \to x_2 \\ & A_8 \to x_2. & \end{array}$$

Sei p eine Funktion mit drei Argumenten und seien f, g und h jeweils Funktionen mit zwei Argumenten. Weiterhin seien a und b Konstantensymbole und x_1, x_2 Variablen.

Mit dieser Grammatik kann man zum Beispiel den Term aus Beispiel 1.2.1 erzeugen. Der Ableitungsbaum hierzu ist in Abbildung 2.3 zu sehen.

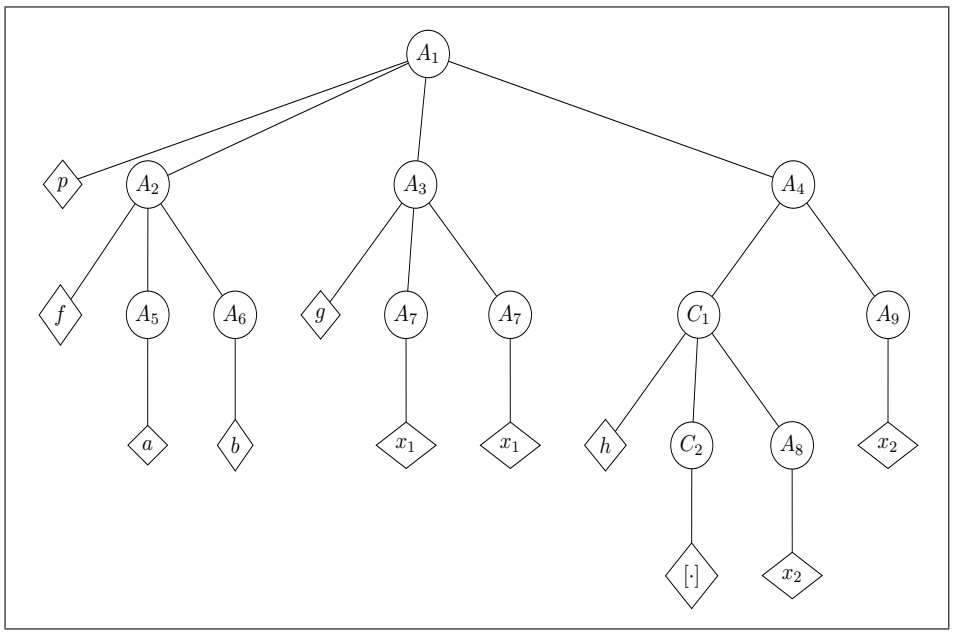


Abbildung 2.3: Ableitungsbaum STG

Man sollte sich nicht durch den Ableitungsbaum verwirren lassen, denn eigentlich wird das Nonterminal A_9 in das Loch von dem Kontext C_1 eingesetzt. Aus dem Baum könnte man naiv auffassen, dass das Nonterminal A_8 in das Loch von dem Kontext C_2 eingesetzt wird. Dies kann aber nicht der Fall sein, weil die Funktion h zwei Argumente erwartet, welche C_2 und A_8 sind (das erste Argument von h ist C_2 und das zweite A_8, also $h(C_2, A_8)$), womit das Nonterminal A_8 nicht im Loch von dem Kontext C_2 eingesetzt werden kann, da dann die Funktion h nur ein Argument hätte.

Beispiel 2.4.19 *Sei* $\mathcal{G} = (\mathcal{TN}, \mathcal{CN}, \Sigma, \mathcal{R})$ *eine Singleton-Baum-Grammatik, mit* $\mathcal{TN} = \{A_1, \ldots, A_4\}$, $\mathcal{CN} = \{C_1, C_2, C_3\}$, $\Sigma = \{p, g, b, x_1\}$ *und sei die Menge der Produktionen* $\mathcal{R}$ *wie folgt*

$$\begin{array}{lll} \mathcal{R} = & A_1 \to gA_4 & A_2 \to A_3A_1 \\ & C_1 \to C_2C_3 & C_2 \to [\cdot] \\ & A_3 \to x_1 & C_3 \to x_1 \\ & A_4 \to pA_2C_1A_2. & \end{array}$$

Dieses Beispiel ist keine Singleton-Baum-Grammatik, denn

1. Die Produktion des Term-Nonterminals A_4 ist nicht zugelassen.
2. Die Produktion des Kontext-Nonterminals C_3 ist nicht zugelassen.
3. Es gibt eine Rekursion zwischen den Term-Nonterminalen A_1, A_4 und A_2 in der Grammatik.

Kapitel 3

Der Algorithmus von Plandowski

Die in Kapitel 2.4 erwähnten Singleton-Kontextfreien-Grammatiken wurden von Plandowski eingeführt [16]. Die Idee hierbei ist, Wörter, die exponentiell lang sein können, in einer kompakten Form als Grammatiken darzustellen. Plandowski hat bewiesen, dass zwei Wörter w_A und w_B in polynomieller Zeit, in Abhängigkeit von der Größe der Grammatik, auf Gleichheit überprüft werden können, wobei das Wort w_A durch das Nonterminal A und das Wort w_B durch das Nonterminal B einer Singleton-Kontextfreien-Grammatik, erzeugt wird. Hierbei handelt es sich um das sogenannte Wortproblem.

Theorem 3.0.20 *Gegeben sei eine Singleton-Kontextfreie-Grammatik $\mathcal{G}$ und zwei Nonterminale N_1 und N_2. Dann kann in polynomieller Zeit, in Abhängigkeit von der Größe der Grammatik, entschieden werden ob $w_a = w_b$.*

Beweis. [16].

Eine zu diesem Theorem äquivalente Aussage ist die Folgende:

Wenn zwei unterschiedliche Grammatiken $\mathcal{G}_1$, $\mathcal{G}_2$ und zwei Nonterminale N_1 aus $\mathcal{G}_1$ und N_2 aus $\mathcal{G}_2$ gegeben sind, kann man in polynomieller Zeit überprüfen, ob $w_{\mathcal{G}_1,N_1} = w_{\mathcal{G}_2,N_2}$ gilt.

Beispiel 3.0.21 *Gegeben sei eine Sprache, deren Wörter folgende Syntax enthält:*

$$P ::= a \mid b \mid P^n \mid (P) \mid PP \ ,$$

wobei a und b Terminalsymbole und n eine natürliche Zahl ist. Mit dieser Grammatik ist es möglich a^{1000} für das Wort $\underbrace{a \ldots a}_{1000}$ zu schreiben. Ein etwas komplizierteres Beispiel wäre $ab^{100}(ab^{100}a)^{15}$.

Wenn man die obigen Terme zuerst expandiert, also das komplette Wort als String betrachtet, dauert die Überprüfung auf Gleicheit der Terme exponentielle Zeit. Ein nichttrivialer Test, der erfolgreich durchgeführt werden sollte, ist, ob $(ab)^{100}a = a(ba)^{100}$ gilt. Es ist einfach diesen Test auf einer kleinen SCFG durchzuführen, wobei die Anzahl der Produktionen polynomiell in der Länge des expandierten Wortes beschränkt ist (hier 21). Beispielsweise, kann der Term a^{1000} durch eine Singleton-Kontextfreie-Grammatik dargestellt werden, die folgende Produktionen enthält:

$$\begin{array}{ll} N_1 \to a & N_2 \to N_1N_1 \\ N_3 \to N_2N_2 & N_4 \to N_3N_3 \\ N_5 \to N_4N_4 & N_6 \to N_5N_5 \\ N_7 \to N_6N_6 & N_8 \to N_7N_7 \\ N_9 \to N_8N_8 & M_9 \to N_9M_8 \\ M_8 \to N_8M_7 & M_7 \to N_7M_6 \\ M_6 \to N_6M_5 & M_5 \to N_5N_3. \end{array}$$

Hier erzeugt das Nonterminal M_9 das Wort a^{1000}. Nachdem das Wort anhand der Regeln erzeugt worden ist, stellt das Theorem 3.0.20 sicher, dass der Gleichheitstest in polynomieller Zeit durchgeführt werden kann.
Die Ergebnise von Plandowski wurden auf die Singleton-Baum-Grammatiken übertragen ([2], [15]), wodurch das Wortproblem auch für Singleton-Baum-Grammatiken entscheidbar ist.

Theorem 3.0.22 *Sei $\mathcal{G}$ eine Singleton-Baum-Grammatik und seien A, B zwei Term-Nonterminale aus $\mathcal{G}$. Dann kann in der Zeit von $\mathcal{O}(|G|^3)$ entschieden werden, ob $w_A = w_B$ gilt.*

3.1 Der Algorithmus

Der Algorithmus arbeitet mit sogenannten Testmengen, welche die Wörter enthalten, die auf Gleichheit überprüft werden sollen. Das Problem hierbei ist, dass ein Wort in der Testmenge exponentiell lang sein kann. Noch allgemeiner könnte das kürzeste Wort einer kontextfreien Sprache, deren Grammatik n Produktionen besitzt, eine exponentielle Länge haben. Deswegen ist es für manche kontextfreie Sprachen unmöglich alle Wörter der Testmenge in polynomieller Zeit aufzulisten. Um dieses Problem zu umgehen, wird jedes Wort in eine kleine kontextfreie Grammatik (SCFG) kodiert. Jedes Wort hat dann ein zugehöriges Nonterminal von dem es erzeugt wird. Diese Nonterminale werden statt den langen Wörtern in die Testmenge eingefügt. Die Definition der Singleton-Kontextfreien-Grammatik ist hierbei um eine Eigenschaft eingeschränkt, nämlich, dass sie in Chomsky-Normalform sein muss, mit der Ausnahme, dass nutzlose Nonterminale in der Grammatik erlaubt sind.

Man kann das Problem wie folgt formulieren:

Das Problem: Sei $\mathcal{G}$ eine Grammatik, die eine Menge von Wörtern erzeugt, und sei $\mathcal{S}$ eine Menge von Tupeln, die aus den Nonterminalen der Grammatik besteht.

Das Ziel: Überprüfe für jedes Paar (A, B) aus $\mathcal{S}$ ob $w_A = w_B$.

Sei $\mathcal{G}$ eine Grammatik mit n Produktionen, die in Chomsky-Normalform ist. Die Länge der kürzesten Worte, die von einem Nonterminal ableitbar sind, überschreiten 2^n nicht. Das bedeutet, dass die Länge von keinem Wort 2^n überschreitet. Die grundlegende Idee des Algorithmus ist es, die Beziehungen zwischen den Wörtern zu betrachten. Der Algorithmus speichert die Beziehungen in einer Menge von Tripeln (A, B, i) ab, wobei A und B Nonterminale oder Terminalsymbole sind, und i eine natürliche Zahl ist. Die Menge von Tripeln kann wiederum in zwei Mengen aufgeteilt werden, welche wir $Suffix$- und $Subword$-Tripel nennen. Sei $|w|$ die Länge eines Wortes w, dann gilt für ein Tripel (A, B, i) das Folgende:

$$(A, B, i) \text{ ist ein } \begin{cases} Suffix - \text{Tripel, wenn } i + |w_B| \geq |w_A|. \\ Subword - \text{Tripel, wenn } i + |w_B| < |w_A|, \text{ mit } i \geq 1. \end{cases}$$

Die Basisrelation unserer Überlegungen heißt $SUFSUB(\mathcal{G})$. Für ein Tripel (A, B, i) gilt:

$(A, B, i) \in SUFSUB(\mathcal{G})$, wenn

- (A, B, i) ein $Suffix$-Tripel ist und $w_A[i{+}1 \ldots |w_A|] = w_B[1 \ldots |w_A|{-}i]$, und
- (A, B, i) ein $Subword$-Tripel ist und $w_A[i + 1 \ldots i + |w_B|] = w_B$.

Also ist nicht jedes $Suffix$- bzw. $Subword$-Tripel in $SUFSUB(\mathcal{G})$ enthalten. Nur Tripel, die der Eigenschaft genügen, dass auch die Symbole an diesen Stellen gleich sind, werden in $SUFSUB(\mathcal{G})$ aufgenommen. Die folgenden Abbildungen sollen $SUFSUB(\mathcal{G})$ veranschaulichen:

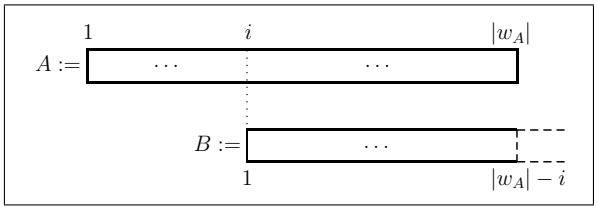


Abbildung 3.1: *Suffix*-Tripel

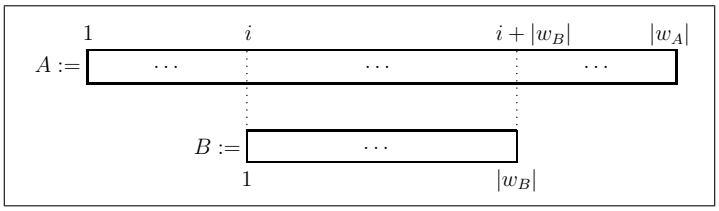


Abbildung 3.2: *Subword*-Tripel

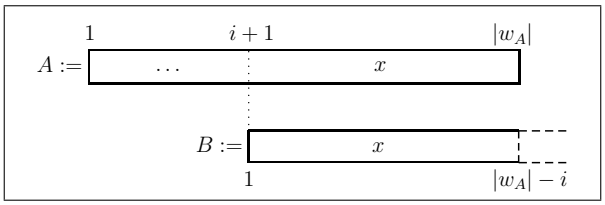


Abbildung 3.3: *Suffix*-Tripel in *SUFSUB*

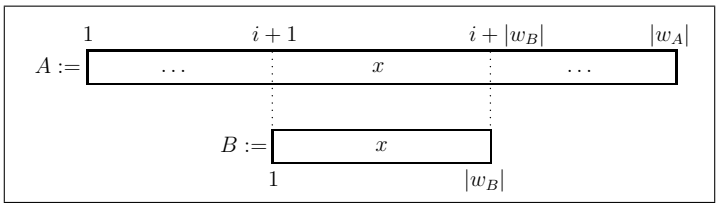


Abbildung 3.4: *Subword*-Tripel in *SUFSUB*

Diesmal sind die Teilwörter x aus w_A und y aus w_B identisch, wobei die $Suffix$- bzw. *Subword*-Tripel-Eigenschaft erfüllt werden muss. Nur in diesem Fall wird ein Tripel in $SUFSUB(\mathcal{G})$ aufgenommen.

Jetzt kann das Problem neu dargelegt werden:

Gegeben sei eine Grammatik $\mathcal{G}$ und eine Menge $\mathcal{S}$ von Tupeln, welche aus den Nonterminalen von $\mathcal{G}$ bestehen. Für jedes Paar $(A, B) \in \mathcal{S}$ soll überprüft werden, ob $|w_A| = |w_B|$ und $(A, B, 0) \in SUFSUB(\mathcal{G})$.

Die neue Darlegung des Problems ist leicht zu verstehen. Die erste Voraussetzung zur Überprüfung, ob zwei Worte gleich sind, ist, die Längengleichheit der Worte ($|w_A| = |w_B|$). Die zweite Voraussetzung ist, dass bei beiden Worten die Zeichen ab der ersten Stelle übereinstimmen müssen ($(A, B, 0) \in SUFSUB(\mathcal{G})$). Nachdem die Problemstellung klar definiert worden ist, kann man überlegen, welche Hilfsmittel benötigt werden: Hierzu wird eine Menge rel von Tripeln definiert. Danach wird eine Funktion $split(A, rel)$ eingeführt, die alle Vorkommen von dem Nonterminal A aus den

Tripeln aus *rel* entfernt, ohne die Information in *rel* zu verlieren. Angenommen, es gibt eine Produktion $A \rightarrow EF$ in der Grammatik. Für ein einziges Tripel aus *rel* ist die Operation $split(A, (B, C, i))$ wie folgt definiert:

- $A \neq B$ und $A \neq C$:

$$split(A, (B, C, i)) = (B, C, i)$$

- $A = B$ und $A \neq C$:

$$split(A, (A, C, i)) := \begin{cases} (E, C, i) \cup (C, F, |w_E - i|) \\ \text{wenn } |w_E| > i \text{ und } |w_C| + i > |w_E|, \\ \\ (E, C, i) \\ \text{wenn } |w_E| > i \text{ und } |w_C| + i \leq |w_E|, \\ \\ (C, F, 0) \\ \text{wenn } |w_E| = i \text{ und } |w_C| \leq |w_F|, \\ \\ (F, C, 0) \\ \text{wenn } |w_E| > i \text{ und } |w_C| > |w_F|, \\ \\ (F, C, i - |w_E|) \\ \text{wenn } |w_E| < i. \end{cases}$$

- $A \neq B$ und $A = C$:

$$split(A, (B, A, i)) := \begin{cases} (B, E, i) \\ \text{wenn } |w_E| + i \geq |w_B|, \\ \\ (B, E, i) \cup (B, F, |w_E + i|) \\ \text{wenn } |w_E| + i < |w_B|. \end{cases}$$

- $A = B$ und $A = C$:

$$split(A,(A,A,i)) := \begin{cases} \emptyset \\ \text{wenn } i = 0, \\ \\ (E,E,i) \cup (E,F,|w_E| - i) \\ \text{wenn } |w_E| > i \geq 1 \text{ und } i \geq |w_F|, \\ \\ (E,E,i) \cup (F,F,i) \cup (E,F,|w_E| - i) \\ \text{wenn } |w_E| > i \geq 1 \text{ und } i < |w_F|, \\ \\ (E,F,0) \\ \text{wenn } |w_E| = i \text{ und } |w_E| \geq |w_F|, \\ \\ (F,E,0) \cup (F,F,i) \\ \text{wenn } |w_E| = i \text{ und } |w_E| < |w_F|, \\ \\ (F,E,i - |w_E|) \\ \text{wenn } |w_E| < i \text{ und } i \geq |w_F|, \\ \\ (F,E,i - |w_E|) \cup (F,F,i) \\ \text{wenn } |w_E| < i \text{ und } i < |w_F|. \end{cases}$$

Für die ganze *rel* Menge wird *split* wie folgt definiert:

$$split(A, rel) = \bigcup_{(B,C,i) \in rel} split(A,(B,C,i)).$$

Die Definition für *split* sieht zwar kompliziert aus, aber die Idee ist das Entfernen des Nonterminals A aus den Tripeln in rel, ohne dabei die Beziehung zwischen den einzelnen Wörtern zu beeinträchtigen. Mit anderen Worten: Es wird anstelle des Nonterminals A die dazugehörige Produktion eingesetzt, womit man im Ableitungsbaum eine Tiefe runter geht (da man eigentlich einen Ableitungsschritt durchgeführt hat). Das nächste Lemma zeigt, dass die *rel* Menge vor der Durchführung von $split(A, rel)$ und danach äquivalent ist.

Lemma 3.1.1 *Es gilt* $rel \subseteq SUFSUB(\mathcal{G})$*, wenn* $split(A, rel) \subseteq SUFSUB(\mathcal{G})$ *für ein beliebiges Nonterminal* A*.*

Beweis. [16], Lemma 7.

Sei $\#suffix(rel)$ die Anzahl der $Suffix$-Tripel und $\#subword(rel)$ die Anzahl der *Subword*-Tripel in rel. Weiterhin gelte:

$$\#sufsub(rel) := \#suffix(rel) + \#subword(rel).$$

Dann ist klar, dass die Anzahl der Tripel in rel $\#sufsub(rel)$ ist.

Lemma 3.1.2 *Sei rel eine Menge von Tripeln, so dass für jedes* $(B, C, i) \in rel$ $|w_A| \geq \max\{|w_B|, |w_C|\}$. *Dann gilt:*

$$\#suffix(split(A, rel)) \leq 3\#sufsub(rel),$$

$$\#subword(split(A, rel)) \leq \#subword(rel) + 3\#suffix(rel).$$

Beweis. [16], Lemma 8.

Die zweite Operation, die für rel eingeführt wird, ist $compact(rel)$. Sie reduziert die Anzahl der $Suffix$-Tripel in rel. Davor benötigen wir noch ein weiteres Lemma.

Lemma 3.1.3 *Wenn x und y Perioden in einem Wort w sind und* $x + y \leq |w|$, *dann ist* $gcd(x, y)$ *auch eine Periode in w, wobei gcd für den größten gemeinsamen Teiler steht.*

Beweis. [16], Lemma 9.

Die Operation $SimpleCompact(rel)$ wird folgendermaßen definiert: Wenn es drei $Suffix$-Tripel $(A, B, i), (A, B, j), (A, B, k)$ in rel gibt, so dass $(j - i) + (k - i) \leq |w_A| - i$ und $i < j < k$ gilt, dann ersetzt die Operation $SimpleCompact$ diese durch zwei Tripel $(A, B, i), (A, B, i + gcd(j - i, k - i))$. Wenn es keine solchen drei Tripel gibt, schlägt die Operation fehl. Ein $compact(rel)$ für die Menge rel entsteht durch die Anwendung von $SimpleCompact(rel)$, bis es fehlschlägt. Das bedeutet, dass ein Aufruf von $compact(rel)$ die Überprüfung von je drei Tripeln auf die obige Eigenschaft (alle möglichen dreier Kombinationen zwischen den Tripeln wird überprüft), durch die Operation $SimpleCompact(rel)$ bezweckt.

Lemma 3.1.4 *Es gilt* $rel \subseteq SUFSUB(\mathcal{G})$ *genau dann, wenn* $compact(rel) \subseteq SUFSUB(\mathcal{G})$

Beweis. [16], Lemma 10.

Das nächste Lemma macht eine Aussage über die Anzahl der $Suffix$-Tripel in rel nach der Operation $compact(rel)$.

Lemma 3.1.5 *Sei rel eine Menge von Tripeln. Dann gilt:*

$$\#suffix(compact(rel)) \leq (2n + 1) * n^2.$$

Beweis. [16], Lemma 11.

Algorithmus *Test;*
{**Eingabe**:

1. Eine Grammatik $\mathcal{G}$,
2. Eine Menge $\mathcal{S}$ von Tupel, die aus den Nonterminaln von $\mathcal{G}$ bestehen.

Ausgabe:

für jedes $(A, B) \in \mathcal{S}$ teste ob $w_A = w_B$ gilt.}

begin
berechne für jedes Nonterminal A die Länge $|w_A|$;
if $\exists (A, B) \in \mathcal{S}$, so dass $|w_A| \neq |w_B|$ ***then***
return false;
else $(A_1, \ldots, A_n) :=$ Sortiere NTs in absteigender Reihenfolge anhand von $|w_A|$;
$rel := \bigcup_{(A,B)\in\mathcal{S}}(A, B, 0)$;
for $i = 1$ ***to*** n ***do***
begin
$rel := split(A_i, rel)$;
$rel := compact(rel)$;
end;
{$\nexists$ Nonterminal in den Tripel von rel}
if $\exists (a, b, 0) \in suffix$ und $a \neq b$ ***then***
return false;
else
return true;
end

Die Eingabe des Algorithmus ist die Grammatik und die Testmenge, in der sich die Tupel, bestehend aus Nonterminalen, befinden, deren Wörter auf Gleichheit überprüft werden sollen. Als erstes muss die Länge von jedem Nonterminal berechnet werden. Der Trick hierbei ist, nicht die Wörter zu erzeugen, weil man einen exponentiellen Faktor erreicht. Stattdessen muss man auf der gegebenen Grammatik arbeiten. Die Idee hierbei ist, dass von bestimmten Nonterminalen die Längen schon bekannt sind (nämlich den Nonterminalen, die die Blätter, also die Terminalsymbole, erzeugen $(A \rightarrow b)$). Diese müssen auf den rechten Seiten der übrigen Produktionen eingesetzt werden, wodurch neue Nonterminale hinzukommen, deren Längen bekannt sind. Mit den neuen Nonterminalen muss die gleiche Prozedur rekursiv weitergemacht werden, bis man von allen Nonterminalen die Längen berechnet hat. Also arbeitet man den Baum von den Blättern zu der Wurzel auf. Diese Methode findet immer von jedem Nonterminal die Länge, ansonsten ist die Grammatik falsch, denn jedes Nonterminal in der Grammatik hat exakt eine Produktion, dessen linke Seite dieses Nonterminal enthält. Danach betrachtet man die Tupel in $\mathcal{S}$. Die Längen der Wörter, die durch die Nonterminale

in einem Tupel gebildet werden, werden verglichen. Dies wird für jedes Tupel durchgeführt. Falls die Längen in einem Tupel ungleich sind, bricht der Algorithmus ab und liefert das Ergebnis *false*. Dies ist klar, da Wörter ungleicher Länge nicht gleich sein können. Falls alle Wörter in den Tupeln die gleiche Länge haben, sortiert man zunächst die Nonterminale $A_1, \ldots, A_n$ der Grammatik in absteigender Reihenfolge anhand von $|w_{A_i}|$, mit $1 \leq i \leq n$. Danach wird die rel_0 Menge gebildet, indem aus jedem Tupel $(A, B) \in \mathcal{S}$ ein Tripel $(A, B, 0)$ erzeugt wird. Jetzt wird für jedes Nonterminal, beginnend beim Ersten, nach der Sortierung zuerst die Operation *split* und anschließend die Operation *compact* ausgeführt. Nachdem dies für jedes Nonterminal durchgeführt worden ist, darf es in den Tripeln von rel_n keine Nonterminale mehr geben (falls doch, wird *false* zurückgegeben). Wenn jetzt in rel_n ein Tripel $(a, b, 0) \in Suffix$-Tripel existiert und $a \neq b$ ist, dann sind die Wörter nicht gleich und der Algorithmus liefert *false*. Andernfalls sind die Wörter gleich, womit die Nonterminale in den Tupeln der Testmenge das gleiche Wort erzeugen, und der Algorithmus *true* liefert.

Beispiel 3.1.6 *Gegeben sei eine SCFG* $\mathcal{G} = (\mathcal{N}, \mathcal{T}, \mathcal{R})$ *mit* $\mathcal{N} = \{A-1, \ldots, A_6\}$, $\mathcal{T} = \{a\}$ *und sei die Menge der Produktionen* $\mathcal{R}$ *wie folgt*

$$\begin{array}{lll} \mathcal{R} = & A_1 \to A_2A_3 & A_2 \to A_4A_4 \\ & A_3 \to A_5A_5 & A_4 \to a \\ & A_5 \to a & A_6 \to A_3A_2. \end{array}$$

Nun möchten wir wissen, ob die Nonterminale A_1 *und* A_6 *das gleiche Wort bilden.*

Der Algorithmus geht wie folgt vor:

1. Die Eingabe ist die Grammatik $\mathcal{G}$ und die Testmenge $\mathcal{S} = \{(A_1, A_6)\}$.
2. Die Länge jedes Wortes wird berechnet, also:

$$|w_{A_1}| = 4, |w_{A_2}| = 2, |w_{A_3}| = 2, |w_{A_4}| = 1, |w_{A_5}| = 1 \text{ und } |w_{A_6}| = 4.$$

3. Es wird überprüft, ob $|w_{A_1}| \neq |w_{A_6}|$:

$$|w_{A_1}| = 4 = |w_{A_6}| = 4.$$

4. Die Nonterminale werden in absteigender Reihenfolge entsprechend $|w_{A_i}|$, mit $1 \leq i \leq 6$, sortiert:

$$(A_1, A_6, A_2, A_3, A_4, A_5).$$

5. Die Menge rel_0 wird gebildet:

$$rel_0 = \{(A_1, A_6, 0)\}.$$

6. Jetzt wird für jedes Nonterminal, beginnend mit A_1, *split* und *compact* ausgeführt:

   (a) $split(A_1, rel_0)$

   - $split(A_1, (A_1, A_6, 0))$. Der zweite Fall von *split* tritt ein, denn $A_1 = A_1$ und $A_1 \neq A_6$. Wir benötigen noch die Produktion von A_1, $A_1 \rightarrow A_2A_3$. Nun überprüfen wir welche Eigenschaft zutrifft:
     - $|w_{A_2}| = 2 > i = 0$ (trifft zu),
       $|w_{A_6}| + i = 4 + 0 > |w_{A_2}| = 2$ (trifft zu).

     Also wird das Tripel $(A_1, A_6, 0)$ durch zwei neue Tripel $(A_2, A_6, 0)$ und $(A_6, A_3, 2)$ ersetzt und die neue *rel* Menge ist $rel_1 = \{(A_2, A_6, 0), (A_6, A_3, 2)\}$.
   - $compact(rel_1) = rel_1$.

   (b) $split(A_6, rel_1)$

   - $split(A_6, (A_2, A_6, 0))$. Der dritte Fall von *split* tritt ein, denn $A_6 \neq A_2$ und $A_6 = A_6$. Wir benötigen noch die Produktion von A_6, $A_6 \rightarrow A_3A_2$. Nun überprüfen wir welche Eigenschaft zutrifft:
     - $|w_{A_3}| + i = 2 + 0 \geq |w_{A_2}| = 2$ (trifft zu).

     Also wird das Tripel $(A_2, A_6, 0)$ durch ein neues Tripel $(A_2, A_3, 0)$ ersetzt.
   - $split(A_6, (A_6, A_3, 2))$. Der zweite Fall von *split* tritt ein, denn $A_6 = A_6$ und $A_6 \neq A_3$. Nun überprüfen wir welche Eigenschaft zutrifft:
     - $|w_{A_3}| > 2$ (trifft nicht zu).
     - $|w_{A_3}| > 2$ (trifft nicht zu).
     - $|w_{A_3}| = 2$ (trifft zu),
       $|w_{A_3}| = 2 \leq |w_{A_2}| = 2$ (trifft zu).

     Das Tripel $(A_6, A_3, 2)$ wird durch ein neues Tripel $(A_3, A_2, 0)$ ersetzt. Die neue *rel* Menge ist $rel_2 = \{(A_2, A_3, 0), (A_3, A_2, 0)\}$.
   - $compact(rel_2) = rel_2$.

   (c) $split(A_2, rel_2)$

   - $split(A_2, (A_2, A_3, 0))$. Der zweite Fall von *split* tritt ein. Nun überprüfen wir welche Eigenschaft zutrifft:
     - $|w_{A_4}| = 1 > i = 0$ (trifft zu),
       $|w_{A_3}| + i = 2 + 0 > |w_{A_4}| = 1$ (trifft zu).

     Also wird das Tripel $(A_2, A_3, 0)$ durch $(A_4, A_3, 0)$ und $(A_3, A_4, 1)$ ersetzt.

- $split(A_2, (A_3, A_2, 0))$. Der dritte Fall von *split* tritt ein:
  - $|w_{A_4}| + i = 1 + 0 \geq |w_{A_3}| = 2$ (trifft nicht zu).
  - $|w_{A_4}| + i = 1 + 0 < |w_{A_3}| = 2$ (trifft zu).

  Das Tripel $(A_3, A_2, 0)$ wird durch $(A_3, A_4, 0)$ und $(A_3, A_4, 1)$ ersetzt. Die neue *rel* Menge ist $rel_3 = \{(A_4, A_3, 0), (A_3, A_4, 0), (A_3, A_4, 1)\}$.
- $compact(rel_3) = rel_3$.

(d) $split(A_3, rel_3)$

- $split(A_3, (A_4, A_3, 0))$. Der dritte Fall von *split* tritt ein. Nun überprüfen wir, welche Eigenschaft zutrifft:
  - $|w_{A_5}| + i = 1 + 0 \geq |w_{A_4}| = 1$ (trifft zu).

  $(A_4, A_3, 0)$ wird durch $(A_4, A_5, 0)$ ersetzt.
- $split(A_3, (A_3, A_4, 0))$. Der zweite Fall von *split* tritt ein:
  - $|w_{A_5}| = 1 > i = 0$ (trifft zu),
    $|w_{A_4}| + i = 1 + 0 > |w_{A_5}| = 1$ (trifft nicht zu).
  - $|w_{A_5}| = 1 > i = 0$ (trifft zu),
    $|w_{A_4}| + i = 1 + 0 \leq |w_{A_5}| = 1$ (trifft zu).

  Das Tripel $(A_3, A_4, 0)$ wird durch $(A_5, A_4, 0)$ ersetzt.
- $split(A_3, (A_3, A_4, 1))$. Der zweite Fall von *split* tritt ein:
  - $|w_{A_5}| = 1 > i = 1$ (trifft nicht zu).
  - $|w_{A_5}| = 1 > i = 1$ (trifft nicht zu).
  - $|w_{A_5}| = 1 = i = 1$ (trifft zu),
    $|w_{A_4}| = 1 \leq |w_{A_5}| = 1$ (trifft zu).

  $(A_3, A_4, 1)$ wird ersetzt durch $(A_4, A_5, 0)$. Die neue *rel* Menge ist $rel_4 = \{(A_5, A_4, 0), (A_4, A_5, 0)\}$.
- $compact(rel_4) = rel_4$.

(e) $split(A_4, rel_4)$

- $split(A_4, (A_5, A_4, 0))$. Der dritte Fall von *split* tritt ein:
  - $|a| + i = 1 + 0 \geq |w_{A_5}| = 1$ (trifft zu).

  $(A_5, A_4, 0)$ wird durch $(A_5, a, 0)$ ersetzt.
- $split(A_4, (A_4, A_5, 0))$. Der zweite Fall von *split* tritt ein:
  - $|a| = 1 > i = 0$ (trifft zu),
    $|w_{A_5}| + i = 1 + 0 > |a| = 1$ (trifft nicht zu).
  - $|a| = 1 > i = 0$ (trifft zu),
    $|w_{A_5}| + i = 1 + 0 \leq |a| = 1$ (trifft zu).

  Das Tripel $(A_4, A_5, 0)$ wird durch $(a, A_5, 0)$ ersetzt. Die neue *rel* Menge ist $rel_5 = \{(A_5, a, 0), (a, A_5, 0)\}$.
- $compact(rel_5) = rel_5$.

(f) $split(A_5, rel_5)$

- $split(A_5, (A_5, a, 0))$. Der zweite Fall von *split* tritt ein:

  - $|a| = 1 > i = 0$ (trifft zu),
    $|a| + i = 1 + 0 > |a| = 1$ (trifft nicht zu).

  - $|a| = 1 > i = 0$ (trifft zu),
    $|a| + i = 1 + 0 \leq |a| = 1$ (trifft zu).

  $(A_5, a, 0)$ wird durch $(a, a, 0)$ ersetzt.

- $split(A_5, (a, A_5, 0))$. Der dritte Fall von *split* tritt ein:

  - $|a| + i = 1 + 0 \geq |a| = 1$ (trifft zu).

  Das Tripel $(a, A_5, 0)$ wird durch $(a, a, 0)$ ersetzt. Die neue *rel* Menge ist $rel_6 = \{(a, a, 0)\}$.

- $compact(rel_6) = rel_6$.

7. In den Tripeln der endgültigen Menge rel_6 tauchen keine Nonterminale mehr auf.

8. Das Tripel $(a, a, 0)$ wird auf die letzte Eigenschaft überprüft:

   * $\exists (a, b, 0) \in suffix$? Ja, $(a, a, 0) \in suffix$,

   * $a \neq b$? Nein, $a = b = a$.

9. Der Algorithmus liefert *true*, das Nonterminal A_1 und A_6 bilden das gleiche Wort.

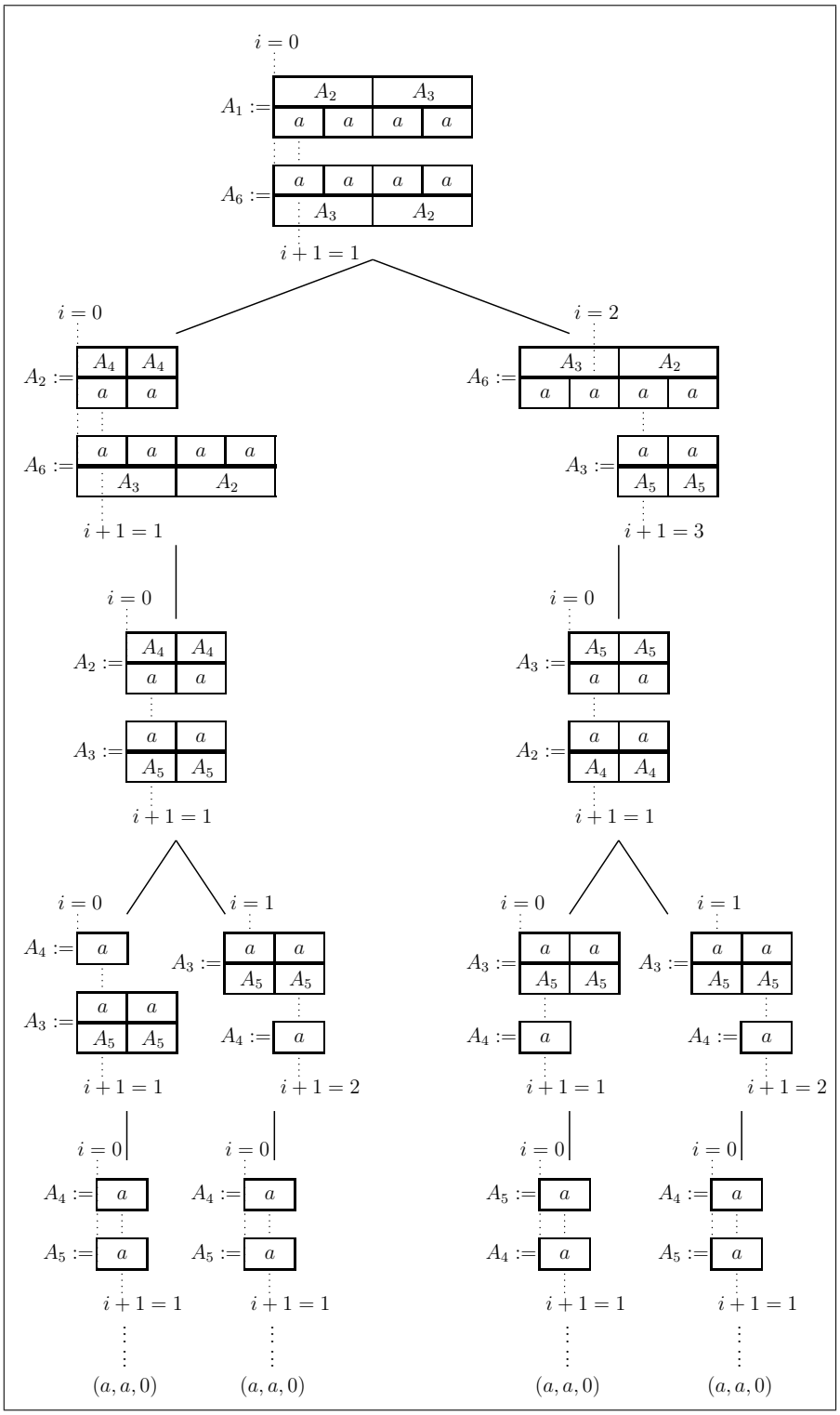


Abbildung 3.5: Split-Baum für Beispiel 3.9

Die Abbildung 3.5 verdeutlicht die Arbeitsweise von *split*. Es ist leicht zu sehen, wieso man die Information zwischen den Wörtern tatsächlich nicht verliert. Jede Höhe im Baum repräsentiert eine *rel* Menge (rel_i entspricht der *rel* Menge in der Höhe i, wobei die Wurzel die Höhe $i = 0$ besitzt). In der Wurzel steht die Menge $rel_0 = \{(A_1, A_6, 0)\}$, also die Testmenge, die der Algorithmus zur Überprüfung bekommt. Nun teilt *split* das Tripel $(A_1, A_6, 0)$ in zwei neue Tripel $(A_2, A_6, 0)$ und $(A_6, A_3, 2)$ auf. Die Produktion von A_1 wurde an den entsprechenden Stellen bei A_6 aufgeteilt, womit keine Position zwischen den Wörtern übersprungen, verändert oder gelöscht worden ist. Das Gleiche wird nun mit A_6 für die neuen Tripel durchgeführt usw., bis man an den Blättern ankommt, also die Tripel in *rel* keine Nonterminale mehr enthalten. Es ist klar, dass in einer Menge gleiche Elemente nur einmal aufgelistet werden. Aber um die Arbeitsweise von *split* zu verstehen, ist im Baum diese Mengeneigenschaft nicht vorhanden. Die Blätter, also die letzte *rel* Menge, zeigen nämlich, dass jede Stelle der Wörter w_{A_1} und w_{A_6} verglichen wird. Dies geschieht nicht durch das Erzeugen der Wörter, um sie dann Stelle für Stelle zu vergleichen, sondern *split* arbeitet auf der Grammatik, ohne die Wörter zu expandieren.

3.2 Die Laufzeit

Theorem 3.2.1 *Der Algorithmus Test hat eine von der Größe der Eingabe abhängige polynomielle worst-case Laufzeit.*

Beweis. [16], Theorem 12 .

Sei rel_i die Menge *rel* vor der i-ten Iteration des Algorithmus. Die Korrektheit des Algorithmus folgt aus Lemma 3.1.1 und Lemma 3.1.4. Um zu zeigen, dass der Algorithmus eine polynomielle worst-case Laufzeit hat, genügt es zu beweisen, dass die Anzahl der $sufsub$-Tripel in rel_i ($\#sufsub(rel_i)$) polynomiell beschränkt ist. Anfangs ist $\#subword(rel_0) = 0$ und $\#suffix(rel_0) = |\mathcal{S}| \leq n^2$. Das ist klar, denn *rel* erzeugt am Anfang aus allen Tupeln $\in \mathcal{S}$ Tripel der Form $(A, B, 0)$ (welche $Suffix$-Tripel sind) und die Anzahl der Tupel in $\mathcal{S}$ kann höchstens n^2 sein, wobei n die Anzahl der Nonterminale in der Grammatik ist, denn für ein Nonterminal A kann man n viele Tupel bilden (inklusive des Tupels (A, A)) und somit für n Nonterminale insgesamt $n \cdot n = n^2$ viele. Aus Lemma 3.1.2 folgt:

$$\#suffix(rel_i) \leq (2n + 1) \cdot n^2, \text{ für } i \geq 1.$$

Da die Operation *compact Subword*-Tripel nicht beachtet, folgt:

$$\#subword(rel_i) \leq \#subword(rel_{i-1}) + 3\#suffix(rel_{i-1}), \text{ für } i \geq 1.$$

Man beachte den lezten Fall in *split*, in dem $A = B$ und $A = C$ (also (A, A, i)) ist. Es ist klar, dass es sich hierbei nur um ein $Suffix$-Tripel handeln kann. Der worst-case in diesem Fall wäre, wenn *split* solch ein Tripel durch drei neue Tripel ersetzen würde. Allgemein betrachtet wäre also der worst-case Fall, wenn alle *Subword*-Tripel erhalten blieben, alle $Suffix$-Tripel die obige Form (A, A, i) hätten und *split* jedes von ihnen durch drei Neue ersetzen würde. Jetzt kann man wegen Lemma 3.1.2 folgern, dass

$$\#subword(rel_i) \leq \#subword(rel_{i-1}) + 3(2n+1) \cdot n^2$$

gilt. Da die Ungleichung für ein beliebiges i gilt, entsteht die folgende Rekursion:

$$\begin{aligned}
\#subword(rel_i) &\leq \#subword(rel_{i-1}) + 3(2n+1) \cdot n^2 \\
\#subword(rel_i) &\leq \#subword(rel_{i-2}) + 2 \cdot 3(2n+1) \cdot n^2 \\
\#subword(rel_i) &\leq \#subword(rel_{i-3}) + 3 \cdot 3(2n+1) \cdot n^2 \\
&\vdots \\
&\vdots \\
&\vdots \\
\#subword(rel_i) &\leq \underbrace{\#subword(rel_0)}_{0} + i \cdot 3(2n+1) \cdot n^2.
\end{aligned}$$

Durch Auflösung der obigen Rekursion erhalten wir:

$$\#subword(rel_i) \leq 3i(2n+1) \cdot n^2.$$

Aus dem Algorithmus ist leicht zu erkennen, dass es höchstens soviele Iterationsschritte geben kann, wieviele Nonterminale die Grammatik besitzt (zu beachten ist die ***for***-Schleife). Also gilt $i \leq n$ und somit

$$\#subword(rel_i) \leq 3n(2n+1) \cdot n^2.$$

Jetzt kann man insgesamt folgern:

Wir wissen das $\#sufsub(rel_i) = \#suffix(rel_i) + \#subword(rel_i)$ gilt.

Nun kann man umformen:

$$\#sufsub(rel_i) = \#subword(rel_i) + \#suffix(rel_i).$$

Mit Lemma 3.5 und $\#subword(rel_i) \leq 3n(2n+1) \cdot n^2$ folgt:

$$\#sufsub(rel_i) \leq 3n(2n+1) \cdot n^2 + (2n+1) \cdot n^2,$$

und somit gilt

$$\#sufsub(rel_i) \leq (3n+1)(2n+1) \cdot n^2.$$

■

Theorem 3.2.2 *Eine Menge von Gleichungen zwischen Wörtern aus einer Grammatik, wobei eine Gleichung einem Tupel aus einer Testmenge entspricht, kann in polynomieller Zeit überprüft werden.*

Beweis. [16], Theorem 13 .

Kapitel 4

Unifikation unter Kompression

Die Unifikation ist eine Erweiterung des Wortproblems, weil zusätzliche Vorkommen von Variablen erlaubt sind, welche in ihrer gegebenen Reihenfolge substituiert werden, um potenzielle Gleichheit zu erhalten.

Definition 4.0.3 *Die first-order Unifikation enthält eine STG mit first-order Termen, Kontexten und zwei Term-Nonterminalen A_s und A_t von G mit $s = w_{G,A_s}$ und $t = w_{G,A_t}$.*

Es soll entschieden werden, ob s und t unifizierbar sind. Die Frage ist, ob die Berechnung einen allgemeinsten Unifikator liefert.
Wenn $\langle$ G, A_s, A_t $\rangle$ eine Eingabe für die STG-Unifikation ist, besteht die Aufgabe im Auffinden einer Substitution σ mit $\sigma(w_{G,A_s}) = \sigma(w_{G,A_t})$.
Gewöhnliche Algorithmen zur Term-Unifikation müssen Unterterme berechnen, Substitution durchführen, nach unterschiedlichen Stellen suchen, nach Vorkommen einer bestimmten Variable in Termen suchen, usw. Diese Operationen werden in dieser Diplomarbeit auf Eingabetermen, die durch STGs repräsentiert werden, implementiert. Um die Position von Termen zu bestimmen, benutzen wir SCFGs. Damit die Übersicht bewährt bleibt, benennen wir diese Nonterminale Positionsnonterminale (s. Definition 2.4.14).

4.1 Auffinden der ersten unterschiedlichen Stelle

Nachdem wir den Algorithmus von Plandowski behandelt haben, steht uns ein Verfahren zu Verfügung, mit dem wir in polynomieller Zeit überprüfen können, ob zwei Nonterminale das selbe Wort bilden. Um herauszufinden, an welcher Stelle der Unterschied ist, gibt es die Möglichkeit, jeweils Präfixe und Suffixe von diesen Nonterminalen zu erzeugen. Dabei haben diese Präfixe und Suffixe eine Länge von der Hälfte ihrer Nonterminale. Anschließend testet man wieder mit dem Testalgorithmus von Plandowski die beiden Nonterminale, die die Präfixe darstellen, auf Gleichheit. Durch diesen Schritt wird immer garantiert, dass man die erste unterschiedliche Stelle findet. Bei

Erfolg können somit die Präfixe ignoriert werden, da die potenzielle unterschiedliche Position in den Suffixen existiert. Die erneute Abfrage kann auf die Suffixe angewendet werden, womit bei jedem Durchlauf einmal die Laufzeit von Plandowskis Algorithmus benötigt wird. Bei Misserfolg wird die erneute Abfrage auf die Präfixe angewendet, und die Suche kann von Neuem beginnen. Da in jeder Abfrage auf die Hälfte der Länge der Terme der vorherigen Terme reduziert wird, existiert hier die Eigenschaft der binären Suche. Die Laufzeit der binären Suche ist $O(\log n)$ in der Länge der Terme. Im schlimmsten Fall müssen also $O(\log n)$ rekursive Aufrufe gemacht werden, um ein Ergebnis zu erhalten. Wie schon anfangs erwähnt, arbeitet dieser Algorithmus mit binärer Sucheigenschaft auf Nonterminalen und nicht auf den *tatsächlichen Termen*[1]. Denn das würde im schlimmsten Fall einen Effizienzverlust bedeuten, wenn die Terme exponentielle Länge hätten. Deswegen sind Grammatiken platzsparende Beschreibungen für solche Terme (s. Beispiel 3.0.21). Zum Beispiel können die beiden Worte $(ab)^{1000}a = a(ba)^{1000}$ mit ungefähr 16 Produktionen durch eine Grammatik dargestellt werden.
Innerhalb der Produktionen gibt es zwei *Startnonterminale*, die abgeleitet diese Worte produzieren. Von diesen *Startnonterminalen* werden Präfix- bzw. Suffixnonterminale gebildet, welche genau die Hälfte der Länge haben. An dieser Stelle werden eventuell alle Längen [2] der Nonterminale der anfänglichen Grammatik benötigt. Für die Längenberechnung betrachten wir Beispiel 1 aus [16].

Beispiel 4.1.1 *Die Grammatik definiert Worte exponentieller Länge mit den Produktionen*
$A_i \to A_{i-1}A_{i-1}$ *mit* $1 \leq i \leq k$, $A_0 \to a$.

Um die Länge für jedes Nonterminal A_i berechnen zu können, fängt man bei A_0 an, welches die Länge 1 hat, und addiert sukzessive alle Vorkommen auf den rechten Seiten der Produktionen, in denen A_0 auftaucht, von der untersten Ebene anfangend.

Beispiel 4.1.2
Für $k = 5$ sieht es folgendermaßen aus:

$A_5 \to A_4A_4$	$A_4 \to A_3A_3$	$A_3 \to A_2A_2$	$A_2 \to A_1A_1$	$A_1 \to A_0A_0$	$A_0 \to 1$
$A_5 \to A_4A_4$	$A_4 \to A_3A_3$	$A_3 \to A_2A_2$	$A_2 \to A_1A_1$	$A_1 \to 1+1$	
$A_5 \to A_4A_4$	$A_4 \to A_3A_3$	$A_3 \to A_2A_2$	$A_2 \to 2+2$		
$A_5 \to A_4A_4$	$A_4 \to A_3A_3$	$A_3 \to 4+4$			
$A_5 \to A_4A_4$	$A_4 \to 8+8$				
$A_5 \to 16+16$					
$A_5 \to 32$	$A_4 \to 16$	$A_3 \to 8$	$A_2 \to 4$	$A_1 \to 2$	$A_0 \to 1$

[1] Terme, die expandiert als String existieren, und nicht von einer Grammatik beschrieben werden

[2] in [16] steht, dass in polynomieller Zeit die Längen berechnet werden

Nachdem man die Längen hat, können die Präfixe bzw. Suffixe gebildet werden. Wenn von A_5 ein Präfix der Länge 10 berechnet werden soll, dann wendet man folgenden textuell beschriebenen Algorithmus an: Zuerst betrachtet man das **erste** Nonterminal, welches auf der rechten Seite von A_5 existiert. Dann holt man aus der Längenliste die Länge dieses Nonterminals, in diesem Fall $A_4 \rightarrow 16$. Da $16 > 10$, betrachtet man das **erste** Nonterminal, das auf der rechten Seite von A_4 existiert. Dann holt man aus der Längenliste die Länge dieses Nonterminals, in diesem Fall $A_3 \rightarrow 8$. Jetzt ist $8 < 10$. Nun kann das **erste** Nonterminal A_3 auf der rechten Seite von diesem Präfix

$$Pre_{10} \rightarrow A_3 \ldots,$$

platziert werden. Da schon ein Nonterminal der Länge 8 erfolgreich gefunden wurde und eine Differenz von 2 besteht, betrachtet man den **rechten** Nachbarn von A_3. In diesem Fall ist es wieder A_3. Da nun die Länge von $A_3 = 8$ ist, und $8 > 2$, betrachtet man das **erste** Nonterminal, das auf der rechten Seite von A_3 auftaucht. In diesem Fall ist es A_2 mit einer Länge von 4. Die Bedingung $4 > 2$ ist erfüllt, somit betrachtet man das **erste** Nonterminal auf der rechten Seite von A_2. Dieses Nonterminal A_1 hat eine Länge von 2. Die Bedingung $2 == 2$ ist erfüllt. Das **zweite** Nonterminal A_1 auf der rechten Seite von diesem Präfix,

$$Pre_{10} \rightarrow A_3 A_1$$

kann jetzt platziert werden und der Algorithmus erfolgreich enden. Für eine Produktion ist das Beispiel 5.2.6 gegeben. Die Produktion von diesem Präfix ist,

Produktion (N ''Pre_10'' [N ''A_3'', N ''A_1''] .

Wichtig ist hierbei, dass jedes dazugekommene Nonterminal von rechts konkateniert wird, um das Präfix richtig zu bilden:

```
    Produktion (N "Pre_10" []++[(N "A_3")]++[(N "A_1")] }
--> Produktion (N "Pre_10" [N "A_3",N "A_1"]}.
```

Dieses Prinzip ist auch übertragbar auf die Bildung eines Suffixes der Länge 10 von A_5. Man schaut sich jetzt nicht das erste Nonterminal, sondern das letzte Nonterminal auf der rechten Seite von A_5 an, und wendet dieses Prinzip wie oben an. Solange die gesuchte Länge kleiner ist, als die gefundenen Längen, schaut man sich das letzte Nonterminal an, der auf der rechten Seite von dem Nonterminal mit der gefundenen Länge existiert. Das wiederholt man solange, bis die gesuchte Länge größer ist, als die gefundene Länge. Nun kommt das Konkatenieren des Nonterminals A_3 von dem Suffix an der letzten Stelle,

$$Suff_{10} \rightarrow .....A_3.$$

Jetzt besteht wieder eine Differenz von 2, aber diesmal schaut man sich den linken Nachbarn von A_3 an. Das geht solange, bis man bei A_1 gelandet ist,

welches genau die Länge 2 hat. Zum Schluss wird A_1 als vorletztes konkateniert

$$Suff_{10} \rightarrow A_1 A_3.$$

Die Produktion von diesem Suffix ist,

Produktion (N ‘‘Suff_10’’ [N ’’A_1’’, N ‘‘A_3‘‘].

Wichtig ist hierbei, dass jedes dazugekommene
Nonterminal von links konkateniert wird, um das Suffix richtig zu bilden:

```
    Produktion (N "Suff_10" [(N "A_1")]++[(N "A_3")]++[] }
--> Produktion (N "Suff_10" [N "A_1",N "A_3"] }.
```

Mit den oben beschriebenen Verfahren können wir den binären Suchalgorithmus mit Hilfe von Plandowskis Testalgorithmus implementieren. In [13] wird solch eine Funktion namens **index** beschrieben. Die Funktion **index** arbeitet nicht direkt auf einer gegebenen STG, sondern in der preorder-traversierten Darstellung von den Worten. Diese Darstellung ist die Preorder-Grammatik der STG. Diese Grammatik ermöglicht, auf einer Singleton-Tree-Grammar, die zusätzlich Kontexte hat, die unterschiedliche Stelle zu finden [3]. Diese Überlegungen wurden in [2] behandelt. Die Singleton-Tree-Grammar wird in eine Singleton-Contextfree-Grammar umgewandelt, welche diese Preorder-Eigenschaft hat. Der nächste Abschnitt behandelt die Umwandlungsregeln und beschreibt diese ausführlich.

4.1.1 Berechnen der Preorder-Traversierung eines Terms

Die folgenden Ausführungen entnehmen wir aus [13]. Zwei verschiedene Bäume können die selbe Preorder-Traversierung haben. Wenn sie aber Terme über einer festen Signatur darstellen, wobei die Stelligkeit jeder Funktion eine feste Zahl ist, dann ist die Preorder-Traversierung für jeden Term eindeutig. Bei einem Term t existiert eine bijektive Abbildung zwischen den Indizes $\{1, \ldots, |pre(t)|\}$ und den Positionen $P(t)$ von t, welche jede Position $p \in P(t)$ dem Index $\{1, \ldots, |pre(t)|\}$ zuordnet, der das $root(t|p)$ beim Preorder-Durchlauf darstellt. Wir können zwei rekursive Funktionen $pIndex(t,p) \rightarrow \{1, \ldots, |Pre(t)|\}$ und $iPos(t,i) \rightarrow P(t)$ definieren:

$$pIndex(t,p) := \begin{cases} 1, \text{ wenn } p = \lambda \\ (1 + |t_1| + \cdots + |t_{i-1}|) + pIndex(t_i, p), \text{ wenn } p = i.p \\ \text{und } t = f(t_1, \ldots, t_m), \end{cases}$$

[3] Durch die preorder Traversierung findet eine Spaltung von Kontext-Nonterminalen in linke und rechte Teile. Dadurch entfallen die Kontexte

und

$$iPos(t,i) := \begin{cases} \lambda, \text{ wenn } i = 1 \\ i.iPos(t_i,k), \text{ wenn } i = 1 + |t_1| + \cdots + |t_{i-1}| + k \\ \text{und } t = f(t_1,\ldots,t_m) \text{ mit } 1 \leq k \leq |t_i|. \end{cases}$$

Es wird in [2] beschrieben, wie aus einer gegebenen STG G, eine SCFG Pre_G konstruiert wird, welche die preorder-traversierten Terme von G repräsentieren. Die Konstruktionvorschriften zeigt die folgende Abbildung. Für jedes Term-Nonterminal A und seiner Produktion $A \to \alpha$ von G ist eine dementsprechende Produktion $\mathcal{P}_A \to \alpha'$ von Pre_G Voraussetzung, um das Nonterminal $\mathcal{P}_A$ von Pre_G in folgende Beziehung zu setzen: $w_{Pre_G,\mathcal{P}_\mathcal{A}} = pre(w_{G,A})$. In der Grammatik befinden sich auch Kontext-Nonterminale, die in preorder Nonterminale einmal links vom Loch ($\mathcal{L}_C$) und einmal in preorder Nonterminale rechts vom Loch ($\mathcal{R}_C$) transformiert werden.

$$\begin{array}{lcl}
A \to f(A_1,\ldots,A_m) & \Rightarrow & \mathcal{P}_A \to f(\mathcal{P}_{A_1},\ldots,\mathcal{P}_{A_m}) \\
A \to C_1 A_2 & \Rightarrow & \mathcal{P}_A \to \mathcal{L}_{C_1}\mathcal{P}_{A_2}\mathcal{R}_{C_1} \\
A \to A_1 & \Rightarrow & \mathcal{P}_A \to \mathcal{P}_{A_1} \\
C \to C_1 C_2 & \Rightarrow & \begin{cases} \mathcal{L}_C \to \mathcal{L}_{C_1}\mathcal{L}_{C_2} \\ \mathcal{R}_C \to \mathcal{R}_{C_2}\mathcal{R}_{C_1} \end{cases} \\
C \to f(A_1,\ldots,A_{i-1},C_i,A_{i+1},\ldots,A_m) & \Rightarrow & \begin{cases} \mathcal{L}_C \to f(A_1\ldots A_{i-1}\mathcal{L}_{C_i}) \\ \mathcal{R}_C \to \mathcal{R}_{C_i}\mathcal{P}_{A_{i+1}}\ldots\mathcal{P}_{A_n} \end{cases} \\
C \to [\cdot] & \Rightarrow & \begin{cases} \mathcal{L}_C \to \lambda \\ \mathcal{R}_C \to \lambda \end{cases}
\end{array}$$

Abbildung 4.1: Transformationsregeln für Preorder-Traversierung Pre_G

Bei diesen Transformationsregeln ist ein A ein Term-Nonterminal und ein C ein Kontext-Nonterminal. Falls ein Term-Nonterminal A eine Funktion f produziert, wird die Regel übernommen. Man schreibt lediglich das $\mathcal{P}$ vor jedem Term-Nonterminal, um das Preorder-Nonterminal zu verdeutlichen, welches vorher ein Term-Nonterminal in der STG war. Das Term-Nonterminal A kann auch ein Kontext C_1 und ein A_2 produzieren. Hierbei schreibt man vor das A_2 wieder ein $\mathcal{P}$, wobei ein Kontext-Nonterminal C_1 aufgespalten wird in linke und rechte Teile, die in der Preorder-Grammatik Nonterminale vom Typ $\mathcal{L}$ und $\mathcal{R}$ heissen. Die enstandenen $\mathcal{L}$- bzw. $\mathcal{R}$-Nonterminale in der Preorder-Grammatik ermöglichen uns, dass A_2 an die richtige Stelle im Kontext, nämlich in das Loch von C_1 zu setzen. Ein Term-Nonterminal kann zuletzt wieder ein Term-Nonterminal bilden, dies ist ein λ-*rule*.

Schließlich kann ein Kontext C auch eine Funktion f bilden, wobei in den Argumenten an der i-ten Stelle ein Kontext C_i enthalten ist. Laut der Transformationregel wird die Funktion genau an der Stelle des C_i in einen linken Teil und rechten Teil aufgespalten. Somit bildet der linke Teil von C mit dem Nonterminal $\mathcal{L}$ die Kette im Funktionsargument bis zum linken $\mathcal{L}_{C_i}$ Nonterminal. Der rechte Teil von C mit dem Nonterminal $\mathcal{R}$ bildet die Kette der Symbole, welche Argumente sind, die nach dem C_i auftauchen. Zuletzt bildet ein Kontext das Loch, nämlich der unterste Kontext in der Hierarchie. Dieses Loch erzeugt ein λ, d.h. es kann vernachlässigt werden.
Anhand eines Beipiels sollen die einzelnen Regeln, die in Abbildung 4.1.1 aufgelistet sind, nochmal nachvollzogen werden.

Beispiel 4.1.3 *Sei $\mathcal{G} = (\mathcal{TN}, \mathcal{CN}, \Sigma, \mathcal{R})$ eine Singleton-Baum-Grammatik, mit $\mathcal{TN} = \{A_1, \ldots, A_9\}$, $\mathcal{CN} = \{C_1, C_2, C_3, C_D\}$, $\Sigma = \{p, f, h, a, b, x_1, x_2\}$ und sei die Menge der Produktionen $\mathcal{R}$ wie folgt:*

$$\begin{array}{lll} \mathcal{R} = & A_s \to pA_3A_4 & \\ & A_3 \to gA_7A_7 & A_7 \to x_1 \\ & A_4 \to C_2A_9 & C_1 \to hA_{10}C_0A_8 \\ & C_2 \to C_1C_D & A_8 \to x_2 \\ & A_8 \to x_2 & C_0 \to [\cdot] \\ & C_D \to C_1C_1 & A_9 \to a \\ & A_{10} \to b. & \end{array}$$

Sei p eine Funktion mit zwei Argumenten und seien f, g und h jeweils Funktionen mit zwei Argumenten. Weiterhin seien x_1, x_2 Variablen.

Die nächste Abbildung zeigt die Transformationen für dieses Beispiel. Es wird immer eine Produktion bezüglich einer Regel gezeigt.

$$\begin{array}{lcl} A_s \to p(A_3, A_4) & \Rightarrow & \mathcal{P}_{A_s} \to p(\mathcal{P}_{A_3}, \mathcal{P}_{A_4}) \\ A_4 \to C_2A_9 & \Rightarrow & \mathcal{P}_{A_4} \to \mathcal{L}_{C_2}\mathcal{P}_{A_9}\mathcal{R}_{C_2} \\ C_2 \to C_1C_D & \Rightarrow & \left\{ \begin{array}{l} \mathcal{L}_{C_2} \to \mathcal{L}_{C_1}\mathcal{L}_{C_D} \\ \mathcal{R}_{C_2} \to \mathcal{R}_{C_D}\mathcal{R}_{C_1} \end{array} \right. \\ C_1 \to h(A_{10}, C_0, A_8) & \Rightarrow & \left\{ \begin{array}{l} \mathcal{L}_{C_1} \to h(\mathcal{P}_{A_{10}}, \mathcal{L}_{C_0}) \\ \mathcal{R}_{C_1} \to \mathcal{R}_{C_0}\mathcal{P}_{A_8} \end{array} \right. \\ C_0 \to [\cdot] & \Rightarrow & \left\{ \begin{array}{l} \mathcal{L}_{C_0} \to \lambda \\ \mathcal{R}_{C_0} \to \lambda \end{array} \right. \end{array}$$

Abbildung 4.2: Transformationen für Beispiel 4.1.3

Die Grammatik wurde wie oben beschrieben umgewandelt. Die nächste Abbildung zeigt den Ableitungsbaum für diese Grammatik.

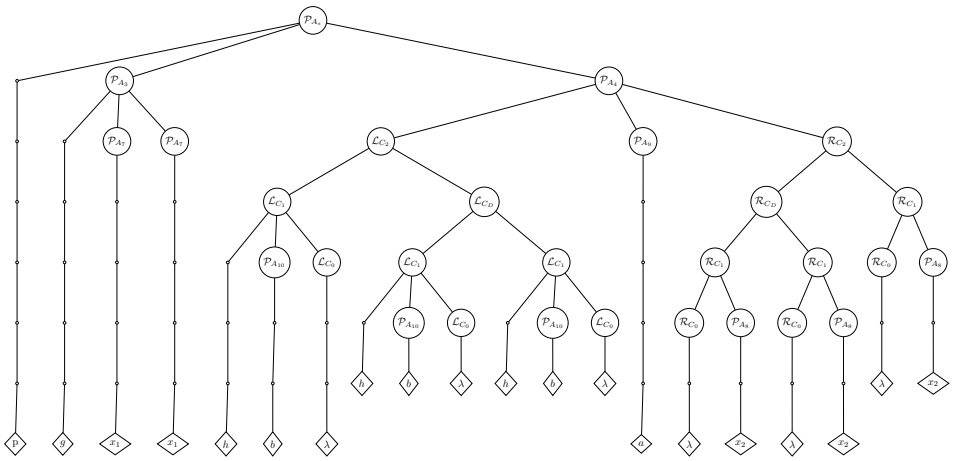


Abbildung 4.3: Ableitungsbaum

Die Blätter dieses Ableitungsbaumes erzeugen den Term von $\mathcal{P}_{A_s}$

$$p(g(x_1, x_1), h(b, h(b, h(b, a, x_2), x_2), x_2), x_2)).$$

Man sieht hier, dass das erste Argument von p, nämlich die Funktion g, vom Einsetzen in die Löcher von den Kontexten nicht betroffen ist. Im zweiten Argumenten von p hat die Funktion h ein Loch in seinem zweiten Argument. Wenn Löcher auftreten, sieht man sie an den Blättern als λ's beschrieben. Diese λ-Blätter haben keine Funktionalität oder Länge, sondern dienen lediglich als Hinweis dafür, dass genau an dieser Stelle eine Einsetzung stattfindet. Das Einsetzen wird hier gespart, denn man muss die λ-Blätter ignorieren und die Symbole in den Blättern von links nach rechts aufschreiben um den Term zu erhalten. Anhand der Abbildung 4.3 erkennt man die Vorteile der Preorder-Grammatik im Vergleich zur STG. Man muss nur den STG-Baum aufbauen und versuchen das Wort abzuleiten, was komplizierter ist.
Nachdem gezeigt wurde, wie man die Längen von Nonterminalen und die Preorder-Grammatik effizient aufbaut, stehen nun die Eingaben für die **index** Funktion zu Verfügung. Wir definieren eine weitere STG mit einem Startnonterminal A_t und vereinigen die Produktionsmenge mit der Produktionsmenge aus Beispiel 4.1.3.

Beispiel 4.1.4 *Sei $\mathcal{G} = (\mathcal{TN}, \mathcal{CN}, \Sigma, \mathcal{R})$ eine Singleton-Baum-Grammatik, mit $\mathcal{TN} = \{A_t, B_1, \cdots, B_{10}\}$, $\mathcal{CN} = \{C_{1_1}, C_0\}$,*
$\Sigma = \{p, f, h, a, b, y_1, y_2, y_3, y_4\}$ und sei die Menge der Produktionen $\mathcal{R}$ wie folgt:

$$\begin{aligned}
\mathcal{R} = \quad & A_t \to pB_3B_4 & & \\
& B_3 \to y_1 & & B_4 \to C_{1_1}B_9 \\
& C_{1_1} \to hB_{10}C_0B_8 & & B_8 \to y_4 \\
& B_{10} \to y_2 & & B_9 \to y_3 \\
& C_0 \to [\cdot] & &
\end{aligned}$$

Sei p eine Funktion mit zwei Argumenten und seien f, g und h jeweils Funktionen mit zwei Argumenten. Weiterhin seien y_1, y_2, y_3, y_4 Variablen.

Die Eingaben für **index** sind die beiden Nonterminale $\mathcal{P}_{A_t}$ und $\mathcal{P}_{A_s}$ und die vereinigte Grammatik von Beispiel 4.1.3 und Beispiel 4.1.4. Expandiert erzeugt das Nonterminal $\mathcal{P}_{A_t}$ den Term

$$p(y_1, h(y_2, y_3, y_4)$$

und das Nonterminal $\mathcal{P}_{A_s}$ den Term

$$p(g(x_1, x_1), h(b, h(b, h(b, a, x_2), x_2), x_2)).$$

Anhand der Terme sieht man, dass der Unterschied an der zweiten Stelle liegt. Die Funktion **index** liefert in diesem Fall die Zahl zwei. In der zweiten Zeile wird überprüft, ob einer der Nonterminale ein Term der Länge 1 bildet. Wenn dieser Fall eintritt, dann beendet die Funktion ihre Ausführung und liefert zusätzlich die eins[4]. Sonst wird die Länge des kürzeren Nonterminals durch 2 geteilt und in einer Variablen m gespeichert. Da $|\mathcal{P}_{A_t}| = 6$ der kürzere der Beiden ist, ergibt sich für $m = 3$. Nach den Anweisungen aus Zeile 5 und 6 erweitern wir die Grammatik um neue Nonterminale $\mathcal{P}_{A_{t'}}$ und $\mathcal{P}_{A_{s'}}$, welche Präfixe von $\mathcal{P}_{A_s}$ und $\mathcal{P}_{A_t}$ darstellen und die Länge $m = 3$ haben müssen. Folglich haben wir $\mathcal{P}_{A_{t'}} \rightarrow p, B_3, h$ und $\mathcal{P}_{A_{s'}} \rightarrow p, g, A_7$.

4.1.2 Die index-Funktion

INDEX$(p1, p2, P)$

```
1  ▷ Voraussetzung: w_{p1} und w_{p2} unterscheiden sich an einer Stelle.
2  if min(|w_{p1}|, |w_{p2}|) = 1
3     then return 1
4     else  m ← min(|w_{p1}|, |w_{p2}|)div2
5           ▷  Erweitere P, um neue Nonterminale p'_1 und p'_2,
6           ▷ welche die Präfixe der Länge m von w_{p1} und w_{p2}
7           ▷ darstellen.
8           if (w'_{p1} ≠ w'_{p2})
9              then return INDEX(p'_1, p'_2, P)
10             else
11                   ▷ Erweitere P, um neue Nonterminale p''_1 und p''_2,
12                   ▷  welche die Suffixe der Länge |w_{p1}| − m von w_{p1}
13                   ▷  und |w_{p2}| − m von w_{p2}, und
14                   return m + INDEX(p''_1, p''_2, P)
```

Anschließend wird in der Zeile 7 überprüft, ob die beiden Nonterminale das gleiche Wort bilden. Diese Abfrage nutzt den Plandowski Testalgorithmus.

[4] s. Zeile 3 nächste Seite

In diesem Fall ist $w_{\mathcal{P}_{A_{t'}}} \neq w_{\mathcal{P}_{A_{s'}}}$. Das bedeutet, solange die Abfrage ungleich ist, wird **index** auf diesen Präfixen angewendet, also index($P_{A'_t}$, $P_{A'_s}$, $P \cup P_{A'_t} \cup P_{A'_s}$) bis der Abbruch in Zeile 3 zutrifft. Wenn aber die Präfixe gleich sind, dann werden zusätzlich die Suffixe gebildet und der Aufruf über die Suffixe gemacht. Im nächsten Aufruf wird in Zeile 2 die Länge des kürzeren Nonterminals durch 2 geteilt und in einer Variablen m gespeichert. Da beide Nonterminale die selbe Länge 3 haben, ist $m = 1$ durch ganzzahlige Division mit 2. Nach den Anweisungen aus Zeile 5 und 6 erweitern wir die Grammatik um neue Nonterminale $\mathcal{P}_{prefix_{A_{t''}}}$ und $\mathcal{P}_{prefix_{A_{s''}}}$, welche Präfixe von $\mathcal{P}_{A_{s'}}$ und $\mathcal{P}_{A_{t'}}$ darstellen und die Länge $m = 1$ haben müssen. Folglich haben wir $\mathcal{P}_{prefix_{A_{t''}}} \to p$ und $\mathcal{P}_{prefix_{A_{s''}}} \to p$. Anschließend wird in der Zeile 7 überprüft, ob die beiden Nonterminale das gleiche Wort bilden. Diese Abfrage nutzt den Plandowski Testalgorithmus.
In diesem Fall ist $w_{\mathcal{P}_{prefix_{A_{t''}}}} = w_{\mathcal{P}_{prefix_{A_{s''}}}}$. Die Präfixe sind gleich, folglich erweitern wir die Grammatik nach den Anweisungen aus Zeile 10, 11 und 12 um neue Nonterminale $\mathcal{P}_{suffix_{A_{t''}}}$ und $\mathcal{P}_{suffix_{A_{s''}}}$, welche Suffixe von $\mathcal{P}_{A_{s'}}$ und $\mathcal{P}_{A_{t'}}$ darstellen und die Länge $|\mathcal{P}_{A_{s'}}| - m = 3 - 1 = 2$ von $\mathcal{P}_{A_{s'}}$ und $|\mathcal{P}_{A_{t'}}| - m = 3 - 1 = 2$ von $\mathcal{P}_{A_{t'}}$ haben müssen. Folglich haben wir $\mathcal{P}_{suffix_{A_{t''}}} \to B_3, h$ und $\mathcal{P}_{suffix_{A_{s''}}} \to g, A_7$, so dass der nächste Aufruf der Zeile 13 entsprechend

$$\overbrace{m}^{1} + \text{index}(\mathcal{P}_{suffix_{A_{t''}}}, \mathcal{P}_{suffix_{A_{s''}}}, P \cup \mathcal{P}_{suffix_{A_{t''}}} \cup \mathcal{P}_{suffix_{A_{s''}}})$$

lautet. Im nächsten Aufruf ist das Minimum der Beiden 2, also ist $m = 1$. Jetzt wird P wieder um neue Nonterminale erweitert, welche das Präfix

$$\mathcal{P}_{prefix_{A_{t'''}}} \to B_3$$

der Länge $\overbrace{m}^{1}$ von $\mathcal{P}_{suffix_{A_{t''}}}$ ist und das Präfix

$$\mathcal{P}_{prefix_{A_{s'''}}} \to g$$

der Länge $\overbrace{m}^{1}$ von $\mathcal{P}_{suffix_{A_{s''}}}$ ist.
Nach Zeile 7 ist $w_{\mathcal{P}_{prefix_{A_{t'''}}}} \neq w_{\mathcal{P}_{prefix_{A_{s'''}}}}$, also rufen wir

$$\text{index}(\mathcal{P}_{prefix_{A_{t'''}}}, \mathcal{P}_{prefix_{A_{s'''}}}, P \cup \mathcal{P}_{prefix_{A_{t'''}}} \cup \mathcal{P}_{prefix_{A_{s'''}}})$$

auf. Dieser Aufruf verzweigt sofort in die Zeile 3, weil das kürzere Nonterminal die Länge 1 hat. Addiert man alle Rückgabewerte, die es bis jetzt gab, lautet das Ergebnis 2. Somit haben wir die erste unterschiedliche Stelle in den Termen an der zweiten Position.

Lemma 4.1.5 *Sei G eine SCFG der Grösse n, und seien p_1 und p_2 Nonterminale von G, so dass $w_{p_1} \neq w_{p_2}$. Die Berechnung der ersten Stelle, an der sich w_{p_1} und w_{p_2} unterscheiden, benötigt $O(|P|^4)$.*

Beweis. O.B.d.A., können wir annehmen, dass P in Chomsky-Normal-Form ist. Die Umwandlung geschieht in linearem Zeit- bzw. Platzbedarf. Da die beiden w_{p_1} und w_{p_2} durch $2^{|P|}$ nach oben beschränkt sind, produziert der oben beschriebene binäre Suchalgorithmus höchstens $|P|$ rekursive Aufrufe, wobei jeder Aufruf nach [21] $O(|P^3|)$ benötigt. Die Laufzeit für einen binären Suchalgorithmus ist somit $O(|P^4|)$.

■

Die **index** Funktion ist notwendig, um die erste mögliche Unifizierung durchzuführen. Mit Hilfe dieser Zahl kann die Positionsgrammatik bestimmt werden, welche die Angabe über die unterschiedliche Stelle macht. Die Positionsgrammatik ist speziell nur für diese Stelle und arbeitet parallel mit der STG, um die Unterterme, die die unterschiedliche Stelle beherrbergen, in den zu unifizierenden Termen zu identifizieren. Die Positionsgrammatik ist eine platzsparende Beschreibung und eignet sich für die Identifikation hervorragend. Sie ist eine wiederholt komprimierte Darstellung, die auf der gegebenen Grammatik arbeitet. Im nächsten Abschnitt wird die Bildung der Positionsgrammatik genauer beschrieben.

4.2 Bildung der Positionsgrammatik

Die Abbildung 4.5 erklärt die Vorgehensweise der Umformungsregeln. Dabei beginnt man anfangs für ein gegebenes $\mathcal{P}_{N,k}$ durch wiederholtes Anwenden der Regeln, bis die erste Regel zutrifft. Das k ist auf 1 reduziert worden und durch die Anweisung unter dem Strich soll man ein neues Nonterminal $\mathcal{P}_{N,1} \rightarrow \lambda$ erzeugen. Diese Regel ist die Abbruchbedingung und ist sozusagen die Pfeilspitze des Pfeiles, welches auf die unterschiedliche Stelle in der STG zeigt.
In der zweiten Regel bildet das Nonterminal N aus der STG eine Funktion mit m Argumenten. Jetzt wird stets solange mit einer Laufvariable $i \geq 1$ über die Längen in den Nonterminalen aufaddiert, bis die Bedingung zutrifft. Man überspringt alle Argumente und sucht das Nonterminal, welches in der Länge plus aller vorherigen Nonterminalen aufaddiert passt. Die Laufvariable gibt uns dabei die Stelle des Nonterminals an, in der die Suche eventuell weitergeführt wird. Nachdem die Bedingung passt und das N_i gefunden wurde, bildet man ein neues Positionsnonterminal $\mathcal{P}_{N_i,k-k'}$. Der nächste Aufruf wird mit diesem Nonterminal weitergeführt, dessen korrespondierendes Nonterminal aus der STG gesucht wird. Das alte k ist aktualisiert worden, weil man viele Stellen bzw. Nonterminale, die irrelevant sind, übersprungen hat. Die gesammte Länge k' dieser Nonterminale wird aus dem alten k abgezogen.

Bei den dritten, vierten und fünften Regeln haben wir die übrigen zulässigen Formen der STG. In der Definition 2.4.9 kann nachgelesen werden, welche Formen eine Singleton-Tree-Grammar erlaubt. Wenn in der dritten Regel ein Nonterminal N einen Kontext C_1 und einen Term-Nonterminal N_2 produziert, dann handelt es sich um ein Term-Nonterminal. Wenn das Nonterminal N jedoch zwei Kontexte auf der rechten Seite hat, dann handelt es sich um ein Kontext-Nonterminal. Zusätzlich muss die Bedingung $1 < k \leq |w_{\mathcal{L}_{C_1}}|$ gelten. Die Länge $|w_{\mathcal{L}_{C_1}}|$ nimmt hier Bezug auf die Preorder-Länge des Nonterminals C_1. Denn laut Abbildung 4.1.1 werden Kontext-Nonterminale in linke $\mathcal{L}$ und rechte $\mathcal{R}$ Teile aufgespalten. Das Zutreffen der Bedingung besagt, dass die gesuchte Stelle in der linken Hälfte von C_1 ist. Dementsprechend ensteht ein neues Nonterminal $\mathcal{P}_{C_1,k}$, das in diesen Kontext verzweigt. Das k wird nicht verändert, weil kein Überspringen von Nonterminalen stattfand. In der vierten Regel ist $|w_{\mathcal{L}_{C_1}}| < k \leq |w_{\mathcal{L}_{C_1}}| + |w_{N_2}|$. Hier ist die gesuchte Stelle in N_2, daraus folgt die Anweisung für das Erzeugen des neuen Nonterminals $\mathcal{H}_{C_1}\mathcal{P}_{N_2,k-k'}$. Von dem alten k wird das neue k' abgezogen, weil man um k' Stellen weiterläuft.
In der letzten Regel befindet sich die gesuchte Stelle in der rechten Hälfte vom Kontext C_1, weil die Bedingung $|w_{\mathcal{L}_{C_1}}| + |w_{N_2}| < k$ zutrifft. Dementsprechend findet die Suche, wie in der dritten Regel, wieder in C_1 weiter, allerdings muss die Länge von N_2 von dem k abgezogen werden. In unserer Längenberechnung haben Nonterminale, die Löcher produzieren, die Länge 0, im Gegensatz zu [13]. Hierdurch kommt es zu einer kleinen Abweichung beim Erzeugen des neuen Nonterminals in der letzten Regel. Wir zählen die $(+1)$ nicht mit, während sie in [13] dazugezählt wird, weil die Längen von Nonterminalen, die Löcher produzieren dort mitbewertet werden. Die Abbildung 4.4 soll nochmals die Zusammenhänge für die letzten drei Regeln verdeutlichen.

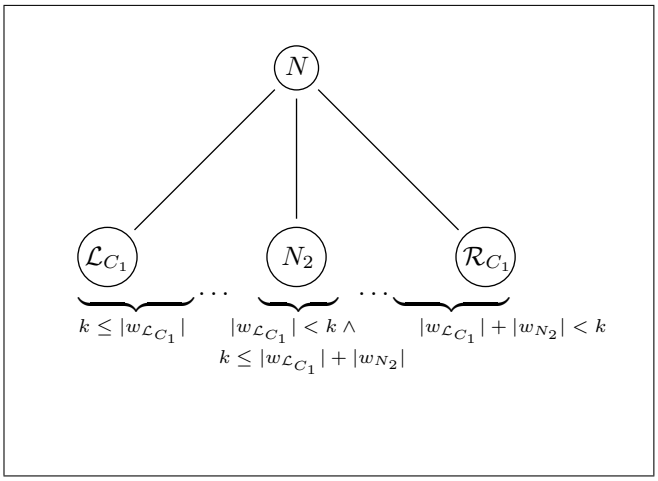


Abbildung 4.4: Ableitungsbaum

Die folgenden Aussagen und die Regeln für die Bildung einer Positionsgrammatik (SCFG) *Pos* entnehmen wir aus [13]. Mit Hilfe der Indexzahl k aus dem vorherigen Abschnitt, möchte man die SCFG Pos generieren. Die jeweiligen separat zu berechnenden SCFGs Pos_{A_t} bzw. Pos_{A_s} enthalten zwei Nonterminale $pos_{A_t,k}$ bzw. $pos_{A_s,k}$ welche die Wörter $w_{pos_{A_t}}$ bzw. $w_{pos_{A_s}}$ repräsentieren, die die erste unterschiedliche Position von den zu unifizierenden Nonterminalen A_t und A_s identifizieren. Um dies zu verallgemeinern möchte man für ein gegebenes Nonterminal N von G und einer gegebenen positiven Zahl k die SCFG und das Nonterminal $\mathcal{P}_{N,k}$ konstruieren. Dieses Nonterminal generiert die Position von dem Symbol, das dem k-ten Index von $\mathbf{pre}(w_N)$ entspricht. Das Nonterminal $\mathcal{P}_{A_t,k}$ oder $pos_{A_t,k}$ ist das p, welches wir berechnen, mit Hilfe der Umformungsregeln, die in der folgenden Abbildung definiert sind. Diese Umformungsregeln erzeugen neue Positionsnonterminale, welche wieder auf diesen Regeln angewendet werden. Dabei ist zu beachten, dass eine Assoziation mit den Nonterminalen aus der STG besteht. Am Anfang fängt man mit $\mathcal{P}_{A_t,k}$ an, dementsprechend nimmt man das passende Nonterminal A_t aus der STG. Kontext-Nonterminale, der Form $C_2 \to C_1 C_D$ (Beispiel 4.1.3) überführen wir folgendermaßen:

Beispiel 4.2.1

$$\mathcal{H}_{C_2} \to \mathcal{H}_{C_1}\mathcal{H}_{C_D},\ \mathcal{H}_{C_1} \to 2\mathcal{H}_{C_0},\ \mathcal{H}_{C_D} \to \mathcal{H}_{C_1}\mathcal{H}_{C_1},\ \mathcal{H}_{C_0} \to \lambda$$

Das Kontext-Nonterminal $C_1 \to hA_{10}C_0A_8$ hat an der zweiten Stelle des Funktionsargumentes den Kontext. Das Überführen in die Positionsgrammatik wird in der obigen Zeile gezeigt. Das neu enstandene Nonterminal hat auf der rechten Seite die Zahl für die Stelle des auftretenden Kontextes, konkateniert mit diesem Kontext-Nonterminal. Die Laufzeit für das Erzeugen dieser Positionsnonterminale ist linear.

$$\frac{(N \rightarrow \alpha) \wedge (k = 1)}{\mathcal{P}_{N,k} \rightarrow \lambda} \tag{4.1}$$

$$\frac{(N \rightarrow f(N_1, \ldots, N_m)) \wedge (1 + |w_{N_1}| + \cdots + |w_{N_{i-1}}| = k' < k \leq k' + |w_{N_i}|)}{\mathcal{P}_{N,k} \rightarrow i\mathcal{P}_{N_i,k-k'}} \tag{4.2}$$

$$\frac{(N \rightarrow C_1 N_2) \wedge (1 < k \leq |w_{\mathcal{L}_{C_1}}|)}{\mathcal{P}_{N,k} \rightarrow \mathcal{P}_{C_1,k}} \tag{4.3}$$

$$\frac{(N \rightarrow C_1 N_2) \wedge (k' = |w_{\mathcal{L}_{C_1}}| < k \leq |w_{\mathcal{L}_{C_1}}| + |w_{N_2}|)}{\mathcal{P}_{N,k} \rightarrow \mathcal{H}_{C_1}\mathcal{P}_{N_2,k-k'}} \tag{4.4}$$

$$\frac{(N \rightarrow C_1 N_2) \wedge (|w_{\mathcal{L}_{C_1}}| + |w_{N_2}| < k) \wedge (k' = |w_{N_2}|)}{\mathcal{P}_{N,k} \rightarrow \mathcal{P}_{C_1,k-k'}} \tag{4.5}$$

Abbildung 4.5: Konstruktionsregeln für die SCFG, die zum k-ten Index in $\mathbf{pre}(w_N)$ korrespondierende Position berechnet

Das nächste Beispiel beinhaltet die vereinigte Grammatik von Beispiel 4.1.3 und 4.1.4.

Beispiel 4.2.2 *Sei* $\mathcal{G} = (\mathcal{TN}, \mathcal{CN}, \Sigma, \mathcal{R} = \mathcal{R}_1 \cup \mathcal{R}_2)$ *eine Singleton-Baum-Grammatik, mit* $\mathcal{TN} = \{A_t, A_s, B_1, \cdots, B_{10}, A_1, \ldots, A_9\}$,
$\mathcal{CN} = \{C_{1_1}, C_1, C_2, C_3, C_D, C_0\}$,
$\Sigma = \{p, g, h, a, b, y_1, y_2, y_3, y_4, x_1, x_2\}$ *und sei die Menge der Produktionen* $\mathcal{R}$ *wie folgt:*

$$\begin{array}{lllll}
\mathcal{R}_1 = & A_t \rightarrow pB_3B_4 & & \mathcal{R}_2 = & A_7 \rightarrow x_1 \quad A_s \rightarrow pA_3A_4 \\
& B_3 \rightarrow y_1 \quad B_4 \rightarrow C_{1_1}B_9 & & & A_9 \rightarrow a \quad A_3 \rightarrow gA_7A_7 \\
& C_{1_1} \rightarrow hB_{10}C_0B_8 \quad B_8 \rightarrow y_4 & & & A_8 \rightarrow x_2 \quad C_2 \rightarrow C_1C_D \\
& B_{10} \rightarrow y_2 \quad B_9 \rightarrow y_3 & & & A_8 \rightarrow x_2 \quad A_4 \rightarrow C_2A_9 \\
& C_0 \rightarrow [\cdot] & & & A_{10} \rightarrow b \quad C_D \rightarrow C_1C_1 \\
& & & & C_0 \rightarrow [\cdot] \quad C_1 \rightarrow hA_{10}C_0A_8
\end{array}$$

Sei p *und* g *eine Funktion mit zwei Argumenten und sei* h *eine Funktion mit drei Argumenten. Weiterhin seien* $y_1, y_2, y_3, y_4, x_1, x_2$ *Variablen.*

Es soll nun mit dieser Grammatik gezeigt werden, wie man die Positionsgrammatik für die erste unterschiedliche Stelle berechnet. Zusätzlich brauchen wir noch die Längenliste der Nonterminale, die in der Tabelle 4.1 stehen. Da wir im vorherigen Abschnitt schon unser $k = 2$ haben, müssen wir jeweils

$\mathcal{P}_{A_t,2}$ und $\mathcal{P}_{A_s,2}$ auf die Regeln aus Abbildung 4.5 anwenden. Wir beginnen mit $\mathcal{P}_{A_t,2}$ und wenden die zweite Regel an.

$$(1.Schritt) \quad \frac{A_t \to pB_3B_4 \wedge (1 = k' < 2 \leq 1 + \overbrace{|w_{B_3}|}^{1})}{\mathcal{P}_{A_t,2} \to 1\mathcal{P}_{B_3,2-1}}$$

Der nächste Aufruf findet mit $\mathcal{P}_{B_3,1}$ statt. Hierfür wird die erste Regel angewendet, also die Abbruchbedingung, weil $k = 1$ ist. Infolgedessen haben wir

$$(2.Schritt) \quad \frac{(B_3 \to y_1) \wedge (k = 1)}{\mathcal{P}_{B_3,1} \to \lambda}.$$

Längen der STG-Nonterminale	Längen der Preorder-Nonterminale
TN A_s → Laenge 14	P A_s → Laenge 14
TN A_4 → Laenge 10	P A_4 → Laenge 10
TN A_t → Laenge 6	L C_2 → Laenge 6
CN C_2 → Laenge 9	R C_2 → Laenge 3
CN C_D → Laenge 6	P A_t → Laenge 6
TN B_4 → Laenge 4	P B_4 → Laenge 4
CN C_{1_1} → Laenge 3	R C_D → Laenge 2
CN C_1 → Laenge 3	L C_D → Laenge 4
TN A_3 → Laenge 3	L C_{1_1} → Laenge 2
TN A_9 → Laenge 1	L C_1 → Laenge 2
CN C_0 → Laenge 0	R C_1 → Laenge 1
TN A_8 → Laenge 1	R C_{1_1} → Laenge 1
TN A_7 → Laenge 1	P A_3 → Laenge 3
TN A_{10} → Laenge 1	P B_3 → Laenge 1
TN A_8 → Laenge 1	P B_9 → Laenge 1
TN B_8 → Laenge 1	P B_{10} → Laenge 1
TN B_{10} → Laenge 1	P B_8 → Laenge 1
TN B_9 → Laenge 1	P A_8 → Laenge 1
TN B_3 → Laenge 1	P A_{10} → Laenge 1
	P A_7 → Laenge 1
	R C_0 → Laenge 0
	L C_0 → Laenge 0
	P A_9 → Laenge 1

Tabelle 4.1: Längentabelle

Nun muss für $\mathcal{P}_{A_s,2}$ auch die zweite Regel angewendet werden,

$$(1.Schritt) \quad \frac{A_s \to pA_3A_4 \wedge (1 = k' < 2 \leq 1 + \overbrace{|w_{A_3}|}^{3})}{\mathcal{P}_{A_s,2} \to 1\mathcal{P}_{A_3,2-1}}$$

Der nächste Aufruf fängt mit dem neu erzeugten Nonterminal $\mathcal{P}_{A_3,1}$ an. Es ist wieder die erste Regel, weil $k = 1$ ist.

$$(2.Schritt) \quad \frac{(A_3 \rightarrow y_1) \wedge (k = 1)}{\mathcal{P}_{A_3,1} \rightarrow \lambda}$$

Nachdem auch hier die Abbruchbedingung eingetreten ist, kommen noch zusätzlich, die im Beispiel 4.2.1 gezeigten Kontext-Nonterminale hinzu. Die Positionsgrammatik für die erste unterschiedliche Stelle wird folgendermaßen festgehalten:

Beispiel 4.2.3 *Sei* $\mathcal{G} = (\mathcal{P}, \mathcal{H}, \Sigma, \mathcal{R})$ *eine Position-SCFG, mit* $\mathcal{P} = \{P_{A_s,2}, P_{A_t,2}, P_{B_3,1}, P_{A_3,1}\}$, $\mathcal{H} = \{\mathcal{H}_{C_{1_1}}, \mathcal{H}_{C_1}, \mathcal{H}_{C_2}, \mathcal{H}_{C_3}, \mathcal{H}_{C_D}, \mathcal{H}_{C_0}\}$, $\Sigma = \mathbb{N} \setminus 0$ *und sei die Menge der Produktionen* $\mathcal{R}$ *wie folgt:*

$$\begin{array}{lll} \mathcal{R} = & P_{A_s,2} \rightarrow 1P_{A_3,1} & \mathcal{H}_{C_{1_1}} \rightarrow 2\mathcal{H}_{C_0} \\ & P_{A_3,1} \rightarrow \lambda & \mathcal{H}_{C_2} \rightarrow \mathcal{H}_{C_1}\mathcal{H}_{C_D} \\ & P_{A_t,2} \rightarrow 1P_{B_3,1} & \mathcal{H}_{C_D} \rightarrow \mathcal{H}_{C_1}\mathcal{H}_{C_1} \\ & P_{B_3,1} \rightarrow \lambda & \mathcal{H}_{C_1} \rightarrow 2\mathcal{H}_{C_0} \\ & & \mathcal{H}_{C_0} \rightarrow \lambda \end{array}$$

An diesem Beispiel sieht man, dass die $\mathcal{H}$*-Typen nirgends in den rechten Seiten von den* $\mathcal{P}$*-Typen auftauchen. Der Grund ist, dass das Zeichen an der zweiten Stelle im Term nicht mit den Kontext-Nonterminalen in der STG in Berührung kommt.*

Durch Induktion in der Tiefe von N bzw. $\mathbf{d}\text{epth}(N)$ ist es überprüfbar, dass jedes $\mathcal{P}_{N,k}$ jeweils $iPos(w_N, k)$ generiert. Jedes $\mathcal{P}_{N,k}$ erzeugt durch Anwendung der obigen Regeln höchstens ein neues $\mathcal{P}_{N',k'}$ und das korrespondierende N' genügt der Bedingung $\mathbf{d}\text{epth}(N') < \mathbf{d}\text{epth}(N)$. Daher braucht man höchstens $\mathbf{d}\text{epth}(G)$ von den Umformungsregeln aus der Abbildung 4.5. Aus diesen Überlegungen folgt das nächste Lemma.

Lemma 4.2.4 *Sei* G *eine STG, und seien* A_s *und* A_t *Term-Nonterminale von* G*, so dass* $w_{A_s} \neq w_{A_t}$*. Es ist möglich eine SCFG* P *mit Positionsnonterminalen* p*, die die erste unterschiedliche Position in* w_{A_s} *und* w_{A_t} *identifizieren, in* $O(|G^4|)$ *Zeit zu konstrurieren. Zusätzlich gilt auch,* $|P| \leq |G|$ *und* $\mathbf{d}epth(P) \leq \mathbf{d}epth(G)$

Beweis. [13] Lemma 3.8.
In unserer Implementierung, die im Kapitel 5 ausführlich behandelt wird, bilden wir die Positionsgrammatik mit der Funktion posgrammar. Mit Hilfe dieser erzeugten Positionsgrammatik kommen wir zum nächsten Schritt, nämlich dem Auffinden der Unterterme, worin die unterschiedliche Stelle existiert. Gesucht ist also jeweils $w_{A_t}|_{w_{\mathcal{P}_{A_t,2}}}$ und $w_{A_s}|_{w_{\mathcal{P}_{A_s,2}}}$. Die Funktion für das Auffinden der Unterterme haben wir p_ext genannt. Nachdem diese

Unterterme berechnet wurden, muss die Eigenschaft der first-order Unifizierbarkeit überprüft werden. Die nächsten Abschnitte dienen der Erläuterung dieser Probleme. Zuerst kommen die in [13] definierten Regeln für p_ext und danach der Robinson-Algorithmus in pseudocode Darstellung.

4.3 Erweiterung der STG durch die Substitution

Bevor wir zu den in [13] definierten p_ext Regeln kommen, soll hier im Detail im Falle einer Erweiterung der Grammatik gezeigt werden, wie man ein bestimmtes Suffix von w_C von einem Kontext-Nonterminal C von G bildet [22].

Definition 4.3.1 *Sei G eine STG und sei C ein Kontext-Nonterminal von G. Sei l eine natürliche Zahl mit* $l \leq |hp(w_C)|$. *Wir definieren* ***Suff(G,C,l)*** *als Erweiterung für G rekursiv folgendermaßen:*

SUFF(G, C, l)
1 **if** $l = 0$
2 **then** SUFF$(G, C, l) := G$
3 ▷ In allen übrigen Fällen nehmen wir $l > 0$ an
4 **if** (C $\rightarrow$ $C_1C_2 \in$ G) $\wedge$ $l <$ $|\, hp(w_{C_1})|$
5 **then** SUFF(G, C, l) beinhaltet SUFF(G, C_1, l), welches ein
6 Nonterminal C_1' hat,das ein Suffix von w_{C_1} ist mit
7 $|\, hp(w_{C_1'})| = |\, hp(w_{C_1})| - l$, zusätzlich
8 kommt die Regel hinzu: $C' \rightarrow C_1'C_2$
9 ▷ C' ist ein neu dazugekommenes Nonterminal
10 **if** (C $\rightarrow$ $C_1C_2 \in$ G) $\wedge$ $l \geq$ $|\, hp(w_{C_1})|$
11 **then** mit $l' := l - |\, hp(w_{C_1})|$ definieren wir
12 SUFF(G, C, l) als SUFF(G, C_2, l')
13 **if** (C $\rightarrow$ $f(A_1, \ldots, A_{i-1}, C_i, A_{i+1}, \ldots, A_m) \in G$
14 **then** definieren wir SUFF(G, C, l) als SUFF$(G, C_i, l-1)$
15 ▷ In allen übrigen Fällen ist SUFF(G, C, l) undefiniert

Abbildung 4.6: Bildung von Suffixen

Die hier gezeigte $Suff$- Funktion wird beim Berechnen des Unterterms in der Funktion $pExt$ in Abbildung 4.7, Zeile 5 aufgerufen. Wir werden weiter unten $pExt$ erklären und anschließend die Regeln auf die STG vom Beispiel 4.2.2 und der zugehörigen Positionsgrammatik in Beispiel 4.2.3 anwenden. Denn $pExt$ erwartet im ersten Argument die STG G, im zweiten Argument das Nonterminal N, dessen Unterterm berechnet wird. Dieser Unterterm

wird mit Hilfe des Positionsnonterminals p im dritten Argument und der Positionsgrammatik P berechnet.

Lemma 4.3.2 *Sei G eine STG, die first-order Terme und Kontexte beschreibt. Sei C ein Kontext-Nonterminal von G, und l eine natürliche Zahl mit $l \leq |hp(w_C)|$. Schließlich ist $G' = Suff(G, C, l)$ berechenbar in $O(|G|)$. Es kommen höchstens $depth(C)$ viele neue Kontext-Nonterminale hinzu, wobei das Kontext-Nonterminal C' von G' den Suffix von w_C mit $|hp(w_{C'})| = |hp(w_C)| - l$ erzeugt. Für jedes neue Kontext-Nonterminal N ist $depth_{G'}(N) \leq depth_G(C)$ und $depth(G') = depth(G)$.*

Beweis. [13] Lemma 3.2.

Dieses Lemma besagt, dass sich die Tiefe der Grammatik nicht ändert. Das neu dazugekommene Kontext-Nonterminal C' erzeugt im Endeffekt den möglichen Unterterm, der auf die Variable abgebildet wird. Das Erweitern von G um ein Nonterminal, das $w_N|_{w_p}$ erzeugt, für ein Nonterminal N von G und ein Positionsnonterminal p von P ist in polynomieller Zeit berechenbar. Dies wurde in [22] schon gezeigt. An dieser Stelle können die Regeln für die Erweiterung der Grammatik definiert werden.

Definition 4.3.3 *Sei G eine STG, die first-order Terme und Kontexte beschreibt und sei P eine SCFG, die Positionen beschreibt. Sei p ein Positionsnonterminal von P und N ein Nonterminal von G. Folgende rekursiv definierte Funktion pExt(G,N,p,P) soll die Grammatik G erweitern.*

pExt(G, N, p, P)

if $w_p = \lambda$ (das leere Wort)
  then pExt$(G, N, p, P) := G$
    ▷ In allen übrigen Fällen nehmen wir $w_p \neq \lambda$ an
if (N $\rightarrow$ $C_1N_2 \in$ G) $\wedge$ $w_p < hp(w_{C_1})$
  then pExt(G, N, p, P) beinhaltet Suff$(G, C_1, |w_p|)$, welches ein
    Nonterminal $C_1^{'}$ hat, das ein Suffix von w_{C_1} ist mit
    $|\,hp(w_{C_1^{'}})| = |\,hp(w_{C_1})| - |w_p|$, zusätzlich
    kommt die Regel hinzu: $N^{'} \rightarrow C_1^{'}N_2$
    ▷ $N^{'}$ ist ein neu dazugekommenes Nonterminal
if (N $\rightarrow$ $C_1N_2 \in$ G) $\wedge$ w_p ist *disjoint* von $hp(w_{C_1})$
  then pExt(G, N, p, P) := pExt(G, C_1, p, P)
if (N $\rightarrow$ $C_1N_2 \in$ G) $\wedge$ $hp(w_{C_1}) \leq w_p$
  then Erweitere P mit $depth(p)$ neuen Nonteminalen, wobei das
    Anfangsnonterminal p' das Suffix von w_p erzeugt mit der
    Länge $|w_p| - |\,hp(w_{C_1})|$, und definieren
    pExt(G, N, p, P) := pExt$(G, N_2, p^{'}, P)$
if (N $\rightarrow$ $f(N_1, \ldots, N_m) \in G \wedge i \leq w_p$ mit $1 \leq i \leq n$
  then Erweitere P mit $depth(p)$ neuen Nonteminalen, wobei das
    Anfangsnonterminal p' das Suffix von w_p erzeugt mit der
    Länge $|p| - 1$ und definieren,
    pExt(G, N, p, P) := pExt$(G, N_i, p^{'}, P)$
▷ In allen übrigen Fällen ist pExt(G, N, p, P) undefiniert

Abbildung 4.7: Bildung des Unterterms $w_N|_{w_p}$

Nachdem oben die Eingaben für *pExt* kurz erläutert wurden, kann nun der Pseudocode genauer beschrieben werden. Die erste Zeile überprüft das Positionsnonterminal p auf das leere Wort. Falls dieser Fall zutrifft, bricht die Ausführung ab und die Grammatik wird zurückgegeben. In allen übrigen Fällen bildet p nicht das leere Wort. Die restlichen Abfragen überprüfen die rechten Seiten des Nonterminals N und verzweigen dementsprechend in den *then*-Zweig und führen die Anweisungen aus. Zusätzlich wird das gegebene p mit dem $hp(w_{C_1})$ verglichen. Da hier die Terme voneinander auf die Präfix-Eigenschaft überprüft werden [5], würde der expandierte Vergleich (Erzeugen der Wörter) der Terme für bestimmte Grammatiken erheblichen Zeit- und Platzaufwand verursachen [6]. Man könnte diese Ineffienz in den Griff bekommen, falls viele Wiederholungen im erzeugten Term enthalten sind (s. Beispiel 5.1.1). Allerdings existieren Grammatiken, die keine Wiederholun-

[5] $w_p < hp(w_{C_1})$ bedeutet, ob w_p ein Präfix von $hp(w_{C_1})$
[6] Aus unseren Erfahrungen ist ein *stack overflow* unvermeidlich

gen [7] beinhalten. In der gesammten Implementierung wird zu keiner Zeit mit expandierten Termen gearbeitet. Um dies an dieser Stelle zu garantieren, benutzen wir den Plandowski Testalgorithmus um den Präfix-Vergleich durchzuführen. Für $wp < hp(w_{C_1})$ werden die folgenden Schritte benötigt:

- Erzeuge ein Präfix $b^{'}$ von C_1 der Länge $|w_p|$
- Prüfe die Testmenge $\{(b^{'}, C_1)\}$ mit Plandowski
- Wenn Plandowski = True, dann Präfix, ansonsten kein Präfix

Um das Präfix b' erzeugen zu können, muss zuzätzlich noch die Bedingung $|w_p| < |hp(w_{C_1})|$ gelten. Eine solche Präfix-Abfrage findet in der Zeile 4 statt. Falls die Präfix-Eigenschaft zutrifft, dann bildet man einen Suffix von C_1. Dieses neue Suffix hat die Länge $|hp(C_1)|-|w_p|$. Das bedeutet, die ersten $|w_p|$ Stellen werden ignoriert, danach sucht man nach dem Unterterm, der genau bis zur Stelle $|hp(C_1)|$ geht. Hier existiert eine Verbindung zwischen der Länge des Positionsnonterminals p und der Länge des Kontext-Nonterminals C_1.

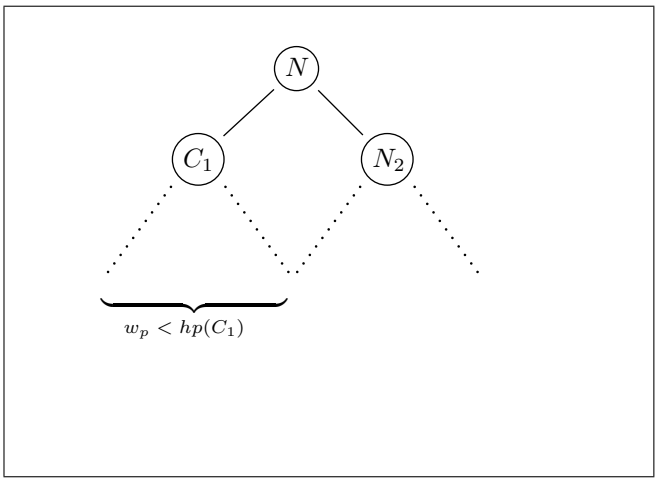


Abbildung 4.8: Ableitungsbaum

Wenn wp kein Präfix von $hp(w_{C_1})$ ist, macht der Algorithmus mit den übrigen Anweisungen weiter. In Zeile 11 wird genau der umgekehrte Fall getestet. Hier wird nach der Bedingung $hp(w_{C_1}) < wp$ geprüft. Man wiederholt die obigen drei Schritte, indem diesmal ein Präfix $b^{'}$ von wp der Länge C_1 gebildet wird. Anschließend kommt der Plandowski-Test. Falls der Test *True* ist, dann wird P erweitert, um ein Suffix $p^{'}$ von w_p mit der Länge $|w_p|-|hp(w_{C_1})|$. Weiterhin geht der rekursive Aufruf mit diesem $p^{'}$ in Zeile 14 weiter. Wenn

[7] mit Wiederholungen meinen wir run's, siehe das Beispiel 5.1.1

keiner dieser Fälle zutrifft, dann sind die beiden Nonterminale *disjoint* zueinander. Das heisst, kein Nonterminal ist der Präfix von einem Anderen. In diesem Fall ruft man die Funktion mit C_1 in Zeile 10 auf. In Zeile 15 haben wir den Funktionsfall. Hier läuft man an die Stelle in den Argumenten soweit, bis uns die Positionsgrammatik stoppt, wenn in Zeile 15 die Bedingung für das größte i zutrifft. Mit diesem i wird das Nonterminal (Zeile 18) identifiziert, und der Aufruf mit diesem Nonterminal weitergeführt.
Nimmt man das Positionsnonterminal $p = P_{A_s,2}$ aus dem Beispiel 4.2.3 und das Term-Nonterminal $N = A_s$ aus dem Beispiel 4.2.2, dann ist der erste Aufruf $pExt(G, A_s, P_{A_s,2}, P)$. G ist die STG-Grammatik aus dem Beispiel 4.2.2 und P die Positionsgrammatik aus dem Beispiel 4.2.3. In Abbildung 4.7 trifft die Zeile 15 zu, weil

$$A_s \to pA_3A_4 \wedge 1 \leq \overbrace{P_{A_s,2}}^{1\lambda}$$

zutrifft. Dementsprechend wird P um die Anweisungen zwischen Zeile 12-14 erweitert. Das heisst, es wird ein neues Nonterminal $p^{'}$ erzeugt, der ein Suffix von $P_{A_3,1}$ mit einer Länge um eins kürzer ist. Das würde bedeuten $p^{'} \to \lambda$. Der nächste Aufruf lautet $pExt(G, A_3, p^{'}, P)$ und wir erhalten laut Zeile 1

$$p^{'} \to \lambda.$$

Die Abbruchbedingung traf zu, und die Rückgabe ist die erweiterte Grammatik G, die den Unterterm A_3 beinhaltet.
Für $p = P_{A_t,2}$ mit der gegebenen Grammatik und der gegebenen Positionsgrammatik erhalten wir den Unterterm B_3. Nachdem die Unterterme A_3 und B_3 berechnet wurden, kann im nächsten Abschnitt der Unifikationsalgorithmus von Robinson besprochen werden.

Lemma 4.3.4 *Sei G eine STG, die first-order Terme und Kontexte beschreibt und sei P eine SCFG, die Positionen beschreibt. Sei p ein Positionsnonterminal von P und N ein Nonterminal von G. Es kann in $O(|G| + |P|^4)$ überprüft werden, ob w_p eine gültige Position von w_N ist. Darüberhinaus ist $G^{'} = pExt(G, N, p, P)$ in $O(|G| + |P|^4)$ berechenbar. Es kommen höchstens $depth(N)$ neue Nonterminale hinzu und ein Nonterminal von $G^{'}$ erzeugt $w_N|_{w_p}$. Schließlich gilt für jedes Nonterminal $N^{'}$: $depth_{G^{'}}(N^{'}) \leq depth_G(N)$, mit $depth(G^{'}) = depth(G)$.*

Beweis. [13] Lemma 3.4.

4.4 Der Unifikationsalgorithmus

Die Unifikation auf Termen wendet die Substitution an, wenn eine Variable isoliert wurde. Die Substitution wird in dieser Diplomarbeit auf STG-Termen

angewendet. In STG-Grammatiken haben Variablen, Terminalsymbole die Stelligkeit 0. Um eine Substitution zu erzeugen, ersetzt man diese Variable, indem sie in ein Term-Nonterminal umgewandelt wird und mit der entsprechenden Belegung auf die rechte Seite gesetzt wird. Die Definition dieses Substitutionsschrittes halten wir folgendermaßen fest:

Definition 4.4.1 *Sei G eine STG und sei X ein Terminal, das eine Variable erster Ordnung repräsentiert, sowie A ein Term-Nonterminal von G. Dann ist $\{X \mapsto A\}(G)$ definiert als die erweiterte STG um die Produktion $X \to A$, hierbei wird X in ein Term-Nonterminal konvertiert.*

Beispiel 4.4.2 *Für die gegebene STG haben wir die Unterterme mit Hilfe von pExt berechnet. Das liefert uns ein neues Nonterminal N'_{A_s}, das $w_{A_s}|_{w_{A_s,2}}$ erzeugt. Wir müssen überprüfen, dass die Variable y_1 nicht in $w_{A_s}|_{w_{A_s,2}}$ auftaucht, sonst wäre es ein occurs check. Diese Suche braucht lineare Zeit [22]. Schliesslich führen wir die folgende Substitution wie in Definition 4.4.1 durch:*

$$\{y_1 \mapsto N'_{A_s}\}(G).$$

Die neue Grammatik würde dann folgendermaßen aussehen:

$$\{B_3 \to y_1, y_1 \to N'_{A_s}, N'_{A_s} \to A_3, A_3 \to gA_7A_7, A_7 \to x_1, \ldots\ldots\}.$$

Die ersten zwei Produktionen erweitern G um G'. Insbesondere gilt:

$$w_{G',N'_{A_s}} = w_{G,A_s}|_{w_{P_{A_s},2}} = w_{G,A_s}|_{1\lambda} = A_3 = gx_1x_1.$$

Daraus folgt:

$$w_{G',A_t} = p(y_1, h(y_2, y_3, y_4)) = p(w_{G,'N'_{A_s}}, h(y_2, y_3, y_4)) = p(g(x_1, x_1), h(y_2, y_3, y_4)).$$

Diese Stelle war also unifizierbar. Die Lösung ist in G' enthalten. Wenn mehrere solcher Substitutionen angewendet werden, kann im Allgemeinen die Tiefe des Nonterminals in G wachsen. Um zu zeigen, dass das Tiefenwachstum während mehrerer Unifikationen polynomiell beschränkt ist, brauchen wir eine Tiefendefinition $Vdepth$, welches keine Vergrößerung nach der Unifikation zu Folge hat. Es erlaubt uns das endgültige Wachstum nach oben abzuschätzen. Die Definition von $Vdepth$ ist identisch mit $depth$, ansonsten 0 für Nonterminale N, die einer speziellen Menge V angehören, mit folgender Bedingung:

Definition 4.4.3 *Sei $\mathcal{G} = (\mathcal{TN}, \mathcal{CN}, \Sigma, \mathcal{R})$ eine Singleton-Baum-Grammatik, und V eine Teilmenge von $\mathcal{TN} \cup \Sigma$. V heisst eine λ-Menge für G, wenn für jedes Nonterminal A in V, die Produktion von G folgende Form hat: $A \to \alpha$ ist eine λ-Regel.*

Definition 4.4.4 *Sei $\mathcal{G} = (\mathcal{TN}, \mathcal{CN}, \Sigma, \mathcal{R})$ eine Singleton-Baum-Grammatik, und V eine λ-Menge für G. Für jedes Nonterminal N in G, die Definition von $Vdepth_{G,V}(N) = Vdepth_V(N) = Vdepth(N)$, wenn G oder V klar vom Kontext sind, folgendermaßen (ausserdem ist $max(\emptyset) = 0$):*

$$Vdepth(N) := \begin{cases} 0, \textit{ wenn } N \in V \\ 1 + max\{Vdepth(N') \mid \exists (N \rightarrow \beta_1 N' \beta_2) \in G \\ \wedge\, N' \in (\mathcal{TN} \cup \mathcal{CN})\}, \textit{ ansonsten} \end{cases}$$

$Vdepth(G)$ ist das Maximum der $Vdepth$'s von den Nonterminalen.

Das folgende Lemma folgt aus den obigen Definitionen und sagt aus, dass eine Substitution $Vdepth$ nicht ändert.

Lemma 4.4.5 *Sei G und V wie oben definiert, sei $X \in V$ ein Terminal-Symbol der Stelligkeit 0, und sei A ein Term-Nonterminal von G.*
Sei $G' = \{X \mapsto A\}(G)$. Dann gilt für jedes Nonterminal N von G, dass $Vdepth_{G'}(N) = Vdepth_G(N)$.

Wir müssen noch zeigen, dass $Vdepth$ während der Konstruktion von G in $pExt(G, A, p, P)$ nicht wächst. Schließlich muss gezeigt werden, dass $Vdepth$ während der Konstruktion von $Suff(G, C, l)$ nicht wächst. Die Beweise können im Anhang von [13] nachgelesen werden.

Lemma 4.4.6 *Sei G eine STG, C ein Kontext-Nonterminal von G, V die Menge von Terminalen und Term-Nonterminalen von G. Sei l eine natürliche Zahl kleiner als $|hp(w_C)|$ und $G' = Suff(G, C, l)$. Dann gilt für jedes Nonterminal N von G, dass $Vdepth_G(N) = Vdepth_{G'}(N)$ und für jedes neue Nonterminal N in G' und nicht in G gilt, dass $Vdepth_{G'}(N) \leq Vdepth_G(C)$. Darüberhinaus kann man sagen, dass die Anzahl neu hinzugekommmener Nonterminale durch $Vdepth_G(N)$ beschränkt wird.*

Lemma 4.4.7 *Sei G eine STG und P eine SCFG, sei N ein Nonterminal von G, V eine λ-Menge für G, p ein Positionsnonterminal von P, so dass $w_p \in Pos(w_N)$ und sei $G' = pExt(G, N, p, P)$. Dann gilt für jedes Nonterminal N' von G, dass $Vdepth_G(N') = Vdepth_{G'}(N')$ und für jedes neue Nonterminal N'' in G' und nicht in G gilt, dass $Vdepth_{G'}(N'') \leq Vdepth_G(N)$. Darüberhinaus kann man sagen, dass die Anzahl neu hinzugekommmener Nonterminale durch $Vdepth_G(N)$ beschränkt wird.*

Die Tatsache, dass $Vdepth$ durch die grundlegenden Operationen wie der Berechnung von Untertermen (Lemma 4.4.6) oder der Substitution (Lemma 4.4.7) nicht verändert wird, dient als Beweis für die endgültige Grösse der STG, die polynomiell beschränkt ist.

```
1  ▷ Eingabe: Komprimierte Terme s und t
2  while s ≠ t
3        do
4           ▷ Suche nach der ersten unterschiedlichen Stelle, so dass
5           root(s|_p) ≠ root(t|_p)
6           if beide root(s|_p) und root(t|_p) Funktionssymbole
7              then return Die eingegebenen Terme s und t sind nicht
8                   unifizierbar
9              else  Entweder (s|_p) oder (t|_p) ist eine Variable x,
10                  z.B (s|_p)
11                  und beide sind unterschiedlich
12                  if x existiert in (t|_p)
13                     then return Die eingegebenen Terme s und t
14                          sind nicht unifizierbar (occurs check)
15                     else  return Erweitere die komprimierte
16                          Darstellung um die Anweisung {x ↦ t|_p}
17
```

Abbildung 4.9: Unifikationsalgorithmus auf STG-komprimierten Termen

Dieser Algorithmus wurde aus [13] entnommen.
Der in Abbildung 4.9 gezeigte Algorithmus ist überschaubar bzw. ein Standardverfahren. Die meisten *first-order*-Unifikationsalgorithmen sind Ableitungen dieses Schemas. Manche repräsentieren die Terme mit DAGs (gerichteten azyklischen Graphen), um das Speicherwachstum (gegeben durch Wiederholungen oder identischen Mustern in den Termen) zu vermeiden. In unserer Diplomarbeit sind die Terme durch STGs komprimiert. Die Eingabe ist eine STG und zwei Nonterminale A_s und A_t, welche die Terme s und t repräsentieren. In den vorherigen Abschnitten sahen wir, wie man die Unterterme für die erste unterschiedliche Position p berechnet. Durch $pExt$ können wir die Unterterme $s|_p$ und $t|_p$ berechnen. Die $root(s|_p)$ und $root(t|_p)$ sind voneinander verschieden. Man ersetzt die Variable $x = s|_p$ überall durch $t|_p$.
Der Algorithmus braucht polynomielle Zeit: Sei n und m der Anfangswert von $depth(G)$ und $|G|$. Wir definieren die Menge aller *first-order*-Terme am Anfang der Berechnung (bevor einer von ihnen in ein Term-Nonterminal umgewandelt wurde). Zu dieser Zeit ist $Vdepth(G) = n$. Der Wert $Vdepth(G)$ bleibt während dem Durchlauf dank Lemma 4.4.6 und Lemma 4.4.7 erhalten. Darüberhinaus sagt uns Lemma 4.4.7, dass höchstens n neue Nonterminale bei jedem Schritt hinzukommen. Weil insgesammt höchstens $|V|$ Aufrufe gemacht werden, ist die endgültige Größe von G durch $m + |V| \cdot n$ beschränkt. Jeder Aufruf kostet höchstens $O(|G|^4)$, demnach folgt daraus:

Theorem 4.4.8 *First-order Unifikation zweier Terme, die durch eine STG dargestellt werden, braucht polynomielle Zeit* $O(|V|\cdot(m+|V|\cdot n)^4)$. *Hierbei ist* m *die Eingabegrammatik in STG,* n *repräsentiert die Tiefe, und* V *die Menge unterschiedlicher Variablen, die in den Termen auftauchen. Dies gilt sowohl für das Entscheidungsproblem, als auch für die Berechnung des allgemeinsten Unifikanten.*

Kapitel 5

Implementierung

5.1 Bemerkungen zu Haskell

Haskell's Typchecker überprüft jeden gültigen Ausdruck auf einen Typ und macht dabei keine Ausnahmen, so dass es keine ungetypten Ausdrücke gibt. Nach erfolgreichem Typcheck können nach dieser Theorie keine Typfehler zur Laufzeit enstehen. Diese Art des Typchecks nennt man *starke, statische Typisierung* [18].
Eine andere Variante des Typchecks, ist die *schwache, statische Typisierung*, die den Typcheck zur Kompilierzeit vornimmt, jedoch können zur Laufzeit immer noch Typfehler auftreten, weil für den Programmierer Löcher zugelassen sind. In solchen Fällen wäre ein dynamischer Typcheck notwendig. Normalerweise haben im Programm alle Konstanten, Variablen (Bezeichner), Funktionen und Prozeduren einen Typen [18].
Die *monomorphe Typisierung* erzwingt, dass alle Objekte und insbesondere Funktionen einen eindeutigen Typ haben. Also existieren keine Typvariablen, bzw. es gibt keine Allquantoren in den Typen. Übertragen auf Haskell müsste man zwei verschiedene Längenfunktionen für Listen von Zahlen und Listen von Strings definieren [18].
Die *polymorphe Typisierung* erlaubt es Funktionen zu schreiben, die auf mehreren Typen arbeiten, wobei der Funktion schematische Typen (Typvariablen) zugeteilt werden. Alle aktuellen Parameter der Funktionsaufrufe müssen diesem Schema entsprechen. Diese Art der Typisierung nennt man *parametrischen Polymorphismus*. Dadurch ist es möglich für Listen mit verschiedenen Typen, die diesem Schema entsprechen, nur eine Längenfunktion zu definieren [18]
Die verzögerte Auswertung (*lazy evaluation*) verhindert die unnötige Auswertung von Ausdrücken, die für das Ergebnis nicht relevant sind. Das heisst, ein Ausdruck wird nur dann ausgewertet, wenn er gebraucht wird [1]. In imperativen Programmiersprachen werden die Ausdrücke ausgewertet, bis sie

[1] *nicht-strikte Auswertung*

ein Ergebnis liefern [2], um dann weitere Ausführungsschritte zu vollziehen [3]. Die Funktionen liefern immer einen Wert, wobei im Gegensatz zu funktionalen Programmiersprachen, die Funktionsargumente als eingesetzer Ausdruck nicht ausgewertet zurückgeliefert werden [4]. Beide Auswertungsstrategien haben ihre Vorteile und Nachteile. Die normale Auswertung muss darauf achten, dass durch Substitutionen keine Konflikte mit Variablennamen innerhalb eines Ausdrucks und den formalen Parametern einer Funktion enstehen, wenn die Funktion in den Ausdruck eingesetzt wird.
Der Speicherbedarf wächst, wenn Ausdrücke immer substituiert werden, ohne eine Auswertung. Einen Vorteil der normalen Auswertung gegenüber der applikativen Auswertung ist im Folgenden zu sehen:

Listing 5.1: Endlosschleife bei applikativer Auswertungsreihenfolge

```
1 bot       =  bot
2 test x y =  if x == 0
3                then 0
4                else y
```

Die Funktion `bot` ist so definiert, dass sie sich im Funktionsrumpf nochmal aufruft. Die Ausführung würde eine Endlosschleife bedeuten. Wenn `test 0 bot` aufgerufen wird, führt dies bei applikativer Auswertung zu einer Endlosschleife (wegen der strikten Auswertung), während die normale Auswertung das Ergebnis als 0 ausgibt (aufgrund der nicht-strikten Auswertung).

5.1.1 Datentypen

Datentypen in Haskell können mit dem Schlüsselwort **data** deklariert werden. Dieses Konstrukt erleichtert die Umsetzung im Programmcode. Damit können eigene Datenstrukturen definiert werden, welche von Haskell internen Typen, wie Int, String, Char oder Float abstrahieren. Zum Beispiel ist die Deklaration einer Datenstruktur über Bäume mit folgenden Zeilen gegeben:

Listing 5.2: Datentyp Baum

```
1 data Baum = Baum Int (Baum ) (Baum ) | Blatt Int
```

Die rechte Seite definiert die Datenkonstruktoren, die zu diesem Datentyp gehören. In diesem Fall heissen die Datenkonsruktoren `Baum` und `Blatt`. Die Konstruktoren können den selben Namen haben, wie der Datentyp. Den Datenkonstruktoren folgen die Typparameter, was im Falle `Baum` ein Grundtyp Int und zwei `Baum`-Typen sind. Eine besondere Eigenschaft dieser Datenstruktur ist ihre rekursive Form. Bei rekursiven Datenstrukturen stehen die Datentypen als Eingabeparameter für die Datenkonstruktoren.
Instanzen dieses Typs würden dann folgendermaßen aussehen:

[2] *strikte Auswertung*
[3] Die appl. Auswertung ist als "evaluate the argument and then apply" bekannt [17]
[4] Die normale Auswertung ist als "fully expand and then reduce" bekannt [17]

Listing 5.3: eine Instanz

```
1 nameDerInstanz :: Baum
2 nameDerInstanz = Baum 1 ( Baum 2
3                           (Blatt 3)
4                           (Blatt 4))
5                         ( Baum 5
6                           (Blatt 6)
7                           (Blatt 7))
```

Durch die Definition eines abstrakten Datentyps (ADT) wird der erste Schritt der Anforderung, nämlich die Form der Problemstellung, sichtbar. Die Information, die man verarbeiten möchte, wird in eine für das Problem beherrschbare Struktur gespeichert. In den ADT-Baum können Zahlen aus einer Liste in einer beliebigen Reihenfolge abgespeichert werden. Man kann aus einer Liste mit zufällig angeordneten Zahlen den Baum füllen. Folgende Funktion fügt Zahlen in den ADT-Baum ein:

Listing 5.4: Einfügen und Erzeugen eines Baumes

```
1 einfuegen :: [Int] -> Baum
2 einfuegen [ ]    = Blatt 0
3 einfuegen [x]    = Blatt x
4 einfuegen (x:xs) = Baum  x (einfuegen ys )
5                            (einfuegen zs )
6                    where
7                      m      = div (length xs) 2
8                      (ys,zs)= splitAt m xs
```

Nun können kontextfreie Grammatiken auf diese Weise beschrieben werden. Kontextfreie Grammatiken haben auf der linken Seite immer ein Nonterminal und können auf der rechten Seite beliebig viele Nonterminale oder Terminale haben (s. Kapitel 2.2). Wir brauchen einen Nonterminaletyp N und einen Terminaltyp T, welche zu einer Produktion vom Typ Prod folgendermaßen zusammengesetzt werden:

Listing 5.5: Datentyp für eine kontextfreie Grammatik

```
1
2 data Prod = Produktion (Symbol) [Symbol]
3
4 data Symbol = T String | TN String | I Integer
```

Beim ADT Prod ist auf der rechten Seite jede mögliche Kombination von Nonterminalen und Terminalen möglich, wobei die linke Seite als Symbol nur ein Element enthält, welches einzig vom Typ Nonterminal sein darf. Eine Grammatik, die eine Sprache beschreibt, kann nun als eine Liste von Prod's beschrieben werden. Wobei die erzeugte Sprache eine Konkatenation von Zeichenketten ist, welche in Terminalen definiert werden. Somit kann Beispiel 2.2.2, das eine Zeichenkette aus beliebig tief verschachtelten Funktionen bildet, in Haskell deklariert werden.

Listing 5.6: Grammatik für beliebig verschachtelte Funktion

```
doll ::[Prod]
doll =[Produktion (TN "S") [TN "F",TN "K_1",TN "A",
                            TN "S",TN "B",TN "K_2"],
       Produktion (TN "S") [T "x"],
       Produktion (TN "F") [T "f"],
       Produktion (TN "K_1")[T "("],
       Produktion (TN "K_2")[T ")"],
       Produktion (TN "A") [T "a"],
       Produktion (TN "B") [T "b"]
      ]
```

In Listing 5.6 sieht man eine Grammatik, die Zeichenketten produziert, welche sich immer wieder selbst reproduzieren. Um ein Wort aus dieser Grammatik abzuleiten, fängt man mit dem Startsymbol an und ersetzt die Nonterminalen, die auf den rechten Seiten auftauchen, sukzessive durch ihre Produktionen. In Haskell kann solch eine Funktion realisiert werden, wobei die Angabe eines Zählers zusätzlich die Verschachtelungstiefe angibt.

5.1.2 Module

Module sind in Haskell Dateien mit der Endung .hs. An den Anfang einer Datei kommt das Schlüsselwort **module** und anschließend der Modulname, wobei der Name der Datei und der Modulname gleich heissen müssen. Das Schlüsselwort **module** ist dabei optional, d.h. fehlt die Moduldeklaration, so wird die Datei stets als Modul namens Main geladen. Die Deklaration eines Moduls in einer Datei hat die folgende Form:

Listing 5.7: Eine Moduldeklaration für Schachbrett.hs

```
module Schachbrett (Brett(..),Position,nächsterZug,getPosition,
                    setPosition)
where
data Brett    = [(X,Y,Figur),(X,Y,Figur,(X,Y,Figur),(.),(.),..]
data X        = 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8
data Y        = A | B | C | D | E | F | G | H
data Figur    = König | Dame | Turm | Springer | Läufer | Bauer
                | Leer
data Position = X Y
.....
nächterZug    :: Figur -> Position -> Position
.....
getPosition   :: Figur -> Position
setPosition   :: Figur -> Position
getGeburtstag :: Date
..... usw .....
```

Das Prinzip der Module besteht darin, ein größeres Programm in kleinere, überschaubare Teile zu zerlegen. Die Inhalte der Module bestehen aus Funktionen, die für einen bestimmten Zweck gedacht sind. Es besteht eine Gemeinsamkeit zwischen den Funktionen innerhalb eines Moduls.

Wenn man die Funktionen eines Moduls in einem anderen Modul laden möchte, nutzt man die **import** Anweisung. Durch die Angabe von runden Klammern hinter den Modulnamen kann man die gewünschten Datentypen oder Funktionen aus diesem Modul laden. Nun soll das Modul Spiel.hs, das Modul Schachbrett.hs durch Importieren benutzen. Hierbei ist zu beachten, dass alle Module im selben Ordner enthalten sind.

Listing 5.8: Ein Import für Spiel.hs

```
1module Spiel
2where
3import Schachbrett (Brett, nächterZug, getPosition, ...)
4
5data Spieler = String
6 ....
7 ....
8spielen :: ......
9  if (istFigur getPosition brett)
10  then
11        if  (getFigur getPosition brett < meineFigur)
12        then lösch Figur
13        else nächterZug meineFigur aktuellepos    ......
```

Ausser der **import** Anweisung existiert auch der Export in Modulen. Damit verbietet man Modulen, den Import mancher Funktionen oder Datentypen. Kritische Objekte (kritisch bzgl. der Sicherheit) können somit gekapselt [5] werden und Programme erhalten von aussen keinen Zugriff auf diese Objekte. In Listing 5.7 sieht man, dass in der Export-Anweisung alle Objekte ausser getGeburtstag aufgelistet sind, womit der Zugriff auf diese Funktion verboten wird.

5.1.3 Tupel und Listen

In Haskell spielen Listen und Tupel eine große Rolle, da ein großer Teil der Probleme auf der Manipulation von Inhalten dieser Datenstrukturen basieren. Zum Beispiel können mit Listen von Floats, Werte aller erdenklichen mathematischen Funktionen abgespeichert und weiterverarbeitet werden. Eine Liste von Strings können die Wörter eines Buches abspeichern, um zum Beispiel später die Anzahl der Vorkommen eines Wortes zu bestimmen. Bei Listen gibt es die Einschränkung, dass die Elemente den selben Typ haben müssen. Um das Problem mit der Anzahl der Wörter zu lösen, bräuchte man zwei Listen, die separat Integer und Strings abspeichern. Wenn man zusätzlich noch die Zeilennummer und die Seitennummer des ersten Auftretens speichern möchte, wären noch zusätzlich zwei weitere Listen nötig. Abhilfe schafft hier das Tupel, denn Tupel können Elemente unterschiedlichen Typs beinhalten. Ein Tupel der Form $(Arzt, 100, 2, 1)$ sagt aus, dass das Wort Arzt

[5] Datenkapselung (information hiding)

hundert mal im gesammten Buch und zum ersten Mal auf Seite 2 in der ersten Zeile aufgetaucht ist. Jetzt kann man eine Liste von diesem speziellen Tupel erstellen, womit man für jedes in dem Buch auftauchende Wort solch ein Tupel in die Liste schreibt.
Durch Kombination von Tupeln und Listen kann ein einfaches Kompressionsverfahren entwickelt werden. Wenn eine Liste existiert, in der ein String aus lauter Zahlen besteht, ist es möglich diese lange Zahlenkette effizient zu komprimieren. Die Kompression ist umso besser, je größere Blocks (runs) auftauchen.

Beispiel 5.1.1 *Eine lange Zeichenkette wie*

$$\overbrace{\text{“}1111111111111}^{Block1}\overbrace{2222222222222}^{Block2}\overbrace{6666666666666}^{Block3}\overbrace{88.....\text{“}}^{Block4}$$

hat nach der genannten Kompression die Form

$$[(''1'', 13), (''2'', 15), (''6'', 15), (''8'', ..)....].$$

Eine weitere Möglichkeit effizient zu arbeiten, besteht in der Nutzung von Tupeln in Funktionsargumenten. Auf der Suche nach einem bestimmten Element in einer Liste aus Zahlen kann ein (Bool,Int)-Typ vorteilhaft sein:

Listing 5.9: Tupel

```
1
2is_elem a []       = (False,a)
3is_elem a (x:xs)   = if a==x then (True,x)
4                           else is_elem a xs
```

```
*Main> is_elem 2 [1..]
(True,2)
```

Die Funktion is_elem wird im Quellcode aufgerufen und der Zugriff auf die Tupelelemente geschieht mit den eingebauten Funktionen **fst** und **snd**.

Listing 5.10: fst und snd

```
1
2ausgabe x = if fst(x)
3            then ("Element in der Liste",snd(x))
4            else ("Element nicht in der Liste",snd(x))
```

Wenn is_elem in ausgabe aufgerufen wird, ist der Rückgabewert ein Tupel bestehend aus einem Hinweisstring und der Zahl.

Beispiel 5.1.2

```
*Main> ausgabe (is_elem 3 [1..2])
("Element nicht in der Liste",3)
*Main>
```

Tupel können bei bestimmten Situationen nützlich sein, z.B wenn die Ausgabe einer Funktion (Int,String) ist. Die Bedingung sei ein String zu erzeugen, der keine doppelten Elemente enthält. Dabei können die Elemente im String zufällig angeordnet sein. Das nächste Listing zeigt eine Funktion, die eine Liste erzeugt, jedoch nicht zufällig verteilt. Hierbei wird ein Zähler mitgegeben, der an ein String konkateniert wird, und die Funktion kette n-mal den Zähler erhöht.

Listing 5.11: fst und snd

```
1
2kette n zaehler string =
3         if n == 0
4         then (zaehler, string)
5         else kette (n-1) (zaehler+1) (string++(show zaehler))
```

```
*Main> kette 5 1 []
(6,"12345")
*Main>
```

Durch die Übergabe des Zählers wird im nächsten Aufruf von kette ein String gebildet, der unterschiedlich ist. Dadurch können eindeutige Strings erzeugt werden. Bei Übergabe einer Liste von Produktionen kann durch Weitergabe eines Zählers vermieden werden, dass beim nächsten Aufruf wieder die selben Nonterminale verwendet werden, obwohl sie schon existieren. Durch die Weitergabe des Zählers enstehen immer neue Nonterminale, die rechts auch etwas Neues produzieren. Dieser Sachverhalt wird in der Implementierung der Chomsy-Regeln nochmal hervorgehoben.

5.1.4 Ströme und unendliche Listen

Listen können in Haskell unendlich wachsen. Die Eingabe [1..] im Interpreter führt zu einer unendlichen Zahlenreihe. Die Frage ist nun, wie man auf diesen Listen operieren kann. Während die Liste aufgebaut wird, werden Elemente einzeln auf die Funktion angewendet, so dass ein Zwischenresultat ensteht. Die Funktionen müssen so geschrieben werden, dass sie auf unendlichen Listen arbeiten können. Folgende Ausführungen entnehmen wir aus [20]. Die beiden Funktionen **foldl** und **foldr** verknüpfen alle Elemente in einer Liste. Die Verknüpfung kann vielfältig sein.

- Summe aller Elemente einer Liste.

- Produkt aller Elemente einer Liste.
- Vereinigung aller Mengen (in einer Liste).
- Alle Listen in einer Liste von Listen zusammenhängen.

Die Berechnung erfolgt auf zwei Arten:

1. Summe von $(1, 2, 3, 4)$ in der Form $(((1+2)+3)+4)$
2. Summe von $(1, 2, 3, 4)$ in der Form $(1+(2+(3+4)))$

Die erste Variante beginnt schon mit dem ersten Element zu arbeiten, während die zweite Variante noch keinen auswertbaren Ausdruck erzeugt und lediglich auf das Listenende wartet. Unendliche Listen in Haskell können als Ströme aufgefasst werden. Die Modellvorstellung ist, dass man einen Eingabe-Strom nur einmal lesen kann, und nur einen kleinen Teil davon zur Verarbeitung bzw. zur Erzeugung eines Ausgabestroms verwenden bzw. speichern kann. Auf Grund der verzögerten Auswertung ist es in Haskell möglich unendliche Listen zu verarbeiten. Die Eingabe `[1..]` wird als Strom aller nätürlichen Zahlen interpretiert. Nimmt man immer das aktuelle Element ohne eine Referenz auf den Anfang des Stroms, so sorgt die Endrekursions-Optimierung zusammen mit dem Garbage Collector [6] dafür, dass viele intererassante Anwendungen den Eigabestrom bzw. die Eingabeströme mit konstantem Speicher verarbeiten können.

Beispiel 5.1.3

```
Prelude> [1..]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,.........

Prelude> map (*3) [1..]
[3,6,9,12,15,18,21,24,27,30,33,36,39,42,45,48,51,54,57,60.....
```

Hier wird jeweils ein Element aus der Liste genommen und mit drei multipliziert.

Die Abfrage nach der Länge einer Liste führt zu einer nichtterminierenden Abfrage.

Beispiel 5.1.4

```
Prelude> length [1..]
^CInterrupted
```

Beispiel 5.1.5 *Man kan eine Liste xs danach fragen, ob sie n oder mehr Elemente enthält, mittels*

[6] automatische Speicherbereinung

```
drop n xs /= [],
```

und Abfragen mit

```
lengthge str n = drop n xs /= []
```

```
*Main> lengthge [1..] 2
True
*Main> lengthge [1..10] 20
False
```

Haskell hat weitere vordefinierte Funktionen, die für Ströme verwendbar sind. Zum Beispiel filter, nub, scanr, scanl, concat, foldr, zip, zipWith, usw..

Beispiel 5.1.6
Man kan eine Liste xs danach fragen, ob sie n oder mehr Elemente enthält, mittels

```
Prelude> filter (\x -> x > 12 && x < 20) [1..]
[13,14,15,16,17,18,19 ^CInterrupted.
Prelude>
```

Beispiel 5.1.7 *Man kan eine Funktion schreiben, die jedes k-te Element aus der Liste holt:*

```
strom_ktes xs  = let (y:ys) = drop k xs
                 in
                     y:(strom_ktes ys k)
```

```
*Main> strom_ktes [1..] 3
[4,8,12,16,20,24,28,32,36,40,44,48,52,56,60,64,68,72,.....
```

Der Operator (++) konkateniert zwei Listen und macht daraus eine Liste :

```
Prelude> [1,2]++[3,4,3]++[3]
[1,2,3,4,3,3]
Prelude>
```

Ein weiterer Operator auf Listen ist map, der eine Funktion als Eingabe und eine Liste hat, wobei die Elemente in der Liste einzeln auf die Funktion angewendet werden und als Ausgabe die resultierende Liste zurückgegeben wird.

```
Prelude> map (+2) [1..10]
[3,4,5,6,7,8,9,10,11,12]
Prelude>
```

Der Operator concat erzeugt aus einer Liste aus Listen eine flache Liste :

```
Prelude> concat [[1],[2],[3],[4]]
[1,2,3,4]
Prelude>
```

Für eine detaillierte Einführung in Haskell empfehlen wir [19], [25], [24], [23] und [26].

5.2 Erklärungen

Die Implementierung besteht aus 22 Modulen, wobei hier die wichtigsten erläutert werden.

5.2.1 Modul Datentypen

In diesem Modul befinden sich unter anderem die Datentypen für Produktionen der Singleton-Tree-Grammatik, der Singleton-Kontextfreien-Grammatik, der Positionsgrammatik und der Grammatik, die als Eingabe für Plandowskis Algorithmus dient. Folgende Tabelle hilft zur Übersicht:

Produktion	**Datentyp**
Singleton-Tree-Grammatik (STG)	`STG_Prod` und `STGProd`
Singleton-Kontextfreie-Grammatik (SCFG)	`ProdPre`
Positionsgrammatik	`Pos`
Grammatik für Plandowski	`Prod`

Betrachten wir nun die einzelnen Datentypen etwas genauer:

Die STG ist die erste Eingabe für den Unifikationsalgorithmus. Wir benutzen für eine Produktion der STG die Datentypen `STG_Prod` und `STGProd`. Der Datentyp `STG_Prod` sieht wie folgt aus:

```
1 data STG_Prod = STG_Prod (STG_Pre_Symbol) [STG_Pre_Symbol]
```

Der Konstruktor hat als ersten Parameter ein `STG_Pre_Symbol`, der folgenden Typ besitzt:

```
1 data STG_Pre_Symbol = TN String | CN String | P String
2                     | L String | R  String | F String
3                     | X String | K  String
4                     | Laenge Integer | Loch | Lambda
```

Die einzelnen Konstruktoren haben folgende Bedeutung:

TN: Dies sind die Term-Nonterminale der Singleton-Tree-Grammatik.

CN: Dies sind die Kontext-Nonterminale der Singleton-Tree-Grammatik.

P: Aus einem `TN` der STG wird nach der Preorder-Traversierung ein `P` in der ProdPreOrder-Grammatik (SCFG).

L: Die `CN` der STG werden bei der Preorder-Traversierung in die linke und rechte Hälfte des Lochs aufgeteilt. `L` stellt die linke Hälfte des Lochs (des Kontexts) in der ProdPreOrder-Grammatik dar.

R: `R` stellt die rechte Hälfte des Lochs (des Kontexts) in der ProdPreOrder-Grammatik dar.

F: `F` sind Funktionssymbole mit Stelligkeit > 0 und werden sowohl in der STG, als auch in der ProdPreOrder-Grammatik verwendet.

X: `X` sind Variablen und werden sowohl in der ProdPreOrder-Grammatik, als auch in der STG verwendet.

K: `K` sind Konstanten und werden sowohl in der STG, als auch in der ProdPreOrder-Grammatik verwendet.

Laenge: Repräsentiert die Länge des Wortes, dass durch ein Nonterminal gebildet wird. Sie wird sowohl in der STG, als auch in der ProdPreOrder-Grammatik verwendet.

Loch: Dies ist das Loch in einem Kontext der STG, in dem ein Term eingesetzt werden kann.

Lambda: Aus einem Loch in der STG wird nach der Preorder-Traversierung ein `Lambda` in der ProdPreOrder-Grammatik.

Hierbei erwarten die Konstruktoren `TN, CN, F, X, K, P, L` und `R` noch einen String als Eingabe, der jeweils den Namen des Term-Nonterminals, des Kontext-Nonterminals, der Funktion, der Variablen, usw. spezifiziert. Der Konstruktor `Laenge` erwartet noch einen Integer als Eingabe, während `Loch` und `Lambda` keine Parameter besitzen.

Der erste Parameter von `STG_Prod` stellt die linke Seite einer Produktion dar. Der zweite Parameter ist eine Liste von `STG_Pre_Symbol`'en. Dieser Parameter stellt die rechte Seite der Produktion dar. Da es mehrere Symbole auf der rechten Seite einer Produktion geben kann, handelt es sich diesmal um eine Liste. Jetzt können wir eine Produktion in der Singleton-Tree-Grammatik erstellen.

Beispiel 5.2.1 *Angenommen wir möchten, dass die STG folgende Produktion besitzt:*

$$A \to f(B, D)$$

Hierbei sind A, B und D Term-Nonterminale und f ein Funktionssymbol. Im Programm würde diese Produktion wie folgt aussehen:

```
STG_Prod (TN "A") [F "f", TN "B", TN "D"].
```

Da die Grammatik aus mehreren Produktionen bestehen kann, ist es klar das die Singleton-Tree-Grammatik eine Liste von `STG_Prod` Datentypen ist (`[STG_Prod]`). Jetzt können wir eine kleine Singleton-Tree-Grammatik betrachten.

Beispiel 5.2.2

```
1  grammatik :: [STG_Prod]
2  grammatik = [STG_Prod (TN "V")    [F "z",TN "B'",TN "A_s"],
3               STG_Prod (TN "W")    [F "z",TN "A_s",TN "A_s"],
4               STG_Prod (TN "A_s")  [F "g",TN "A",TN "A"],
5               STG_Prod (TN "A")    [CN "C_2",TN "A'"],
6               STG_Prod (TN "A'")   [K "a"],
7               STG_Prod (TN "B'")   [X "x"],
8               STG_Prod (CN "C_2")  [CN "C_1",CN "C_1"],
9               STG_Prod (CN "C_1")  [CN "C_0",CN "C_0"],
10              STG_Prod (CN "C_0")  [F "f",TN "A'",CN "C'"],
11              STG_Prod (CN "C'")   [F "j",CN "C''",TN "A'"],
12              STG_Prod (CN "C''")  [Loch] ]
```

Da die Implementierung keinen Parser beinhaltet, ist dieser Datentyp vor allem für Benutzer geeignet, die sich mit den Regeln der Singleton-Tree-Grammatik gut auskennen. Wie auch leicht zu sehen ist, müssen nämlich die Produktionen bereits den Regeln entsprechend eingegeben werden. Zwar

werden bei falschen Eingaben viele Fehler abgefangen, aber das ständige Nachschauen und Ausbessern der Produktionen kann auf Dauer sehr lästig sein. Deswegen haben wir auch einen zweiten, benutzerfreundlichen Datentypen für die Produktionen verwendet. Dieser Datentyp heisst, wie auch schon vorher erwähnt, `STGProd` und sieht wie folgt aus:

```
data STGProd =
              AfA    TermNT FunSym [TermNT]
            | ACA    TermNT ContextNT TermNT
            | CLoch  ContextNT
            | CCC    ContextNT ContextNT ContextNT
            | CfAC   ContextNT FunSym [TermNT] ContextNT [TermNT]
            | Lam    TermNT TermNT
            | Ax     TermNT Variable
```

Dieser Datentyp besitzt sieben Konstruktoren, die jeweils den Regeln der Singleton-Tree-Grammatik entsprechen. Die folgende Darstellung zeigt welcher Konstruktor welcher Regel in der Singleton-Tree-Grammatik entspricht (siehe Definition 2.4.9):

$$
\begin{array}{lcl}
\texttt{AfA} & \Longleftrightarrow & A \to f(A_1, \ldots, A_m) \\
\texttt{ACA} & \Longleftrightarrow & A \to C_1 A_2 \\
\texttt{CLoch} & \Longleftrightarrow & C \to [\cdot] \\
\texttt{CCC} & \Longleftrightarrow & C \to C_1 C_2 \\
\texttt{CfAC} & \Longleftrightarrow & C \to f(A_1, \ldots, A_{i-1}, C_i, A_{i+1}, \ldots, A_m) \\
\texttt{Lam} & \Longleftrightarrow & A \to A_1(\lambda - \mathit{Produktion}) \\
\texttt{Ax} & \Longleftrightarrow & A \to x
\end{array}
$$

Die Parameter `TermNT, ContextNT, FunSym` und `Variable` der Konstruktoren sind jeweils vom Typ String. Mit den obigen Regeln sind auch die Bedeutungen der jeweiligen Parameter leicht zu verstehen:

- `TermNT` steht für Term-Nonterminal, `ContextNT` steht für Kontext-Nonterminal, `FunSym` steht für Funktionssymbol und `Variable` steht wieder für eine Variable.

- `AfA  TermNT FunSym [TermNT]`: Das erste `TermNT` steht auf der linken Seite der Produktion. `|FunSym|`ist das Funktionssymbol und `[TermNT]` sind die Argumente dieser Funktion (als Liste, da eine Funktion mehrere Argumente haben kann), die sich auf der rechten Seite der Produktion befinden. Falls die Funktion keine Argumente besitzt (`[TermNT]`=[ ]), wird das `FunSym` als Konstante angesehen, womit das `TermNT` eine Konstante produziert.

- `ACA  TermNT ContextNT TermNT`: Das erste `TermNT` steht auf der linken Seite der Produktion. Das `ContextNT` ist das erste Symbol auf der rechten Seite der Produktion, in dessen Loch das zweite `TermNT` eingesetzt wird.

- `CLoch ContextNT`: Das `ContextNT` steht für das Kontext-Nonterminal, das ein Loch produziert. Hier gibt es keine spezifische Eingabe für die rechte Seite der Produktion.
- `CCC ContextNT ContextNT ContextNT`: Das erste `ContextNT` steht für die linke Seite und die restlichen zwei für die rechte Seite der Produktion, wobei hier das zweite `ContextNT` in das Loch des ersten `ContextNT` eingesetzt wird.
- `CfAC ContextNT FunSym [TermNT] ContextNT [TermNT]`: Das erste `ContextNT` ist die linke Seite der Produktion. Die rechte Seite fängt mit einem Funktionssymbol an, das auf jeden Fall genau ein `ContextNT` in den Argumenten hat. Vor und nach diesem `ContextNT` können beliebig viele Term-Nonterminale als Argumente vorkommen (also auch gar keine, da beide `[TermNT]` =[ ] sein können).
- `Lam TermNT TermNT`: Das erste `TermNT` ist die linke und das Zweite die rechte Seite der Produktion.
- `Ax TermNT Variable`: Das `TermNT` auf der linken Seite produziert eine Variable auf der rechten Seite.

Nun können wir eine Produktion mit diesem Datentypen betrachten.

Beispiel 5.2.3 *Angenommen wir möchten, dass die STG folgende Produktion besitzt:*

$$A \to f(B, D)$$

Hierbei sind A, B und D Term-Nonterminale und f ein Funktionssymbol. Im Programm würde diese Produktion wie folgt aussehen:

```
AfA "A" "f" ["B","D"]
```

Da wir eine Liste von Produktionen für eine Singleton-Tree-Grammatik benötigen, führen wir einen neuen Typen `STG`

```
1 newtype STG = STG [STGProd]
```

ein, der als Parameter eine Liste von `STGProd`'s erwartet. Jetzt können wir die vorher mit `STG_Prod` aufgebaute Singleton-Tree-Grammatik aus Beispiel 5.2.2 mit dem neuen Typ betrachten.

```
1 grammatik :: STG
2 grammatik = STG [AfA "V" "z" ["B'","A_s"],
3                  AfA "W" "z" ["A_s","A_s"],
4                  AfA "A_s" "g" ["A","A"],
5                  ACA "A" "C_2" "A'",
6                  AfA "A'" "a" [],
7                  Ax  "B'" "x",
```

```
8                CCC   "C_2"  "C_1"  "C_1",
9                CCC   "C_1"  "C_0"  "C_0",
10               CfAC  "C_0"  "f"  ["A'"]  "C'"  [],
11               CfAC  "C'"  "j"  []  "C''"  ["A'"],
12               CLoch "C''"  ]
```

Der Datentyp `STGProd` ist in der Tat benutzerfreundlich, da die Regeln direkt an den Konstruktoren ablesbar sind. Aber trotzdem können sich unerwartete Fehler einschleichen. Bei einer größeren Grammatik kann leicht die Übersichtlichkeit verloren gehen. Dadurch kann es passieren, dass einem Term-Nonterminal und einem Kontext-Nonterminal der gleiche Name (String) gegeben wird. In diesem Fall könnte der Algorithmus später Fehler produzieren, da er nicht weiss, zu welchem Konstruktor der String gehört. Um dies zu verhindern, wird vorher überprüft, ob der Schnitt der Menge von den Strings der Term-Nonterminalen und der Menge der Strings der Kontext-Nonterminalen leer ist. Falls dies nicht zutrfifft, wird der Benutzer darauf hingewiesen, sonst gibt es keine Probleme mit der Grammatik. Dieses Missverständniss kann bei den `STG_Prod`'s nicht auftreten, da man die Konstruktoren immer vor den Strings mitnimmt (zum Beispiel `TN "A"` und `CN "A"`).
Die Wahl, welcher Typ benutzt werden soll, ist dem Benutzer überlassen. Aber intern arbeitet der Algorithmus auf `STG_Prod` Typen. Deswegen werden die `STGProd` Typen nach der Eingabe direkt in `STG_Prod` umgewandelt, was keinerlei Komplikationen oder Fehler verursacht. Die Ausgabe wird dann für den Benutzer wieder in `STGProd` konvertiert. Hierfür benutzen wir die Funktion *stgprod_ in_ stg*, die folgende Umwandlungen vornimmt:

$$
\begin{array}{lcl}
(STG\_Prod\ (TN\ a)\ [K\ f]) & \Longrightarrow & (AfA\ a\ f\ [\,]) \\
(STG\_Prod\ (TN\ a)\ [TN\ b]) & \Longrightarrow & (Lam\ a\ b) \\
(STG\_Prod\ (TN\ a1)\ [CN\ c, TN\ a2]) & \Longrightarrow & (ACA\ a1\ c\ a2) \\
(STG\_Prod\ (TN\ a)\ [X\ x]) & \Longrightarrow & (Ax\ a\ x) \\
(STG\_Prod\ (TN\ a)\ ((F\ f) : ys)) & \Longrightarrow & (AfA\ a\ f\ (nimmStr\ ys\ [\,])) \\
(STG\_Prod\ (CN\ a)\ [Loch]) & \Longrightarrow & (CLoch\ a) \\
(STG\_Prod\ (CN\ c1)\ [CN\ c2, CN\ c3]) & \Longrightarrow & (CCC\ c1\ c2\ c3) \\
(STG\_Prod\ (CN\ c1)\ ((F\ f) : ys)) & \Longrightarrow & let \\
 & & \quad t = (teile\_in\_3\ ys\ [\,]\ [\,]) \\
 & & \quad u = (first\_d\ t) \\
 & & \quad v = (second\_d\ t) \\
 & & \quad w = (third\_d\ t))\ in \\
 & & (CfAC\ c1\ f\ u\ v\ w)
\end{array}
$$

In der fünften Regel verwenden wir eine Funktion *nimmstr*. Sie erhält als Eingabe die Rechte Seite *ys* der Produktion $(STG\_Prod\ (TN\ a)\ ((F\ f) : ys))$, also eine Liste von `STG_Pre_Symbol`'en, nimmt von jedem Konstruktor den dazugehörigen String und liefert eine Liste von diesen Strings als Ergebnis zurück. In der letzten Regel benutzen wir die Funktion *teile_ in_ 3*. Die Idee

hierbei ist die rechte Seite ys der Produktion $(STG\_Prod\ (CN\ c1)\ ((F\ f) : ys))$ in drei Stücke u, v und w aufzuteilen, die wie folgt gebildet werden:

$$Sei\ ys = [\ \underbrace{TN\ a_1, \ldots, TN\ a_{i-1}}_{u},\ \underbrace{CN\ c}_{v}\ , \underbrace{TN\ a_{i+1}, \ldots, TN\ a_n}_{w}\ ],$$

wobei u und w nur die Strings der Term-Nonterminale und v den String des Kontext-Nonterminals beinhaltet ($u = [a_1, \ldots, a_{i-1}], w = [a_{i+1}, \ldots, a_n]$ und $v = c$).

Die einzigen relevanten Datentypen für den Benutzer sind `STG_Prod` und `STGProd`. Die restlichen Typen werden intern zur Weiterverarbeitung der Singleton-Tree-Grammatik benötigt. Zum Beispiel benötigen wir den Datentyp `ProdPre` wenn die Singleton-Tree-Grammatik in preorder traversiert wird und dadurch eine Singleton-Kontextfreie-Grammatik (SCFG) entsteht. Der Datentyp `ProdPre` ist dann eine Produktion dieser Grammatik und sieht wie folgt aus:

```
1 data ProdPre = ProdPreOrder (STG_Pre_Symbol) [STG_Pre_Symbol]
```

Der Konstruktor `ProdPreOrder` erwartet ein `STG_Pre_Symbol` als ersten Parameter und eine Liste von `STG_Pre_Symbol`'en (`[STG_Pre_Symbol]`) als zweiten Parameter. Wir haben den Typ `STG_Pre_Symbol` und die dazugehörigen Konstruktoren während der Erläuterung des Datentyps `STG_Prod` schon ausführlich besprochen. Die Singleton-Kontextfreie-Grammatik wird als Liste von `ProdPre`'s (`[ProdPre]`) dargestellt. Wir betrachten folgendes Beispiel:

Beispiel 5.2.4 *Die resultierende SCFG nach der Preorder-Traversierung der Singleton-Tree-Grammatik aus Beispiel 5.2.2 sieht wie folgt aus:*

```
1  grammatik :: [ProdPre]
2  grammatik = [ProdPreOrder (P "V")   [F "z",P "B'",P "A_s"],
3               ProdPreOrder (P "W")   [F "z",P "A_s",P "A_s"],
4               ProdPreOrder (P "A_s") [F "g",P "A",P "A"],
5               ProdPreOrder (P "A")   [L "C_2",P "A'",R "C_2"],
6               ProdPreOrder (P "A'")  [K "a"],
7               ProdPreOrder (P "B'")  [X "x"],
8               ProdPreOrder (L "C_2") [L "C_1",L "C_1"],
9               ProdPreOrder (R "C_2") [R "C_1",R "C_1"],
10              ProdPreOrder (L "C_1") [L "C_0",L "C_0"],
11              ProdPreOrder (R "C_1") [R "C_0",R "C_0"],
12              ProdPreOrder (L "C_0") [F "f",P "A'",L "C'"],
13              ProdPreOrder (R "C_0") [R "C'"],
14              ProdPreOrder (L "C'")  [F "j",L "C''"],
15              ProdPreOrder (R "C'")  [R "C''",P "A'"],
16              ProdPreOrder (L "C''") Lambda,
17              ProdPreOrder (L "C_2") Lambda ]
```

Ein weiterer Datentyp, der benötigt wird ist `Position`. Wie der Name auch schon verrät, ist dies der Datentyp für die Produktionen der Positionsgram-

matik. Diese Grammatik resultiert aus den Regeln in Abschnitt 4.2 (Abbildung 4.5), die sowohl die Singleton-Tree-Grammatik, als auch die Singleton-Kontextfreie-Grammatik (die ProdPreOrder-Grammatik) benutzen. Der Datentyp sieht wie folgt aus:

```
data Position = Pos (Pos_Symbol) [Pos_Symbol]
```

Der Konstruktor `Pos` erwartet als ersten Parameter ein `Pos_Symbol` und als zweiten Parameter eine Liste von `Pos_Symbol`'en. Nun betrachten wir den Typ von `Pos_Symbol`:

```
data Pos_Symbol = PT String | H String | Z Integer
```

Die einzelnen Konstruktoren haben folgende Bedeutung:

PT: Dies sind die Positionsnonterminale der Positionsgrammatik. Aus einem `TN` in der Singleton-Tree-Grammatik wird ein `PT` in der Positionsgrammatik.

H: Dies sind die Kontext-Nonterminale der Positionsgrammatik. Aus einem `CN` in der Singleton-Tree-Grammatik wird ein `H` in der Positionsgrammatik.

Z: Sie geben den Weg zu der Position an, in dem sich die zu unifizierenden Terme unterscheiden (es sind Zahlen mit deren Hilfe der Weg zu einer Position im Term gefunden wird).

Die Konstruktoren `PT` und `H` erwarten jeweils noch einen String als Eingabe, der die Namen der Positionsnonterminale bzw. Kontext-Nonterminale der Positionsgrammatik spezifiziert. Der Konstruktor `Z` hingegen erwartet einen Integerwert. Dieser bestimmt für ein Nonterminal, das auf der rechten Seite ein Funktionssymbol der Stelligkeit > 0 besitzt, in welchem Argument (in welchem Unterterm) der Pfad zu der Position liegt, worin sich die Terme unterscheiden. Zum Beispiel ist `Z 2` in der Produktion $A \to f(B, C, D)$ ein Indiz dafür, dass der Pfad im zweiten Argument der Funktion f, nämlich dem Nonterminal C, liegt. Die Positionsgrammtik wird durch eine Liste von `Position`'s (`[Position]`) dargestellt. Für die Term-Nonterminale `TN V` und `TN W` sehen die Positionsgrammatiken wie folgt aus:

Beispiel 5.2.5 *Positionsgrammatik für die Term-Nonterminale* `TN V` *und* `TN W` *aus Beispiel 5.2.2:*

```
grammar_V :: [Position]
grammar_V = [Pos (PT "TN_V,2") [Z 1,PT "TN_B',1"],
             Pos (PT "TN_B',1") [],
             Pos (H "C_2") [H "C_1",H "C_1"],
             Pos (H "C_1") [H "C_0",H "C_0"],
             Pos (H "C_0") [Z 2,H "C'"],
             Pos (H "C'") [Z 1,H "C''"],
```

```
8                 Pos (H "C' '")        []  ]
9
10
11grammar_W :: [Position]
12grammar_W = [Pos (PT "TN_W,2")    [Z 1,PT "TN_A_s,1"],
13              Pos (PT "TN_A_s,1") [],
14              Pos (H "C_2")       [H "C_1",H "C_1"],
15              Pos (H "C_1")       [H "C_0",H "C_0"],
16              Pos (H "C_0")       [Z 2,H "C'"],
17              Pos (H "C'")        [Z 1,H "C' '"],
18              Pos (H "C' '")      []  ]
```

Wie auch schon vorher erwähnt, kann der Algorithmus von Plandowski unabhängig von dem Unifikationsalgorithmus getestet (aufgerufen) werden. Hierfür verwenden wir den Datentypen `Prod`, welcher eine Produktion der Eingabegrammatik (SCFG) darstellt:

```
1data Prod = Produktion (Symbol) [Symbol]
```

Der Konstruktor `Produktion` erwartet ein `Symbol` als ersten und eine Liste von `Symbol`'en (`[Symbol]`) als zweiten Parameter. Der Typ von `Symbol` is hierbei:

```
1data Symbol = T String | N String | I Integer
```

Die Konstruktoren `T` und `N` erwarten jeweils einen String und der Konstruktor `I` ein Integer als Parameter. Die Bedeutung der Konstruktoren ist leicht nachzuvollziehen:

T: Das `T` steht für ein Terminalsymbol in der Grammatik.

N: Das `N` steht für ein Nonterminal in der Grammatik.

I: Das `I` rapräsentiert die Länge eines Wortes das durch ein Nonterminal gebildet wird (dies benötigen wir bei der Längenberechnung für jedes Nonterminal).

Dass die rechte Seite einer Produktion eine Liste von `Symbol`'en ist, kann leicht verwirren, da der Algorithmus von Plandowski eine Grammatik als Eingabe erwartet, die schon in Chomsky-Normalform ist, aber die Liste mehr als zwei `Symbol`'e beinhalten könnte und somit diese Voraussetzung nicht erfüllt. In der Tat haben wir bewusst diese Möglichkeit offen gelassen, damit der Benutzer sich nicht darum kümmern muss. Die Grammatik wird, bevor wir sie dem Algorithmus übergeben, in Chomsky-Normalform gebracht. Der Benutzer kann sich also ganz auf das Ergebnis konzentrieren, ohne sich Gedanken über die Eingabegrammatik machen zu müssen. Das einzige Detail, dass der Benutzter wissen muss, ist, dass die Grammatik als eine Liste von `Prod`'s (`[Prod]`) dargestellt wird. Betrachten wir uns nun eine Grammatik:

Beispiel 5.2.6 *Eine Grammatik die als Eingabe für den Algorithmus von Plandowski dient:*

```
grammatik :: [Prod]
grammatik = [Produktion (N "A") [N "B", N "C", N "C", T "d"],
             Produktion (N "K") [N "C", N "B"],
             Produktion (N "B") [N "C", N "D"],
             Produktion (N "C") [N "D", N "E", T "x"],
             Produktion (N "D") [N "G",N "F"],
             Produktion (N "E") [N "F",N "G"],
             Produktion (N "F") [T "c"],
             Produktion (N "G") [T "c"],
             Produktion (N "V") [N "B", N "C", N "C", T "d"]]
```

Ein Datentyp, den wir intern während der Längenberechnung von jedem Nonterminal benötigen, ist `ProdGen`:

```
data ProdGen = ProdSymSym (Symbol) (Symbol,Symbol)
             | ProdIntSym (Symbol) (Integer, Symbol)
             | ProdSymInt (Symbol) (Symbol,Integer)
             | ProdInt    (Symbol) (Integer)
```

Um die Längen der Nonterminale effizient zu berechnen, beginnen wir mit den Blättern (den Terminalsymbolen, die alle Länge 1 besitzen) und arbeiten uns dann entlang den Kanten (bottom-up) im Baum bis zur Wurzel hinauf (eine genauere Beschreibung befindet sich in einem späteren Abschnitt). Dabei können unterschiedliche Situationen auftreten, in denen uns dieser Datentyp weiterhilft. Alle Konstruktoren erwarten als ersten Parameter eine Eingabe vom Typ `Symbol`. Die Konstruktoren und deren zweiten Parameter haben folgende Bedeutung:

ProdSymSym: Repräsentiert ein Nonterminal, dessen Länge noch unbekannt ist (also von keinem der zwei Nachfolger im Baum ist die endgültige Länge bekannt). Dass die rechte Seite ein Tupel ist, kann leicht eingesehen werden, da die Grammatik in Chomsky-Normalform ist.

ProdIntSym: Repräsentiert ein Nonterminal, dessen Länge zum Teil bekannt ist (die Länge des linken Nachfolgers ist bekannt, aber die des Rechten noch nicht). Die Länge des ersten Nonterminals auf der rechten Seite der Produktion ist bekannt, aber die des zweiten Nonterminals ist noch unbekannt.

ProdSymInt: Umgekehrter Fall von `ProdIntSym`. Die Länge des Nonterminals ist wieder nur zum Teil bekannt (die Länge des linken Nachfolgers ist unbekannt, aber dafür die Länge des rechten Nachfolgers). Diesmal jedoch ist die Länge des ersten Nonterminals auf der rechten Seite der Produktion unbekannt, aber die des zweiten Nonterminals ist schon komplett berechnet worden.

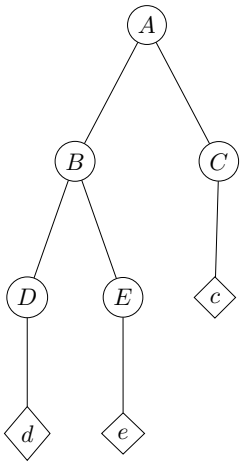
ProdInt: Repräsentiert die Nonterminale, deren komplette Länge bekannt ist (also von beiden Nachfolgern im Baum wurden die kompletten Längen schon berechnet, oder das Nonterminal hat nur ein Terminalsymbol produziert, womit dessen Länge von Anfang an schon feststeht und nicht neu berechnet werden muss).

Die Vorgehensweise der Längenberechnung wird in einem späteren Abschnitt sehr detailliert beschrieben. Das folgende Beispiel soll nur die Verwendung des Datentyps `ProdGen` verdeutlichen.

Beispiel 5.2.7 *Angenommen wir möchten von folgender Grammatik die Länge jedes Nonterminals berechnen:*

```
1 grammatik :: [Prod]
2 grammatik = [ Produktion (N "A") [N "B",N "C"],
3               Produktion (N "B") [N "D",N "E"],
4               Produktion (N "C") [T "c"],
5               Produktion (N "D") [T "d"],
6               Produktion (N "E") [T "e"] ]
```

Die Darstellung der Grammatik als Baum:



1.Nach der Berechnung der Längen für die Blätter:

```
1 längen :: [ProdGen]
2 längen = [ProdSymSym (N "A") (N "B",N "C"),
3           ProdSymSym (N "B") (N "D",N "E"),
4           ProdInt    (N "C") (1),
5           ProdInt    (N "D") (1),
6           ProdInt    (N "E") (1) ]
```

2. (o.B.d.A.) Nach dem Durchlauf für Nonterminal C:

```
1 längen :: [ProdGen]
2 längen = [ProdSymInt (N "A") (N "B",1),
3           ProdSymSym (N "B") (N "D",N "E"),
4           ProdInt    (N "C") (1),
5           ProdInt    (N "D") (1),
6           ProdInt    (N "E") (1) ]
```

3. (o.B.d.A.) Nach dem Durchlauf für Nonterminal D:

```
1 längen :: [ProdGen]
2 längen = [ProdSymInt (N "A") (N "B",1),
3           ProdIntSym (N "B") (1,N "E"),
4           ProdInt    (N "C") (1),
5           ProdInt    (N "D") (1),
6           ProdInt    (N "E") (1) ]
```

4. (o.B.d.A.) Nach dem Durchlauf für Nonterminal E:

```
1 längen :: [ProdGen]
2 längen = [ProdSymInt (N "A") (N "B",1),
3           ProdInt    (N "B") (2),
4           ProdInt    (N "C") (1),
5           ProdInt    (N "D") (1),
6           ProdInt    (N "E") (1) ]
```

5. (o.B.d.A.) Nach dem Durchlauf für Nonterminal B:

```
1 längen :: [ProdGen]
2 längen = [ProdInt (N "A") (3),
3           ProdInt (N "B") (2),
4           ProdInt (N "C") (1),
5           ProdInt (N "D") (1),
6           ProdInt (N "E") (1) ]
```

5.2.2 Modul ChomskyProd

Die Chomsky-Normalform wurde ausführlich im Kapitel 2.3 besprochen. In diesem Modul wird eine Grammatik vom Datentyp `Prod` in Chomsky-Normalform gebracht. Wie in dem Abschnitt vorher erwähnt, handelt es sich hierbei um die Eingabegrammatik des Algorithmus von Plandowski. Wir verwenden folgende Regeln, um die Grammatik in Chomsky-Normalform zu überführen:

0. Ersetze alle Regeln der Form $A \to B$ und $B \to x$ durch neue Regeln $A \to x$ und $B \to x$ (allgemeiner: ersetzte Regeln der Form $A_1 \to A_2, A_2 \to A_3, \ldots, A_n \to A_{n+1}, A_{n+1} \to x$ durch neue Regeln $A_1 \to x, A_2 \to x, \ldots, A_n \to x, A_{n+1} \to x$).

1. Alle Regeln der Form $A \to x$ werden unverändert übernommen.

2. Für alle Terminalsymbole die auf den rechten Seiten der Produktionen nicht alleine stehen, werden die dazugehörigen Nonterminale (die diese Terminalsymbole erzeugen) eingesetzt. (Die Regeln $A \to x$ und $B \to \ldots x \ldots$ werden durch die neuen Regeln $A \to x$ und $B \to \ldots A \ldots$ ersetzt).

3. Es kann Terminalsymbole geben, die durch kein Nonterminal erzeugt werden. Für jedes solche Terminalsymbol wird ein neues Nonterminal eingeführt, das dieses Terminalsymbol produziert. Danach werden diese Terminalsymbole auf allen rechten Seiten der Produktionen durch das dazugehörige neue Nonterminal ersetzt. Zum Beispiel wird die Regel $A \to BxC$ (wobei das Terminalsymbol x durch kein Nonterminal erzeugt wird) durch zwei neue Regeln $D \to x$ und $A \to BDC$ ersetzt.

4. Nun haben alle Regeln die Form $A \to B_1 \ldots B_n$. Alle Regeln für die $n = 2$ gilt, werden unverändert übernommen. Für die restlichen Regeln werden neue Nonterminale $Z_1, \ldots, Z_{n-2}$ eingeführt, so dass die Regel $A \to B_1 \ldots B_n$ durch neue Regeln $A \to B_1 Z_1, Z_1 \to B_2 Z_2, \ldots, Z_{n-3} \to B_{n-2} Z_{n-2}, Z_{n-2} \to B_{n-1} B_n$ ersetzt wird. Nun entsprechen alle Produktionen der Grammatik der Chomsky-Normalform.

In der Implementierung stellt die Funktion *cnf_ reg_ 0* die Regel 0 dar und sieht wie folgt aus:

```
cnf_reg_0 :: [Prod] -> [Prod]
cnf_reg_0 prod = let stp = (split_in_tupel_prod1 prod) in
                 durchlaufe_lambda_rules (fst stp) (snd stp) []
```

Die erste Unterfunktion, die aufgerufen wird, ist *split_in_tupel_prod1*. Diese Funktion erhält als Eingabe die komplette Grammatik $\mathcal{G}$. Sie durchläuft die Grammatik und teilt die Produktionen in zwei Listen *prods* und *lambda_prods* auf. Eine Produktion wird in die Liste *lambda_prods* aufgenommen, wenn sie von der Form

`Produktion _ [N a]`

ist. Alle restlichen Produktionen, die dieser Form nicht entsprechen, kommen in die Liste *prods*. Das Ergebnis von *split_in_tupel_prod1* ist das Tupel (*prods*, *lambda_prods*). Die Funktion *durchlaufe_lambda_rules* erhält dann als erste Eingabe die *prods* Liste, als zweite Eingabe die *lambda_prods* Liste und als dritte Eingabe eine leere Liste. Sie enthält zwei weitere Unterfunktionen, nämlich *subst_lambda_rules_in_prods* und *subst_lambda_rules_in_lambdas*, denen sie jeweils die *prods* Liste bzw. die *lambda_prods* Liste als erste Eingabe übergibt. Beide erhalten als zweite Eingabe die erste Produktion aus der *lambda_prods* Liste. Die Funktion *subst_lambda_rules_in_prods* durchläuft für diese Produktion die komplette *prods* Liste und geht dabei wie folgt vor:

Sei $A \to B$ die lambda-Produktion, für die *subst_lambda_rules_in_prods* die *prods* Liste durchlaufen muss.

1. Falls es eine Produktion $B \to C_1 \dots C_n$ (mit $n \geq 2$) in der *prods* Liste gibt, dann behalte diese und füge eine neue Produktion $A \to C_1 \dots C_n$ in die *prods* Liste hinzu.

2. Falls es keine Produktion für das Nonterminal B in der *prods* Liste gibt, dann mache nichts und gib die *prods* Liste unverändert zurück.

Die Funktion *subst_lambda_rules_in_lambdas* hingegen durchläuft für diese Produktion die komplette restliche *lambda_prods* Liste und geht dabei wie folgt vor:

Sei $A \to B$ die lambda-Produktion für die *subst_lambda_rules_in_lambdas* die *lambda_prods* Liste durchlaufen muss.

1. Falls es eine Produktion $B \to C$ in der *lambda_prods* Liste gibt, dann behalte diese und füge eine neue Produktion $A \to C$ in die *lambda_prods* Liste hinzu.

2. Falls es keine Produktion für das Nonterminal B in der *lambda_prods* Liste gibt, dann mache nichts und gib die *lambda_prods* Liste unverändert zurück.

Die beiden Funktionen werden so lange aufgerufen, bis sie für jedes Element der *lambda_prods* Liste die oben genannten Eigenschaften überprüft haben. Aber sobald sie das gesuchte Nonterminal (im obigen Beispiel B) in den Listen gefunden haben, müssen sie die jeweiligen Restlisten nicht mehr durchsuchen, da jedes Nonterminal genau eine Produktion in der Grammatik besitzt (wegen der SCFG Eigenschaft, Definition 2.4.1). Dies wird von der Funktion *durchlaufe_lambda_rules* erzwungen, da sie erst abbricht, wenn sie alle Elemente der *lambda_prods* Liste an die beiden Unterfunktionen übergeben hat. Das Ergebnis ist die Ausgabeliste von *subst_lambda_rules_in_prods*. Diese Liste enthält keine Produktionen der Form $A \to B$.

Beispiel 5.2.8 *Angenommen auf die folgende Grammatik $\mathcal{G}$ soll die Regel* 0 *angewandt werden:*

$$\begin{array}{rcll} \mathcal{G} & = & A \to B & E \to F \\ & & B \to CD & F \to y \\ & & C \to x & G \to CDyFF \\ & & D \to E & H \to GFzv, \end{array}$$

$A, B, \ldots, H$ sind hierbei Nonterminale und v, x, y und z sind Terminalsymbole.

1. *Eingabe von cnf_reg_0 ist $\mathcal{G}$.*
2. *Eingabe von split_in_tupel_prod1 ist $\mathcal{G}$.*
   Ausgabe von split_in_tupel_prod1 ist:

$$\begin{array}{rcl} prods & = & \begin{array}{lcl} B & \to & CD \\ C & \to & x \\ F & \to & y \\ G & \to & CDyFF \\ H & \to & GFzv \end{array} \end{array} \qquad \begin{array}{rcl} lambda\_prods & = & \begin{array}{lcl} A & \to & B \\ D & \to & E \\ E & \to & F \end{array} \end{array}$$

3. *Eingabe von durchlaufe_lambda_rules ist prods, lambda_prods und [ ].*
   (a) *Eingabe von subst_lambda_rules_in_prods ist prods und $A \to B$ (die erste Produktion aus der lambda_prods Liste (o.B.d.A.)).*
   Ausgabe von subst_lambda_rules_in_prods nach dem Durchlauf für $A \to B$ ist:

$$\begin{array}{rcl} prods & = & \begin{array}{l} B \to CD \\ C \to x \\ F \to y \\ G \to CDyFF \\ H \to GFzv \\ A \to CD \end{array} \end{array}$$

Endgültige Ausgabe von subst_ lambda_ rules_ in_ prods ist:

$$\begin{array}{rcll} prods & = & A \to CD & E \to y \\ & & B \to CD & F \to y \\ & & C \to x & G \to CDyFF \\ & & D \to y & H \to GFzv \end{array}$$

(b) *Eingabe von subst_ lambda_ rules_ in_ lambdas ist lambda_ prods (ohne die Produktion $A \to B$) und $A \to B$ (die erste Produktion aus der lambda_ prods Liste (o.B.d.A.)).*
Ausgabe von subst_ lambda_ rules_ in_ lambdas nach dem Durchlauf für $A \to B$ ist:

$$\begin{array}{rcl} lambda\_prods & = & D \to E \\ & & E \to F \end{array}$$

Endgültige Ausgabe von subst_ lambda_ rules_ in_ lambdas ist:

$$lambda\_prods = \emptyset$$

4. *Endgültige Ausgabegrammatik $\mathcal{G}'$ von durchlaufe_ lambda_ rules und somit auch von cnf_ reg_ 0 ist:*

$$\begin{array}{rcll} prods & = & A \to CD & E \to y \\ & & B \to CD & F \to y \\ & & C \to x & G \to CDyFF \\ & & D \to y & H \to GFzv \end{array}$$

Die Regeln 1 und 2 beinhaltet die Funktion *cnf_ reg_ 2*:

```
1 cnf_reg_2 :: [Prod] -> ([Prod],[Prod],[Prod])
2 cnf_reg_2 prod = let stp = (split_in_tupel_prod prod) in
3     durchlaufe_tprods (first stp) (second stp) (thrd stp) []
```

Die Eingabe dieser Funktion ist die Ausgabegrammatik $\mathcal{G}'$ von *cnf_ reg_ 0*. Als erste Unterfunktion wird *split_ in_ tupel_ prod* ausgeführt, die $\mathcal{G}'$ als Eingabe erhält. Sie durchläuft die Grammatik und teilt die Produktionen in drei Listen, nämlich *nprods, tprods* und *t_ n_ prods* auf. In die *nprods* Liste kommen Produktionen, auf deren rechten Seiten nur Nonterminal vorhanden sind, d.h. Produktionen der Form $A \to B_1 \dots B_n$, wobei $n \geq 2$ und $A, B_1, \dots, B_n$ Nonterminale sind. Um dies festzustellen, benutzen wir die Funktion *nurN*, die als Eingabe eine rechte Seite von einer Produktion (also eine `STG_Pre_Symbol` Liste, `[STG_Pre_Symbol]`) kriegt und überprüft, ob alle vorhandenen Elemente Nonterminale sind. Falls dies zutrifft, liefert sie True als Ergebnis. Die *tprods* Liste enthält alle Produktionen der Form $A \to x$, wobei A ein Nonterminal und x ein Terminalsymbol ist. Hierbei genügt es

abzufragen, ob die Länge der rechten Seite einer Produktion gleich eins ist, um zu bestimmen, welche Produktionen in die Liste aufgenommen werden und welche nicht. Diese Vorgehensweise funktioniert immer, da alle lambda-Produktionen in *cnf_ reg_ 0* eliminiert worden sind und daher alle restlichen Produktionen, deren rechte Seiten die Länge eins haben, die Form $A \rightarrow x$ besitzen müssen. Die übriggebliebenen Produktionen werden in die *t_ n_ prods* Liste aufgenommen, da sie auf der rechten Seite sowohl Nonterminale haben, als auch Terminalsymbole besitzen (also die Form $A \rightarrow Symbol_1 \ldots Symbol_n$ besitzen, wobei mindestens ein $Symbol_i$ ein Terminalsymbol und mindestens ein $Symbol_j$ ein Nonterminal sein muss, mit $n \geq 2$; $0 \leq i, j \leq n$ und $i \neq j$). Die Ausgabe von der Funktion *split_ in_ tupel_ prod* ist das Tripel ([*nprods*],[*tprods*],[*t_ n_ prods*]).
Die Ausgabe von *split_ in_ tupel_ prod* dient als Eingabe für die Funktion *durchlaufe_ tprods*, wobei [*nprods*] die erste, [*tprods*] die zweite, [*t_ n_ prods*] die dritte Eingabe ist. Zusätzlich kommt noch eine leere Liste als vierte Eingabe dazu. Die erste und dritte Eingabe werden an die Unterfunktion *setze_ in_ tnprods_ ein* als dessen erste und dritte Eingabe übergeben. Als zweite Eingabe kriegt *setze_ in_ tnprods_ ein* nur eine Produktion aus der *tprods* Liste. Für diese Produktion durchläuft sie die ganze *t_ n_ prods* Liste und geht dabei wie folgt vor:

Sei $A \rightarrow x$ die Produktion, für die *setze_ in_ tnprods_ ein* die *t_ n_ prods* Liste durchlaufen muss, wobei A ein Nonterminal und x ein Terminalsymbol ist. Nimm die erste Produktion aus der *t_ n_ prods* Liste. Angenommen (o.B.d.A.) diese Produktion sei $B \rightarrow Symbol_1 \ldots Symbol_n$ mit $n \geq 2$.

1. Falls das Terminalsymbol x auf der rechten Seite dieser Produktion vorkommt, ersetze alle Vorkommen mit dem Nonterminal A (Bsp:. $n = 4$, $B \rightarrow Symbol_1 x Symbol_3 x$, dann entsteht nach der Ersetzung von x die Produktion $B \rightarrow Symbol_1 A Symbol_3 A$). Falls dies nicht zutrifft, nimm die nächste Produktion aus der *t_ n_ prods* Liste und beginne von vorne.

2. Überprüfe, ob nach dem Ersetzen des Terminalsymbols alle Symbole auf der rechten Seite der Produktion Nonterminale sind. Falls dies zutrifft, füge diese Produktion in die *nprods* Liste hinzu. Falls dies nicht zutrifft, füge diese Produktion wieder in die *t_ n_ prods* Liste hinzu.

3. Mache solange weiter, bis alle Produktionen in der *t_ n_ prods* Liste überprüft worden sind. Die Ausgabe ist ein neues Tripel (*temp_ 1*, *temp_ 2*,*temp_ 3*), wobei *temp_ 1* die neue *nprods* Liste, *temp_ 2* die *tprods* Liste und *temp_ 3* die neue *t_ n_ prods* Liste ist.

Die Überprüfung der rechten Seite auf das Vorkommen des Terminalsymbols x im ersten Schritt wird mit der Funktion *enthalten* durchgeführt. Diese Funktion erhält ein Symbol als erste und eine Liste von Symbolen als

zweite Eingabe und überprüft, ob das erste Symbol in der Liste vorkommt. Falls dies zutrifft, liefert sie True. Die Funktion *ersetze1* ist für die Ersetzung des Terminalsymbols x durch das zugehörige Nonterminal zuständig. Die Eingabe ist ebenfalls ein Symbol und eine Liste von Symbolen. Hierbei durchläuft *ersetze1* die Liste und ersetzt alle Symbole x mit dem Nonterminal A und gibt die neue Liste aus. Im zweiten Schritt benutzen wir wieder die Funktion *nurN* um festzustellen, ob die Produktion in die *nprods* oder *t_n_prods* Liste eingefügt werden soll. Die Rekursion von *durchlaufe_tprods* findet mit der Ausgabe von *setze_in_tnprods_ein* statt. Dabei ist die erste Eingabe die neue *nprods* Liste (first *setze_in_tnprods_ein*), die zweite Eingabe ist die *tprods* Liste (ohne die schon überprüfte Produktion) und die dritte Eingabe ist die neue *t_n_prods* Liste (thrd *setze_in_tnprods_ein*). Die Rekursion läuft so lange, bis alle Produktionen der *tprods* Liste an die Funktion *setze_in_tnprods_ein* weitergegeben worden sind. Die endgültige Ausgabe ist das Tripel (*nprods*,*tprods*,*t_n_prods*), wobei die Listen *nprods* und *t_n_prods* aus der Ausgabe von der Funktion *setze_in_tnprods_ein* übernommen werden. Schauen wir nun die Fortsetzung des Beispiels 5.2.8 an:

Beispiel 5.2.9 *Sei die Grammatik $\mathcal{G}'$ aus Beispiel 5.2.8 die Eingabegrammatik der Funktion cnf_reg_2.*

1. *Eingabe von split_in_tupel_prod ist $\mathcal{G}'$.*
   Ausgabe von split_in_tupel_prod ist:

$$\begin{array}{rcl} nprods & = & \begin{array}{lcl} A & \rightarrow & CD \\ B & \rightarrow & CD \end{array} \end{array} \qquad \begin{array}{rcl} tprods & = & \begin{array}{lcl} C & \rightarrow & x \\ D & \rightarrow & y \\ E & \rightarrow & y \\ F & \rightarrow & y \end{array} \end{array}$$

$$\begin{array}{rcl} t\_n\_prods & = & \begin{array}{l} G \rightarrow CDyFF \\ H \rightarrow GFzv \end{array} \end{array}$$

2. *Eingabe von durchlaufe_tprods ist nprods, tprods, t_n_prods und [ ].*

   (a) *Die Eingabe von setze_in_tnprods_ein ist nprods, die Produktion $C \rightarrow x$ (die erste Produktion aus der tprods Liste (o.B.d.A)), t_n_prods.*
   Ausgabe von setze_in_tnprods_ein nach dem Durchlauf für $C \rightarrow x$ ist:

$$\begin{array}{rcl} nprods & = & \begin{array}{lcl} A & \rightarrow & CD \\ B & \rightarrow & CD \end{array} \end{array} \qquad \begin{array}{rcl} tprods & = & \begin{array}{lcl} D & \rightarrow & y \\ E & \rightarrow & y \\ F & \rightarrow & y \end{array} \end{array}$$

$$t\_n\_prods = G \to CDyFF \quad H \to GFzv$$

Die Ausgabe nach dem Durchlauf der zweiten Produktion $D \to y$ ist:

$$nprods = A \to CD \quad B \to CD \quad G \to CDDFF \qquad tprods = E \to y \quad F \to y$$

$$t\_n\_prods = H \to GFzv$$

Die endgültige Ausgabe von setze_in_tnprods_ein ist:

$$nprods = A \to CD \quad B \to CD \quad G \to CDDFF \qquad tprods = \emptyset$$

$$t\_n\_prods = H \to GFzv$$

3. *Die endgültige Ausgabe von durchlaufe_tprods und von cnf_reg_2 ist:*

$$nprods = A \to CD \quad B \to CD \quad G \to CDDFF \qquad tprods = C \to x \quad D \to y \quad E \to y \quad F \to y$$

$$t\_n\_prods = H \to GFzv$$

Die Funktion *cnf_reg_2_5* reflektiert die Regel 3:

```
cnf_reg_2_5 :: ([Prod],[Prod],[Prod]) -> [Prod] -> [Prod] ->
               Integer -> ([Prod],[Prod],[Prod])

cnf_reg_2_5 (nprod,tprod,[]) [] neuprods zaehler =
            cnf_reg_3 (nprod,tprod,[]) neuprods zaehler

cnf_reg_2_5 (nprod,tprod,[]) neu_t_n_prods neuprods zaehler =
  (cnf_reg_2_5 (nprod,tprod,neu_t_n_prods) [] neuprods zaehler)

cnf_reg_2_5 (nprod,tprod,(x:t_n_prod))
            neu_t_n_prods neuprods zaehler =
 let v = cnf2_5_sub nprod tprod x neu_t_n_prods neuprods
                    zaehler in
     cnf_reg_2_5 (first_2 v,second_2 v,t_n_prod)
                 (third_2 v) (fifth_2 v) (frth_2 v)
```

Die erste Eingabe von *cnf_reg_2_5* ist das Ausgabetripel (*nprods*,*tprods*, *t_n_prods*) der Funktion *cnf_reg_2*. Die zweite Eingabe ist eine leere Liste *neu_t_n_prods*, die mit eventuell neu entstehenden *t_n_prods* gefüllt wird. Die dritte Eingabe ist ebenfalls eine leere Liste *neuprods*, die mit den neuen Produktionen für die Terminalsymbole gefüllt wird. Ein kurzes Beispiel zur Verdeutlichung der Listen *neu_t_n_prods* und *neuprods*:

Beispiel 5.2.10 *Sei $A \rightarrow Bxy$ eine Produktion, wobei A, B Nonterminale und x, y Terminalsymbole sind, die noch von keinem Nonterminal produziert werden. Dann wird im ersten Durchlauf folgendes gemacht:*

1. *Für das Terminalsymbol x wird ein neues Nonterminal X erzeugt und die Produktion $X \rightarrow x$ in die Liste neuprods eingefügt.*

2. *Das Terminal x wird auf der rechten Seite der Produktion $A \rightarrow Bxy$ ersetzt. Die dadurch entstehende Produktion $A \rightarrow BXy$ ist immernoch ein t_ n_ prod und wird deshalb in die Liste neu_ t_ n_ prods konkateniert.*

Die vierte Eingabe ist eine Zahl vom Typ Integer (der Zähler). Bei jedem neu erzeugten Nonterminal wird der Zähler um eins erhöht. Die Idee hierbei ist, dass jedes neu erzeugte Nonterminal, sowohl in der Funktion *cnf_ reg_ 2_ 5*, als auch in der Funktion *cnf_ reg_ 3* (auf die wir später eingehen werden), eindeutig ist, d.h. nur ein mal verwendet (erzeugt) wird. Dies garantieren wir, indem die Funktion *cnf_ reg_ 2_ 5* nach dem kompletten Durchlauf den aktuellen Zähler an die Funktion *cnf_ reg_ 3* übergibt.
Die Funktion *cnf_ reg_ 2_ 5* besitzt zwei Rekursionen. Zuerst wird die *t_ n_ prods* Liste, von der ersten Eingabe, komplett durchlaufen. Dies ist die erste Rekursion. Hierbei können, wie im obigen Beispiel erläutert, neue *t_ n_ prods* entstehen, die in die *neu_ t_ n_ prods* Liste eingefügt werden. Wenn nach der ersten Rekursion die *neu_ t_ n_ prods* Liste leer ist, also keine neuen *t_ n_ prods* entstanden sind, wird die Funktion *cnf_ reg_ 3* aufgerufen. Falls die *neu_ t_ n_ prods* Liste nicht leer ist, findet die zweite Rekursion statt. Dabei wird *cnf_ reg_ 2_ 5* mit dem Tripel (*nprods*,*tprods*,*t_ n_ prods*:=*neu_ t_ n_ prods*) einer leeren Liste *neu_ t_ n_ prods* und der Liste *neuprods* und dem aktuellen Zähler aufgerufen. Im Prinzip wird diesmal die *neu_ t_ n_ prods* Liste durchlaufen und das Ganze von vorne gestartet.
Wir benötigen zusätzlich die Unterfunktion *cnf2_ 5_ sub*, die während der ersten Rekursion benutzt wird. Sie erhält als erste Eingabe die *nprods* Liste, als zweite Eingabe die *tprods* Liste, als dritte Eingabe die erste Produktion aus der *t_ n_ prods* Liste, als vierte Eingabe die *neu_ t_ n_ prods* Liste, die fünfte Eingabe ist die *neuprods* Liste und die Sechste der Zähler. Die Funktion *cnf2_ 5_ sub* geht dann wie folgt vor:

Angenommen die erste Produktion aus der *t_ n_ prods* Liste sei $A \rightarrow Bxy$, wobei A, B Nonterminale und x, y Terminalsymbole sind.

1. Finde das erste Terminalsymbol auf der rechten Seite dieser Produktion (es muss immer mindestens eines geben, da die Produktion sonst nicht in der *t_ n_ prods* Liste wäre). Hier ist es das Terminalsymbol x.

2. Überprüfe ob es schon eine Produktion für das Terminalsymbol x in der *neuprods* Liste gibt (es ist selbstverständlich, dass nicht in der

tprods Liste gesucht wird, da sonst die Funktion *cnf_ reg_ 2* dieses Terminalsymbol schon durch das dazugehörige Nonterminal ersetzt haben müsste). Falls dies zutrifft, gehe zu Schritt a, sonst zu Schritt b (da am Anfang die neuprods Liste leer ist, gehen wir zu Schritt b).

(a) Finde das zugehörige Nonterminal aus der *neuprods* Liste und setze es für das Terminalsymbol x ein. Überprüfe, ob nach dem Einsetzen nur Nonterminale auf der rechten Seite der Produktion vorhanden sind. Falls dies zutrifft, gehe zu Schrit i, sonst zu Schritt ii.

i. Füge die resultierende Produktion in die *nprods* Liste hinzu und gib als Ergebnis

$$(nprods, tprods, neu\_t\_n\_prods, Zaehler, neuprods)$$

zurück.

ii. Füge die resultierende Produktion in die *neu_ t_ n_ prods* Liste hinzu (da auf der rechten Seite der Produktion immernoch mindestens ein Terminalsymbol vorhanden ist) und gib als Ergebnis

$$(nprods, tprods, neu\_t\_n\_prods, Zaehler, neuprods)$$

zurück.

(b) Erzeuge ein neues Nonterminal X und eine neue Produktion $X \rightarrow x$. Füge diese Produktion in die *tprods* Liste und in die *neuprods* Liste hinzu. Ersetze das Terminalsymbol x durch das Nonterminal X auf der rechten Seite von $A \rightarrow Bxy$. Überprüfe, ob die dadurch resultierende Produkion nur Nonterminale auf der rechten Seite besitzt. Falls dies zutrifft, gehe zu Schritt i, sonst zu Schritt ii. (wenn wir in der Produktion $A \rightarrow Bxy$ das Terminalsymbol x durch das neue Nonterminal X ersetzen, entsteht die neue Produktion $A \rightarrow BXy$, welche immernoch ein Terminalsymbol y auf der rechten Seite besitzt. Also gehen wir zu Schritt ii.)

i. Füge die Produktion in die *nprods* Liste hinzu und gib

$$(nprods, tprods, neu\_t\_n\_prods, Zaehler + 1, neuprods)$$

zurück.

ii. Füge die Produktion in die *neu_ t_ n_ prods* Liste hinzu und gib

$$(nprods, tprods, neu\_t\_n\_prods, Zaehler + 1, neuprods)$$

zurück.

Für den ersten Schritt benutzen wir eine Unterfunktion *findeT*, die als Eingabe eine Liste von `Symbol`'en erhält. Sie fängt am Anfang der Liste an und überprüft jedes einzelne `Symbol` auf den davorstehenden Konstruktor. Sobald das erste `Symbol` mit einem `T` davor gefunden wird, liefert *findeT* dieses `Symbol` als Ergebnis. Falls kein Treffer erzielt wird, gibt die Funktion eine Fehlermeldung zurück. Für den zweiten Schritt benutzen wir die Funktion *elementR1*, die als erste Eingabe ein `Symbol` und als zweite Eingabe die *tprods* Liste kriegt. Sie überprüft, ob das `Symbol` auf der rechten Seite einer Produktion aus der *tprods* Liste vorkommt. Wenn das zutrifft, gibt sie ein Tupel mit True und dem linken Nonterminal der dazugehörigen Produktion zurück. Der Schritt 1 a) benötigt die Funktionen *ersetzeTN* und die uns schon bekannte Funktion *nurN*. Die erste Eingabe von *ersetzeTN* ist das zu ersetzende Terminalsymbol, die zweite Eingabe ist das Nonterminal mit dem das Terminalsymbol ersetzt wird und die dritte Eingabe ist eine Liste von `Symbol`'en, in denen die Ersetzung stattfinden soll. Danach überprüft *nurN* ob auf der rechten Seite nur Nonterminale vorhanden sind. Für Schritt 2 b) benötigen wir ebenfalls die Funktion *ersetzeTN* und *nurN*. Betrachten wir nun die Fortsetzung des Beispiels 5.2.9:

Beispiel 5.2.11 *Sei das Ausgabetripel (nprods,tprods,t_n_prods) aus Beispiel 5.2.9 die erste Eingabe für die Funktion cnf_reg_2_5.*

1. *Die Eingabe von cnf_reg_2_5 ist (nprods,tprods,t_n_prods), die Liste neu_t_n_prods (beim ersten Aufruf [ ]), die Liste neuprods (beim ersten Aufruf auch [ ]) und der Zähler (beim ersten Aufruf ist der Zähler auf 1 gesetzt).*

2. *Die Eingabe von cnf2_5_sub ist die Liste nprods,die Liste tprods, die erste Produktion aus der t_n_prods Liste (hier gibt es nur eine Produktion in der t_n_prods Liste, nämlich $H \to GFzv$), die neu_t_n_prods Liste, die neuprods Liste und der Zähler.*

   (a) *Ausgabe nach dem ersten Durchlauf von cnf2_5_sub ist:*

$$\begin{array}{rcl} nprods' & = & A \to CD \\ & & B \to CD \\ & & G \to CDDFF \end{array} \qquad \begin{array}{rcl} tprods' & = & C \to x \\ & & D \to y \\ & & E \to y \\ & & F \to y \\ & & N_1 \to z \end{array}$$

$$neu\_t\_n\_prods' = H \to GFN_1v \qquad Zaehler = 2$$

$$neuprods' = N_1 \to z$$

Der nächste Aufruf von cnf_reg_2_5 findet statt mit:

$$cnf\_reg\_2\_5\ (nprods', tprods', \emptyset)\ neu\_t\_n\_prods'\ neuprods'\ 2$$

womit die zweite Rekursion eintritt, also:

$$cnf\_reg\_2\_5\ (nprods', tprods', neu\_t\_n\_prods')\ [\,]\ neuprods'\ 2.$$

(b) Ausgabe nach dem zweiten Durchlauf von cnf2_5_sub ist:

$$\begin{array}{rcl} nprods'' & = & \begin{array}{lcl} A & \rightarrow & CD \\ B & \rightarrow & CD \\ G & \rightarrow & CDDFF \\ H & \rightarrow & GFN_1N_2 \end{array} \end{array} \qquad \begin{array}{rcl} tprods'' & = & \begin{array}{lcl} C & \rightarrow & x \\ D & \rightarrow & y \\ E & \rightarrow & y \\ F & \rightarrow & y \\ N_1 & \rightarrow & z \\ N_2 & \rightarrow & v \end{array} \end{array}$$

$$neu\_t\_n\_prods'' = \emptyset \qquad Zaehler = 3$$

$$neuprods'' = \begin{array}{l} N_1 \rightarrow z \\ N_2 \rightarrow v \end{array}$$

Der nächste Aufruf von cnf_reg_2_5 findet statt mit:

$$cnf\_reg\_2\_5\ (nprods'', tprods'', \emptyset)\ \emptyset\ neuprods''\ 3$$

womit die Funktion beendet wird.

3. Die endgültige Ausgabe von cnf_reg_2_5 ist:

- $(nprods'', tprods'', \emptyset)$
- $neuprods''$
- 3

Die dritte (und somit letzte) Regel wird von der Funktion *cnf_reg_3* ausgeführt. Die Eingabe von *cnf_reg_3* ist, wie oben erläutert, die Ausgabe von *cnf_reg_2_5*. In die dritte Liste des Eingabetripels werden die Produktionen hinzugefügt, die der Chomsky-Normalform genügen (diese Liste wird als leere Liste von *cnf_reg_2_5* übergeben). Diese Liste nennen wir *fertigenprods*. Die Funktion *cnf_reg_3* sieht wie folgt aus:

```
cnf_reg_3 :: ([Prod],[Prod],[Prod]) -> [Prod] -> Integer ->
             ([Prod],[Prod],[Prod])

cnf_reg_3 ([],tprods,fertigenprods) neuprods zaehler=
          (fertigenprods,tprods,neuprods)

cnf_reg_3 (((Produktion (N a) xs ):nprods),tprods,fertigenprods)
          neuprods zaehler =

      if ((length xs)==2)
      then
      cnf_reg_3 (nprods,tprods,
                 [(Produktion (N a) xs )]++fertigenprods)
                 neuprods zaehler
      else
          if (length xs)<2
          then error "Fehler bei cnf_reg_0"
          else
          let rb  = (runterbrechen zaehler (Produktion (N a) xs)
                     fertigenprods neuprods)
              bir = (first3 rb)
              iki = (second3 rb)
              uec = (thrd3 rb) in
          cnf_reg_3 (nprods,tprods,bir) uec iki
```

Alle Produktionen der Grammatik, die kein Terminalsymbol produzieren, befinden sich in der *nprods* Liste und haben die Form:

$$A \to B_1 \dots B_n, \text{ mit } n \geq 2 \text{ und } A, B_1, \dots, B_n \text{ sind Nonterminale.}$$

Die Produktionen, die der Chomsky-Normalform nicht entsprechen, haben dann die Form:

$$A \to B_1 \dots B_n, \text{ mit } n > 2 \text{ und } A, B_1, \dots, B_n \text{ sind Nonterminale.}$$

Das einzige, was *cnf_reg_3* machen muss, ist, zu überprüfen, für welche Produktionen aus der *nprods* Liste $n == 2$ gilt und diese dann als endgültige Produktionen in die *fertigenprods* Liste einzufügen. Für die restlichen Produktionen mit $n > 2$ muss die Funktion *cnf_reg_3* die rechte Seite bis auf zwei Nonterminale herunterbrechen. Während diesem Vorgang werden neue Nonterminale und dazugehörige neue Produktionen erzeugt. Die Funktion geht dabei wie folgt vor:

Sei $A \to BCDE$ die erste Produktion aus der *nprods* Liste, wobei A, B, C, D und E Nonterminale sind.

1. Überprüfe, ob die rechte Seite der Produktion gleich zwei ist. Falls dies zutrifft, gehe zu Schritt a), sonst zu Schritt 2 (die rechte Seite der Produktion $A \to BCDE$ ist größer als 2, deswegen gehen wir zu Schritt 2).

(a) Füge die Produktion in die fertigenprods Liste hinzu und nimm die nächste Produktion aus der *nprods* Liste. Beginne mit Schritt 1.

2. Falls die rechte Seite die Länge 3 besitzt und es schon ein Nonterminal in der *füreinprodfertig* Liste gibt, das die letzten zwei Nonterminale der rechten Seite erzeugt, nimm dieses Nonterminal und setze es für die letzten zwei Nonterminale ein (also aus den Regeln $X \rightarrow UVW$ und $Z \rightarrow VW$ werden zwei neue regeln $X \rightarrow UZ$ und $Z \rightarrow VW$). Füge die dadurch resultierende Produktion in die *fertigenprods* Liste und die *neuprods* Liste hinzu und gib als Ergebnis

$$(fuereinprodfertig, zaehler, neuprods)$$

aus. Nimm die nächste Produktion aus der *nprods* Liste und beginne mit Schritt 1.
Falls die rechte Seite die Länge 3 besitzt, aber es kein Nonterminal in der *füreinprodfertig* Liste gibt, das die letzten zwei Nonterminale der rechten Seite produziert, gehe zu Schritt a). Falls die rechte Seite nicht die Länge 3 besitzt, gehe zu Schritt 3 (die rechte Seite der Produktion $A \rightarrow BCDE$ ist größer als 3, deswegen gehen wir zu Schritt 3).

(a) Überprüfe ob es ein Nonterminal in der *füreinprodfertig* Liste gibt, das die ersten zwei Nonterminale der rechten Seite erzeugt. Falls dies zutrifft, setze dieses Nonterminal für die ersten zwei Nonterminale auf der rechten Seite ein (also aus den Regeln $X \rightarrow UVW$ und $Z \rightarrow UV$ werden zwei neue Regeln $X \rightarrow ZW$ und $Z \rightarrow UV$). Füge die dadurch resultierende Produktion in die *fertigenprods* Liste und die neuprods Liste hinzu und gib als Ergebnis

$$(fuereinprodfertig, zaehler, neuprods)$$

aus. Nimm die nächste Produktion aus der *nprods* Liste und beginne mit Schritt 1.
Falls es kein Nonterminal in der *füreinprodfertig* Liste gibt, das die ersten zwei Nonterminale der rechten Seite erzeugt, gehe zu Schritt b).

(b) Erzeuge ein neues Nonterminal das die letzten zwei Nonterminale der rechten Seite erzeugt. Setze dieses Nonterminal auf der rechten Seite für diese Nonterminale ein (also aus der Regel $X \rightarrow UVW$ werden zwei neue Regeln $X \rightarrow UN_{zaehler}$ und $N_{zaehler} \rightarrow VW$). Füge die daraus resultierenden Produktionen in die *füreinprodfertig* Liste und die *neuprods* Liste hinzu und gib als Ergebnis

$$(fuereinprodfertig, zaehler + 1, neuprods)$$

aus. Nimm die nächste Produktion aus der *nprods* Liste und beginne mit Schritt 1.

3. Falls es in der *füreinprodfertig* Liste ein Nonterminal gibt, das die rechte Seite ab dem zweiten Nonterminal produziert, nimm dieses Nonterminal und setze es für die Nonterminale auf der rechten Seite ein (also aus den Regeln $X \to TUVW$ und $Z \to UVW$ werden zwei neue Regeln $X \to TZ$ und $Z \to UVW$). Füge die dadurch resultierende Produktion in die *füreinprodfetig* Liste und die *neuprods* Liste ein. Nimm die Produktion des eingesetzten Nonterminals (also $Z \to UVW$) und beginne mit Schritt 2.
   Falls es so ein Nonterminal nicht gibt, gehe zu Schritt 4 (da die *füreinprodfertig* Liste noch leer ist, gehen wir zu Schritt 4).

4. Erzeuge ein neues Nonterminal, das die komplette Rechte Seite ab dem zweiten Nonterminal produziert. Ersetze die entsprechenden Nonterminale auf der rechten Seite durch das neue Nonterminal (also wird die Produktion $A \to BCDE$ durch zwei neue Produktionen $A \to BN_{zaehler}$ und $N_{zaehler} \to CDE$ ersetzt). Füge die dadurch entstehende Produktion in die *füreinprodfertig* Liste hinzu. Erhöhe den Zähler um eins, nimm die Produktion des neuen Nonterminals (also $N_{zaehler} \to CDE$) und beginne mit Schritt 2.

In den Regeln 1, 2, 3 und 4 wird die Unterfunktion *runterbrechen* benutzt. Sie erhält als erste Eingabe den Zähler, als zweite Eingabe die erste Produktion aus der *nprods* Liste, als dritte Eingabe die *füreinprodfertig* Liste und als vierte (und letzte) Eingabe die *neuprods* Liste. Sie bricht die rechte Seite der Produktion so lange runter, bis sie der Chomsky-Normalform entspricht. Die dabei neu erzeugten Nonterminale und die dazu entsprechenden Produktionen werden in die *füreinprodfertig* und *neuprods* Liste hinzugefügt. Die Überprüfung, ob es ein Nonterminal in der *füreinprodfertig* Liste gibt, (in den Regeln 2 und 3) wird mit der Funktion *istenthalten* durchgeführt. Dabei kriegt sie als erste Eingabe eine Liste von `Symbol`'en und als zweite Eingabe die *füreinprodfertig* Liste. Sie überprüft, ob die rechte Seite einer Produktion aus der *füreinprodfertig* Liste der Liste von `Symbol`'en entspricht. Falls sie solch eine Produktion findet, liefert sie ein Tupel mit True und dem linken Nonterminal der Produktion zurück. Die Abfragen nach den Längen der rechten Seiten werden mit der Haskell internen *length* Funktion durchgeführt. Die Rekursin von *cnf_ reg_ 3* findet wie folgt statt:

$$cnf\_reg\_3(nprods, tprods, fertigenprods)neuprodszaehler.$$

Hierbei ist *nprods* die gleiche Liste wie am Anfang, aber ohne die Produktion für die der Durchlauf gemacht worden ist. In der *tprods* Liste befinden sich immer die Produktionen wie am Anfang. Diese Liste wird nicht verändert. Die *fertigenprods* Liste wird von der Ausgabe der Funktion *runterbrechen* übernommen, sowie die Liste *neuprods* und der Zähler. Die Funktion bricht ab, wenn für alle Produktionen aus der *nprods* Liste die rechten Seiten überprüft

(und eventuell aufgeteilt) worden sind. Das Ergebnis ist dann das Tripel

$$cnf\_reg\_3(fertigenprods, tprods, neuprods),$$

wobei die *fertigenprods* Liste alle Produktionen der Form $A \to B_1B_2$ (A, B_1 und B_2 sind Nonterminale), die *tprods* Liste alle Produktionen der Form $C \to x$ (C ist ein Nonterminal und x ein Terminalsymbol) und die *neuprods* Liste alle in den Regeln *cnf_ reg_ 2_ 5* und *cnf_ reg_ 3* neu dazugekommenen Produktionen (die beide Formen besitzen können) enthält. Wichtig ist, dass nun alle Produktionen der Chomsky-Normalform entsprechen.

Betrachten wir nun die Fortsetzung des Beispiels 5.2.11:

Beispiel 5.2.12 *Sei die Eingabe von cnf_ reg_ 3 das Tripel (nprods,tprods, $\emptyset$:=fertigenprods), die neuprods Liste und die Zahl 3 (die Ausgabe von cnf_ reg_ 2_ 5):*

$$\begin{array}{lcl} nprods & = & A \to CD \\ & & B \to CD \\ & & G \to CDDFF \\ & & H \to GFN_1N_2 \end{array} \qquad \begin{array}{lcl} tprods & = & C \to x \\ & & D \to y \\ & & E \to y \\ & & F \to y \\ & & N_1 \to z \\ & & N_2 \to v \end{array}$$

$$fertigenprods = \emptyset \qquad Zaehler = 3$$

$$neuprods = \begin{array}{l} N_1 \to z \\ N_2 \to v \end{array}$$

1. *Die Produktionen von A und B haben rechte Seiten der Länge 2 und werden deshalb in die fertigenprods Liste übernommen:*

$$\begin{array}{lcl} nprods & = & G \to CDDFF \\ & & H \to GFN_1N_2 \end{array} \qquad \begin{array}{lcl} tprods & = & C \to x \\ & & D \to y \\ & & E \to y \\ & & F \to y \\ & & N_1 \to z \\ & & N_2 \to v \end{array}$$

$$\begin{array}{lcl} fertigenprods & = & A \to CD \\ & & B \to CD \end{array} \qquad Zaehler = 3$$

$$neuprods = \begin{array}{l} N_1 \to z \\ N_2 \to v \end{array}$$

2. *Sei nun die Produktion $G \to CDDFF$ die Eingabe.*
   Die erste Eingabe von runterbrechen ist der Zähler, die zweite Eingabe die Produktion $G \to CDDFF$, die dritte Eingabe die fertigenprods Liste und die vierte Eingabe die neuprods Liste.
   Nach dem Durchlauf sieht die Ausgabe von runterbrechen wie folgt aus:

$$\begin{array}{rcl} fuereinprodfertig & = & N_3 \to CN_4 \\ & & N_4 \to DN_5 \\ & & N_5 \to DN_6 \\ & & N_6 \to FF \\ & & A \to CD \\ & & B \to CD \end{array} \qquad Zaehler = 7$$

$$\begin{array}{rcl|rcl} neuprods & = & N_4 \to DN_5 & N_1 \to z \\ & & N_5 \to DN_6 & N_2 \to v \\ & & N_6 \to FF & N_3 \to CN_4 \end{array}$$

3. *Die endgültige Ausgabe von runterbrechen ist:*

$$\begin{array}{rcl} fuereinprodfertig & = & N_3 \to CN_4 \\ & & N_4 \to DN_5 \\ & & N_5 \to DN_6 \\ & & N_6 \to FF \\ & & N_7 \to GN_8 \\ & & N_8 \to FN_9 \\ & & N_9 \to N_1N_2 \\ & & A \to CD \\ & & B \to CD \end{array} \qquad Zaehler = 10$$

$$\begin{array}{rcl|rcl} neuprods & = & N_1 \to z & N_6 \to FF \\ & & N_2 \to v & N_7 \to GN_8 \\ & & N_3 \to CN_4 & N_8 \to FN_9 \\ & & N_4 \to DN_5 & N_9 \to N_1N_2 \\ & & N_5 \to DN_6 \end{array}$$

4. *Die endgültige Ausgabe von cnf_reg_3 ist das Tripel*

   - $(fertigenprods, tprods, neuprods)$ *mit:*

$$\begin{array}{rcl|rcl} fertigenprods & = & A \to CD & N_6 \to FF \\ & & B \to CD & N_7 \to GN_8 \\ & & N_3 \to CN_4 & N_8 \to FN_9 \\ & & N_4 \to DN_5 & N_9 \to N_1N_2 \\ & & N_5 \to DN_6 \end{array}$$

$$
\begin{array}{rcllll}
tprods & = & C & \rightarrow & x & F \rightarrow y \\
 & & D & \rightarrow & y & N_1 \rightarrow z \\
 & & E & \rightarrow & y & N_2 \rightarrow v
\end{array}
$$

$$
\begin{array}{rcllll}
neuprods & = & N_1 & \rightarrow & z & N_6 \rightarrow FF \\
 & & N_2 & \rightarrow & v & N_7 \rightarrow GN_8 \\
 & & N_3 & \rightarrow & CN_4 & N_8 \rightarrow FN_9 \\
 & & N_4 & \rightarrow & DN_5 & N_9 \rightarrow N_1N_2 \\
 & & N_5 & \rightarrow & DN_6 &
\end{array}
$$

Die Ausgabe von *cnf_reg_3* wird dann von der Hauptfunktion *chomsky* bearbeitet, womit die endgültige Ausgabe erhalten wird. Die Hauptfunktion *chomsky* sieht wie folgt aus:

```
1 chomsky :: [Prod] -> Integer -> [Prod]
2 chomsky prod zaehler = let chs = chomsky_sub prod zaehler in
3                            (first chs ++ second chs)
4
5 chomsky_sub :: [Prod] -> Integer -> ([Prod],[Prod],[Prod])
6 chomsky_sub prod zaehler = let cnf2 = (cnf_reg_2 prod) in
7                             if thrd cnf2 == []
8                             then cnf_reg_3 cnf2 [] zaehler
9                             else cnf_reg_2_5 cnf2 [] [] zaehler
```

In der Unterfunktion *chomsky_sub* wird entschieden, ob nach der Anwendung der Funktion *cnf_reg_2* direkt die Funktion *cnf_reg_3* ausgeführt werden soll, oder ob zuerst die Funktion *cnf_reg_2_5* und dann die Funktion *cnf_reg_3* ausgeführt werden soll. Es geht also um die Ersparnis der Funktion *cnf_reg_2_5*. Dies ist dann der Fall, falls es nach der Anwendung der Funktion *cnf_reg_2* keine *t_n_prods* gibt (also wenn die dritte Liste des Ausgabetripels von *cnf_reg_2* leer ist). Dann können wir direkt zur Funktion *cnf_reg_3* übergehen. Ausserdem beginnt hier die Übergabe des Zählers von *chomsky* an die Funktion *chomsky_sub* und von *chomsky_sub* an die weiteren Unterfunktionen *cnf_reg_3* oder *cnf_reg_2_5*. Die Ausgabe von *chomsky_sub* ist das Ausgabetripel von *cnf_reg_3* und die endgültige Ausgabe von *chomsky* ist die Konkatenation der ersten und zweiten Liste dieses Tripels (die Bedeutung der *neuprods* Liste wird in einem späteren Abschnitt erklärt). Die Fortsetzung des Beispiels 5.2.12 ist dann:

Beispiel 5.2.13

1. *Sei das Ausgabetripel von cnf_reg_3 aus Beispiel 5.2.12, die Ausgabe von chomsky_sub:*

$$
\begin{array}{rcllll}
fertigenprods & = & A & \rightarrow & CD & N_6 \rightarrow FF \\
 & & B & \rightarrow & CD & N_7 \rightarrow GN_8 \\
 & & N_3 & \rightarrow & CN_4 & N_8 \rightarrow FN_9 \\
 & & N_4 & \rightarrow & DN_5 & N_9 \rightarrow N_1N_2 \\
 & & N_5 & \rightarrow & DN_6 &
\end{array}
$$

$$
\begin{array}{rcllll}
tprods & = & C & \rightarrow & x & F \rightarrow y \\
 & & D & \rightarrow & y & N_1 \rightarrow z \\
 & & E & \rightarrow & y & N_2 \rightarrow v
\end{array}
$$

$$
\begin{array}{rcllll}
neuprods & = & N_1 & \rightarrow & z & N_6 \rightarrow FF \\
 & & N_2 & \rightarrow & v & N_7 \rightarrow GN_8 \\
 & & N_3 & \rightarrow & CN_4 & N_8 \rightarrow FN_9 \\
 & & N_4 & \rightarrow & DN_5 & N_9 \rightarrow N_1N_2 \\
 & & N_5 & \rightarrow & DN_6 &
\end{array}
$$

2. *Die endgültige Ausgabe der Funktion chomsky, und somit die Chomsky-Normalform der Grammatik $\mathcal{G}$ aus Beispiel 5.2.8, ist dann die Konkatenation von der fertigenprods Liste und der tprods Liste:*

$$
\begin{array}{rcllll}
\mathcal{G}_{\mathcal{CHNF}} & = & N_1 & \rightarrow & z & N_9 \rightarrow N_1N_2 \\
 & & N_2 & \rightarrow & v & A \rightarrow CD \\
 & & N_3 & \rightarrow & CN_4 & B \rightarrow CD \\
 & & N_4 & \rightarrow & DN_5 & C \rightarrow x \\
 & & N_5 & \rightarrow & DN_6 & D \rightarrow y \\
 & & N_6 & \rightarrow & FF & E \rightarrow y \\
 & & N_7 & \rightarrow & GN_8 & F \rightarrow y \\
 & & N_8 & \rightarrow & FN_9 &
\end{array}
$$

5.2.3 Modul Plandowski2

Das Modul Plandowski2 beinhaltet die Hauptfunktion *plandowski* des Algorithmus von Plandowski. Den Algorithmus selbst haben wir bereits ausführlich in Kapitel 3 erklärt. Nun betrachten wir die interne Darstellung. Die erste Unterfunktion, die benötigt wird, ist die *wordLength* Funktion. Sie berechnet die Längen der Wörter, die von jedem Nonterminal der Grammatik gebildet werden. Das Wichtige hierbei ist es zu vermeiden, jedes einzelne Wort zu produzieren und dann Zeichen für Zeichen von links nach rechts zu durchlaufen. Diese naive Vorgehensweise kann eine exponentielle Laufzeit hervorbringen. Die Funktion *wordLength* stellt eine clevere Variante der Längenberechnung dar. Sie geht wie folgt vor:

1. Teile die Grammatik $\mathcal{G}$ in zwei Listen *mengeN* und *mengeT* auf, wobei in die Liste *mengeN* Produktionen der Form

   $$A \rightarrow B_1B_2, \ (A, B_1 \text{ und } B_2 \text{ sind Nonterminale})$$

   und in die Liste *mengeT* Produktionen der Form

   $$A \rightarrow x, \ (A \text{ ist ein Nonterminal und } x \text{ ein Terminalsymbol})$$

eingefügt werden. Die Längen von den Produktionen in der *mengeT* Liste sind bekannt, da sie nur ein Terminalsymbol produzieren. Also erzeugt jedes Nonterminal in dieser Liste ein Wort der Länge 1.

2. Ersetze die Vorkommen aller Nonterminale aus der *mengeT* Liste auf den rechten Seiten der Produktionen in der *mengeN* Liste durch die dazugehörigen Längen. Nachdem dies gemacht worden ist, muss mindestens von einem Nonterminal aus der *mengeN* Liste die komplette Länge bekannt sein. Wenn dies nicht so wäre, gäbe es Nonterminale, die keine Produktion besitzen, womit die Grammatik nicht terminieren würde. Füge die neuen Nonterminale, von denen die Längen komplett berechnet worden sind, in die *mengeT* Liste hinzu.

3. Führe Schritt 2 sukzessive so lange durch, bis von jedem Nonterminal aus der *mengeN* Liste die komplette Länge berechnet worden ist, also alle Nonterminale aus der *mengeN* Liste in die *mengeT* Liste übernommen werden sind, wodurch *mengeN*=[ ] wird.

Die Funktion *wordLength* sieht wie folgt aus:

```
1 wordLength :: [Prod] -> [ProdGen]
2 wordLength xs = let spl = split_in_two xs [] [] in
3                 snd (wordLengthCalc spl (snd spl))
```

Für den ersten Schritt ist die Funktion *split_in_two* zuständig. Der zweite Schritt wird von der Funktion *termsubst* durchgeführt, und für die Rekursion im dritten Schritt verwenden wir die Funktion *wordLengthCalc*:

```
1 wordLengthCalc :: ([ProdGen],[ProdGen]) -> [ProdGen] ->
2                   ([ProdGen],[ProdGen])
3
4 wordLengthCalc ([],ts_neu) ts = ([],ts_neu)
5
6 wordLengthCalc (ns,ts_neu) [] =
7                   wordLengthCalc (ns,ts_neu) ts_neu
8
9 wordLengthCalc (ns,ts_neu) (x:ts) =
10                  wordLengthCalc (termSubst ns x [] ts_neu) ts
```

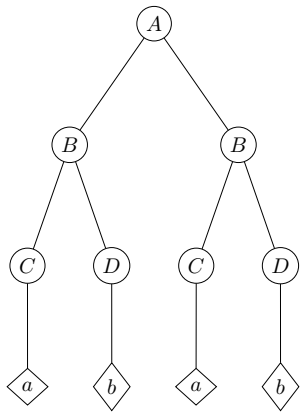
Die erste Eingabe ist das Tupel *(mengeN,mengeT)*, welches das Ausgabetupel von *split_in_two* ist. Als zweite Eingabe erhält sie nur *mengeT*. Dann übergibt sie der Funktion *termsubst* als erste Eingabe die Liste *mengeN*, als zweite Eingabe das erste Element aus der *mengeT* Liste, als dritte Eingabe eine leere Liste *ns_neu* und als vierte Eingabe die komplette *mengeT* Liste. Die Funktion *termsubst* durchläuft dann für ein Nonterminal, dessen komplette Länge schon bekannt ist (also für die zweite Eingabe), die ganze *mengeN* Liste (also die erste Eingabe) und setzt die Länge auf den rechten Seiten, in denen das Nonterminal vorkommt, ein. Hierbei werden alle Nonterminale aus der *mengeN* Liste (also der ersten Eingabe), deren Längen komplett berechnet worden sind, in die *mengeT* Liste (also in die vierte

Eingabe) aufgenommen. Die restlichen Nonterminale, von denen nichts oder nur ein Teil der Längen berechnet worden sind, werden in die *ns_ neu* Liste, die eine leere Liste war, hinzugefügt. Wenn alle Elemente der *mengeN* Liste überprüft worden sind, bricht die Funktion ab und liefert das Tupel *(ns_ neu,ts_ neu)* zurück, wobei *ts_ neu* alle Elemente der *mengeT* Liste von der Eingabe und die Nonterminale, deren Längen ganz berechnet worden sind, beinhaltet. Wenn die Liste *ns_ neu* leer zurückgeliefert wird, dann gibt es kein Nonterminal mehr, dessen Länge nicht berechnet worden ist. Die Funktion *wordLengthCalc* bricht in diesem Fall ab und liefert das Tupel *([ ],ts_ neu)* als Ergebnis. Die endgültige Ausgabe von *wordLength* ist die Liste *ts_ neu*, welche die Längen aller Nonterminale der Grammatik beinhaltet. Falls die Liste *ns_ neu* nicht leer zurückgeliefert wird, aber dafür alle Elemente der *mengeT* auf den rechten Seiten eingesetzt worden sind, ruft sich die Funktion *wordLengthCalc* rekursiv mit den Parametern *(ns_ neu,ts_ neu)* und *ts_ neu* auf, bis *termsubst* die Längen von allen Nonterminalen der Grammatik berechnet hat (also bis *termsubst* die *ns_ neu* Liste leer zurückliefert). Ein ausführliches Beispiel wurde im Abschnitt 5.2.1 (im Beispiel 5.2.7) durchgerechnet. Trotzdem ist es wichtig nochmal zu erwähnen, dass jede Produktion der Form $A \to x$ (wobei A ein Nonterminal und x ein Terminalsymbol ist) ein Blatt in einem Baum ist und damit die Länge der Wörter, die durch solche Nonterminale A erzeugt werden, gleich 1 ist. Danach setzt man die Längen dieser Blätter im Vaterknoten ein, erhält dadurch neue Nonterminale, deren Längen berechnet worden sind, setzt dann diese in ihrem Vaterknoten ein usw., bis man die Wurzel erreicht und ihre Länge berechnet hat. Die Berechnung findet von den Blättern zu der Wurzel statt (bottom-up). Gleiche Teilbäume werden dabei nicht zweimal berechnet. Dies ist der Schlüsselgedanke in der Funktion *wordLength*. Zur Verdeutlichung führen wir ein kleines Beispiel durch.

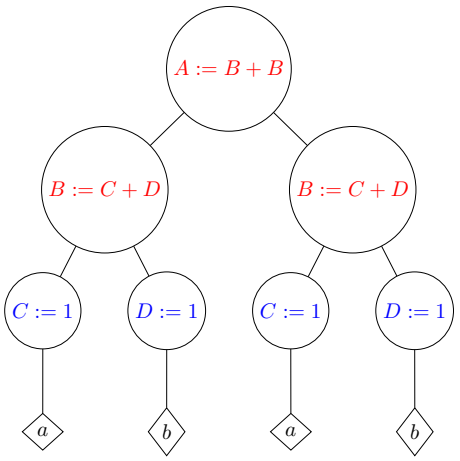
Beispiel 5.2.14 *Sei die Grammatik $\mathcal{G}$, von dessen Nonterminalen die Längen berechnet werden sollen, wie folgt gegeben:*

$$\begin{array}{rcl@{\qquad}rcl} \mathcal{G} & = & A \to BB & C & \to & a \\ & & B \to CD & D & \to & b. \end{array}$$

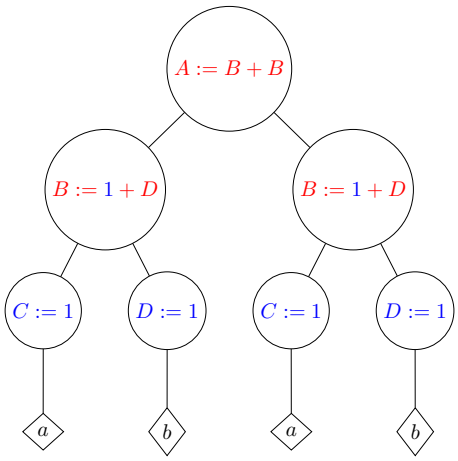
Der dazugehörige Baum ist:



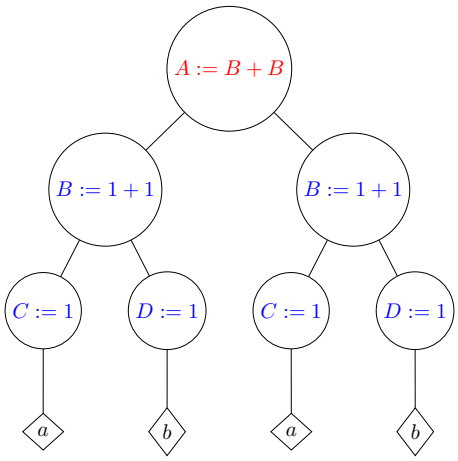
Alle Nonterminale deren Produktionen die Form $X \rightarrow y$ besitzen, wobei X ein Nonterminal und y ein Terminalsymbol ist, haben die Länge 1:



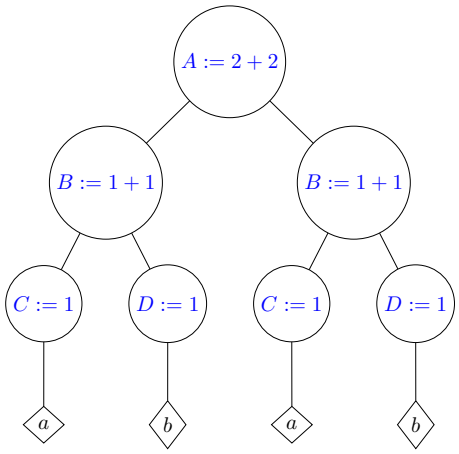
Setze die Länge des Nonterminals C auf allen rechten Seiten der Nonterminale, deren Längen noch nicht bekannt sind, ein:



Setze die Länge des Nonterminals D auf allen rechten Seiten der Nonterminale, deren Längen noch nicht bekannt sind, ein:



Die Länge des Nonterminals B ist nun komplett berechnet worden. Setze dessen Länge auf allen rechten Seiten der Nonterminale, deren Längen noch nicht bekannt sind, ein:



Die Länge jedes Nonterminals aus der Grammatik $\mathcal{G}$ ist berechnet worden. Daraus resultiert das Endergebnis:

$$\begin{array}{lllllll} A & = & 4 & \quad C & = & 1 \\ B & = & 2 & \quad D & = & 1. \end{array}$$

Nachdem die Längen von jedem Nonterminal berechnet worden sind, betrachtet der Algorithmus die eingegebene Testmenge. Die Testmenge ist eine Liste von Tupeln der Form (A, B), wobei A und B Nonterminale der eingegebenen Grammatik sind. Nun muss überprüft werden, ob die Nonterminale in den jeweiligen Tupeln die gleiche Länge besitzen. Hierfür benutzen wir die Funktion *compareLength*.

```
1 compareLength :: [ProdGen] -> [(Symbol,Symbol)]
2                -> [(Symbol,Symbol)]
3
4 compareLength lliste testmenge =
5           compareLength_sub lliste testmenge [] testmenge
6
7
8 compareLength_sub :: [ProdGen] -> [(Symbol,Symbol)]
9                   -> [(Symbol,Symbol)] -> [(Symbol,Symbol)]
10                  -> [(Symbol,Symbol)]
11
12 compareLength_sub (x:ns) ss ss_neu testmenge =
13     compareLength_sub ns (fst(substN x ss [] ss_neu))
14                          (snd(substN x ss [] ss_neu)) testmenge
15
16 compareLength_sub [] ss ss_neu testmenge =
17           if length(ss_neu) == length(testmenge)
18           then testmenge
19           else []
```

Die erste Eingabe von *compareLength* ist die Liste mit den Längen aller Nonterminale der Grammatik (also die Ausgabe von *wordLength*, nämlich

lliste) und die zweite Eingabe ist die zu überprüfende Testmenge. Die Idee dabei ist die folgende:

1. Nimm das erste Nonterminal mit dessen Länge aus der *lliste* (sei dieses Nonterminal A, mit der Länge $a = 3$).

2. Durchlaufe mit diesem Nonterminal die komplette Testmenge und setze die Länge in den Tupeln ein, in denen das Nonterminal vorkommt. Dabei können aus einem Tupel $(\texttt{N}\ b, \texttt{N}\ c)$ folgende Tupel entstehen:

   (a) Falls $(\texttt{N}\ b) = A$, aber $(\texttt{N}\ c) \neq A$: $(\texttt{I}\ a, \texttt{N}\ c)$, also $(3, \texttt{N}\ c)$.

   (b) Falls $(\texttt{N}\ b) \neq A$, aber $(\texttt{N}\ c) = A$: $(\texttt{N}\ b, \texttt{I}\ a)$, also $(\texttt{N}\ b, 3)$.

   (c) Falls $(\texttt{N}\ b) \neq A$, und $(\texttt{N}\ c) \neq A$: $(\texttt{N}\ b, \texttt{N}\ c)$.

   (d) Falls $(\texttt{N}\ b) = A$, und $(\texttt{N}\ c) = A$: $(\texttt{I}\ a, \texttt{I}\ a)$, also $(3, 3)$.

   Die Tupel in a), b) und c) sind noch nicht fertig bearbeitet worden und müssen nochmal zum Überprüfen weitergegeben werden. Das Tupel in d) ist komplett fertig und kann in die endgültige Ausgabe übernommen werden.

3. Wiederhole den zweiten Schritt für das nächste Element aus der *lliste*. Hierbei werden nur die Tupel in a), b) und c) überprüft.

Für die Schritte zwei und drei benutzen wir die Funktion *termsubst*. Die Rekursion findet mittels der Funktion *compareLength_ sub* statt. Sie übergibt die einzelnen Elemente der *lliste* an die Funktion *termsubst* und garantiert somit das von allen Tupeln in der Testmenge die Längen berechnet werden. Nach dem Durchlauf für ein Element nimmt *compareLength_ sub* die Ausgabeliste von *termsubst*, übergibt diese Liste und das nächste Element aus *lliste* als Eingabe an *termsubst*. Diese Rekursion wird so lange durchgeführt, bis *compareLength_ sub* alle Elemente aus *lliste* an *termsubst* übergeben hat, also bis alle Tupel aus zwei Zahlen bestehen. Falls es ein Tupel $(\texttt{I}\ a, \texttt{I}\ b)$ mit $a \neq b$ gibt, liefert *compareLength_ sub* (und somit auch *compareLength*) eine leere Liste als Ergebnis. Dies bedeutet, dass die Testmenge ungültig ist. Diese Entscheidung wird mit einem Längenvergleich von zwei Listen gefällt. Wenn alle Nonterminale in den Tupeln der Testmenge die gleiche Länge besitzen, kommen diese auch in der Ausgabeliste von *termsubst* vor, womit die Länge der Testmenge und der Ausgabeliste gleich sein muss. Falls es ein Tupel in der Testmenge gibt, dessen Nonterminale unterschiedliche Längen besitzen, wird es nicht in die Ausgabeliste von *termsubst* hinzugefügt, womit die Ausgabeliste mindestens ein Element weniger besitzt als die Testmenge. Das nächste Beispiel veranschaulicht die Vorgehensweise.

Beispiel 5.2.15 *Sei die folgende Grammatik $\mathcal{G}$ und die Testmenge als Eingabe gegeben:*

$$\mathcal{G} = \begin{array}{l} A \to BC \\ B \to b \\ C \to c \end{array} \qquad testmenge = [(A, C)]$$

Die *lliste* ist dann:

$$lliste = [(A[2]), (B[1]), (C[1])]$$

Angenommen *compareLength_ sub* übergibt $(A[2])$ als erstes Element aus der *lliste* an *termsubst*. Die Ausgabe von *termsubst* ist dann:

$$testmenge' = [(2, C)]$$

Sei nun $(B[1])$ die Eingabe für *termsubst*. Die Ausgabe sieht wie folgt aus:

$$testmenge'' = [(2, C)]$$

Die letzte Eingabe für *termsubst* ist $(C[1])$ und die dazugehörige Ausgabe:

$$testmenge''' = [\,]$$

Jetzt überprüfen wir, ob die Längen der Liste *testmenge* und der Liste $testmenge'''$ gleich sind:

$$(length\ testmenge = 1) \neq (length\ testmenge''' = 0)$$

Die Ausgabe von *compareLength* ist eine leere Liste.

Nachdem der Algorithmus die Ausgabe von *compareLength* erhalten hat, bricht er entweder ab und liefert False als Ergebnis (falls die Ausgabe von *compareLength* eine leere Liste ist), oder er führt die Operation *split* durch. Jedoch wird vorher die Funktion *prod_ length* benötigt. Die Aufgabe dieser Funktion ist es aus der Liste der Produktionen und der Längenliste eine Liste von Tupeln zu erzeugen, wobei jedes Tupel aus einer Produktion und der dazugehörigen Länge besteht. Betrachten wir die Grammatik aus Beispiel *compareLength*. Die Ausgabe der Funktion *prod_ length* wäre hierfür

$$lpliste = [(A \to BC, A[2]), (B \to b, B[1]), (C \to c, C[1])].$$

Dieses Format ist sehr nützlich für die Funktion *split*, denn sie benötigt sowohl die Produktion, als auch die Länge eines Nonterminals. Beides ist in einem Tupel der *lpliste* vorhanden. Nachdem die *lpliste* gebildet worden ist, wird sie der Funktion *split_ rek_ gen* übergeben, sowie die *lliste* und die Testmenge. Hier werden drei weitere Unterfunktionen aufgerufen. Die Funktion

qsort_ length sortiert die Tripel der *lpliste* in absteigender Reihenfolge anhand der Längen. Was noch für die Eingabe von *split* fehlt, ist die Menge rel_0. Sie wird mit der Funktion *createRel* aus der eingegebenen Testmenge gebildet, indem aus allen Tupeln (A, B) durch Hinzunahme der 0, Tripel $(A, B, 0)$ gebildet werden. Diese Tripel werden lexikographisch, mittels der Funktion *sort*, sortiert (zum Beispiel wird aus der Liste $[(B, B, 0), (B, A, 0)]$ nach der Sortierung die Liste $[(B, A, 0), (B, B, 0)]$). Die Ausgaben von *qsort_ length* und *sort* sowie die Testmenge werden an die Funktion *split_ rek* weitergegeben. Diese Funktion ist für die for-Schleife im Algorithmus zuständig. Sie übergibt die Elemente der absteigend sortierten *lpliste*, beginnend beim ersten, einzeln an die Funktion *split*. Wenn sie alle Elemente weitergegeben hat, liefert sie als Ergebnis die letzte Ausgabe von *split* zurück, welche wiederum die Ausgabe der Funktion *compactRel* repräsentiert. Die Aufgabe von *split* und wie das ganze genau funktioniert wurde im Abschnitt 3.1 beschrieben. Die Funktion überprüft die Tripel auf die gewünschte Eigenschaft der Regeln von der Operation *split*. Nach jedem *split* Aufruf, folgt ein direkter Aufruf von *compactRel*. Sie überprüft die Tripel auf weitere Eigenschaften der Operation $Compact(rel)$, die ebenfalls im Abschnitt 3.1 vorhanden sind. Während der Operation *split* können neue Tripel entstehen, die schon in der *rel* Menge vorhanden sind. Die naive Vorgehensweise wäre, vor jedem Einfügen der Tripel, die ganze Liste durchzugehen und zu überprüfen, ob das jeweilige Tripel schon vorhanden ist. Jedoch verfolgen wir einen anderen Ansatz. Da unsere erste *rel* Menge sortiert ist (rel_0), fügen wir die neu erzeugten Tripel an die richtige Stelle ein, womit alle *rel* Mengen immer sortiert sind. Falls an dieser Stelle ein gleiches Tripel vorhanden ist, wird das neu erzeugte Tripel nicht eingefügt. Für dieses sortierte Einfügen ist die Funktion *(>+>)* zuständig. Sie erwartet als erste und zweite Eingabe eine Liste, wobei die zweite Liste sortiert sein muss. Wenn dies der Fall ist, fügt sie die Elemente aus der ersten Liste an die richtigen Stellen in der zweiten Liste ein. Wenn das einzufügende Element in der sortierten Liste schon vorhanden ist, wird einfach mit dem nächsten Element der ersten Liste weitergemacht, ohne das aktuelle Element in die zweite Liste einzufügen. In der Implementierung ist die sortierte Liste die *rel* Menge und die einzufügende Liste die neu erzeugten Tripel. Die Sortierung der Tripel in den *rel* Mengen ist auch für die Funktion *compactRel* sehr hilfreich. Wenn zwei aufeinander folgende Tripel an den ersten zwei Stellen unterschiedliche Symbole besitzen, kann das erste Tripel unverändert in die Ausgabe übernommen werden, da es wegen der lexikographischen Ordnung in der kompletten Restliste keine zwei Tripel mit den gleichen Symbolen an den ersten zwei Stellen mehr geben kann. Das Gleiche gilt für drei aufeinander folgende Tripel. Falls die ersten zwei Tripel die gleichen Symbole an den entsprechenden Stellen besitzen, aber das dritte nicht, können die ersten zwei ohne weiteres in die Ausgabe übernommen werden. *Rel* Mengen mit nur einem Tripel oder nur zwei Tripeln können direkt ausgegeben werden. Desweiteren können Tripel, die mindestens ein Termi-

nalsymbol enthalten, unverändert in die Ausgabe übernommen werden.
Nach dem letzten Durchlauf von *compactRel* dürfen in den Tripeln der Ausgabemenge keine Nonterminale mehr vorhanden sein. Also müssen alle Tripel die Form (a, b, i) besitzen, wobei a, b Nonterminale und i ein Integer ist. Ausserdem darf es in der Ausgabe kein Suffix-Tripel $(a, b, 0)$, mit $a \neq b$ geben (also das nicht in der Relation $SUFSUB$ vorhanden ist). Anders ausgedrückt, müssen alle Tripel in der Ausgabe in der $SUFSUB$ Menge vorhanden sein. Diese zwei Eigenschaften überprüft die Funktion *equal*. Falls die Ausgabemenge diese Eigenschaften erfüllt, liefert die Funktion *equal* und somit auch die Hauptfunktion *plandowski* True, andernfalls False zurück. Die Hauptfunktion *plandowski* sieht wie folgt aus:

```
1 plandowski :: [Prod] -> [(Symbol,Symbol)] -> Bool
2 plandowski grammar set=
3     let
4         lliste  = wordLength grammar
5         lpliste = prod_length grammar lliste
6         split   = (split_rek_gen lliste set lpliste)
7     in
8         if (compareLength lliste set) == []
9         then False
10        else
11              if split == []
12              then True
13              else equal split
```

5.2.4 Modul PlandowskiProd

Das Module PlandowskiProd beinhaltet die Funktion *plandowskiProd*. Dies ist die Funktion für den Algorithmus von Plandowski auf dem Datentypen `Prod`. Sie importiert die Funktion *chomsky* aus dem Modul ChomskyProd und die Funktion *plandowski* aus dem Modul Plandowski2. Beide Funktionen wurden schon ausführlich besprochen. Die Funktion *plandowskiProd* sieht wie folgt aus:

```
1 plandowskiProd :: [Prod] -> [(Symbol,Symbol)] -> Bool
2 plandowskiProd x testset = let
3                                grammar = chomsky x 1
4                            in
5                            plandowski grammar testset
```

5.2.5 Modul Plandowski_STGProd

In diesem Modul kann der Algorithmus von Plandowski auf eine Singleton-Tree-Grammatik angewendet werden. Die erste Eingabe ist die komplette Grammatik und die zweite Eingabe die Testmenge. Die Testmenge besteht in diesem Fall aus Tupeln mit `STG_Pre_Symbol`'en. Die Hauptfunktion ist *plandowskiSTG_ Prod* und sieht wie folgt aus:

```
plandowskiSTG_Prod:: [STG_Prod] ->
                     [(STG_Pre_Symbol,STG_Pre_Symbol)] ->
                     Bool

plandowskiSTG_Prod stgprods set =
        let
            prod    = (stg_in_prod stgprods)
            chomsk  = (chomsky prod 1)
            testset = (presym_in_sym set)
        in
            plandowski chomsk testset
```

Wir benutzen wieder die Funktion *chomsky* aus dem Modul ChomskyProd und die Funktion *plandowski* aus dem Modul Plandowski2. Aber um diese Funktionen anwenden zu können, müssen deren Eingaben vom Typ `Prod` sein. Das bedeutet, dass wir den Typ `STG_Prod` in den Typ `Prod` umwandeln müssen. Dies erledigt die Funktion *stg_ in_ prod*. Sie geht wie folgt vor:

1. Entferne alle Produktionen von Kontext-Nonterminalen, die ein Loch produzieren und deren Vorkommen auf allen rechten Seiten der restlichen Produktionen. Wenn danach neue Kontext-Nonterminale entstehen, die ein Loch produzieren, wiederhole diesen Schritt für diese Kontext-Nonterminale. Gehe so lange rekursiv vor, bis es kein Kontext-Nonterminal mehr gibt, das ein Loch produziert. Zum Beispiel werden die Produktionen $A \rightarrow C_1B$, $B \rightarrow x$, $C_1 \rightarrow C_2C_2$ und $C_2 \rightarrow [\,]$ durch zwei neue Produktionen $A \rightarrow B$ und $B \rightarrow x$ ersetzt.

2. Wandle alle Produktionen vom Typ `STG_Prod` in Produktionen vom Typ `Prod` um, indem folgende Ersetzungen gemacht werden:

   (a) Ersetze die Produktion

   $$(\texttt{STG\_Prod}\ (\texttt{TN}\ a)\ [S_1, \ldots, S_n])$$

   durch die neue Produktion

   $$(\texttt{Produktion}\ (\texttt{N}\ (\text{“}TN - \text{“} + +a))\ [S'_1, \ldots, S'_n]),$$

   wobei die Liste $[S'_1, \ldots, S'_n]$ aus der Liste $[S_1, \ldots, S_n])$ wie folgt entsteht:

   i. Für alle S von 1 bis n:
   ii. Falls S_1 ein $(\texttt{TN}\ a)$ war, dann ist S'_1 vom Typ $(\texttt{N}\ (''TN - \text{“} + +a))$,
   iii. Falls S_1 ein $(\texttt{CN}\ a)$ war, dann ist S'_1 vom Typ $(\texttt{N}\ (''CN - \text{“} + +a))$,
   iv. Falls S_1 ein $(\texttt{F}\ a)$ war, dann ist S'_1 vom Typ $(\texttt{T}\ (''F - \text{“} + +a))$,
   v. Falls S_1 ein $(\texttt{K}\ a)$ war, dann ist S'_1 vom Typ $(\texttt{T}\ (''K - \text{“} + +a))$,

vi. Falls S_1 ein (`X` a) war, dann ist S'_1 vom Typ (`T` ($''X-$"$++a$)).

(b) Ersetze die Produktion

$$(\texttt{STG\_Prod}\ (\texttt{CN}\ a)\ [S_1, \ldots, S_n])$$

durch die neue Produktion

$$(\texttt{Produktion}\ (\texttt{N}\ (\text{“}CN - \text{“} + +a))\ [S'_1, \ldots, S'_n]).$$

Gehe wie im vorherigen Schritt vor um aus der Liste $[S_1, \ldots, S_n]$ die Liste $[S'_1, \ldots, S'_n]$ zu erhalten.

Für den ersten Schritt benutzen wir die Funktion *lochReduktion*. Diese Funktion teilt zuerst mit Hilfe der Funktion *split_ in_ tupel* die Produktionen in zwei Listen *temp1* und *temp2* auf, wobei in die *temp2* Liste nur Produktionen der Form

$$(\texttt{STG\_Prod}\ (\_)\ [\,])$$

hinzugefügt werden. Alle anderen Produktionen kommen in die Liste *temp1*. Nachdem sie die Produktionen aufgeteilt und die Loch-Produktionen in *temp2* hinzugefügt hat, durchläuft sie für eine Produktion aus *temp2* die komplette *temp1* Liste und löscht das dazugehörige Nonterminal aus allen rechten Seiten der Produktionen aus *temp1*, in denen dieses Nonterminal vorkommt. Dabei übergibt die Funktion *durchlaufeLoch* die Produktionen aus *temp2* nacheinander an die Funktion *durchlaufeProd*, für welche sie die *temp1* Liste durchläuft. Für das Durchlaufen der rechten Seite einer Produktion wird die Funktion *entferneProd* benutzt, die zusätzlich überprüft, ob nach dem Entfernen des jeweiligen Nonterminals noch andere Symbole auf der rechten Seite vorhanden sind. Falls nicht, so wird diese Produktion in die Liste *temp2* hinzugefügt und bei einem späteren Durchlauf aus den rechten Seiten der Produktionen entfernt. Der zweite Schritt wird von der Funktion *umwandeln_ sub* durchgeführt, wobei für die Unterschritte *i*) bis *vi*) die Funktion *umwandeln1* benutzt wird.
Nun wurde die komplette `STG_Prod` Grammatik in eine `Prod` Grammatik umgewandelt, ohne dabei die Informationen der originalen Wörter zu verlieren. Dies wird dadurch garantiert, indem die jeweiligen Konstruktoren der Nonterminale aus der `STG_Prod` Grammatik als Strings in die `Prod` Grammatik mitgenommen werden (im zweiten Schritt). Dies spezifiziert jedes Nonterminal und somit jede Produktion aus der `STG_Prod` Grammatik eindeutig in der `Prod` Grammatik. Wenn dies nicht gemacht wird, kann es zum Beispiel während der Eingabe der Testmenge [(`TN` a, `CN` a)] zu einem Fehler kommen, da aus beiden Nonterminalen ein (`N` a) in der `Prod` Grammatik entstehen würde.
Das einzige was noch gemacht werden muss, ist die Umwandlung der Testmenge, welche Tupel vom Typ `STG_Pre_Symbol` enthält. Hierfür verwenden

wir die Funktion *presym_ in_ sym*. Die Eingabe ist die Testmenge (eine Liste von (`STG_Pre_Symbol`,`STG_Pre_Symbol`)) und die Ausgabe ist die neue Testmenge, die Tupel vom Typ `Prod` enthält. Dabei geht sie wie folgt vor:

1. Wandle alle Tupel vom Typ `STG_Pre_Symbol` in Tupel vom Typ `Prod` um indem folgende Ersetzungen gemacht werden:

   (a) Falls das Tupel (`TN` a, `TN` b) ist, ersetze es durch das neue Tupel (`N` ($“TN-“++a$), `N` ($“TN-“++b$)),

   (b) Falls das Tupel (`TN` a, `CN` b) ist, ersetze es durch das neue Tupel (`N` ($“TN-“++a$), `N` ($“CN-“++b$)),

   (c) Falls das Tupel (`CN` a, `TN` b) ist, ersetze es durch das neue Tupel (`N` ($“CN-“++a$), `N` ($“TN-“++b$)),

   (d) Falls das Tupel (`CN` a, `CN` b) ist, ersetze es durch das neue Tupel (`N` ($“CN-“++a$), `N` ($“CN-“++b$)).

2. Wiederhole Schritt 1, bis alle Tupel umgewandelt worden sind und gib die neue Liste als Ergebnis zurück.

5.2.6 Modul PlandowskiSTG

Das Modul PlandowskiSTG beinhaltet die Funktion *plandowskiSTG* und sieht wie folgt aus:

```
1 plandowskiSTG :: STG -> [(String,String)] -> Bool
2 plandowskiSTG x set = let
3                           grammar = convertSTGtoSTG_Prod x
4                           testset = findpre set grammar
5                       in
6
7                       plandowskiSTG_Prod grammar testset
```

Dies ist die Funktion für den Algorithmus von Plandowski auf dem Datentypen `STG`. Sie importiert die Funktion *plandowskiSTG_ Prod* aus dem Modul PlandowskiSTG_Prod, den wir im vorherigen Abschnitt besprochen haben. Die Idee ist es, die `STG` Grammatik in eine Grammatik vom Typ `STG_Prod` umzuwandeln und den restlichen Ablauf dem Modul PlandowskiSTG_Prod zu überlassen. Die Umwandlung in den `STG_Prod` Typ übernimmt die Funktion *convertSTGtoSTG_ Prod* aus dem Modul Datentypen. Folgende Tabelle zeigt, aus welchen Produktionen der `STG` Grammatik die Produktionen der `STG_Prod` Grammatik werden:

STG	**STG_Prod**
(`AfA` a f [ ])	`STG_Prod` (`TN` a) [K f]
(`AfA` a f as)	`STG_Prod` (`TN` a) ((`F` f) : (map `TN` as))
(`ACA` $a1$ c $a2$)	`STG_Prod` (`TN` $a1$) [`CN` c, `TN` $a2$]
(`CLoch` c)	`STG_Prod` (`CN` c) [`Loch`]
(`CCC` $c1$ $c2$ $c3$)	`STG_Prod` (`CN` $c1$) [`CN` $c2$, `CN` $c3$]
(`CfAC` $c1$ f $as1$ $c2$ $as2$)	`STG_Prod` (`CN` $c1$) ((`F` f : (map `TN` $as1$)) ++ ((`CN` $c2$) : (map `TN` $as2$)))
(`Lam` $a1$ $a2$)	`STG_Prod` (`TN` $a1$) [`TN` $a2$]
(`Ax` a x)	`STG_Prod` (`TN` a) [`X` x]

Bevor *convertSTGtoSTG_Prod* diese Transformationen vornimmt, überprüft sie die Eingabegrammatik auf Korrektheit der Singleton-Tree-Grammatik Regeln. Die zuständige Funktion heisst *checkSTG*. Sie überprüft die Grammatik auf folgende Eigenschaften:

1. Alle Term-Nonterminale und Kontext-Nonterminale müssen unterschiedliche Bezeichner, bzw. Strings besitzen.
2. Alle Term-Nonterminale und Variablen müssen unterschiedliche Bezeichner, bzw. Strings besitzen.
3. Alle Term-Nonterminale und Funktionssymbole [7] müssen unterschiedliche Bezeichner, bzw. Strings besitzen.
4. Alle Kontext-Nonterminale und Variablen müssen unterschiedliche Bezeichner, bzw. Strings besitzen.
5. Alle Kontext-Nonterminale und Funktionssymbole müssen unterschiedliche Bezeichner, bzw. Strings besitzen.
6. Alle Variablen und Funktionssymbole müssen unterschiedliche Bezeichner, bzw. Strings besitzen.
7. Jedes Term-Nonterminal muss eine Produktion besitzen.
8. Jedes Kontext-Nonterminal muss eine Produktion besitzen.
9. Jedes Term-Nonterminal darf nicht mehr als eine Produktion besitzen.

[7] Konstanten sind Funktionssymbole mit der Stelligkeit 0

10. Jedes Kontext-Nonterminal darf nicht mehr als eine Produktion besitzen.

11. Jede Funktion muss eine feste Stelligkeit besitzen, d.h. sie muss an unterschiedlichen Stellen immer die selbe Anzahl an Argumenten enthalten, die der Stelligkeit entspricht.

Die Eigenschaften 1 bis 10 werden von den Funktionen *getSets*, *checkSets* und *mkChecks* überprüft. Die erste Funktion, die aufgerufen wird, ist *getSets*. Sie bekommt die `STG` Grammatik als Eingabe und liefert ein Tupel mit fünf Listen zurück, die jeweils *termNT, contextNT, funsym, vars* und *defnts* heissen. In die *termNT* Liste kommen alle Strings von den Term-Nonterminalen, die in den Produktionen der Grammatik vorkommen. Die *contextNT* Liste hingegen beinhaltet alle Strings der Kontext-Nonterminale der Grammatik. Alle Strings der Funktionssymbole kommen in die *funsym* Liste, während die Strings von den Variablen in die *vars* Liste hinzugefügt werden. Die Strings von allen linken Seiten der Produktionen kommen in die *defnts* Liste, die ausschließlich Term-Nonterminale und Kontext-Nonterminale sein können. Hierbei können in diesen Listen Strings doppelt vorkommen, denn *getSets* überprüft jedes Symbol in den Produktionen einzeln. Da zum Beispiel ein Term-Nonterminal auf der rechten Seite von mehreren Produktionen auftauchen kann, wird es in diesem Fall mehrmals in die *termNT* Liste hinzugefügt. Dies gilt auch für die restlichen Listen, wobei es in der *defnts* Liste normalerweise nicht passieren darf. Darauf gehen wir später genauer ein. Nun folgt ein kleines Beispiel zur Verdeutlichung:

Beispiel 5.2.16 *Sei die* `STG` *Grammatik wie folgt:*

```
1 gstg :: STG
2 gstg = STG [AfA    "A_t" "g" ["B","A"],
3             AfA    "A_s" "g" ["A","A"],
4             ACA    "A" "C_2" "A'",
5             AfA    "A'" "a" [],
6             ACA    "B" "C_1" "B'",
7             Ax     "B'" "x",
8             CCC    "C_2" "C_1" "C_1",
9             CCC    "C_1" "C_0" "C_0",
10            CfAC   "C_0" "f" [] "C'" [],
11            CLoch  "C'"]
```

Die Ausgabe der Funktion getSets ist:

$$termNT = [\text{“}A_t\text{“}, \text{“}B\text{“}, \text{“}A\text{“}, \text{“}A_s\text{“}, \text{“}A\text{“}, \text{“}A\text{“}, \text{“}A\text{“}, \text{“}A'\text{“}, \text{“}B\text{“}, \text{“}B'\text{“}, \text{“}B'\text{“}],$$

$$contextNT = [\text{“}C_2\text{“}, \text{“}C_1\text{“}, \text{“}C_2\text{“}, \text{“}C_1\text{“}, \text{“}C_1\text{“}, \text{“}C_1\text{“}, \text{“}C_0\text{“}, \text{“}C_0\text{“}, \text{“}C_0\text{“}, \text{“}C'\text{“}, \text{“}C'\text{“}],$$

$$funsym = [\text{“}g\text{“}, \text{“}g\text{“}, \text{“}a\text{“}, \text{“}f\text{“}],$$

$$vars = [\text{“}x\text{“}],$$

$$defnts = [\text{“}A_t\text{“}, \text{“}A_s\text{“}, \text{“}A\text{“}, \text{“}A'\text{“}, \text{“}B\text{“}, \text{“}B'\text{“}, \text{“}C_2\text{“}, \text{“}C_1\text{“}, \text{“}C_0\text{“}, \text{“}C'\text{“}].$$

Diese Listen übergibt *getSets* der Funktion *checkSets*. Hier werden aus den Listen *termNT, contextNT, funsym* und *vars*, die mehrmals vorkommenden Elemente entfernt, womit die neuen Listen *tntStrings, cntStrings, funStrings* und *varStrings* entstehen. Ausserdem wird die Liste *defnts* in zwei neue Listen *defTNTs* und *defCNTs* aufgeteilt, wobei in der *defTNTs* Liste alle Strings der Term-Nonterminalen und in der *defCNTs* Liste alle Strings der Kontext-Nonterminalen vorhanden sind. Die Fortsetzung vom Beispiel 5.2.16 ergibt:

$$tntStrings = [\text{“}A_t\text{“}, \text{“}B\text{“}, \text{“}A\text{“}, \text{“}A_s\text{“}, \text{“}A'\text{“}, \text{“}B'\text{“}],$$

$$cntStrings = [\text{“}C_2\text{“}, \text{“}C_1\text{“}, \text{“}C_0\text{“}, \text{“}C'\text{“}],$$

$$funStrings = [\text{“}g\text{“}, \text{“}a\text{“}, \text{“}f\text{“}],$$

$$varStrings = [\text{“}x\text{“}],$$

$$defTNTs = [\text{“}A_t\text{“}, \text{“}A_s\text{“}, \text{“}A\text{“}, \text{“}A'\text{“}, \text{“}B\text{“}, \text{“}B'\text{“}],$$

$$defCNTs = [\text{“}C_2\text{“}, \text{“}C_1\text{“}, \text{“}C_0\text{“}, \text{“}C'\text{“}].$$

Nachdem die neuen Listen aufgestellt worden sind, werden folgende Tupel erzeugt:

- $(tntStrings, cntStrings)$,
- $(tntStrings, varStrings)$,
- $(tntStrings, funStrings)$,
- $(cntStrings, varStrings)$,
- $(cntStrings, funStrings)$ und
- $(varStrings, funStrings)$.

Jetzt wird überprüft, ob jeweils der Schnitt der ersten und der zweiten Liste in einem Tupel leer ist. Diese Überprüfungen spiegeln die Eigenschaften 1 bis 6 wieder. Unserem Beispiel ergibt

$$tntStrings \cap cntStrings = \emptyset,$$

$$tntStrings \cap varStrings = \emptyset,$$

$$tntStrings \cap funStrings = \emptyset,$$

$$cntStrings \cap varStrings = \emptyset,$$

$$cntStrings \cap funStrings = \emptyset,$$

$$varStrings \cap funStrings = \emptyset,$$

womit die ersten 6 Eigenschaften erfüllt sind. Als nächstes überprüfen wir, ob jedes Term-Nonterminal eine Produktion besitzt. Hierfür betrachten wir

die Listen *tntStrings* und *defTNTs*, wobei jedes String aus der *tntStrings* Liste in der *defTNTs* Liste vorhanden sein muss, da *defTNTs* alle Term-Nonterminale beinhaltet, die eine Produktion besitzen. Falls ein String in der *tntStrings* Liste existiert, der nicht in der *defTNTs* Liste vorhanden ist, bedeutet dies, dass es ein Term-Nonterminal gibt, das nichts produziert (Eigenschaft 7). In unserem Beispiel ist jedes Element aus *tntStrings* auch in *defTNTs* enthalten. Die Listen *cntStrings* und *defCNTs* müssen ebenfalls überprüft werden, da jedes in der Grammatik vorhandene Kontext-Nonterminal eine Produktion besitzen muss. Die *defCNTs* Liste beinhaltet alle Kontext-Nonterminale, die eine Produktion haben und die *cntStrings* Liste beinhaltet alle Kontext-Nonterminale, die in der Grammatik auftauchen. Alle Kontext-Nonterminale, die in unserem Beispiel in der Grammatik vorhanden sind, besitzen auch eine Produktion, womit die Eigenschaften 7 und 8 erfüllt sind. Wie schon vorhin erwähnt, sollten in den Listen *defTNTs* und *defCNTs* keine doppelten Elemente vorhanden sein, da sowohl jedes Term-Nonterminal, als auch jedes Kontext-Nonterminal nur eine Produktion besitzen darf. Um festzustellen, ob dies tatsächlich der Fall ist, vergleichen wir die Länge der jeweiligen Liste vor dem Eliminieren und nach dem Eliminieren mehrmals vorhandener Elemente. Es ist leicht zu erkennen, dass die Längen vorher und nachher identisch sein müssen, da sonst mindestens ein Term-Nonterminal bzw. Kontext-Nonterminal zweimal vorhanden ist, womit dieses Nonterminal zwei Produktionen besitzt (Bsp.: Angenommen ein beliebiges Nonterminal N_1 hat zwei Produktionen: $length(defTNTs = [N_1, N_1, N_2]) = 3 > length(defTNTs = [N_1, N_2]) = 2$(nach Elimination mehrmals vorkommender Elemente)). In unserem Beispiel sind die Längen von den Listen *defTNTs* bzw. *defCNTs* vor der Eliminierung und nach der Eliminierung mehrmals vorhandener Elemente gleich, womit die Eigenschaften 9 und 10 erfüllt sind.
Jetzt muss noch die letzte Eigenschaft überprüft werden. Dafür benutzen wir die Funktion *checkFnArities*, die als Eingabe die `STG` Grammatik erhält. Sie betrachtet nur Produktionen der Form

$$\texttt{AfA}\ a\ f\ as$$

und

$$\texttt{CfAC}\ c1\ f\ as1\ c2\ as2,$$

da die Funktionssymbole nur in diesen Produktionen auftauchen. Ausserdem füllt die Funktion eine leere Liste *arities* mit Tupeln (f, n) (wobei f ein Funktionssymbol und n die dazugehörige Stelligkeit ist) wie folgt:

- Wenn die Produktion die Form `AfA` $a\ f\ as$ besitzt, überprüfe, ob ein Tupel mit f an erser Stelle in der *arities* Liste existiert. Wenn dies zutrifft, mache mit Schritt 1 weiter, sonst mit Schritt 2.

1. Gib das zweite Element n, also die Stelligkeit von f aus und vergleiche, ob $length\ as = n$ gilt. Falls $length\ as \neq n$, dann besitzt die gleiche Funktion f zwei verschiedene Stelligkeiten, womit *checkFnArities* abbricht und eine Fehlermeldung liefert. Sonst mache mit der nächsten Produktion weiter.
2. Füge das Tupel $(f, length\ as)$ in die *arities* Liste ein und mache mit der nächsten Produktion weiter.

- Wenn die Produktion die Form `CfAC` $c1\ f\ as1\ c2\ as2$ besitzt, überprüfe, ob ein Tupel mit f an erser Stelle in der *arities* Liste existiert. Wenn dies zutrifft, mache mit Schritt 1 weiter, sonst mit Schritt 2.

  1. Gib das zweite Element n, also die Stelligkeit von f aus und vergleiche, ob $(length\ as1) + (length\ as2) + 1 = n$ gilt. Falls $(length\ as1) + (length\ as2) + 1 \neq n$, besitzt die gleiche Funktion f zwei verschiedene Stelligkeiten, womit *checkFnArities* abbricht und eine Fehlermeldung liefert. Sonst mache mit der nächsten Produktion weiter.
  2. Füge das Tupel $(f, (length\ as1) + (length\ as2) + 1)$ in die *arities* Liste ein und mache mit der nächsten Produktion weiter (die 1 wird wegen dem Kontext-Nonterminal auf der rechten Seite dazuaddiert).

Jedes mal, wenn eine neue Funktion f gefunden wird, überprüft *checkFnArities* zuerst, ob f in der *arities* Liste vorhanden ist. Sie bricht ab wenn alle Produktionen der Grammatik überprüft worden sind und liefert True als Ergebnis zurück. In unserem Beispiel kommt die Funktion g zweimal vor ((`AfA` “A_t“ “g“ [“B“, “A“])und(`AfA` “A_s“ “g“ [“A“, “A“])) und besitzt bei beiden die gleiche Stelligkeit. Die Funktionen f und a kommen jeweils nur einmal vor, womit die Grammatik auch die 11te Eigenschaft erfüllt. Erst wenn alle Eigenschaften erfüllt sind, wird die `STG` Grammatik in eine `STG_Prod` Grammatik umgewandelt.
Nachdem die Grammatik dem gewünschten Datentyp entspricht, muss noch die Testmenge umgewandelt werden. Das Problem hierbei ist, dass die Tupel der Testmenge nur aus Strings bestehen, von denen wir keine weiteren Informationen besitzen. Das bedeutet, wir wissen nicht, ob es sich bei den Strings in den Tupeln um ein Term-Nonterminal oder ein Kontext-Nonterminal handelt. Dies kann mit der Funktion *findpre* herausgefunden werden. Die Eingabe von *findpre* ist die Testmenge und die umgewandelte Grammatik. Sie geht wie folgt vor:

1. Durchlaufe für ein Tupel (a, b) der Testmenge die komplette `STG_Prod` Grammatik. Setze die Zähler za für a und zb für b auf 0. Erzeuge eine leere Liste *neup*.

2. Sei die erste Produktion aus der `STG_Prod` Liste (`STG_Prod` (`TN` s_1) d) (oder (`STG_Prod` (`CN` s_1) d)):

   (a) Wenn *za* und *zb* gleich 1 sind, mache aus der Liste *neup* ein Liste mit einem Tupel. Sonst gehe zu b).

   (b) Wenn *za*= 0 und *zb*= 1, überprüfe, ob $a = s_1$. Wenn das zutrifft, erhöhe *za* auf 1 und füge `TN` a in die *neup* Liste hinzu. Nimm die nächste Produktion und beginne mit Schritt 2. Sonst gehe zu c).

   (c) Wenn *za*= 1 und *zb*= 0, überprüfe, ob $b = s_1$. Wenn das zutrifft, erhöhe *zb* auf 1 und füge `TN` b in die *neup* Liste hinzu. Nimm die nächste Produktion und beginne mit Schritt 2. Sonst gehe zu d).

   (d) Wenn $a = s_1$ und $b = s_1$ gib das Ergebnis [(`TN` a, `TN` b)] zurück und beginne mit dem nächsten Tupel aus der Testmenge bei Schritt 1.

   (e) Wenn $a = s_1$ und $b \neq s_1$, erhöhe *za* auf 1 und füge `TN` a in die Liste *neup* hinzu. Nimm die nächste Produktion und beginne mit Schritt 2.

   (f) Wenn $a \neq s_1$ und $b \neq s_1$, erhöhe *zb* auf 1 und füge `TN` b in die Liste *neup* hinzu. Nimm die nächste Produktion und beginne mit Schritt 2.

Die Idee mit der Umwandlung von *neup* in eine Liste mit nur einem Tupel funktioniert, da *neup* höchstens zwei Elemente besitzt und die Umwandlung nach dem Hinzufügen des zweiten Elements geschieht (da sowohl *za*, als auch *zb* gleich 1 ist). Dafür ist die Funktion *machtupel* zuständig. Für den ersten Schritt benutzen wir die Funktion *findpre_ sub* und für den zweiten Schritt die Funktion *findpre_ sub'*. Das Ergebnis von *findpre* ist die neue Testmenge mit den Tupeln vom Typ `STG_Pre_Symbol`. Jetzt kann der Algorithmus von Plandowski durchgeführt werden.

5.2.7 Modul PlandowskiUnif

Dieses Modul beinhaltet den Algorithmus von Plandowski, der im Unifikationsalgorithmus verwendet wird. Die Funktion heisst *plandowski* und sieht wie folgt aus:

```
1 plandowski :: [Prod] -> [Prod] -> [ProdGen]
2            -> [(Symbol,Symbol)] -> Bool
3 plandowski grammar prodohnelen prodmitlen set =
4     let
5         lliste  = wordLength prodohnelen prodmitlen
6         lpliste = prod_length grammar allelen
7         allelen = lliste++prodmitlen
8         split   = (split_rek_gen allelen set lpliste)
9     in
10        if (compareLength allelen set) == []
```

```
11          then False
12          else
13               if split == []
14               then True
15               else equal split
```

Alle Funktionen und Vorgehensweisen sind die Gleichen, die im Modul Plandowski2 besprochen worden sind. Es gibt nur zwei kleine Unterschiede. Der erste Unterschied ist leicht an der Eingabe der Funktion *plandowski* zu sehen. Es sind zwei Eingaben mehr geworden, nämlich die Liste *prodohnelen* vom Typ `Prod` und die Liste *prodmitlen* vom Typ `ProdGen`. Die Idee steckt am Anfang des Unifikationsalgorithmus. Hier werden nämlich von allen Nonterminalen der Eingabegrammatik die einzelnen Längen schon berechnet. Diese müssen in den Aufrufen von *plandowski* nicht nochmal berechnet werden und stellen deswegen mit ihren Längen die *prodmitlen* Liste dar. Die einzigen Nonterminale, von denen die Längen berechnet werden müssen, sind die neu erzeugten Nonterminale, die während der Umwandlung in die Chomsky-Normalform entstehen. Diese werden im Verlauf der Funktion *chomsky* von den restlichen Nonterminalen getrennt und in die Liste *neuprods* hinzugefügt, die wir im Modul ChomskyProd besprochen haben. Diese Liste stellt die *prodohnlelen* Liste dar.
Der zweite Unterschied liegt in der Eingabe von der Funktion *wordLength*. Sie erhält statt der ganzen Grammatik, die Listen *prodohnelen* und *prodmitlen*. Hiermit vermeiden wir den Durchlauf der *prodmitlen* Liste innerhalb der Unterfunktion *split_in_two*. Diese Liste wird an die zweite Liste des Ausgabetupels von *split_in_two* konkateniert. Der restliche Ablauf ist wie im Modul Plandowski2 beschrieben.

5.2.8 Modul Chomsky

Die Funktion *chomsky* in diesem Modul wird während der Ausführung des Unifikationsalgorithmus benutzt und sieht wie folgt aus:

```
1 chomsky :: [ProdPre] -> Integer -> ([Prod],[Prod])
2 chomsky prod zaehler = let chs = chomsky_sub prod zaehler in
3                        (first chs ++ second chs,thrd chs)
4
5 chomsky_sub :: [ProdPre] -> Integer -> ([Prod],[Prod],[Prod])
6 chomsky_sub prod zaehler = let cnf2 = (cnf_reg_2 prod) in
7                            if thrd cnf2 == []
8                            then cnf_reg_3 cnf2 [] zaehler
9                            else
10                           cnf_reg_2_5 cnf2 [] [] zaehler
```

Die ersten Unterschiede zum Modul ChomskyProd liegen im Eingabetyp und der Ausgabe der Funktionen. Die Eingabe für das Modul ChomskyProd ist eine `STG_Prod` Grammatik und die Eingabe für das Modul Chomsky ist eine `ProdPre` Grammatik. Während im Modul ChomskyProd die Ausgabe nur

die Grammatik in Chomsky-Normalform ist, wird im Modul Chomsky ein Tupel von zwei Listen ausgegeben, wobei die erste Liste wieder die Grammatik in Chomsky-Normalform ist und die zweite Liste, die während dem Durchlauf in den Regeln *cnf_ reg_ 2_ 5* und *cnf_ reg_ 3* neu erzeugte Produktionen besitzt. Der Unterschied liegt in der Verwendung der *neuprods* Liste. Ein weiterer Unterschied ist, dass nach der Eingabe der `ProdPre` Grammatik die Lambda-Produktionen entfernt werden. Hierfür verwenden wir die Funktion *epsilonReduktion*. Das Prinzip dieser Funktion ist das Gleiche, wie das Prinzip der Funktion *lochReduktion* aus dem Modul PlandowskiSTG_Prod und wird deswegen nicht weiter erläutert. Nach der Eliminierung der Lambda-Produktionen wird die `ProdPre` Grammatik in eine `Prod` Grammatik umgewandelt. Dies erledigt die Funktion *umwandeln*, dessen Vorgehensweise die Gleiche ist, wie die der Funktion *umwandeln_ sub* aus dem Modul PlandowskiSTG_Prod. Deswegen gehen wir nur auf die Umwandlung der `ProdPre` Produktionen ein.

1. Ersetze die Produktion

   $$(\texttt{ProdPreOrder}\ (\texttt{P}\ a)\ [S_1, \ldots, S_n])$$

   durch die neue Produktion

   $$(\texttt{Produktion}\ (\texttt{N}\ (\text{“}P\text{“} + + a))\ [S'_1, \ldots, S'_n]),$$

   wobei die Liste $[S'_1, \ldots, S'_n]$ aus der Liste $[S_1, \ldots, S_n]$) wie folgt entsteht:

   (a) Für alle S von 1 bis n:
   (b) Falls S_1 ein (`P` a) war, dann ist S'_1 vom Typ (`N` (“P“ $+ +a$)),
   (c) Falls S_1 ein (`L` a) war, dann ist S'_1 vom Typ (`N` (“L“ $+ +a$)),
   (d) Falls S_1 ein (`R` a) war, dann ist S'_1 vom Typ (`N` (“R“ $+ +a$)),
   (e) Falls S_1 ein (`F` a) war, dann ist S'_1 vom Typ (`T` (“F“ $+ +a$)),
   (f) Falls S_1 ein (`K` a) war, dann ist S'_1 vom Typ (`T` (“K“ $+ +a$)),
   (g) Falls S_1 ein (`X` a) war, dann ist S'_1 vom Typ (`T` (“X“ $+ +a$)).

2. Ersetze die Produktion

   $$(\texttt{ProdPreOrder}\ (\texttt{L}\ a)\ [S_1, \ldots, S_n])$$

   durch die neue Produktion

   $$(\texttt{Produktion}\ (\texttt{N}\ (\text{“}L\text{“} + + a))\ [S'_1, \ldots, S'_n]).$$

   Gehe wie im vorherigen Schritt vor, um aus der Liste $[S_1, \ldots, S_n]$ die Liste $[S'_1, \ldots, S'_n]$ zu erhalten.

3. Ersetze die Produktion

$$(\texttt{ProdPreOrder}\ (\texttt{R}\ a)\ [S_1, \ldots, S_n])$$

durch die neue Produktion

$$(\texttt{Produktion}\ (\texttt{N}\ (\text{“}R\text{“} + + a))\ [S'_1, \ldots, S'_n]).$$

Gehe wie im Schritt 1 vor, um aus der Liste $[S_1, \ldots, S_n]$ die Liste $[S'_1, \ldots, S'_n]$ zu erhalten.

Der komplette restliche Ablauf ist wie der im Modul ChomskyProd beschriebene.

5.2.9 Modul Index

Wir fangen mit dem Modul Index.Index an. Wie der Name es andeutet, beinhaltet dieses Modul die index-Funktion.

Listing 5.12: Eingabeparameter

```
index :: ProdPre->ProdPre->[ProdPre]->Integer->[ProdPre]->Integer
index x y prods zaehler lliste =
```

Die Eingabeparameter x und y sind die `ProdPre`-Typen von den jeweils zu unifizierenden Nonterminalen. Die Liste der Produktionen *prods* ist ebenfalls vom Typ `ProdPre`. Die dritte Eingabe *zaehler* ist vom Typ *Integer*. Dieser Zähler spielt bei der Bildung von Präfixen und Suffixen die Rolle der Namensgebung. Damit werden doppelte Namensgebungen für neu entstandene Nonterminale verhindert, um Berechnungsfehler im Programm vorzubeugen. Bei Namensgebungen wird mit der Anweisung *show zaehler* der String an den Namen konkateniert. Die `ProdPre`-Darstellung der Terme brauchen wir für das Auffinden der ersten unterschiedlichen Stelle. Zusätzlich benötigen wir die Längen der einzelnen Nonterminale, die allerdings in `ProdPre`-Nonterminale transformiert wurden (s. Transformationregeln in der Abbildung 4.1.1).

```
getlx          = (getLength_prod x lliste)
getly          = (getLength_prod y lliste)
```

Die Funktion *getLength_prod x lliste* durchläuft in linearer Zeit die Liste jeweils für x und y und speichert die Längen in *getlx* und *getly* ab.

```
let
pr = (ProdPreOrder (P "s") [])) in

prefix_x=traverse_prefix (x) (prods++[x,y]) (m)
                         (zaehler) pr lliste 0

prefix_y=traverse_prefix (y) (prods++[x,y]) (m)
                         (zaehler+1) pr lliste 0

```

```
10 suffix_x=traverse_suffix (x) (prods++[x,y]) (getlx - m)
11                          (zaehler) pr lliste 0
12
13 suffix_y=traverse_suffix (y) (prods++[x,y]) (getly - m)
14                          (zaehler+1) pr lliste 0
```

An dieser Stelle werden jeweils die Suffixe oder Präfixe gebildet, um später die Berechnung entweder auf die Präfixe oder Suffixe weiterzuführen. Das Prinzip der Präfix- bzw Suffixbildung wurde im Kapitel 4.5 erklärt. Wir beschreiben hier die Implementierung nur für *traverse_prefix*.

```
1 let
2  leer  = (ProdPreOrder p [])
3  füll  = (ProdPreOrder (P h) liste)
4  nleer = (ProdPreOrder p (x:xs))
5  rück  = (ProdPreOrder (P (show zaehler)) (liste))
6 in
7
8 traverse_prefix :: ProdPre->[ProdPre]->Integer->Integer->
9                    ProdPre->[ProdPre]->Integer->ProdPre
10
11 traverse_prefix leer  prods m zaehler füll lliste lprefix = rück
12 traverse_prefix nleer prods m zaehler füll lliste lprefix =
13          let summe  = lerste + lprefix
14              lerste = (getLength x lliste)
15          in
16          if lerste == 0
17          then
18          traverse_prefix (ProdPreOrder p xs) prods m zaehler
19                           rück lliste lprefix
20          else
21          if summe == m
22          then (ProdPreOrder (P (show zaehler)) (liste++[x]))
23          else
24          if summe < m
25          then
26
27          traverse_prefix (ProdPreOrder p xs) prods m zaehler
28          (ProdPreOrder (P (show zaehler)) (liste++[x])) lliste
29          (lprefix+lerste)
30          else
31
32          traverse_prefix (getProd x prods) prods m zaehler
33          (ProdPreOrder (P (show zaehler)) liste) lliste lprefix
```

Die erste Eingabe *nleer* ist das Nonterminal, dessen Präfix der Länge m berechnet werden soll. In der letzten Eingabe *lprefix* wird die Länge des Präfixes schrittweise aufgebaut, entsprechend den Verzweigungen wird *lprefix* ,entweder in Zeile 18 und 31 nicht aktualisiert übergeben, oder in Zeile 27 aktualisiert übergeben. In Zeile 16 werden die einzelnen Elemente auf ihre Länge überprüft. Falls sie 0 ist, wird die Funktion auf des nächste Element des Nonterminals aufgerufen. In Zeile 20 ist $summe = m$, somit kann die Berechung beendet und das Präfix zurückgegeben werden. Solange in Zeile

23 die Bedingung zutrifft, addiert man *lerste* zu *lprefix*, konkateniert das x (bei Präfixen wird von rechts konkateniert) und übergibt dem nächsten Aufruf das aktualisierte *lprefix*. Falls die Bedingung nicht zutrifft, war *summe* zu groß, und mit *getProd x* findet der Aufruf mit einem neuen Nonterminal statt, der auf der rechten Seite von dem ersten Nonterminal steht. Man muss an dieser Stelle mit dem neuen Nonterminal weitermachen, weil die Länge überschritten wurde. In Zeile 11 ist die rechte Seite des übergebenen Nonterminals zu Ende durchlaufen worden, und die Rückgabe ist das Präfix *rück*, dessen rechte Seite *liste* die Länge m hat. Um die Funktion *traverse_suffix* zu erhalten, konkateniert man in Zeile 27 von links, und ruft in Zeile 30 die Funktion *getProd x* auf das letzte Element auf. Weiterhin wird in *index* die Funktion *chomsky* aufgerufen.

```
1 chomsk = (chomsky ([prefix_x]++[prefix_y]++prods) (zaehler+2))
```

Die Funktion *chomsky* wurde im Abscnitt 5.2.8 besprochen. Chomskys Ausgabe benötigen wir für die Funktion *plandowski*, die im Abschnitt 5.2.7 besprochen wurde. Bemerkenswert ist hier, dass der Zähler bei jedem Aufruf um eins höher ist, im Vergleich zu den übergebenen Zählerwerten in den Suffix- und Präfixfunktionen. Diese Maßnahme dient dem Vermeiden von doppelten Namensgebungen. Die Ausgabe von *chomsky* ist ein Tupel mit zwei Listen, wobei in der ersten Liste Produktionen enthalten sind, von denen die Längen bekannt sind. Die zweite Liste beinhaltet neu hinzugekommene Produktionen, deren Länge unbekannt sind. Das ganze hat den Vorteil, dass beim Aufruf von *plandowski* die Übergabe der Produktionen mit und ohne Länge geschieht.

```
1 let
2  testset   = [(N ("P"++show zaehler), N ("P"++show (zaehler+1)))]
3  prelängen= (prodint++prodgenprefix)
4  mitlen    = (fst chomsk)
5  ohnlen    = (snd chomsk)
6 in
7
8 plandows = plandowski mitlen ohnlen prelängen testset
```

Dies verkürzt die Berechnungszeit der Längen, denn die Längen der Produktionen in der Liste *mitlen* müssen nicht nochmal berechnet werden. Eine weitere wichtige Funktion, die in *index* aufgerufen wird, ist die Funktion *pruefrechts*.

```
1 pruefrechts:: ProdPre -> ProdPre -> Bool
2 pruefrechts (ProdPreOrder a b) (ProdPreOrder c d) =
3              if b == d
4              then True
5              else False
```

Ihre Aufgabe besteht darin, für zwei gegebene Nonterminale zu überprüfen, ob die rechten Seiten gleich sind. Durch Hinzunahme dieser Unterfunktion werden eventuelle *plandowski* Aufrufe gespart. Zum Beispiel kann bei der

Unifikation die unterschiedliche Stelle am Ende der beiden Termen existieren. Dies hat zu Folge, dass die gebildeten Präfixe überwiegend gleich sind. Anstatt die polynomielle Laufzeit von *plandowski* abzuwarten, würde ein linearer Durchlauf die Gleichheit feststellen, und einen Beitrag zur Beschleunigung der Laufzeit leisten. Der hier gezeigte Quellcode von *index* stimmt in der Reihenfolge nicht mit unserer Implementierung überein. An dieser Stelle kommt die Hauptaufgabe der *index* Funktion. In Zeile 10 prüft *pruefR* ob die Präfixe gleich sind.

```
1 let
2     w        = (min getlx getly)
3     m        = (div w 2)
4     preläng  = (lliste++[lenprefix_x,lenprefix_y])
5     sufläng  = (lliste++[lensuffix_x,lensuffix_y]))
6 in
7
8  if w == 1
9 then 1
10 else
11 if pruefR
12 then
13 (m + (index suffix_x suffix_y (prods) (zaehler+2) sufläng)
14 else
15 if plandows == False
16 index prefix_x prefix_y (prods) (zaehler+2) preläng
17 else
18 (m + (index suffix_x suffix_y (prods) (zaehler+2) sufläng)
```

Wenn die Präfixe syntaktisch [8] gleich sind, wird *index* mit m addiert auf die Suffixe aufgerufen. Das m ist die Länge, die im Falle von Suffixaufrufen addiert werden muss. Die Abbruchbedingung in Zeile 7 tritt ein, wenn eine von den Längen gleich eins ist. Das zurückgelieferte k signalisiert die erste unteschiedliche Stelle und wird bei der Bildung der Position-Grammatik benötigt.

5.2.10 Modul Posgrammar

Wir beschreiben das Modul Posgrammar.Posgrammar Die *posgrammar*-Funktion ruft die *posgram*-Funktion auf, die sich aus der Vereinigung der Menge der `H`- und `PT`-Nonterminale zusammen setzt [9]. Die Bildung der `H`-Typen übernimmt die *hpos*-Funktion in Zeile 12 und die Bildung der `PT`-Typen übernimmt die *ppos*-Funktion in Zeile 10.

```
1 posgrammar :: STG_Prod -> [STG_Prod] -> [ProdPre] -> [STG_Prod]
2            -> Integer
3            -> (Position,[Position],Integer)
4 posgrammar a stgprods prodpre lliste indx =
5              posgram a stgprods prodpre lliste indx
```

[8] Wenn die rechten Seiten der Präfixe gleich sind, bilden sie auch die gleichen Wörter

[9] siehe 4.2.1

```
6
7
8 posgram :: STG_Prod -> [STG_Prod] -> [ProdPre] -> [STG_Prod]
9         -> Integer
10          -> (Position,[Position],Integer)
11 posgram x stgprods prodpre lliste k =
12   let
13      posgr = ppos x stgprods prodpre lliste k []
14   in
15   ((last posgr),posgr++(hpos(snd(split_in_two stgprods))[]),k)
```

Die *posgrammar*-Funktion übernimmt im ersten Parameter das Nonterminal, das als erstes auf die Transformationsregeln im Kapitel 4.2 in Abbildung 4.5 angewendet wird. Die zweite Eingabe ist die benötigte Singleton-Tree-Grammatik. Desweiteren werden `PrpdPre`- und `STG_Prod`-Längen gebraucht. Die letzte Eingabe ist die *index*-Zahl k, die von *index* zurückgeliefert wird. Die *posgrammar*-Funktion überreicht die Eingabe, ohne Veränderung an die Funktion *posgram*, die als Ausgabe ein dreier Tupel mit dem ersten gebildeten Positions-Nonterminal, die Position-Grammatik und die Zahl k hat. Dieses k wird immer beim Bilden neuer Position-Nonterminale an das Namensstring konkateniert. Dies vereinfacht später die Analyse der Positions-Grammatik, weil die Verbindung zwischen den Position-Nonterminalen besser nachzuvollziehen ist. Wie oben erwähnt, bildet die *hpos* Funktion alle `H`-Typen.

```
1 hpos :: [STG_Prod] -> [Position] -> [Position]
2 hpos                          [] hposition = hposition
3 hpos ((STG_Prod (CN p) xs):ys) hposition =
4   case xs of
5    [Loch]      -> hpos ys ([(Pos (H p) [])]++hposition)
6    [CN a,CN b] -> hpos ys ([(Pos (H p) [H a,H b])]++hposition)
7    otherwise   ->let
8                    fcn = (finde_cn xs 0)
9                    fun = [(Pos (H p) [Z (fst fcn),(snd fcn)])]
10                  in
11                    hpos ys (fun++hposition)
```

Durch *pattern matching* in Zeile 4 schaut man nach Kontext-Nonterminalen. Im *case* werden die einzelnen auftretenden Typen unterschieden, und dementsprechend umgewandelt. Die enstandenen Position-Nonterminale werden in der Liste *hposition* abgespeichert. In Zeile 6 wird das Loch als leere Liste übernommen, in Zeile 7 als `H`-Typen und zuletzt kommt der Funktionsfall ins Spiel. Dabei wird mit der Funktion *finde_ cn* die Stelle des einzigen Kontextes in den Funktionsargumenten festgestellt.

```
1 fine_cn :: [STG_Pre_Symbol] -> Integer -> (Integer,Pos_Symbol)
2
3 find_cn (x:xs) zaehler =
4
5       case x of
6
7           CN a -> error "R. Seite darf nicht mit CN anfangen"
```

```
8             otherwise -> finde_cn_1 xs (zaehler+1)
9
10
11finde_cn_1 :: [STG_Pre_Symbol] -> Integer
12            -> (Integer, Pos_Symbol)
13
14finde_cn_1 [] zaehler = error "Ein. falsch: CN->f(A1...An)"
15finde_cn_1 (x:xs) zaehler =
16
17                    case x of
18                         CN a -> (zaehler,(H a))
19                         otherwise -> finde_cn_1 xs (zaehler+1)
```

Die Funktion *finde_ cn* ruft dabei eine Unterfunktion auf, die ausser der ersten Stelle, die das Funktionszeichen enthält, nach dem Kontext-Nonterminal sucht, und bei Misserfolg zur nächsten Stelle springt und den Zähler erhöht. Die Ausgabe ist ein Tupel mit der Nummer der Stelle und dem Kontext-Nonterminal. Dieses Tupel wird in Zeile 10 von *hpos* beim Erzeugen des Position-Nonterminals aufgerufen. Auf der rechten des Position-Nonterminals, wird dem `Z`-Typ die Zahl, und dem `H`-Typ das Position-Nonterminal übergeben. Ausser den `H`-Typen bildet die *ppos*-Funktion die eigentlichen Transformationen.

```
1ppos :: STG_Prod->[STG_Prod]->[ProdPre]->[STG_Prod]->Integer
2     ->[Position]-> [Position]
3
4ppos (STG_Prod as axs) stgprods prelen lliste k posgrammatik =
```

Bei den Eingaben handelt es sich um das Anfangsnonterminal, die Singleton-Tree-Grammatik, sowie die `ProdPre`- und `STG_Prod`- Längen. Die letzte Eingabe ist die Position-Grammatik, die während der Anwendung der Regeln aufgefüllt wird. Die erste Regel überprüft zuerst das $k = 1$ und bildet die Position-Grammatik vom Typ `PT`. Dabei steht im String zusätzlich der ursprüngliche Typ aus der `STG_Prod`-Grammatik.

```
1let
2   p1 = posgrammatik
3   st = "Module Posgrammar, Funktion fukt (erstes case)"
4in
5
6if k==1
7then
8   case as of
9
10        TN a -> ([(Pos(PT ("TN"++"_"++a++","++"1")) [])]++p1)
11
12        CN a -> ([(Pos(PT ("CN"++"_"++a++","++"1")) [])]++p1)
13
14        otherwise -> error st
15 else
```

Die zweite Regel wird folgendermaßen implementiert:

```
((F a): xs) ->
 let
   gsumme   = (get_summe 1 xs k lliste prelen)
   prod2    = (second gsumme)
   prod3    = (thrd gsumme)
   cnodertn = ((head (drop1 (prod2-1) xs)))
   neuk     = (k-prod3)
   gp       = (getProd_stg cnodertn stgprods)
   p1       = posgrammatik
in
  if nurTN xs
  then
     if first gsumme
     then
       case cnodertn of
            TN v -> let
                      tn' = ("TN"++"_"++v++","++(show neuk))
                      pp  = ([Pos (PT tn) [Z prod2,PT tn']]++p1)
                    in
                      ppos gp stgprods prelen lliste neuk pp
            CN w -> let
                      cn' = ("CN"++"_"++w++","++(show neuk))
                      pp  = ([Pos (PT tn) [Z prod2,PT cn']]++p1)
                    in
                      ppos gp stgprods prelen lliste neuk pp
            otherwise -> error "Module Posgr., Funktion fukt"
      else
         error " "
    else error "TN kann auf der r. Seite nur TN's besitzen"
```

Nachdem im *case* festgestellt wird, dass es sich um ein Term-Nonterminal handelt, wird mit Hilfe der Funktion *get_summe* die Länge der Argumente aufaddiert, um das größte k' herauszufinden.

```
get_summe :: Integer->[STG_Pre_Symbol]->Integer->[STG_Prod]
          ->[ProdPre]-> (Bool,Integer,Integer)

get_summe zaehler (x:xs) k lliste preliste =
if zaehler > (length1 (x:xs))
then (False,0,0)
  else
  if zaehler==1
  then
     if (1 < k) && (k<=(1+(getLength_stg (x) lliste)))
     then (True,1,1)
     else get_summe (zaehler+1) (x:xs) k lliste preliste
  else
     let
      f = (fukt4 y lliste preliste)
      w = [f | y<-(take1 (zaehler-1) (x:xs))]
      k'= foldl (+) (1) w
      h = (head (drop1 (zaehler-1) (x:xs)))
     in
      if ((k'<k)&&(k<=(k'+(fukt4 h lliste preliste))))
```

```
21        then (True, zaehler, k')
22        else get_summe (zaehler+1) (x:xs) k lliste preliste
```

In Zeile 16 wird so lange aufaddiert und in Zeile 18 das k' mit dem k verglichen bis es zutrifft. Die Ausgabe von *get_ summe* ist ein Drei-Tupel mit einem *bool*, der bei Misserfolg für die Fehlermeldung verantwortlich ist, eine Zahl, die das Nonterminal für die weitere Suche angibt, und schließlich das aktualisierte k, das für das neu gebildete Position-Nonterminal im nächsten Aufruf von *ppos* in der zweiten Regel jeweils in Zeile 19 und 24 gebraucht wird. Es bleiben noch die dritte, vierte und die fünfte Regel, welche im folgenden Codefragment implementiert wurden:
In Zeile 14 wird die Bedingung für die erste Regel überprüft. Bei Erfolg wird das Nonterminal, das in *regel_ 1* steht, in der Zeile 16 im nächsten Aufruf an die Position-Grammatik konkateniert. Dabei ändert sich der Wert von k nicht, und *ppos* wird mit der Produktion vom Kontext-Nonterminal in Zeile 1 aufgerufen. Analog dazu sind die Regeln 2 und 3, die jeweils in Zeile 19 überprüft werden. Bei Erfolg der Abfrage wird die Regel 2 angewendet, indem k von der Länge des linken Teils in `ProdPre` des Kontext-Nonterminals von der Zeile 1 subtrahiert wird und das Ergebnis in $neuk'$ steht.

```
1 [CN a,TN b] ->
2  let
3     prod     = (getProd_stg (CN a) stgprods)
4     prod1    = (getProd_stg (TN b) stgprods)
5     laenge   = (getLength (L a) prelen)
6     laenge1  = (getLength_stg (TN b) lliste)
7     neuk     = (k-laenge1)
8     neuk'    = (k-laenge)
9     r1       = [PT ("CN"++"_"++a++","++(show k))]
10    r2       = [H a,PT ("TN"++"_"++b++","++(show neuk'))]
11    r3       = [PT ("CN"++"_"++a++","++(show neuk))]
12    regel_1  = ([(Pos (PT tn) r1)])
13    regel_2  = ([(Pos (PT tn) r2)])
14    regel_3  = ([(Pos (PT tn) r3)])
15    p1       = posgrammatik
16
17   in
18    if (1 < k)&&(k <= laenge)
19     then
20     ppos prod stgprods prelen lliste k (regel_1++p1)
21     else
22
23     if ((laenge)<k) && (k <= laenge+laenge1)
24     then
25     ppos prod1 stgprods prelen lliste neuk' (regel_2++p1)
26     else
27     ppos prod  stgprods prelen lliste neuk  (regel_3++p1)
```

Bei Misserfolg der Abfrage in Zeile 19 wird die dritte Regel angewendet. Hierbei tritt wieder ein aktualisiertes *neuk* im Aufruf in Zeile 23 auf. Dementsprechend wird *regel_ 3* an *posgrammatik* konkateniert. Beim Erzeugen der

Position-Nonterminale speichern wir im String die alten `STG_Prod`-Typen. Es soll dem Zweck der besseren Untersuchung der Nonterminale dienen, um die Zusammenhänge zwischen den verschiedenen Typen festellen zu können. Nachdem die Implementierung der Position-Grammatik erläutert wurde, soll der Schwerpunkt im nächsten Abschnitt auf die Erläuterung der *pExt*-Funktion liegen. Wie schon aus vorherigen Abschnitten bekannt ist, benötigt die *pExt*-Funktion als Eingabe die Position-Grammatik, um den Unterterm zu berechnen.

5.2.11 Modul P_ext

Die *pExt*-Funktion hat insgesammt acht Eingaben. Die ersten vier entsprechen den Ausführungen aus Kapitel 4.3. Zuerst die Singleton-Tree-Grammatik, das `STG_Prod`-Nonterminal, weiterhin das Position-Nonterminal und die Position-Grammatik.

```
1 let
2    nonterm  = (STG_Prod (a) (as))
3    posnterm = (Pos (p) ps)
4    z        = zaehler
5    z2       = zaehler2
6    tnn      = (TN ("N'-"++show zaehler2))
7 in
8
9 p_ext ::[STG_Prod]->STG_Prod->Position->[Position]->Integer
10        ->Integer->[STG_Pre_Symbol]->[STG_Prod]
11        ->([STG_Prod],STG_Prod,Integer,[STG_Prod])
12
13 p_ext stgprods nonterm posnterm posprods z z2 erstesn2 lengstg=
```

Die nächsten zwei Eingaben regeln die interne Namensbildung der Nonterminale. Sie sorgen dafür, dass beim Bilden von Suffixen und Präfixen die Namen der erzeugten Nonterminale eindeutig bleiben, um die Testmenge z.B. bei *plandowski* zu spezifizieren. Die nächste Eingabe *erstesn2* ist eine Liste, die gefüllt wird, um später das erste Element in der Abbruchbedingung mitzunehmen. Schließlich beinhaltet die letzte Eingabe die Längenliste. Unten in Zeile 7 wird überprüft, ob das Position-Nonterminal aus der Eingabe ein λ-rule produziert oder die leere Liste erzeugt. Falls es zutrifft, wird im Falle des `TN`-Typs das `STG_Prod`-Nonterminal aus der Eingabe auf die rechte des neuen N' Nonterminals gespeichert, oder es wird im Falle des `CN`-Typs das `STG_Prod`-Nonterminal aus der Eingabe auf die rechte des neuen N' Nonterminals gespeichert, und zusätzlich das *erstn*2 mitgespeichert, denn es handelt sich um das Nonterminal, das in das Loch des Kontext-Nonterminals eingesetzt wird.

```
1 let
2    a1=[(STG_Prod tnn [a])]
3    a2=[(STG_Prod tnn [a,erstn2])]
4    b1=lengstg++[(STG_Prod tnn [Laenge lenga])]
```

```
5    b2=lengstg++[(STG_Prod tnn [Laenge (lenga+lengn2)])]
6    s1=stgprods
7  in
8  if lambda (Pos (p) ps) posprods lliste || ps==[]
9  then
10  case a of
11      TN l -> (s1++a1,(STG_Prod tnn [a]),zaehler2,b1)
12      CN l -> (s1++a2,(STG_Prod tnn [a,erstn2]),zaehler2,b2)
```

Unten im Quelltext werden drei Fälle unterschieden.

1. Mit Hilfe von *plandow* wird in Zeile 30 überprüft, ob p ein Präfix von `CN` $c1$ ist. Zusätzlich muss die Bedingung mit der Länge $lp = |p| < |c1| = lhc1$ gelten. Um die Präfixüberprüfung zu machen, wird ein Präfix b' von $c1$ mit der Länge lp in Zeile 13 gebildet. Falls hier die Präfixeigenschaft zutrifft, bildet man ein Suffix $suff1$ von $c1 = hprod$ mit der Länge $l1 = |c1| - |p|$ in Zeile 7 und speichert den gefundenen Unterterm an der zweiten Stelle im Ausgabetupel in Zeile 42 ab.

2. Ansonsten prüft *plandow'* in Zeile 24, ob `CN` $c1$ ein Präfix von p ist. In diesem Fall gilt die Bedingung $lp = |p| \geq |c1| = lhc1$. Um die umgekehrte Präfixbildung zu machen, wird ein Präfix a' von p mit der Länge $lhc1$ in Zeile 12 gebildet. Falls hier die Präfixeigenschaft zutrifft, wird $pExt$ auf $n2$ und dem Suffix von p der Länge $|c1| - |p|$ in Zeile 26 aufgerufen.

3. Wenn die Überprüfung in Zeile 24 nicht zutraf, hat die Präfixeigenschaft in den Schritten 1 und 2 nicht gegolten. Es handelt sich um den *disjoint*-Fall. Der Aufruf von $pExt$ wird mit $c1$ und dem p in Zeile 28 fortgeführt.

```
1 case as of
2
3 [CN c1, n2]  ->
4
5 let
6  s2      = (Pos (p) (ps))
7  l_1     = (lHc1 - lp)
8  l_2     = (lp - lHc1)
9  suff1   = traverse_suffix_pos hprod posprods l_1
10                               zaehler dummy2 lliste 0
11 suff2   = traverse_suffix_pos s2 posprods l_2
12                               zaehler dummy2 lliste 0
13 lp      = (getLength_pos p lliste)
14 a'      = traverse_prefix p prodpre lHc1
15                           (zaehler+1) dummy3 prelen 0
16 b'      = traverse_prefix hprodpre prodpre lp
17                           (zaehler+1) dummy3 prelen 0
18 fall2   = (getProd_stg (n2) stgprods)
19 fall22  =([(fst suff2)]++posprods)
```

```
20 fall3   = (getProd_stg (CN c1) stgprods)
21 er      = (erstesn2++[n2])
22 test1   = [((N (wandleum p)), N ("P"++show (zaehler+1)))]
23 test2   = [((N (wandleum (H c1))), N ("P"++show (zaehler+1)))]
24 pg      = (prodint++prodgenprefix')
25 plandow=  plandowski (fst chomsk)  (snd chomsk)  pg  test1
26 plandow'= plandowski (fst chomsk') (snd chomsk') pg  test2
27 abfrage  =
28 if plandow'==True
29  then
30   p_ext stgprods fall2 (fst suff2) fall22 (snd suff2)
31         zaehler2 er lengstg
32  else
33   p_ext stgprods fall3 (Pos (p) ps) posprods zaehler
34         zaehler2 er lengstg
35in
36if lp<lHc1 && plandow==True
37then
38 let
39  w        = (pos_in_sym (fst suff1) zaehler2)
40  v        = (pos_in_prod (fst suff1) stgprods
41                          zaehler2 (lengstg++[wleng]))
42  v'''     = (third1 v)
43  n''      = (STG_Prod tnn [w,n2])
44  np       = ([n'']++(first1 v)++stgprods)
45  wleng    = (STG_Prod w [Laenge v'''])
46  n2leng   = (getLength_stg n2 lengstg)
47  n''leng  = (STG_Prod tnn [Laenge (v'''+n2leng)])
48 in
49 (np,n'',zaehler2,(lengstg++[n''leng]++(second1 v)))
50else abfrage
```

Die Ausgabe von *pExt* ist ein Viertupel, der an der ersten Stelle die Liste der Singleton-Tree-Grammatik und an der zweiten Stelle den gefundenen Unterterm beinhaltet. In Zeile 34 wird mit Hilfe der Funktion *pos_in_prod* gegebenfalls der Unterterm in `STG_Prod` gerechte Form runtergebrochen. Während dieser Prozedur kommen höchstens $depth(C)$ viele Nonterminale dazu (s. Lemma 4.3.2). Im nächsten Abschnitt wird das Modul *Unifikation* besprochen, wobei hier alle vorher besprochenen Module importiert werden.

5.2.12 Modul Unifikation

Die Funktion *unifikationSTG* ist für `STGProd`-Typen konzipiert. Die erste Eingabe ist eine Liste mit den entsprechenden `STGProd`-Typen und die restlichen beiden Eingaben übernehmen die Nonterminale, die zu unifizieren sind. Die übergebene Grammatik wird in die interne `STG_Prod`-Typform konvertiert und in *stgprods* gespeichert. Die Konvertierung übernimmt die Funktion *convertSTGtoSTG_Prod*, die unter anderem auf Falscheingaben reagiert (siehe 5.2.6). Die Funktion *getP* sucht aus der *stgprods* Liste das passende Nonterminal und konvertiert diese ebenfalls in die interne Verarbeitungs-

form.

```
unifikationSTG grammatik nt1 nt2 =
        let
           stgprods = (convertSTGtoSTG_Prod grammatik)
           as       = (getP nt1 stgprods)
           at       = (getP nt2 stgprods)
        in
```

Die konvertierte Eingabe wird der Funktion *unifikation_sub* übergeben. Diese Funktion prüft die Unifizierbarkeit der Terme für die erste unterschiedliche Stelle.

```
case unifikation_sub stgprods as at [] 1 of
      Left err -> putStrLn err
      Right x  -> putStrLn (show ([stgprod_in_stg (fst x) []],
                                  [stgprod_in_stg (snd x) []]))
```

Diese Funktion hat den gleichen Effekt wie *unifikationSTG* mit dem Unterschied, das hier die internen `STG_Prod`-Typen verwendet werden.

```
unifikation stgprods as at =
        case unifikation_sub stgprods as at [] 1 of
                Left err -> putStrLn err
                Right x  -> putStrLn (show x)
```

Die Funktion *unifikation_sub* ruft alle Funktionen auf, welche für die Unifikation gebraucht werden. Es werden die Längen der `STG_Prod`-Nonterminale und der `ProdPre`-Nonterminale berechnet. Die Umformung in `ProdPre`-Typen wird durch die Funktion *transform_p* gewährleistet. Die Funktion *prelen_in_prodlen* wandelt `ProdPre`-Typen in `ProdInt`-Typen um.

```
unifikation_sub :: [STG_Prod]->STG_Prod->STG_Prod->[STG_Prod]
                   -> Integer
                   -> Either String ([STG_Prod],[STG_Prod])

unifikation_sub stgprods as at unifikant zaehler =
let
 lliste      = (wordLength_stg stgprods)
 prelen      = (wordLength prodpre)
 prodpre     = (transform_p stgprods)
 prodint     = (prelen_in_prodlen prelen)
 preas       = (head(transform_p [as]))
 preat       = (head(transform_p [at]))
 indx        = (index preas preat prodpre 1 prelen)
 pas         = (posgrammar as stgprods prelen lliste indx)
 pat         = (posgrammar at stgprods prelen lliste indx)
 fpas        = (first1 pas)
 fpat        = (first1 pat)
 passtg      = (p_ext stgprods as fpas (second1 pas) 1
                      zaehler [] lliste)
 patstg      = (p_ext stgprods at fpat (second1 pat) 1
                      zaehler [] lliste)
 zaehlerNas  = (third passtg)
 zaehlerNat  = (third patstg)
```

```
24 neulenstgas = (fourth passtg)
25 neulenstgat = (fourth patstg)
26 fstrootpas  = (fst(root(second passtg) [] (first passtg)
27                          neulenstgas))
28 fstrootpat  = (fst(root(second patstg) [] (first patstg)
29                          neulenstgat))
30 sndrootpas  = (snd(root(second passtg) [] (first passtg)
31                          neulenstgas))
32 sndrootpat  = (snd(root(second patstg) [] (first patstg)
33                          neulenstgat))
34 chomsk      = (chomsky prodpre 1)
35 testmeng    = [(getLinks as, getLinks at)]
36 plan        = plandowski (fst chomsk) (snd chomsk)
37                          prodint testmeng
38 in
```

Da die meisten Funktionen in den vorherigen Abschnitten besprochen wurden, bleiben noch zwei wichtige Funktionen, die erklärt werden müssen. Eine dieser Funktionen ist *root*:

```
1 root :: STG_Prod->[STG_Prod]->[STG_Prod]->[STG_Prod]
2         -> (STG_Pre_Symbol,STG_Prod)
```

Wie der Name andeutet, berechnet *root* das erste Terminalsymbol, vom Nonterminal aus der ersten Eingabe. Bei diesem Nonterminal handelt es sich um den von *pExt* gefundenen Unterterm (Zeilen 17 und 18) [10]. Jeweils für beide zu unifizierenden Nonterminale wird in diesen Zeilen ein Tupel mit dem Unterterm und der Grammatik zurückgeliefert. Die *root*-Funktion prüft zusätzlich die Längen der rechten Seiten. Da *root* am ersten Nonterminal auf der rechten Seite interresiert ist und den rekursiven Aufruf mit der Produktion dieses Nonterminal fortführt, ist es möglich, dass das falsche Symbol berechnet wird. Dieses Symbol kann das *Loch* sein, jedoch möchte man das erste Symbol, also das Zeichen im *Loch* finden. Um die falsche Berechnung zu vermeiden, prüft man die Längen der Nonterminale. Das *Loch* hat die Länge 0, dementsprechend wird der rekursive Aufruf auf das zweitrechte, drittrechte usw. Nonterminal verlagert, bis ein Nonterminal mit einer Länge größer als 0 gefunden wird. Wenn keine Löcher zu prüfen sind, wird das Ausgabetupel mit den Werten direkt gefüllt, sobald man auf eine Variable, Konstante oder eine Funktion trifft. Im Ausgabetupel steht an der ersten Stelle das *Wurzelzeichen* und an der zweiten Stelle der Unterterm, der an *root* von *pExt* übergeben wurde. Als nächstes soll die Funktion *occurs_check* erläutert werden. Wie der Name andeutet, sucht diese Funktion für eine gegebene Variable nach Vorkommen im gegebenen Unterterm. Diese Funktion wurde effizient für Singleton-Tree-Grammatiken implementiert. Die Funktion liefert *false*, wenn die Variable nicht gefunden wurde. Ansonsten wird die rechte Seite des Nonterminals als *queue* benutzt, wobei neue Elemente von rechts konkateniert werden. Bei diesen Elementen handelt es sich um

[10] *pExt* liefert *passtg* oder *patstg*

die rechten Seiten von TN- bzw. CN-Typen. Die Nonterminale dieser Typen kommen zusätzlich in eine Liste *fertigen*, die signalisiert, dass die Suche der Variablen nicht nochmal stattfinden muss. Auf diese Weise spart man gleiche Teilbäume und vermeidet den möglichen exponetiellen Aufwand. Für den Robinson-Algorithmus aus dem Abschnitt 4.4 können jetzt die Anweisungen ab Zeile 6 ausgeführt werden.
Diese Anweisung muss immer am Anfang gemacht werden, denn nach jedem Aufruf der *unifikation_sub*-Funktion muss mit Hilfe von *plandowski*[11] und den beiden Nonterminalen *as* und *at* getestet werden, ob die Unifikation erfolgreich war, nachdem beide Nonterminale *as* und *at* unifiziert wurden.

```
1 if plan
2 then
3  Right (stgprods, unifikant)
4 else
```

Wenn dieser Test *false* ergibt, müssen die *roots* überprüft werden. Zuerst wählt man die Wurzel eines Unterterms und vergleicht diesen mit weiteren *cases* auf die Wurzel des anderen Unterterms.

```
1 case fstrootpas of
```

Für das Funktionssymbol in *fstrootpas* vergleicht man das Funktionssymbol des anderen Unterterms. Da beide Funktionsterme sind, ist eine Unifikation ausgeschlossen, nach Zeile 7 im Robinson in Kapitel 4.4.

```
1 (F f) -> case fstrootpat of
2         (F g)      ->
3         if (g/=f)
4          then Left "Nicht unifizierbar"
5          else Left "Ein root muss eine Variable sein!"
```

Ansonsten ist die Wurzel *fstrootpat* eine Variable und das neue Nonterminal *nt* kann aufgebaut werden. Zusätzlich wird mit der Funktion *ersetze_R* die Definition 4.4.1 realisiert, nämlich die Umwandlung der gefundenen Variable in ein Term-Nonterminal.

```
1 (X x) ->
2  let
3   nt          = (STG_Prod (TN x) [TN ("N'-"++show zaehlerNat)])
4   unifmenge_1= ([nt]++(ersetze_R (X x) (first passtg)))
5   unif        = ([nt]++unifikant)
6  in
7   if occurs_check (X x) sndrootpas (first passtg) []
8   then Left "1. Nicht unifizierbar! Occurs Check!"
9   else unifikation_sub unifmenge_1 as at unif (zaehler+1)
10
11 otherwise -> Left "Nicht unifizierbar"
```

[11] oben in der Zeile 29 im *let* werden die üblichen Schritte gemacht, um die Eingabe der plandowski-Funktion anzupassen

Wenn diese Schritte erfolgreich waren, kann der Aufruf *unifikation_ sub* mit der erweiterten Grammatik G' und dem erhöhten Zähler in Zeile 8 aufgerufen werden.

5.3 Voraussetzung und Bedienung des Programms

Das Programm enthält keinen Parser, weshalb die Eingabe nicht auf die ganzen Eigenschaften überprüft wird. Damit das Programm einwandfrei getestet und ausgeführt werden kann, müssen einige Voraussetzungen vom Benutzer berücksichtigt werden.

Voraussetzungen für PlandowskiProd

1. Die Eingabegrammatik muss rekursionsfrei sein.
2. Jedes Nonterminal, das in der Grammatik vorkommt, muss genau eine Produktion besitzen.

Die Voraussetzung, dass die Eingabegrammatik in Chomsky-Normalform sein muss, ist nicht vom Benutzer zu beachten. Jede Grammatik wird nach der Eingabe in die Chomsky-Normalform überführt, womit sich der Benutzer keine Gedanken darüber machen muss.
Die Eingabe für die Funktion *plandowskiProd* sieht folgendermaßen aus:

```
grammar :: [Prod]
grammar =  [Produktion (N "A") [N "B",N "C",N "C",T "d"],
            Produktion (N "K") [N "C",N "B"],
            Produktion (N "B") [N "C",N "D"],
            Produktion (N "C") [N "D",N "E",T "x"],
            Produktion (N "D") [N "G",N "F"],
            Produktion (N "E") [N "F",N "G"],
            Produktion (N "F") [T "c"],
            Produktion (N "G") [T "c"],
            Produktion (N "V") [N "B",N "C",N "C",T "d"]]

testset :: [(STG_Pre_Symbol,STG_Pre_Symbol)]
testset =  [(N "A",N "V"),(N "E",N "D")]
```

Der Aufruf in der Konsole wird wie folgt eingegeben:

```
> plandowskiProd grammar testset
```

oder

```
> plandowskiProd grammar [(N "A",N "V"),(N "E",N "D")]
```

Die Ausgabe für diesen Aufruf ist:

```
> True
```

Voraussetzungen für PlandowskiSTG
Für die Funktion in PlandowskiSTG gilt nur die erste Voraussetzung. Bei den `STG`-Typen werden noch zusätzlich einige Eigenschaften überprüft, da diese Eingabe für Benutzer geeignet ist, die mit den Regeln der Singleton-Tree-Grammatiken nicht vertraut sind. Die Einzelheiten wurden im Abschnitt 5.2.6 besprochen. Die Chomsky-Normalform muss nicht erfüllt sein. Betrachten wir nun eine Eingabe für die Funktion *plandowskiSTG*.

```
grammarSTG :: STG
grammarSTG =  STG [AfA "V" "z" ["A_t","A_s"],
                   AfA "W" "z" ["A_s","A_t"],
                   AfA "A_t" "g" ["B","K"],
                   AfA "A_s" "g" ["A","A"],
                   ACA "A" "C_0" "A'",
                   AfA "A'" "a" [],
                   ACA "B" "D" "B'",
                   Ax "B'" "x",
                   CCC "C_1" "C_0" "C_0",
                   CfAC "C_0" "f" ["A'"] "C'" [],
                   ACA "K" "C_0" "B''",
                   Ax "B''" "y",
                   CfAC "C'" "j" [] "C''" ["A'"],
                   CfAC "C''" "g" [] "C_5" ["L"],
                   Ax "L" "o",
                   CLoch "C_5",
                   CCC "D" "C_0" "C_0"]

testset :: [(String,String)]
testset =  [("V", "V"),("A_t","A_s")]
```

Der Aufruf sieht wie folgt aus:

```
> plandowskiSTG grammarSTG testset
```

Die Ausgabe für diesen Aufruf ist:

```
> False
```

Voraussetzungen für PlandowskiSTG_ Prod
Hier gelten die gleichen Voraussetzungen wie für PlandowskiProd. Die Eingabegrammatik muss ebenfalls nicht in Chomsky-Normalform sein. Die Eingabe für die Funktion *plandowskiSTG_ Prod* sieht zum Beispiel wie folgt aus:

```
grammarSTG_Prod :: [STG_Prod]
grammarSTG_Prod =  [STG_Prod (TN "A_t") [F "g",TN "B",TN "A"],
                    STG_Prod (TN "A_s") [F "g",TN "A",TN "A"],
                    STG_Prod (TN "A")   [CN "C_4",TN "A'"],
                    STG_Prod (TN "A'")  [K "a"],
                    STG_Prod (TN "B")   [CN "D",TN "B'"],
                    STG_Prod (TN "B'")  [X "x"],
                    STG_Prod (CN "C_4") [CN "C_3",CN "C_3"],
                    STG_Prod (CN "C_3") [CN "C_2",CN "C_2"],
                    STG_Prod (CN "C_2") [CN "C_1",CN "C_1"],
                    STG_Prod (CN "C_1") [CN "C_0",CN "C_0"],
                    STG_Prod (CN "C_0") [F "f",CN "C'"],
                    STG_Prod (CN "C'")  [Loch],
                    STG_Prod (CN "D")   [CN "C_3",CN "C_2"]]

testset :: [(Symbol,Symbol)]
testset =  [(TN "A_t",TN "A_s")]
```

Der Aufruf sieht wie folgt aus:

```
> plandowskiSTG_Prod grammarSTG_Prod testset
```

Die Ausgabe für diesen Aufruf ist:

```
> False
```

Voraussetzungen für Unifikation
Für die Funktion *unifikationSTG* muss folgendes beachtet werden:

1. Die `STG` Grammatik darf keine Rekursion enthalten.
2. Für keinen der zu unifizierenden Nonterminale sollte ein leerer String eingegeben werden.

Eine Eingabe für die Funktion *unifikationSTG* sieht wie folgt aus:

```
stg :: STG
stg =  STG [AfA   "A_t" "g" ["B","A","K"],
            AfA   "A_s" "g" ["A","A","A"],
            ACA   "A" "C_1" "A'",
            AfA   "A'" "a" [],
            ACA   "B" "D" "B'",
            Ax    "B'" "x",
            CCC   "C_4" "C_3" "C_3",
            CCC   "C_3" "C_2" "C_2",
            CCC   "C_2" "C_1" "C_1",
```

```
        CCC   "C_1" "C_0" "C_0",
        CfAC  "C_0" "f" ["A'"] "C'" ["A'","A'","A'","A''"],
        ACA   "K" "C_1" "B''",
        Ax    "B''" "y",
        AfA   "A''" "b" [],
        CfAC  "C'" "j" [] "C''" ["A'"],
        CLoch "C''",
        CCC   "D" "C_0" "C_0"]
```

Der Aufruf zum Unifizieren der Nonterminale "A_t" und "A_s" wird wie folgt eingegeben:

```
> unifikationSTG stg "A_s" "A_t"
```

Die Ausgabe des Algorithmus ist:

```
([y --> N'-2
  x --> N'-1
  A_t --> g(B,A,K)
  A_s --> g(A,A,A)
  A --> C_1 A'
  A' --> a
  B --> D B'
  B' --> x
  C_4 --> C_3 C_3
  C_3 --> C_2 C_2
  C_2 --> C_1 C_1
  C_1 --> C_0 C_0
  C_0 --> f(A',C',A',A',A',A'')
  K --> C_1 B''
  B'' --> y
  A'' --> b
  C' --> j(C'',A')
  C'' --> []
  D --> C_0 C_0
  N'-1 --> A'
  N'-2 --> A'],
 [y --> N'-2
  x --> N'-1
])
```

Dies ist ein Tupel von zwei Listen, wobei die erste Liste die unifizierte Grammatik darstellt und die zweite Liste die dazugehörige Substitutionsmenge. Bei der Eingabe der Funktion *unifikation* ist folgendes zu beachten:

1. Die `STG_Prod` Grammatik darf keine Rekursion enthalten.

2. Bei der Eingabe sollte als Eingabe keine leere Grammatik eingegeben werden.

3. Jedes Nonterminal, dass in der Grammatik vorkommt, sollte genau eine Produktion besitzen.

Die Eingabegrammatik sei wie folgt gegeben:

```
stgp :: [STG_Prod]
stgp =  [STG_Prod (TN "V")   [F "z",TN "A_t",TN "A_s"],
         STG_Prod (TN "W")   [F "z",TN "A_s",TN "A_t"],
         STG_Prod (TN "A_t") [F "g",TN "B",TN "K"],
         STG_Prod (TN "A_s") [F "g",TN "A", TN "A"],
         STG_Prod (TN "A")   [CN "C_2",TN "A'"],
         STG_Prod (TN "A'")  [K "a"],
         STG_Prod (TN "B")   [CN "D",TN "B'"],
         STG_Prod (TN "B'")  [X "x"],
         STG_Prod (CN "C_4") [CN "C_3",CN "C_3"],
         STG_Prod (CN "C_3") [CN "C_2",CN "C_2"],
         STG_Prod (CN "C_2") [CN "C_1",CN "C_1"],
         STG_Prod (CN "C_1") [CN "C_0",CN "C_0"],
         STG_Prod (CN "C_0") [F "f",TN "A'",CN "C'"],
         STG_Prod (TN "K")   [CN "C_1",TN "B''"],
         STG_Prod (TN "B''") [X "y"],
         STG_Prod (CN "C'")  [F "j",CN "C''",TN "A'"],
         STG_Prod (CN "C''") [Loch],
         STG_Prod (CN "D")   [CN "C_0",CN "C_0"]]
```

Die zu unifizierenden Nonterminale seien (`TN` "V") und (`TN` "W"). Die Funktion muss wie folgt aufgerufen werden:

```
> unifikation stgp (STG_Prod (TN "V") [F "z",TN "A_t",TN "A_s"])
                   (STG_Prod (TN "W") [F "z",TN "A_s",TN "A_t"])
```

Die Ausgabe nach der Unifikation ist:

```
([TN y --> TN N'-2
 ,TN N'-2 --> CN PT 3-2,TN A'
 ,CN PT 3-2 --> CN C_1,CN C''
 ,TN x --> TN N'-1
 ,TN N'-1 --> CN PT 3-1,TN A'
 ,CN PT 3-1 --> CN C_1,CN C''
 ,TN V -->  F z (TN A_t,TN A_s)
 ,TN W -->  F z (TN A_s,TN A_t)
 ,TN A_t -->  F g (TN B,TN K)
 ,TN A_s -->  F g (TN A,TN A)
```

```
 ,TN A --> CN C_2,TN A'
 ,TN A' --> K a
 ,TN B --> CN D,TN B'
 ,TN B' --> TN x
 ,CN C_4 --> CN C_3,CN C_3
 ,CN C_3 --> CN C_2,CN C_2
 ,CN C_2 --> CN C_1,CN C_1
 ,CN C_1 --> CN C_0,CN C_0
 ,CN C_0 -->  F f (TN A',CN C')
 ,TN K --> CN C_1,TN B''
 ,TN B'' --> TN y
 ,CN C' -->  F j (CN C'',TN A')
 ,CN C'' --> []
 ,CN D --> CN C_0,CN C_0],
 [TN y --> TN N'-2
 ,TN x --> TN N'-1
])
```

Hierbei handelt es sich auch um ein Tupel mit zwei Listen. Die erste Liste ist die unifizierte Grammatik und die zweite Liste ist die Substitution. Wir gehen davon aus, dass triviale Falscheingaben vom Benutzer selbst erkannt werden.

Benutzung der ausführbaren Datei

Die ausführbare Datei *wut* wird in der Eingabeaufforderung (Shell) mit den folgenden Parametern aufgerufen:

- ***wut** unifikationSTG Dateiname String1 String2.* Dies ist der Aufruf für den Unifikationsalgorithmus auf den `STG` Typen. Der Dateiname repräsentiert die Datei, in der sich die `STG` Grammatik befindet, die als erste Eingabe an den Algorithmus übergeben wird. String1 und String2 sind die Bezeichner der zu unifizierenden Nonterminale. Die Ausgabe des Algorithmus wird in die Datei *Dateiname.out* geschrieben.

- ***wut** unifikation Dateiname Produktion1 Produktion2.* Dies ist der Aufruf für den Unifikationsalgorithmus auf den `STG_Prod` Typen. In der Datei befindet sich diesmal die `STG_Prod` Grammatik, die als erste Eingabe an den Algorithmus übergeben wird. Produktion1 und Produktion2 sind die Produktionen (vom Typ `STG_Prod`) der zu unifizierenden Nonterminale. Die Ausgabe wird in die neu erzeugte Datei *Dateiname.out* geschrieben.

- ***wut** plandowskiProd Dateiname1 Dateiname2.* Dies ist der Aufruf für den Algorithmus von Plandowski auf die `Prod` Typen. Der Dateiname1

steht für die Datei, in der sich die `Prod` Grammatik befindet, die als erste Eingabe an die Funktion *plandowskiProd* übergeben wird. Die zweite Eingabe ist wieder eine Datei. Sie beinhaltet die Testmenge und wird als zweite Eingabe dem Algorithmus übergeben. Die Ausgabedatei *Dateiname1.out* enthält das Ergebnis.

- ***wut plandowskiSTG Dateiname1 Dateiname2.*** Dies ist der Aufruf für den Algorithmus von Plandowski auf die `STG` Typen. Der Dateiname1 steht für die Datei, in der sich die `STG` Grammatik befindet, die als erste Eingabe an den Algorithmus von Plandowski übergeben wird. Die zweite Eingabe ist eine weitere Datei. In dieser Datei befindet sich die zu überprüfende Testmenge und wird als zweite Eingabe dem Algorithmus übergeben. Die Datei *Dateiname1.out* beinhaltet das Ergebnis.

- ***wut plandowskiSTG_ Prod Dateiname1 Dateiname2.*** Dies ist der Aufruf für den Algorithmus von Plandowski auf die `STG_Prod` Typen. In der ersten Datei befindet sich die `STG_Prod` Grammatik und in der zweiten Datei die zu überprüfende Testmenge. *Dateiname1.out* beinhaltet das Ergebnis des Algorithmus.

Falls der Benutzer eine Eingabe vergisst, wird er in den obigen Fällen mittels einer Meldung darauf hingewiesen.
Eine Datei für den Testaufruf des Unifikationsalgorithmus auf den `STG` Typen sieht dabei wie folgt aus:

```
1STG [AfA "A_t" "g" ["B","A"],
2AfA "A_s" "g" ["A","A"],
3ACA "A" "C_4" "A'",
4AfA "A'" "a" [],
5ACA "B" "D" "B'",
6Ax "B'" "x",
7CCC "C_4" "C_3" "C_3",
8CCC "C_3" "C_2" "C_2",
9CCC "C_2" "C_1" "C_1",
10CCC "C_1" "C_0" "C_0",
11CfAC "C_0" "f" [] "C'" [],
12CLoch "C'",
13CCC "D" "C_3" "C_2"]
```

Für die restlichen Datentypen muss die Eingabe in der entsprechenden Syntax erfolgen. Zu jeder Funktion ist in der beigefügten CD eine Testdatei angelegt worden. Die Aufrufe für die einzelnen Funktionen sehen wie folgt aus:

- *unifikationSTG*: wut unifikationSTG unifikationstg “*A_s*“ “*A_t*“

- *unifikationSTG_ Prod*: wut unifikation unifikationstgprod ’(`STG_Prod` (`TN` “A_t“) [`F` “g“, `TN` “B“, `TN` “A“])’ ’(`STG_Prod` (`TN` “A_s“) [`F` “g“, `TN` “A“, `TN` “A“])’

- *plandowskiSTG_ Prod*: wut plandowskiSTG_Prod plandowskistgprod plandowskistgprodtestset
- *plandowskiSTG*: wut plandowskiSTG plandowskistg plandowskistgtestset
- *plandowskiProd*: wut plandowskiProd plandowskiprod plandowskiprodtestset

Kapitel 6

Testfälle

Die folgenden Testfälle wurden mit der Option *ghc –make -O -o test Main.hs* in der Version 6.10.1 auf einem Intel(R) Pentium(R) D CPU 2.80GHz mit 1024 KB cache und 2060608 kB RAM kompiliert. Das Betriebssystem basiert auf Linux Fedora Version 2.6.26.8-57.fc8.

6.1 Test Plandowski

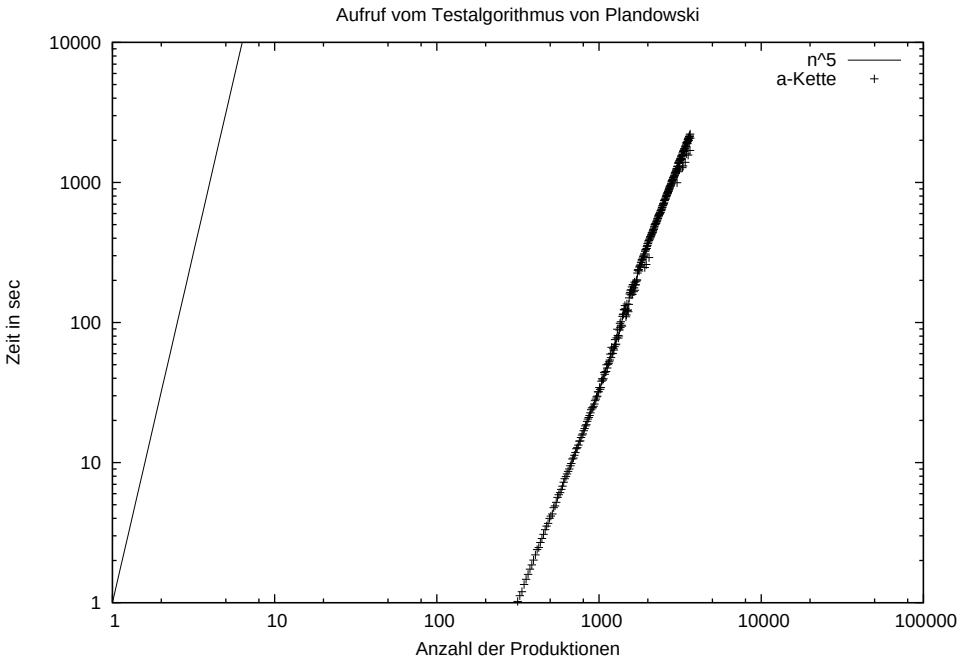
Dieser Test bezieht sich auf eine Grammatik, die mit ungefähr n Produktionen auskommt und dabei einen Term der Länge 2^n erzeugt (s. Beispiel 4.1.1). Die Grammatik hat folgende Gestalt:

$$A_i \rightarrow A_{i-1}A_{i-1}\, mit\, 1 \leq i \leq k, A_0 \rightarrow a$$

$$\cup$$

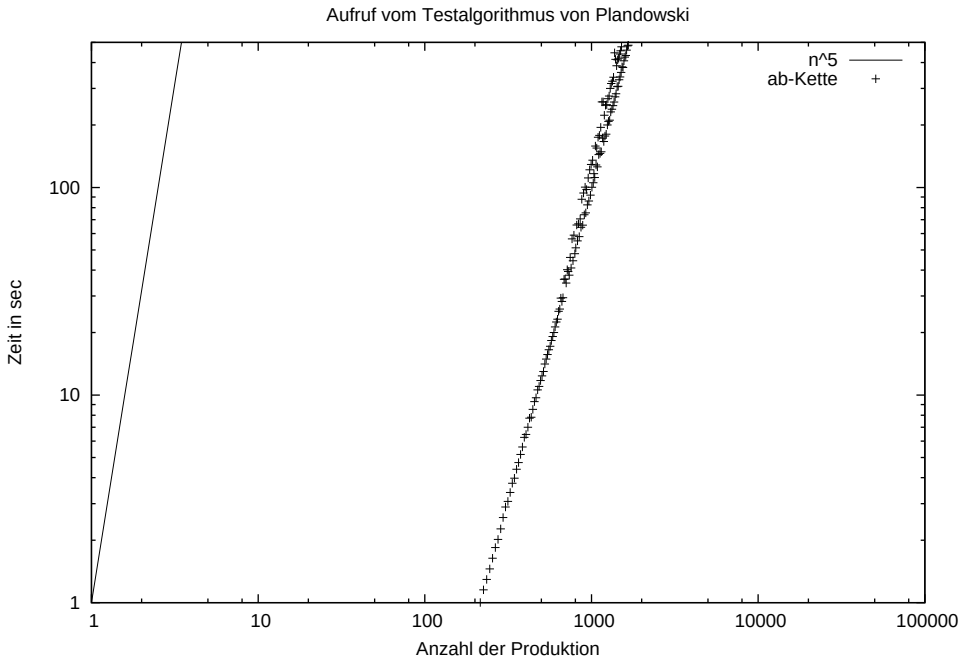
$$B_i \rightarrow B_{i-1}B_{i-1}\, mit\, 1 \leq i \leq k, B_0 \rightarrow a$$

Die Nonterminale A_i und B_i erzeugen Worte mit 2^n vielen a Terminalsymbolen und werden auf Gleichheit überprüft. Bei jedem Durchlauf des Tests wird die Anzahl der Produktionen um fünf weitere erhöht. Folgende Grafik zeigt den zeitlichen Verlauf der Berechnungen:



Beide Achsen entsprechen dem logarithmischen Maß, wobei die y-Achse die Dauer der Berechnung angibt und die x-Achse die Anzahl der Produktionen wiederspiegelt. Man sieht als Resultat eine Gerade, die verglichen mit einer polynomiellen Funktion fünften Grades flacher verläuft.

In der folgenden Graphik geht es um dieselbe Grammatik mit dem Unterschied, dass B_i ein 2^i langes Wort aus bs produziert. Der Verlauf zeigt einen ähnlichen Anstieg wie oben.



Im nächsten Test haben wir eine Grammatik, die der Fibonacci-Folge entsprechende Terme erzeugt. Die Termlängen, ohne den Funktionssymbolen, entsprechen der Fibonacci-Folge. Dieser Test eignet sich gut, weil in den Termen wenig Wiederholungen auftreten. Die Grammatik dafür ist:

$$A_i \to f(A_{i-1}, A_{i-2}), \ldots, A_0 \to 0, \ldots, A_1 \to 1 \; mit \; 1 \leq i \leq k \; , \; k > 1$$

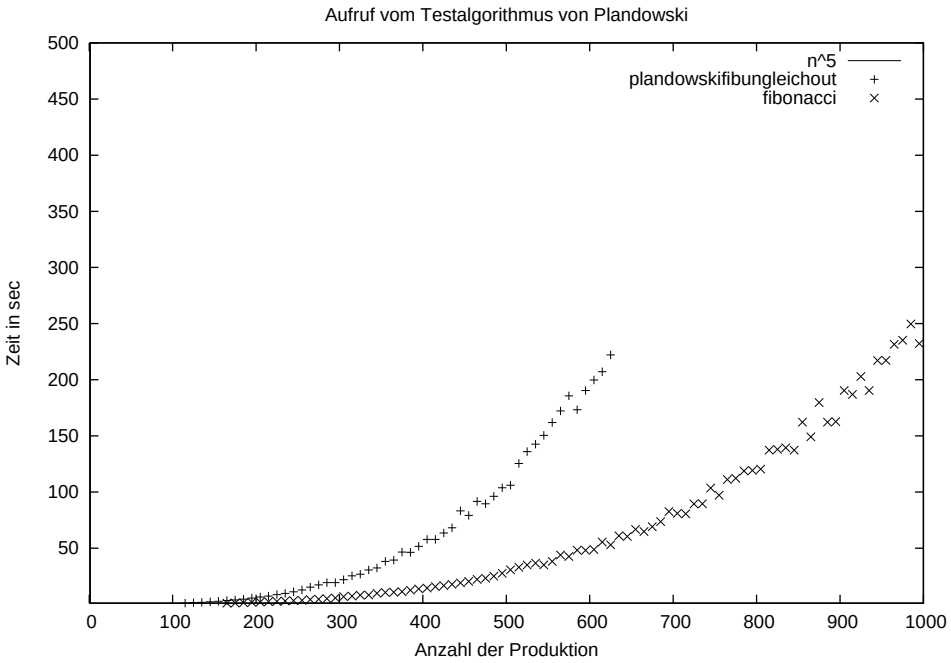
$$\cup$$

$$B_j \to f(B_{j-1}, B_{j-2}), \ldots, B_0 \to x, \ldots, B_1 \to y \; mit \; 1 \leq j \leq k \; , \; k > 1$$

Als Eingabe für den Algorithmus verwenden wir die Nonterminale A_i und B_j. Für denn Fall $i = j$ besitzen die Worte gleiche Längen. Die Grammatik sieht für $i = j = 4$ folgendermaßen aus:

$$
\begin{array}{lcllcl}
A_4 & \rightarrow & f(A_3, A_2) & B_4 & \rightarrow & f(B_3, B_2) \\
A_3 & \rightarrow & f(A_2, A_1) & B_3 & \rightarrow & f(B_2, B_1) \\
A_2 & \rightarrow & f(A_1, A_0) & B_2 & \rightarrow & f(B_1, B_0) \\
A_1 & \rightarrow & 1 & B_1 & \rightarrow & y \\
A_0 & \rightarrow & 1 & B_0 & \rightarrow & x
\end{array}
$$

Bei dem Test werden sowohl die Bedingung $i = j$, als auch die Bedingung $i \neq j$ überprüft. Der Algorithmus von Plandowski ergab folgende Graphik: Anhand der Graphik kann man den Unterschied für die beiden Fälle erkennen. Die Graphik zeigt einen Anstieg für diese Tests, die einen polynomiellen Verlauf andeutet.



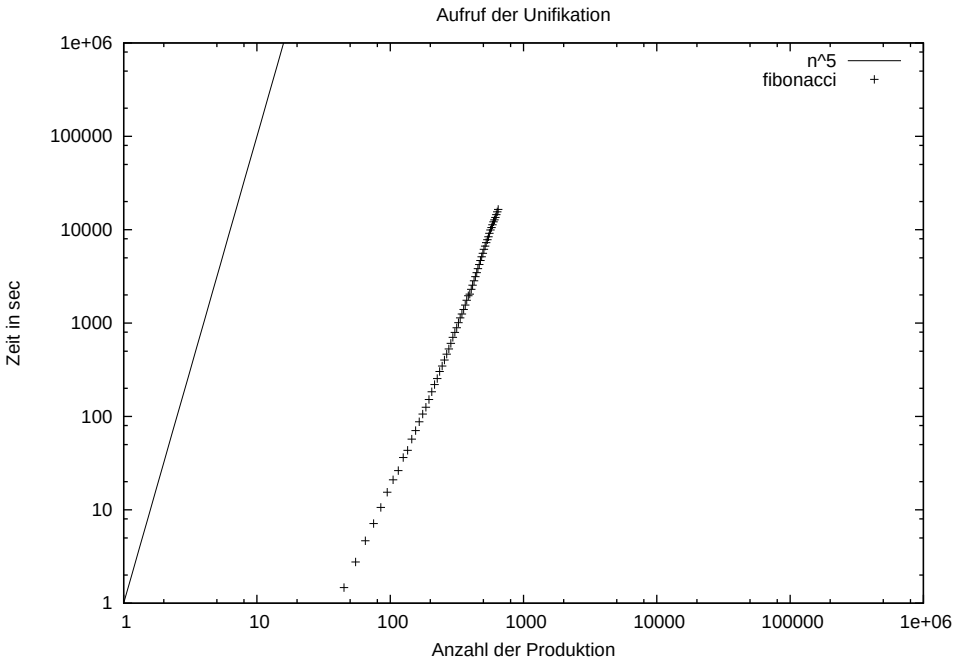
Die folgende Tabelle verschafft einen Überblick über die einzelnen Werte:

n	plandowskiAA	plandowskiAB	plandowskiABBA	plandowskifibout	plandowskifibungleich	a^{2^n}
5	0.0	0.0	0.0	1.0e-3		1.0e-3
105	3.5995e-2	0.110983	0.125981	0.26696	0.775883	7.5989e-2
205	0.283957	0.858869	0.955854	1.90871	6.457019	0.564914
305	0.92886	2.888561	2.991546	6.797966	21.961662	0.999847
405	2.194666	6.457019	6.950944	14.400811	57.813212	8.15976
505	4.150369	12.36312	13.740911	30.719331	106.048878	0.218966
605	7.243899	21.282765	23.108487	48.782584	199.806624	19.698006
705	11.281286	34.626736	38.08321	80.922698	363.285773	0.85687
805	16.879434	51.209215	57.644236	120.421692	568.979502	47.29481
905	24.589262	73.879768	101.981497	190.397055	767.58231	10.708372
1005	34.392772	100.236762	110.841149	240.358461	ka	6.71498
1105	44.674209	144.138088	141.530484	321.908063	ka	57.930193
1205	60.202848	176.970096	193.030654	431.197449	2036.766364	648.765372
1305	80.834711	231.300837	266.156537	608.941426	2679.182703	145.10594
1405	113.743708	282.703023	338.17359	804.827647	3302.692913	21.952663
1505	122.803331	358.444508	499.758026	1068.895504	3904.967355	5.451172
1605	158.927839	432.602234	475.965642	1276.677916	4675.378234	172.689747
1705	201.442376	540.149885	957.0835	1568.777508	5786.368339	839.034447
1805	258.263738	624.980989	848.858954	1939.30118	7251.105664	69.438443
1905	309.615932	844.388634	869.026888	2426.484119	ka	42.703509
2005	367.257168	ka	1212.413684	2337.677619	ka	138.581932
2105	424.946398	ka	1266.300493	3263.163923	ka	61.195698
2205	496.379538	ka	1654.011552	3570.800156	ka	725.857654
2305	554.18975	ka	2228.856163	ka	ka	1.909709
2405	629.992226	ka	2229.492066	ka	ka	ka
2505	704.874842	ka	2930.899436	ka	ka	ka
2605	812.04055	ka	2880.557089	ka	ka	ka
2705	900.173154	ka	ka	ka	ka	ka
2805	1012.060143	ka	ka	ka	ka	ka
2905	1105.697908	ka	ka	ka	ka	ka
3005	1251.178792	ka	ka	ka	ka	ka
3105	1385.045442	ka	ka	ka	ka	ka
3205	1490.985335	ka	ka	ka	ka	ka
3305	1656.58716	ka	ka	ka	ka	ka
3405	1793.48035	ka	ka	ka	ka	ka
3505	1891.42046	ka	ka	ka	ka	ka
3605	2119.156839	ka	ka	ka	ka	ka

Tabelle 6.1: Dauer der Berechnungen in [sec]

6.2 Test Unifikation

Die obige Fibonacci-Grammatik wird ebenfalls auf die Unifikation getestet. Dabei sind die Zeitverläufe in der folgenden Graphik enstanden.



Der Verlauf zeigt an, dass die Berechnung ein polynomielles Wachstum andeutet. Die Werte werden mit einer polynomiellen Funktion fünften Grades verglichen.
Die nächste Grammatik beschreibt Terme, deren Längen ohne die Funktionssymbole der Fakultätsfunktion entsprechen:

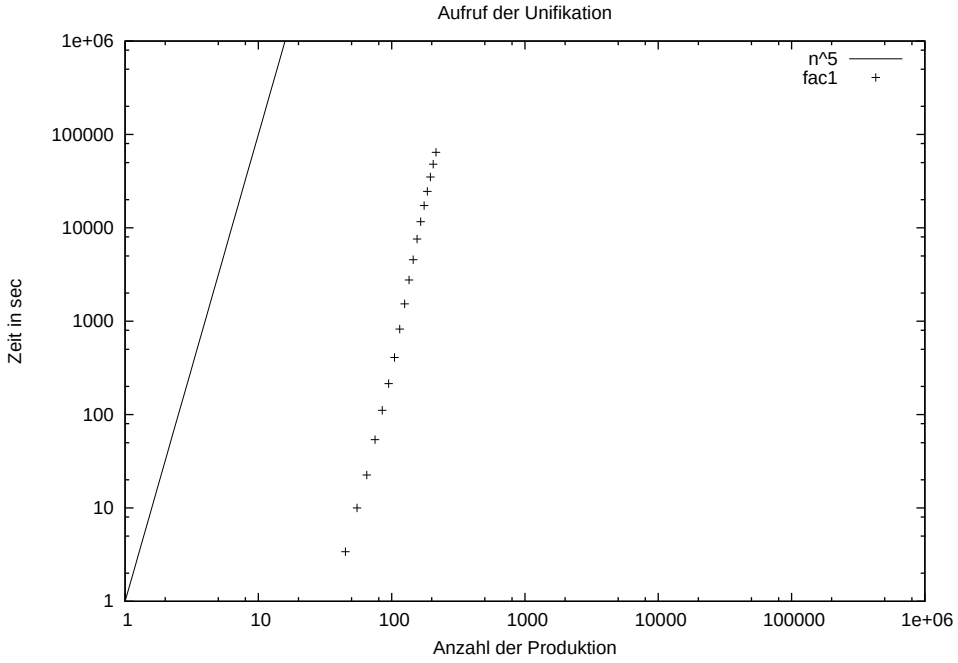
$$A_i \rightarrow f_i(\underbrace{A_{i-1}, \ldots, A_{i-1}}_{i-mal}), \ldots, A_0 \rightarrow 0, A_1 \rightarrow 1$$

$$\cup$$

$$B_j \rightarrow f_j(\underbrace{B_{j-1}, \ldots, B_{j-1}}_{j-1mal}, D), \ldots, B_0 \rightarrow 0, B_1 \rightarrow 1, D \rightarrow x$$

$$mit \;\; 2 \leq i, j \leq k \;,\; k > 2$$

Wir betrachten den Fall mit $i = j$. Die zu unifizierenden Nonterminale sind A_i und B_j, wobei der Unterschied im letzten Argument liegt. Folgende Graphik zeigt den zeitlichen Verlauf für wachsende Eingaben.



Hier sieht man eine leichte Krümmung des Verlaufs. Allerdings handelt es sich hierbei, um eine Grammatik mit zusätzlich neu dazukommenden Funktionen mit wachsender Stelligkeit, in Abhängigkeit von der Anzahl der Produktionen. Dadurch wächst die Anzahl der Zeichen in der Grammatik qua-

dratisch. Mit jedem Inkrementieren von n (Anzahl der Produktionen) kommen $n+(n-1)+\cdots+(n-i)+0 = \frac{n\cdot(n+1)}{2} = O(n^2)$ Zeichen dazu. Für eine Grammatik A_k mit $k \geq n$ ergeben sich Termlängen der Größe $k^2!$. Die Unifizierung dieser Grammatik dauert viel zu lange. In der nächsten Tabelle sind einige Werte aufgelistet. Die Angaben beziehen sich dabei auf Sekunden.

Größe der Grammatik	unifyfacult1	unifyfibo
5	1.999e-3	0.0
25	0.211968	0.249962
45	3.402482	1.470777
65	22.548571	4.658291
85	111.046119	10.606388
105	408.993824	20.947815
125	1532.623006	36.340475
145	4557.459162	57.108319
165	11617.554863	87.612681
185	24580.177245	125.600905
205	47975.088678	183.506104
225	ka	254.271346
245	ka	347.210215
265	ka	464.233426
285	ka	607.033716
305	ka	795.314094
325	ka	1010.365401
345	ka	1250.371915
365	ka	1560.895708
385	ka	1961.723772
405	ka	2297.08879
425	ka	2838.064548
445	ka	3467.857806
465	ka	4240.007421
485	ka	5122.9102
505	ka	6166.175599
525	ka	7270.367736
545	ka	8387.039977
565	ka	9926.544936
585	ka	11283.220689
605	ka	12762.458812
625	ka	14511.124974
645	ka	16469.277289

Literaturverzeichnis

[1] Venturini Zilli, M. Complexity of the Unification Algorithm for first-order Expressions. Calcolo 12, 4 (Oct.-Dec. 1975), 361-372.

[2] G. Busatto, M. Lohrey, and S. Maneth. Efficient memory representation of XML documents. In Proceedings of DBPL 2005, volume 3774 of LNCS, pages 199-216, 2005.

[3] M. Lohrey and S. Maneth. The complexity of tree automata and XPath on grammar-compressed trees. In Proc. of the 10th CIAA ?05, 2005.

[4] Franz Baader, Tobias Nipkow. Term rewriting and all that, Cambridge Univ. Press, 1999

[5] Franz Baader, Wayne Snyder. Unification Theory, Chapter Eight of Handbook of Automated Deduction, Springer Verlag, Berlin (2001)

[6] DOV M.Gabbay, C.J.Hogger, J.A.Robinson. Handbook of Logic in Artificial Intelligence and Logic Programming, Volume 2 Deduction Methodologies, Chapter 2 Unification theory, Oxford Clarendon Press (1994)

[7] J. A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. Journal of the ACM. 1965 ACM Press

[8] T.Kutsia. Introductory Course at ESSLLI 2007, 19th European Summer School in Logic, Language and Information Trinity College, Dublin, Ireland. 6–17 August, 2007 (http://www.risc.uni-linz.ac.at/people/tkutsia/esslli2007; Syntactic Unification)

[9] Wayne Snyder. A Proof Theory for General Unification, Birkhäuser, Boston, Basel, Berlin, 1991

[10] Solving equations in abstract algebras: A rule-based survey of unification, in L.-J. Lassez and G. Plotkin, eds, Computational Logic: Essays in Honor of A. Robinson, MIT Press, Cambridge, MA, 1991

[11] J. Herbrand, Recherches sur la theorie de la demonstration, in W.D.Goldfarb, ed., Logical Writings, Reidel, Dordrecht, 1971

[12] A. Martelli, U. Montanari. An efficient Unification Algorithm, ACM Transactions on Programming Languages and Systems 4(2) 258-282

[13] A. Gascon, G. Godoy and Manfred Schmidt-Schauß. Unification with Singleton Tree Grammars.

[14] J. Hopcroft, J. Ullman, Einführung in die Automatentheorie, formale Sprachen und Komplexitätstheorie Addison-Wesley, 1988

[15] M. Schmidt-Schauß, Polynomial Equality Testing for Terms with Shared Substructures. Technical Report Frank-21, Research Group for Artificial Intelligence and Software Technology, Institut für Informatik,J.W.Goethe-Universität Frankfurt, November 2005

[16] Wojciech Plandowski. Testing equivalence of morphisms in context-free languages. In ESA 94, volume 855 of Lecture Notes in Computer Science,pages 460-470, 1994.

[17] Harold Abelson, Gerald Jay Sussman , Julie Sussman. Structure an Interpretation of Computer Programs The Mit Press, Cambridge, Massachusetts, London, England

[18] Schmidt-Schauß, Manfred. Skript zur Vorlesung Praktische Informatik, Kapitel 3, im Sommersemester 2005. http://www.ki.informatik.uni-frankfurt.de

[19] Hal Daume III, Yet Another Haskell Tutorial, 2002-2006

[20] Schmidt-Schauß, Manfred. Skript zur Vorlesung Einführung in die funktionale Programmierung, Kapitel 2, im Wintersemester 2007/08. http://www.ki.informatik.uni-frankfurt.de

[21] Y. Lifshits. Processing compressed texts: A tractability border. In CPM 2007, number 4580 in LNCS, pages 228?240, 2007.

[22] A. Gascón, G. Godoy, and M. Schmidt-Schauß. Context matching for compressed terms. In 23rd IEEE LICS, pages 93-102, 2008.

[23] Manuel M.T. Chakravarty, Gabriel C. Keller. Einführung in die Programmierung mit Haskell, Pearson Studium 2004.

[24] S. Thompson. Haskell, The Craft of Functional Programming, Prentice Hall 1987.

[25] Simon L. Peyton Jones. The Implementation of Functional Programming Languages, Addison-Wesley Second Edition.

[26] Die offizielle Haskell-Hompage, http://www.haskell.org/