

Bachelorarbeit

# Syntaktische und semantische Analyse von Hyperkanteneretzungsgrammatiken zur Heapabstraktion

Syntactic and Semantic Analysis of Hyperedge  
Replacement Grammars for Heap Abstraction

Gereon Kremer  
Matrikelnummer: 288911

29. September 2011

Gutachter:  
Priv.-Doz. Dr. Thomas Noll  
Prof. Dr. Ir. Joost-Pieter Katoen



# Erklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit selbständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten und nicht veröffentlichten Schriften entnommen sind, sind als solche kenntlich gemacht.

Aachen, den 29. September 2011

(Gereon Kremer)



# Inhaltsverzeichnis

<b>1</b>	<b>Einführung</b>	<b>1</b>
1.1	Motivation und Ziel der Arbeit . . . . .	1
<b>2</b>	<b>Formale Grundlagen</b>	<b>3</b>
2.1	Allgemeine Notationen . . . . .	3
2.2	Hypergraphen . . . . .	4
2.2.1	Grundlegende Definition . . . . .	4
2.2.2	Hypergraphen zur Heapabstraktion . . . . .	6
2.3	Hyperkantenersetzungsgrammatiken . . . . .	8
<b>3</b>	<b>Parsen</b>	<b>15</b>
3.1	Formale Sprachen . . . . .	15
3.1.1	Reguläre Sprachen . . . . .	16
3.1.2	Kontextfreie Sprachen . . . . .	17
3.2	Lexikalische Analyse . . . . .	18
3.2.1	Eindeutigkeit der Zerlegung . . . . .	20
3.2.2	Konstruktion des Lexers als DEA . . . . .	21
3.3	Syntaktische Analyse . . . . .	23
3.3.1	LL(1) vs. LL(2) . . . . .	25
3.3.2	LL(*) . . . . .	27
3.4	Erweiterung von kontextfreien Grammatiken . . . . .	28
<b>4</b>	<b>Fehlerüberprüfung</b>	<b>29</b>
4.1	Zusätzliche Informationen . . . . .	29
4.2	Hyperkantenersetzungsgrammatik . . . . .	30
4.3	Heapabstraktionsgrammatik . . . . .	30
4.3.1	Produktivität . . . . .	31
4.3.2	Wachstum . . . . .	31
4.3.3	Typisierung . . . . .	32
<b>5</b>	<b>Implementierung</b>	<b>33</b>
5.1	Definition der Grammatik . . . . .	33
5.2	Spezifikation des Lexers . . . . .	35
5.3	Spezifikation des Parsers . . . . .	37
5.4	Spezifikation des Baumparsers . . . . .	40
5.5	Implementierung der Fehlerüberprüfungen . . . . .	41
5.5.1	Terminal- und Nichtterminalsymbole . . . . .	42

## Inhaltsverzeichnis

5.5.2	Graphzusammenhang . . . . .	43
5.5.3	Knotennamen . . . . .	44
5.5.4	Knotentypen . . . . .	44
5.5.5	Knotentypisierung . . . . .	45
5.5.6	Reduktionstentakel . . . . .	46
5.5.7	Produktivität . . . . .	48
5.5.8	Wachstum . . . . .	50
5.5.9	Typisierung . . . . .	51
5.6	Weitere Klassen . . . . .	53
5.6.1	Löser für Systeme von Folgerungen . . . . .	53
5.6.2	Abhängigkeiten zwischen <code>Observer</code> -Objekten . . . . .	56
5.6.3	Hinzufügen zum Grammatikobjekt . . . . .	58
5.6.4	Schnittstelle zum Nutzer . . . . .	59
<b>6</b>	<b>Fazit</b>	<b>63</b>
6.1	Zusammenfassung . . . . .	63
6.2	Ausblick . . . . .	63
	<b>Literaturverzeichnis</b>	<b>65</b>

# 1 Einführung

Die Verifikation als Ansatz, um die Korrektheit von Programmen zu beweisen, ist eine der zentralen Anwendungen der theoretischen Informatik. Endliche und endlich repräsentierbare Systeme können häufig mithilfe von Model Checking [BK08] verifiziert werden. Zeigerbasierte Datenstrukturen, wie sie häufig im Heap vorliegen, sind jedoch oft nicht endlich und daher mit Model Checking schwer zu handhaben.

Um auf solche Programme und Datenstrukturen trotzdem das Konzept des Model Checkings anwenden zu können, gibt es zwei zentrale Ansätze. Während das *Infinite State Model Checking* versucht Model Checking auch auf unendlichen Strukturen durchzuführen, versucht die *Abstraktion* unendliche Datenstrukturen auf abstrakte Weise darzustellen um damit eine endliche Gesamtstruktur zu erhalten.

In dieser Arbeit wird der Ansatz der auf Graphen basierten Abstraktion verfolgt. Dabei wird prinzipiell der gesamte Heap als Graph abgebildet und dann Teile des Graphen in einer einzelnen Kante zusammengefasst [HNR10, JHKN11]. Auf diesem, teilweise abstrakten, Graphen können dann herkömmliche Verfahren zum Model Checking, beispielsweise durch eine Modellierung mit LTL oder CTL [BK08], durchgeführt werden.

Die Basis der Abstraktion bilden bei diesem Ansatz Grammatiken, die die erlaubten Abstraktionsschritte definieren, also die Ersetzung eines Teils des Graphen durch eine Kante. Von einem gewissen Startgraphen ausgehend kann mithilfe der Grammatik eine deutlich kompaktere Darstellung der Datenstrukturen erstellt werden.

Die Verifikation wird auf dem abstrahierten Graphen durchgeführt, wobei benötigte Teile des Graphen durch Rückwärtsanwendung der Grammatikregeln wieder zu konkreten Datenstrukturen umgewandelt werden können. Hierbei findet eine durch die Abstraktion gegebene Überapproximation statt. Da einige Informationen über die genaue Struktur der Daten bei einer Abstraktion verloren gehen, müssen anschließend alle möglichen konkreten Strukturen verifiziert werden.

Diese Technik wird vom Juggernaut-Framework [HNR10] implementiert, welches in dieser Arbeit verstärkt betrachtet und erweitert wird. Juggernaut steht dabei, dem Ansatz entsprechend, für **J**ust Use **G**raph **GR**ammars to **N**icely **A**btract **U**nbounded **s**tructures.

## 1.1 Motivation und Ziel der Arbeit

Während Juggernaut bei der eigentlichen Abstraktion bereits gute Fortschritte erzielt, bleibt der Vorgang des Einlesens einer Grammatik noch weit hinter den bereits theoretisch erzielten Erkenntnissen zurück. Das Ziel dieser Arbeit ist es deshalb, einen flexibel

## 1 Einführung

erweiterbaren und benutzerfreundlichen Mechanismus zum Einlesen neuer Grammatiken zu implementieren und in das bestehende Framework zu integrieren.

Aufgrund der bisher wenig spezifischen und zum Teil schwer verständlichen Fehlermeldungen, sollte die Grammatik fehlerhaft sein, war die Konstruktion für jemanden, der mit den Interna des Ansatzes nicht sehr gut vertraut ist, sehr aufwendig. Die neue Lösung sollte daher nur die Eingabe der nötigsten Informationen erfordern und verschiedene Fehlerquellen dediziert abprüfen, um möglichst genaue Fehlermeldungen ausgeben zu können.

Da die Anforderungen an eine Grammatik gut formalisiert sind, wird in dieser Arbeit zunächst diese Formalisierung vorgestellt. Beginnend mit dem Begriff eines Hypergraphens werden all jene Voraussetzungen erläutert, die von einer Grammatik gefordert werden.

Anschließend werden die möglichen Fehler bei der Spezifikation einer Grammatik vorgestellt und konkrete Vorgehensweisen zum Erkennen dieser Fehler beschrieben. Gleichzeitig werden einige Informationen, die bisher manuell angegeben werden mussten, nun automatisch aus der Grammatik ermittelt. Damit lassen sich mögliche Fehler bei der Angabe einer Grammatik verhindern.

Im Zuge der Vereinfachung wurde auch die Syntax der Grammatik als kontextfreie Sprache formalisiert und gegenüber der bisherigen Sprache deutlich vereinfacht. Ein großer Teil der Arbeit wird mithilfe des ANTLR Parsergenerators (ANother Tool for Language Recognition) durchgeführt und das Aussehen der Syntax durch dessen Eigenheiten beeinflusst. Daher wird auch  $LL(*)$ , eine von ANTLR implementierte Erweiterung des  $LL(k)$ -Parsings, in Abschnitt 3.3.1 beschrieben.

Vor einem kurzen Fazit wird die Implementierung des Parsers und der Fehlerprüfungen sowie Möglichkeiten, weitere Eigenschaften der Grammatik zu bestimmen, vorgestellt. Der vorgestellte Quelltext beschränkt sich zwar auf einen Pseudocode mit einer zu Python ähnlichen Syntax, der lauffähige Quelltext in Java kann jedoch auf der beiliegenden CD oder unter <http://moves.rwth-aachen.de/i2/juggrnaut/> eingesehen werden.

## 2 Formale Grundlagen

Der von Juggernaut [HNR10] verfolgte Ansatz zur Verifikation basiert auf einer Formalisierung – nahezu – beliebiger Datenstrukturen im Heap, die als Graphen modelliert werden. Kanten stehen dabei für konkrete Zeiger oder abstrakte Platzhalter für größere Datenstrukturen.

Um eine endliche Darstellung einer möglicherweise unendlichen Datenstruktur zu erhalten, können Teilgraphen durch Platzhalter ersetzt werden. Um die Verifikation auf dieser endlichen Darstellung durchzuführen, können für die Platzhalter auch wieder Graphen eingesetzt werden. Mit diesen Ersetzungen werden ähnlich zum Modell der kontextfreien Sprachen Grammatiken definiert, so dass sich ein analoger Begriff von Terminal- und Nichtterminalsymbolen sowie Sprachen über Graphen ergibt.

Im Laufe dieses Kapitels soll das Modell dieser Grammatiken umfassend eingeführt werden, um darauf aufbauend eine Sprache zur Spezifikation solcher Grammatiken zu entwickeln. Diese kann dann eingelesen und auf mögliche Fehler geprüft werden.

### 2.1 Allgemeine Notationen

Im Folgenden sollen einige Notationen eingeführt werden, die in diesem und folgenden Kapiteln genutzt werden.

**Definition 2.1** (Sequenzen). *Eine Sequenz über einer Menge  $\Sigma$  bezeichnet eine endliche Folge von Elementen aus  $\Sigma$ , also eine Folge aus  $\Sigma \times \dots \times \Sigma$ . Eine Sequenz kann leer sein, dann wird sie mit  $\varepsilon$  bezeichnet. Die Elemente einer Sequenz unterliegen einer durch die Sequenz gegebenen Nummerierung, so dass das  $k$ 'te Element einer Sequenz  $w$  mit  $w(k)$  bezeichnet wird.*

*Die Zugehörigkeit von  $a \in \Sigma$  zu einer Sequenz  $w$  unabhängig von der Reihenfolge in  $w$  kann durch  $a \in w$  ausgedrückt werden. Die Länge einer Sequenz  $w$  wird als  $|w|$  geschrieben.*

**Definition 2.2** (Endlicher Abschluss). *Der endliche Abschluss von einer Menge  $\Sigma$ , geschrieben  $\Sigma^*$ , bezeichnet die Menge der Sequenzen, die durch Konkatenation der Elemente aus  $\Sigma$  gebildet werden können. Die leere Sequenz  $\varepsilon$  ist hierbei grundsätzlich ein Element von  $\Sigma^*$ .*

**Definition 2.3** (Spezielle Funktionen). *Sei eine Funktion  $f : A \rightarrow B$  gegeben. Für eine Sequenz  $x = x_1 \dots x_n \in A^*$  ist  $f(x)$  definiert als  $f(x) = f(x_1 \dots x_n) = f(x_1) \dots f(x_n)$ , im Falle  $x = \varepsilon$  gilt  $f(x) = \varepsilon$ .*

## 2 Formale Grundlagen

Die Identitätsfunktion über einer Menge  $M$  wird mit  $id_M$  angegeben. Die Einschränkung von  $f$  auf den Definitionsbereich  $A'$ , geschrieben  $f' = f \upharpoonright A'$  wird definiert als  $f' : A' \cap A \rightarrow B, a \mapsto f(a)$ .

Um unnötige und wiederholte Erklärungen zu vermeiden, werden einige Formelzeichen als im Kontext bekannt verwendet.

**Definition 2.4** (Formelzeichen). *Die folgenden Formelzeichen werden ohne explizite Einführung gemäß der angegebenen Definition verwendet.*

Definition 2.5:

*Typisiertes Alphabet  $\Lambda$ , Knotentypen  $T$ , Symbole  $\Sigma_N$ , Nichtterminalsymbole  $N$ , Terminalsymbole  $\Sigma$ , Typisierung von Symbolen  $types$ , Rang  $rank$ .*

Definition 2.6:

*Knoten  $V$ , Kanten  $E$ , Benennungsfunktion  $label$ , Typisierungsfunktion  $type$ , Kantenfunktion  $att$ , externe Knoten  $ext$ .*

## 2.2 Hypergraphen

Es soll zunächst die hier verwendete Version von Graphen, die Hypergraphen, definiert werden, sowie die Anschauung von Hypergraphen im Kontext der Heapabstraktion erläutert werden.

Anschließend wird der Formalismus der Hyperkantenersetzung sowie der einer Grammatik basierend auf diesen Ersetzungen vorgestellt. In Kapitel 5 wird dann genau dieser Formalismus genutzt, um solche Ersetzungsgrammatiken einzulesen und in eine für Juggernaut verwendbare Form überführen.

### 2.2.1 Grundlegende Definition

Um in der Modellierung eines Heaps auch den Typ eines Objekts oder einer Variablen berücksichtigen zu können, wird ein typisiertes Alphabet verwendet.

**Definition 2.5** (Typisierte Alphabete). *Sei  $\Sigma_N = \Sigma \cup N$  ein Alphabet aus Terminalsymbolen  $\Sigma$  und Nichtterminalsymbolen  $N$ , wobei  $\Sigma \cap N = \emptyset$ .*

*Ein typisiertes Alphabet  $\Lambda = (\Sigma_N, T, types)$  ist ein Tupel aus dem Alphabet  $\Sigma_N$ , einer Menge von Typen  $T$  und einer Typisierungsfunktion  $types : \Sigma_N \rightarrow T^*$ .*

*Der Rang eines Symbols  $a \in \Sigma_N$  ist gegeben durch  $rank(a) = |types(a)|$ .*

Diese Form der Typisierung stellt den aktuellen Stand von Juggernaut dar, wird aber wahrscheinlich durch eine Typhierarchie erweitert werden [HJ11]. Hier wird diese Typhierarchie nicht verwendet, sondern stets die Gleichheit der Typen gefordert. Das entstandene Modell weist dadurch einige Einschränkungen auf, sollte sich aber zur Verwendung einer Typhierarchie erweitern lassen.

Hypergraphen stellen eine verallgemeinerte Form der Graphen dar. Während normalerweise Kanten als ein Paar von Knoten definiert sind, kann eine Hyperkante, im Folgenden nur Kante, an beliebig vielen Knoten anliegen – insbesondere auch an nur einem Knoten.

**Definition 2.6** (Hypergraph). *Ein Hypergraph  $H = (V, E, label, type, att, ext)$  über dem typisierten Alphabet  $\Lambda$  ist ein Tupel aus Knoten  $V$  und Kanten  $E$ , der Benennungsfunktion  $label : E \rightarrow \Sigma_N$ , der Typisierungsfunktion  $type : V \rightarrow T$ , der Kantenfunktion  $att : E \rightarrow V^*$  und einer Sequenz externer Knoten  $ext \in V^*$ .*

*Es wird gefordert, dass  $types(label(e)) = type(att(e))$  für alle  $e \in E$  gilt, damit gilt implizit auch  $rank(label(e)) = |att(e)|$ . Statt  $rank(label(e))$  ist auch die kürzere Notation  $rank(e)$  erlaubt. Alle Knoten, die nicht durch  $ext$  als extern gekennzeichnet sind, heißen intern.*

*Ein Tupel  $(X, i)$  eines Symbols  $X \in \Sigma_N$  und  $i \in \{1, \dots, rank(X)\}$  heißt Tentakel. Der Begriff wird sowohl für die Tentakel eines Symbols aus  $\Sigma_N$ , als auch für die Tentakel einer Kanten selber verwendet.*

*Die Menge aller Hypergraphen über  $\Lambda$  wird mit  $HG_\Lambda$  bezeichnet.*

Ein Hypergraph besteht aus Knoten aus der Menge  $V$  und Kanten aus der Menge  $E$ . Die  $label$ -Funktion benennt alle Kanten, so dass diese mit den Symbolen aus  $\Lambda$  und den zugehörigen Typsequenzen identifiziert werden können. Falls  $label(e) \in N$  ist, so heißt die Kante  $e$  Nichtterminalkante, anderenfalls gilt  $label(e) \in \Sigma$  und  $e$  heißt Terminalkante.

Die Kantenfunktion  $att$  liefert zu jeder Kante die inzidenten Knoten. Durch die Darstellung dieser Knoten als Sequenz liefert  $att$  zudem eine implizite Nummerierung dieser Knoten. Ist  $e \in \Sigma$  eine Terminalkante, so wird zusätzlich  $rank(e) = 2$  gefordert. Tentakel bezeichnen die Verbindungen eines Nichtterminals zu den jeweils anliegenden Knoten.

**Beispiel 2.1.** *Sei zunächst ein typisiertes Alphabet  $\Lambda$  gegeben mit*

$$\begin{aligned} \Sigma &= \{\mathbf{p}, \mathbf{n}\} & N &= \{L\} \\ T &= L & \Sigma_N &= \Sigma \cup N \\ types = A &\mapsto \begin{cases} LL & \text{falls } A \in \{\mathbf{p}, \mathbf{n}\} \\ LLL & \text{falls } A = L \end{cases} & \Lambda &= (\Sigma_N, T, types) \end{aligned}$$

*Sei außerdem der Hypergraph  $H = (V, E, label, type, att, ext)$  über  $\Lambda$  gegeben mit:*

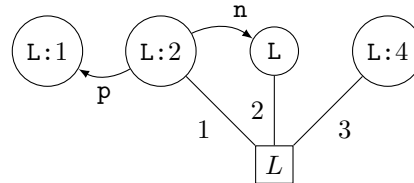
$$\begin{aligned} V &= \{v_1, \dots, v_4\} & type = v &\mapsto L \forall v \in V \\ E &= \{e_p, e_n, e_L\} & ext &= v_1 v_2 v_4 \\ label = e &\mapsto \begin{cases} \mathbf{p} & \text{falls } e = e_p \\ \mathbf{n} & \text{falls } e = e_n \\ L & \text{falls } e = e_L \end{cases} & att = e &\mapsto \begin{cases} v_2 v_1 & \text{falls } e = e_p \\ v_2 v_3 & \text{falls } e = e_n \\ v_2 v_3 v_4 & \text{falls } e = e_L \end{cases} \end{aligned}$$

## 2 Formale Grundlagen

Das typisierte Alphabet  $\Lambda$  enthält zwei Terminale  $\mathbf{n}$  und  $\mathbf{p}$  sowie ein Nichtterminal  $L$ ,  $L$  ist der einzige Typ in  $T$ . Während  $\mathbf{n}$  und  $\mathbf{p}$  die Typsequenz  $LL$  haben, liegen Kanten des Typs  $L$  an drei Knoten des Typs  $L$  an.

Der Graph enthält vier Knoten sowie drei Kanten. In Abbildung 2.1 ist der Hypergraph  $H$  graphisch dargestellt.

Abbildung 2.1: Beispiel für einen Hypergraphen



Die Knoten  $v_1$  bis  $v_4$  sind als Kreise von links nach rechts und die Terminalkanten als einfache gerichtete Kanten dargestellt. Die Nichtterminalkante  $L$  hingegen ist als Rechteck gezeichnet. Die Knoten enthalten ihren Typen als Beschriftung sowie zusätzlich ihre Position in  $ext$ , falls sie dort enthalten sind. Jede Kante  $e$  ist mit ihrem Typen  $label(e)$  gekennzeichnet.

Jedes Tentakel  $(X, i)$  ist als ungerichtete Kante zwischen der Nichtterminalkante des Typs  $X$  und dem anliegenden Knoten dargestellt und mit  $i$  beschriftet.

Da zwei Hypergraphen bereits verschieden sind, wenn die Benennung der Kanten sich ändert, soll nun der Begriff der Isomorphie für Hypergraphen eingeführt werden. Im Folgenden wird nicht zwischen isomorphen Hypergraphen unterschieden.

**Definition 2.7** (Isomorphie von Hypergraphen). *Zwei Hypergraphen sind isomorph, falls sie bis auf Umbenennung von Knoten und Kanten identisch sind.*

Um im Folgenden zu garantieren, dass die repräsentierte Datenstruktur zusammenhängend bleibt, wird der formale Begriff des Graphzusammenhangs benötigt. Dieser soll nun definiert werden.

**Definition 2.8** (Graphzusammenhang). *Ein Weg von einem Knoten  $v_1$  zu einem Knoten  $v_n$  ist eine Sequenz von Knoten  $v_2 \dots v_{n-1}$ , so dass für jedes  $k \in \{1, \dots, n-1\}$  eine Kante  $e_k$  existiert mit  $v_k, v_{k+1} \in att(e_k)$ .*

*Ein Hypergraph  $H$  heißt zusammenhängend, falls für jedes Paar von Knoten aus  $H$  ein Weg zwischen diesen Knoten existiert.*

### 2.2.2 Hypergraphen zur Heapabstraktion

Mit einem Hypergraphen über einem typisierten Alphabet  $\Lambda$  mit  $N = \emptyset$  kann ein konkreter Heap dargestellt werden. Eine einzelne Speicherstelle – oder ein Objekt – wird

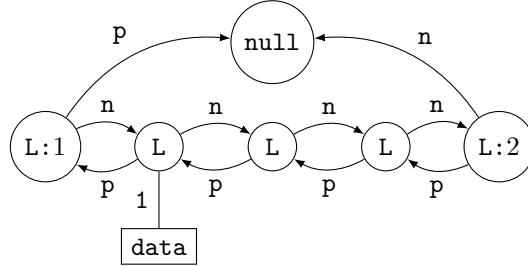
als Knoten modelliert. Globale Variablen  $Var_\Sigma$  werden als Kanten von Rang eins repräsentiert, die zu der entsprechenden Speicherstelle inzident sind, während Selektoren oder Zeiger  $Sel_\Sigma$  als Kanten mit Rang zwei modelliert werden.

Ein Zeiger weist in dieser Interpretation stets vom ersten zum zweiten Knoten. Variablennamen werden durch die Benennungsfunktion  $label$  den Kanten zugewiesen. Kanten mit anderem Grad werden bei der Darstellung eines konkreten Heaps nicht benötigt.

Für das Alphabet  $\Sigma$  gilt dementsprechend  $\Sigma = Var_\Sigma \cup Sel_\Sigma$ . Um Selektoren zu modellieren, die auf kein gültiges Objekt zeigen, gibt es den *Null*-Knoten. Wie in der Praxis haben Zeiger ohne gültiges Ziel den Wert `null`, zeigen also auf den *Null*-Knoten, der selber keine ausgehenden Kanten besitzt.

**Beispiel 2.2.** *Der Hypergraph in Abbildung 2.2 ist ein Beispiel für eine Repräsentation eines konkreten Heaps, der eine doppelt verkettete Liste von Elementen enthält. Knoten werden stets als Kreise und Kanten als Rechtecke abgebildet, Terminalkanten werden durch eine einfache gerichtete Kante dargestellt.*

Abbildung 2.2: Graphische Repräsentation eines Heaps.



Ein Knoten  $v$  ist mit seinem Typ  $type(v)$  beschriftet, externe Knoten zusätzlich mit ihrer Position in  $ext$ . Die Knoten in diesem Beispiel sind Listenknoten mit dem Typ  $L$ . Der Zeiger auf das nächste Element in der Liste ist  $n$ , der Zeiger auf das Vorherige  $p$ . Außerdem existiert eine Variable `data`, die durch ihr einziges Tentakel mit dem zweiten Listenknoten verbunden ist.

Die in Beispiel 2.2 skizzierte Darstellung einer Heapdatenstruktur unterliegt mehr Einschränkungen als durch die Definition der Hypergraphen gegeben ist, es darf bei Variablen und Selektoren nicht zu Namenskonflikten kommen. Dies wird durch den Begriff der Heapkonfiguration definiert.

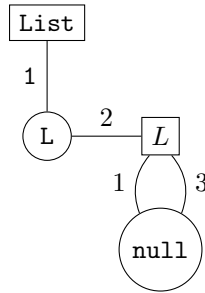
**Definition 2.9** (Heapkonfiguration). *Sei  $H \in HG_\Lambda$ .  $H$  heißt Heapkonfiguration, falls*

1.  $\forall a \in Sel_\Sigma, v \in V_H : |\{e \in E_H \mid att(e)(1) = v, label(e) = a\}| \leq 1$  und
2.  $\forall a \in Var_\Sigma : |\{e \in E_H \mid label(e) = a\}| \leq 1$

*Falls  $H$  Nichtterminalkanten enthält, so heißt  $H$  abstrakt, ansonsten konkret. Die Menge der Heapkonfigurationen über  $\Lambda$  heißt  $HC_\Lambda$ .*

**Beispiel 2.3.** Sei  $L$  eine Nichtterminalkante, die eine doppelt verkettete Liste repräsentiert. Die Heapkonfiguration in Abbildung 2.3 ist dann ein Beispiel für eine Variable, die eine doppelt verkettete Liste enthält. Die globale Variable  $list$  zeigt dabei auf den ersten Knoten der Liste. Weitere Elemente der Liste werden durch das Nichtterminal  $L$  repräsentiert. Die Tentakel jedes Nichtterminals werden dabei entsprechend ihrer Nummer beschriftet.

Abbildung 2.3: Graphische Repräsentation einer abstrakten Heapkonfiguration.



### 2.3 Hyperkantenersetzungsgrammatiken

Wie bereits beschrieben sollen zur Vereinfachung der Heapstruktur einzelne Datenstrukturen oder Teile des Heaps zu einer Nichtterminalkante zusammengefasst werden. Diese Ersetzung nennt man *Abstraktion*, da hier von der Struktur des Heaps abstrahiert wird. Diese Struktur kann zu Teilen wiederhergestellt werden, indem die Nichtterminalkante wieder durch einen Hypergraphen ersetzt wird, der Heap wird hier *konkretisiert*. Der dabei wiederhergestellte Heap ist stets nur eine Approximation des ursprünglichen Heaps, da bei der Abstraktion in der Regel Daten über die genaue Struktur verloren gehen.

Im Folgenden wird ein Formalismus eingeführt, welcher es erlaubt, für eine Nichtterminalkante einen Hypergraphen einzusetzen. Dieser hat Ähnlichkeit zu kontextfreien Grammatiken, da auch hier die Ersetzung einer Kante unabhängig vom Kontext dieser Kante geschieht. Dieser Formalismus ist, da er auf Hypergraphen und nicht auf Wörtern operiert, deutlich ausdrucksstärker als der Formalismus der kontextfreien Grammatiken [Roz97, Kapitel 2.5].

**Definition 2.10** (Hyperkantenersetzungsgrammatiken, HRG). Eine Hyperkantenersetzungsgrammatik (Hyperedge Replacement Grammar – HRG) über dem typisierten Alphabet  $\Lambda$  ist eine Menge von Ersetzungsregeln der Form  $X \rightarrow H$  mit  $X \in N$  und  $H \in HG_\Lambda$  zusammenhängend, so dass  $types(X) = type(ext_H)$ . Dadurch gilt implizit auch  $rank(X) = |ext_H|$ .

Die Menge aller solcher Grammatiken über  $\Lambda$  heißt  $HRG_\Lambda$ .

Im Unterschied zu kontextfreien Grammatiken ist der Startgraph nicht Bestandteil der Grammatik. Daher wird dieser in der Anwendung als Parameter angegeben.

### 2.3 Hyperkantenersetzungsgrammatiken

Die oben erwähnte *Konkretisierung* entspricht einer Regelanwendung einer solchen Grammatik, die *Abstraktion* entsprechend einer umgekehrten Anwendung einer Regel. Damit durch eine Konkretisierung die Datenstruktur nicht getrennt wird und durch eine Abstraktion nicht verschiedene Datenstrukturen zusammengeführt werden können, wird für alle rechten Regelseiten der Graphzusammenhang gefordert. Dies wird in Beispiel 2.6 veranschaulicht.

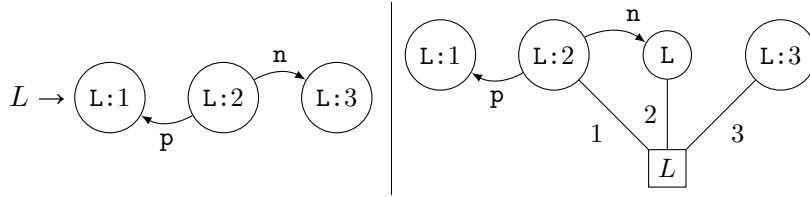
Als Beispiel für eine Hyperkantenersetzungsgrammatik sollen wieder verkettete Listen dienen. Das gezeigte Nichtterminal  $L$  kann im vorher gezeigten abstrakten Heap verwendet werden. Statt der rein formalen Notation wird hier stets die in Beispiel 2.4 verwendete, graphische Notation genutzt.

**Beispiel 2.4.** Sei  $L \in N$  ein Nichtterminal. In Abbildung 2.4 werden zwei Ersetzungsregeln  $L \rightarrow H_1$  und  $L \rightarrow H_2$  dargestellt. Die linke Regelseite  $L$  kann also durch einen der beiden Hypergraphen  $H_1$  oder  $H_2$  ersetzt werden.

Die erste Regel  $L \rightarrow H_1$  stellt dabei eine Art Terminalregel dar. In  $H_1$  sind keinerlei Nichtterminalkanten enthalten, es werden lediglich die Selektoren des zweiten externen Knotens erzeugt.

Die zweite Regel  $L \rightarrow H_2$  hingegen erzeugt neben den Selektoren des zweiten externen Knotens einen weiteren Listenknoten und eine weitere Nichtterminalkante  $L$ .

Abbildung 2.4: Hyperkantenersetzungsgrammatik für eine doppelt verkettete Liste



Bei der Ersetzung einer Nichtterminalkante in einem Hypergraphen  $H$  durch einen Hypergraphen  $H'$  werden die zu der Kante inzidenten Knoten mit den externen Knoten von  $H'$  identifiziert. Die Nichtterminalkante wird dann aus  $H$  entfernt und alle internen Knoten und alle Kanten aus  $H'$  zu  $H$  hinzugefügt.

**Definition 2.11** (Hyperkantenersetzung). Seien  $H, G \in HG_\Lambda$  Hypergraphen über dem typisierten Alphabet  $\Lambda$  und eine Nichtterminalkante  $e \in E_H$  gegeben und es gelte  $\text{type}(\text{att}_H(e)) = \text{type}(\text{ext}_G)$ . O.B.d.A. seien  $V_H \cap V_G = E_H \cap E_G = \emptyset$ .

Die Ersetzung von  $e$  durch  $G$  ist dann definiert als  $H[e/G] = F \in HG_\Lambda$  mit

$$\begin{aligned} V_F &= V_H \cup (V_G \setminus \text{ext}_G), & E_F &= (E_H \setminus \{e\}) \cup E_G, \\ \text{label}_F &= \text{label}_H \upharpoonright E_F \cup \text{label}_G, & \text{type}_F &= \text{type}_H \upharpoonright V_F \cup \text{type}_G, \\ \text{ext}_F &= \text{ext}_H, & \text{att}_F &= \text{att}_H \upharpoonright E_F \cup \text{mod} \circ \text{att}_G \end{aligned}$$

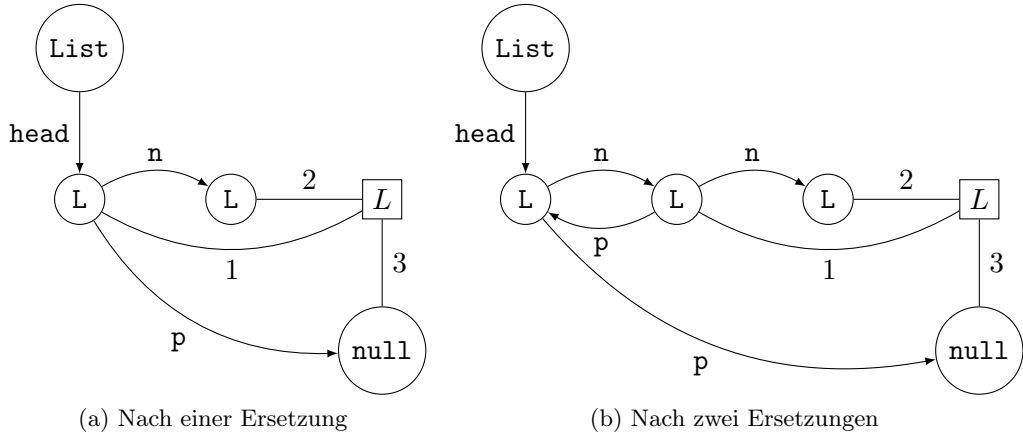
mit  $\text{mod} = \text{id}_{V_F} \cup \{\text{ext}_G(i) \mapsto \text{att}_H(e)(i) \mid i = 1 \dots \text{rank}(e)\}$ .

## 2 Formale Grundlagen

Auch dieser Vorgang soll mit dem bereits bekannten Hypergraph für eine Liste aus Beispiel 2.3 und einer Regel aus Beispiel 2.4 veranschaulicht werden.

**Beispiel 2.5.** *Es sei die Grammatik für das Nichtterminal  $L$  aus Beispiel 2.4 und der abstrakte Heap  $H$  aus Abbildung 2.3 gegeben. Wird nun die zweite Regel der Grammatik zweimal angewendet, so ergeben sich sukzessiv die Heaps aus Abbildung 2.5. Formal wird die Ersetzung  $H[e/H_L]$  angewendet, wobei  $e$  die einzige Nichtterminalkante mit  $\text{label}(e) = L$  bezeichnet. Die Forderung  $\text{rank}(L) = |\text{ext}_{H_L}|$  ist jeweils erfüllt.*

Abbildung 2.5: Heapkonfiguration nach der Ersetzung



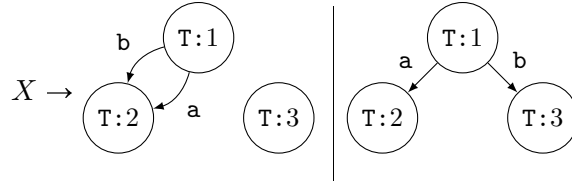
Da die Anwendung einer Ersetzung mit dieser Notation recht sperrig ist, wird nun der Begriff der Ableitung eingeführt. Eine Ableitung bezeichnet die Anwendung einer Ersetzung auf einen Hypergraphen.

**Definition 2.12** (HRG Ableitung). *Seien  $G \in \text{HRG}_\Lambda$ ,  $H, H' \in \text{HG}_\Lambda$ ,  $p = X \rightarrow K \in G$  und  $e \in E_H$  mit  $\text{label}(e) = X$ .  $H'$  ist von  $H$  mit  $p$  ableitbar, falls  $H'$  isomorph ist zu  $H[e/K]$ . Diese Ableitung wird mit  $H \xrightarrow{e,p} H'$  bezeichnet. Es gilt  $H \Rightarrow_G H'$ , falls  $H \xrightarrow{e,p} H'$  für ein  $e \in E_H$  und ein  $p \in G$ .*

Die transitive Hülle über Ableitungen aus  $G$  wird mit  $H \Rightarrow_G^* H'$  bezeichnet.

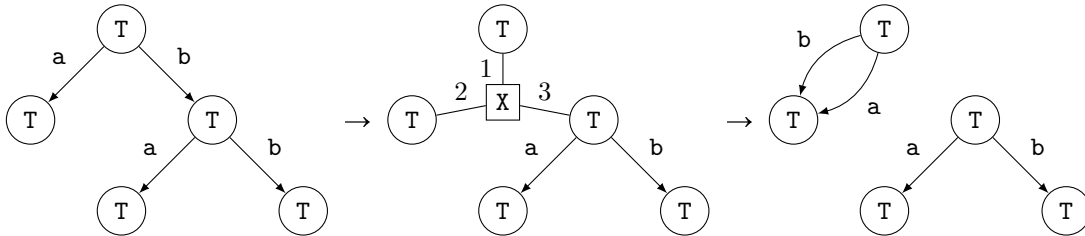
Durch die Anwendung einer solchen Ableitung wird nun deutlich warum gefordert wird, dass die rechten Regelseiten zusammenhängend sein müssen. Dies zeigt das folgende Beispiel 2.6. Diese Forderung stellt eine Einschränkung dar, falls Datenstrukturen im zu verifizierenden Programm potentiell unendlich oft getrennt oder zusammengeführt werden, vereinfacht aber die spätere Arbeit mit der Grammatik enorm.

**Beispiel 2.6.** *Sei eine Regel für das Nichtterminal  $X$  mit einem nicht zusammenhängenden Hypergraphen auf der rechten Regelseite wie in Abbildung 2.6 gegeben.*

Abbildung 2.6: Regel für  $X$  mit nicht zusammenhängender rechter Regelseite


Wird diese Grammatik nun angewendet, so kann es, wie in Abbildung 2.7 gezeigt, zu ungewollten Verfälschungen der Datenstruktur kommen. Im linken Ausgangsgraphen ist ein zusammenhängender Baum gegeben, dieser ist nach einer Abstraktion und der anschließenden Konkretisierung im Endgraphen auf der rechten Seite getrennt.

Abbildung 2.7: Anwendung der Regel auf einen Hypergraphen



Mit dem Begriff einer Grammatik und einer Ableitung kann nun die Sprache einer Grammatik auf einem Ausgangsgraphen eingeführt werden.

**Definition 2.13** (Sprache einer HRG). Sei  $G \in HRG_\Lambda$  und  $H \in HG_\Lambda$ . Die Sprache  $L_G(H)$  ist definiert als  $L_G(H) = \{K \in HG_\Sigma \mid H \Rightarrow_G^* K\}$ . Ist  $G$  im Kontext eindeutig, kann hier ebenso  $L(H)$  statt  $L_G(H)$  geschrieben werden.

Da im Folgenden auch die Sprache eines Nichtterminals von Interesse ist, wird nun ein kanonischer Hypergraph für ein Nichtterminal  $X$ , genannt *Handle*, eingeführt.

**Definition 2.14** (Handle). Sei  $X \in N$  mit  $\text{rank}(X) = n$ . Das  $X$ -Handle ist der Hypergraph  $X^\bullet$ , definiert als  $X^\bullet = (V, E, \text{label}, \text{type}, \text{att}, \text{ext})$  mit

$$\begin{aligned} V &= \{v_1, \dots, v_n\} & E &= \{e\} \\ \text{label} &= e \mapsto X & \text{type} &= \text{types}(\text{label}(e)) \\ \text{att} &= e \mapsto v_1 \dots v_n & \text{ext} &= v_1 \dots v_n \end{aligned}$$

Die Sprache einer Grammatik ausgehend von einem Nichtterminal  $X$  ist somit durch  $L(X^\bullet)$  gegeben. Für abstrakte Heapkonfigurationen  $H \in HC_{\Sigma_N}$  enthält  $L(H)$  alle aus  $H$  ableitbaren konkreten Hypergraphen.

Für einen beliebige Hyperkantenersetzungsgrammatik ist jedoch nicht gewährleistet, dass  $L(H')$  für eine Heapkonfiguration  $H'$  nur gültige Heapkonfigurationen enthält. Um dieses Problem zu lösen wird nun der Begriff der Datenstrukturgrammatik eingeführt.

**Definition 2.15** (Datenstrukturgrammatiken). *Eine Grammatik  $G \in HRG_\Lambda$  heißt Datenstrukturgrammatik (DSG) über dem typisierten Alphabet  $\Lambda$ , falls für alle Nichtterminale  $X$  gilt, dass  $L(X^\bullet) \subseteq HC_\Sigma$ . Die Menge aller DSG über  $\Lambda$  wird mit  $DSG_\Lambda$  bezeichnet.*

Das Problem, ob für eine gegebene HRG  $G \in HRG_\Lambda$  gilt, dass  $G \in DSG_\Lambda$ , ist entscheidbar [JHKN11]. Bevor mit den Heapabstraktionsgrammatiken die Grammatiken definiert werden können, die auch in den folgenden Kapiteln verwendet werden, müssen noch die folgenden Begriffe definiert werden.

Von besonderem Interesse sind ausgehende Kanten eines Knotens in einer konkreten Heapkonfiguration. Sie bezeichnen alle Terminalkanten, die von dem jeweiligen Knoten ausgehen, also durch ihr erstes Tentakel mit ihm verbunden sind.

**Definition 2.16** (Ausgehende Kanten). *Sei  $H \in HC_\Sigma$  eine konkrete Heapkonfiguration mit  $v \in V_H$ . Die Menge der ausgehenden Kanten von  $v$  ist definiert als*

$$out(v) = \{label(e) \mid e \in E_H, att(e)(1) = v\}$$

Für die weitere Verwendung der Grammatik in Juggernaut wird zudem die Information benötigt, ob an einem bestimmten Tentakel mithilfe einer Ersetzungsregel von dem anliegenden Knoten ausgehende Terminalkanten erzeugt werden können. Ist dies nicht der Fall, so heißt ein solches Tentakel Reduktionstentakel.

**Definition 2.17** (Reduktionstentakel). *Ein Tentakel  $(X, i)$  heißt Reduktionstentakel, falls für alle  $H \in L(X^\bullet)$  gilt, dass  $out(ext_H(i)) = \emptyset$ .*

In Beispiel 2.4 sind sowohl  $(L, 1)$  als auch  $(L, 3)$  Reduktionstentakel, da keine ausgehenden Kanten an den externen Knoten 1 oder 3 erzeugt werden können. Das Tentakel  $(L, 2)$  ist hingegen kein Reduktionstentakel, da die ausgehenden Kanten  $n$  und  $p$  in beiden Regeln erzeugt werden.

Damit kann nun der Begriff der Heapabstraktionsgrammatik eingeführt werden.

**Definition 2.18** (Heapabstraktionsgrammatiken). *Eine Grammatik  $G \in DSG_\Lambda$  ist eine Heapabstraktionsgrammatik (HAG) über dem typisierten Alphabet  $\Lambda$ , falls die folgenden Eigenschaften erfüllt sind:*

1. *produktiv:  $\forall X \in N : L(X^\bullet) \neq \emptyset$*
2. *wachsend:  $\forall X \rightarrow H \in G : |E_H| \leq 1 \Rightarrow H \in HG_\Sigma$*
3. *typisiert:  $G$  ist gemäß Definition 2.19 typisiert.*
4. *lokal konkretisierbar*

*Die Menge aller HAGs über  $\Lambda$  wird mit  $HAG_\Lambda$  bezeichnet.*

Die Produktivität ist analog zur Produktivität von kontextfreien Grammatiken definiert. Ein Nichtterminal ist produktiv, falls jede Nichtterminalkante dieses Nichtterminals zu mindestens einer konkreten Heapkonfiguration abgeleitet werden kann.

Ist eine Grammatik nicht produktiv, so kann es zu Situationen kommen, in denen von der aktuellen Hyperkonfiguration keine konkrete Heapkonfiguration erreicht werden kann. In diesem Fall ist entweder keine Regel anwendbar oder es kommt zu einer endlosen Wiederholung von Regeln, die stets wieder Nichtterminalkanten erzeugen.

Eine Grammatik ist wachsend, wenn jede Regel etwas an der Heapkonfiguration ändert in dem Sinne, dass entweder eine neue Terminalkante erzeugt oder die Anzahl der Kanten erhöht wird. Die einzigen Regeln, die diese Eigenschaft verletzen, bilden ein Nichtterminal auf einen Hypergraphen mit nur einer Nichtterminalkante ab, da Hypergraphen mindestens eine Kante beinhalten müssen, um keine isolierten Knoten zu bilden.

Da der Hypergraph und damit seine einzige Hyperkante die selbe Typsequenz wie das Nichtterminal haben muss, ist das Ergebnis nach einer Anwendung dieser Regel isomorph zum ursprünglichen Hypergraphen. Muss also die Umgebung eines bestimmten Knotens konkretisiert werden, kann eine solche Regel in eine Endlosschleife führen und wird daher verboten.

Die Typisierung einer Grammatik ist gegeben, falls alle Regeln eines Nichtterminals stets externe Knoten mit identischen Typsequenzen erzeugen.

**Definition 2.19** (Typisierung). *Eine DSG  $G \in DSG_\Lambda$  ist typisiert, falls für jedes Nichtterminal  $X \in N$  und jedes Tentakel  $(X, i)$  von  $X$  gilt, dass für alle  $H_1, H_2 \in L(X^\bullet)$   $T(X, i) := out_{H_1}(ext_{H_1}(i)) = out_{H_2}(ext_{H_2}(i))$  gilt.*

Die Menge  $T(X, i)$  bezeichnet damit die Menge der Terminalsymbole für das  $i$ 'te Tentakel des Nichtterminals  $X$ . Bei einer Konkretisierung muss für jedes dieser Terminalsymbole an dem jeweiligen externen Knoten eine solche Terminalkante erzeugt werden.

Ist eine Grammatik lokal konkretisierbar, so müssen die ausgehenden Kanten jedes beliebigen Knotens in einem Schritt konkretisiert werden können. Dies wird durch die Transformation der Grammatik in eine lokale Greibach Normalform [JHKN11] erreicht. Dabei werden aus den vorhandenen Regeln weitere Regeln erzeugt, so dass für jedes Tentakel jedes Nichtterminals eine Teilmenge von Regeln existiert, mit denen alle ausgehenden Kanten des jeweiligen externen Knotens erzeugt werden können.

Während die Frage, ob eine gegebene Grammatik lokal konkretisierbar ist, unentscheidbar ist [JHKN11], garantiert die lokale Greibach Normalform diese Eigenschaft. Die Unentscheidbarkeit folgt aus der Unentscheidbarkeit der Sprachinklusion für Sprachen von Hyperkantenersetzungsgrammatiken.

Diese Heapabstraktionsgrammatiken stellen die formale Basis für Juggernaut dar. Der in späteren Kapiteln beschriebene Parser soll genau solche Grammatiken erkennen.



## 3 Parsen

Vor der eigentlichen Verarbeitung von Daten steht stets das Einlesen und das anschließende Strukturieren dieser Daten. Hier soll eine in Textform angegebene Heapabstraktionsgrammatik eingelesen, auf mögliche Fehler geprüft und für die Verifikation mit Juggernaut genutzt werden. Da die Struktur dieser Grammatik hinreichend komplex ist, wird hier mit ANTLR [Par07] auf einen Parsergenerator für nahezu beliebige kontextfreie Grammatiken zurückgegriffen.

Der Begriff Parser bezeichnet im Allgemeinen ein Programm, welches unstrukturierten Text anhand einer gewissen Spezifikation – meistens einer kontextfreien Grammatik – strukturiert. Entspricht die Eingabe nicht der Spezifikation, so wird sie mit einem entsprechenden Fehler verworfen. Im Folgenden wird stets angenommen, dass sich die Eingabe strukturieren lässt, so dass lediglich das Vorgehen zum Erzeugen einer strukturierten Datenstruktur behandelt wird.

Der Prozess des Parsens wird in zwei Phasen aufgeteilt, die lexikalische und die syntaktische Analyse. Die lexikalische Analyse zerlegt die Eingabe in elementare Bestandteile, genannt Symbole. Dies sind in der Regel Namen, Operatoren oder Zahlen. Diese Symbole entsprechen den Terminalsymbolen der kontextfreien Grammatik.

Die syntaktische Analyse strukturiert diese Symbole dann mithilfe der Grammatik. Hier gibt es zwei alternative Ansätze, das *Top-Down-Parsing* (LL-Parsing) und das *Bottom-Up-Parsing* (LR-Parsing). *LL* und *LR* stehen hierbei für die Art der Ableitungen: **L**eft **t**o **R**ight, **L**eftmost derivation und **L**eft to **R**ight, **R**ightmost derivation.

Da in dieser Arbeit mit dem ANTLR-Framework [Par07] gearbeitet wird, welches einen LL-Parser erzeugt, wird im Folgenden nur das Top-Down-Parsing betrachtet. ANTLR bietet als Parsergenerator den Komfort, dass lediglich die zu erkennenden Symbole sowie die zu verwendende Grammatik definiert werden muss. Der eigentliche Programmcode, also Lexer und Parser, wird von ANTLR automatisch generiert und muss nicht mehr angepasst werden.

### 3.1 Formale Sprachen

Als Basis für die lexikalische und syntaktische Analyse dienen die regulären Sprachen und kontextfreien Sprachen. Vor der Definition dieser Sprachen sollen zunächst die Begriffe eines Wortes und einer Sprache eingeführt werden.

**Definition 3.1** (Wörter und Sprachen). *Ein Wort  $w$  über einem Alphabet  $\Sigma$  ist eine Sequenz aus Symbolen  $a \in \Sigma$ , also  $w \in \Sigma^*$ .*

Eine Sprache  $L$  bezeichnet eine endliche oder unendliche Teilmenge  $L \subseteq \Sigma^*$ . Eine Sprache heißt endlich, falls  $|L| \in \mathbb{N}$ .

### 3.1.1 Reguläre Sprachen

Die Klasse der regulären Sprachen stößt bereits schnell an ihre Grenzen, da sie bereits Sprachen wie beispielsweise die Sprache der korrekt geklammerten Ausdrücke nicht mehr enthält. Trotzdem ist sie von großer Relevanz, da sie insbesondere im Zusammenhang mit der Automatentheorie einige eindrucksvolle Resultate wie den Satz von Büchi, Elgot, Trakhtenbrot [Bü60, Elg61, Tra57] liefert und darüber hinaus sehr angenehme Abschlusseigenschaften aufweist.

Reguläre Sprachen werden beim Parsen in der lexikalischen Analyse verwendet. Die verschiedenen elementaren Bestandteile der Eingabe werden stets Elemente regulärer Sprachen sein, so dass die Eingabe in verschiedene Elemente verschiedener regulärer Sprachen aufgeteilt werden muss.

Reguläre Sprachen werden über reguläre Ausdrücke definiert, sowohl die Definition von regulären Ausdrücken als auch der entsprechenden Sprachen erfolgt induktiv über den Aufbau der Ausdrücke.

**Definition 3.2** (Reguläre Ausdrücke und Sprachen). *Ein regulärer Ausdruck  $r$  über einem Alphabet  $\Gamma$  hat eine der folgenden Formen:*

- $\varepsilon$
- $a$  für ein  $a \in \Gamma$
- $(r_1)^*$ , den endlichen Abschluss über einem regulären Ausdruck  $r_1$
- $r_1 \cdot r_2$ , die Verkettung von regulären Ausdrücken  $r_1$  und  $r_2$
- $r_1 + r_2$ , die Vereinigung von regulären Ausdrücken  $r_1$  und  $r_2$

Für einen regulären Ausdruck  $r$  ist die Sprache  $L(r)$  definiert als:

- $\emptyset$ , falls  $r = \varepsilon$
- $\{a\}$ , falls  $r = a \in \Gamma$
- $L(r_1)^*$ , falls  $r = (r_1)^*$
- $L(r_1) \cdot L(r_2)$ , falls  $r = r_1 \cdot r_2$
- $L(r_1) \cup L(r_2)$ , falls  $r = r_1 + r_2$

Die regulären Sprachen sind genau die Sprachen, die über einen regulären Ausdruck definiert werden können.

Die Menge aller regulären Ausdrücke über dem Alphabet  $\Gamma$  wird mit  $\mathfrak{R}(\Gamma)$  bezeichnet.

Um Wörter effizient als Wörter einer regulären Sprache zu identifizieren, die als regulärer Ausdruck gegeben ist, werden endliche Automaten verwendet. Man unterscheidet zwischen deterministischen und nichtdeterministischen Automaten, die allerdings gleich mächtig sind.

Zudem ist eine Umwandlung eines nichtdeterministischen in einen deterministischen Automaten immer möglich, wenn auch mit exponentieller Laufzeit verbunden [HMU02, Kapitel 2.3.5]. Daher wird im Folgenden stets mit deterministischen Automaten gearbeitet.

**Definition 3.3** (Deterministische endliche Automaten). *Ein deterministischer endlicher Automat (DEA)  $\mathfrak{A}$  ist definiert als ein Tupel  $\mathfrak{A} = (\Gamma, Q, F, q_0, \delta)$  über einem Eingabealphabet  $\Gamma$ , einer Menge von Zuständen  $Q$ , einer Menge von Endzuständen  $F \subseteq Q$ , einem Startzustand  $q_0 \in Q$  und einer Übergangsfunktion  $\delta : Q \times \Gamma \rightarrow Q$ .*

Sei  $\hat{\delta} : Q \times \Gamma^* \rightarrow Q$  definiert als

- $\hat{\delta}(q, \varepsilon) = q$
- $\hat{\delta}(q, wa) = \delta(\hat{\delta}(q, w), a)$  für alle  $a \in \Gamma, w \in \Gamma^*$

Die Sprache  $L(\mathfrak{A})$  ist definiert als  $L(\mathfrak{A}) = \{w \mid \hat{\delta}(q_0, w) \in F\}$ . Man sagt, der Automat  $\mathfrak{A}$  erkennt  $L(\mathfrak{A})$ .

Ein Zustand  $q \in Q$  heißt produktiv, falls ein  $w \in \Gamma^*$  existiert, so dass  $\hat{\delta}(q, w) \in F$ . Die Menge aller produktiven Zustände wird mit  $P \subseteq Q$  bezeichnet.

Algorithmen zur Konstruktion eines Automaten  $\mathfrak{A}$  aus einem regulären Ausdruck  $r$ , so dass  $L(\mathfrak{A}) = L(r)$  ist, sind in der Informatik bekannt [HMU02, Kapitel 3.2], daher werden diese hier nicht weiter behandelt.

### 3.1.2 Kontextfreie Sprachen

Die kontextfreien Sprachen bilden eine mächtigere Sprachklasse als die regulären Sprache und werden in der syntaktischen Analyse verwendet. Sie sind über eine Menge von Ersetzungsregeln definiert, die nicht vom Kontext des ersetzten Symbols abhängig sein dürfen.

**Definition 3.4** (Kontextfreie Grammatiken). *Eine kontextfreie Grammatik  $G$  ist definiert als ein Tupel  $G = (N, T, S, R)$  mit einer Menge von Nichtterminalsymbolen  $N$ , einer Menge von Terminalsymbolen  $T$ , einem Startsymbol  $S \in N$  sowie einer Menge von Regeln  $R \subseteq \{N \rightarrow (N \cup T)^*\}$ .*

Die Kontextfreiheit der Regeln ergibt sich daraus, dass die linke Regelseite nur aus einem einzigen Nichtterminal besteht. Könnten dort weitere Symbole stehen, so könnte beispielsweise die Regel  $aX \rightarrow ab$  das Nichtterminal  $X$  durch das Terminal  $b$  ersetzen, falls das  $X$  hinter einem  $a$  steht. So können also weitere Bedingungen an den Kontext eines Nichtterminals gestellt werden. Die Klasse der kontextsensitiven Sprachen ist jedoch oft unbrauchbar, bereits das Leerheitsproblem, also die Frage ob die Grammatik mindestens ein Wort erzeugt, ist unentscheidbar [AB03, Kapitel 9.2].

**Definition 3.5** (Ableitungen, Sprachen). *Eine Ableitung ist eine Anwendung einer Regel  $A \rightarrow \alpha \in R$  auf eine Symbolsequenz  $\beta = \beta_1 A \beta_2 \in (N \cup T)^*$ , geschrieben  $\beta \rightarrow_G \beta_1 \alpha \beta_2$ . Eine Linksableitung  $\xrightarrow{l}_G$  bezeichnet eine Ableitung mit  $\beta_1 \in T^*$ , eine*

Rechtsableitung  $\xrightarrow{r}_G$  entsprechend eine Ableitung mit  $\beta_2 \in T^*$ . Eine Folge von Ableitungen mit beliebigen Regeln wird mit  $\alpha \rightarrow_G^* \beta$  notiert.

Die Sprache  $L(G)$  ist definiert als  $L(G) = \{w \in T^* \mid S \rightarrow_G^* w\}$ . Die kontextfreien Sprachen sind genau die Sprachen, die über eine kontextfreie Grammatik definiert werden können.

Zur kompakteren Schreibweise können zwei Regeln  $A \rightarrow \alpha$  und  $A \rightarrow \beta$  zu einer neuen Regel  $A \rightarrow \alpha \mid \beta$  zusammengefasst werden. Das Symbol  $\mid$  übernimmt hier die logische Bedeutung einer Disjunktion.

**Definition 3.6** (Produktive Nichtterminale). *Ein Nichtterminal  $A$  heißt produktiv, falls es möglich ist, aus  $A$  ein Terminalwort abzuleiten, also  $A \rightarrow^* w, w \in T^*$ , ansonsten heißt  $A$  unproduktiv.*

Unproduktive Nichtterminale können mitsamt aller sie enthaltenden Regeln aus einer Grammatik  $G$  entfernt werden, da für die resultierende Grammatik  $G'$  gilt  $L(G) = L(G')$ . Ohne Beschränkung der Allgemeinheit wird im Folgenden angenommen, dass jede Grammatik nur produktive Nichtterminale enthält. [HMU02, Kapitel 7.1.1]

Ist ein Wort Teil einer kontextfreien Sprache, so kann es durch eine Folge von Ableitungen aus dem Startsymbol der Grammatik erzeugt werden. Werden die Ableitungen auf Links- oder Rechtsableitungen beschränkt, so ist die Folge der Ableitungen sogar eindeutig bestimmt. Ein Wort kann also eindeutig als eine Folge von Ableitungen oder Regeln repräsentiert werden. Wie im folgenden Beispiel 3.1 dargestellt, wird diese Folge von Ableitungen häufig als Baum notiert.

**Beispiel 3.1.** *Sei  $G = (N, T, S, R)$  eine kontextfreie Grammatik mit  $N = \{S, P, A, D\}$ ,  $T = \{0, 1\}$  und  $R = \{S \rightarrow P + S \mid P, P \rightarrow A * P \mid A, A \rightarrow (S) \mid D, D \rightarrow 0 \mid 1 \mid DD\}$ . Die von der Grammatik erzeugte Sprache  $L(G)$  kann als die Sprache der arithmetischen Ausdrücke mit der Addition und Multiplikation über den Binärzahlen interpretiert werden.*

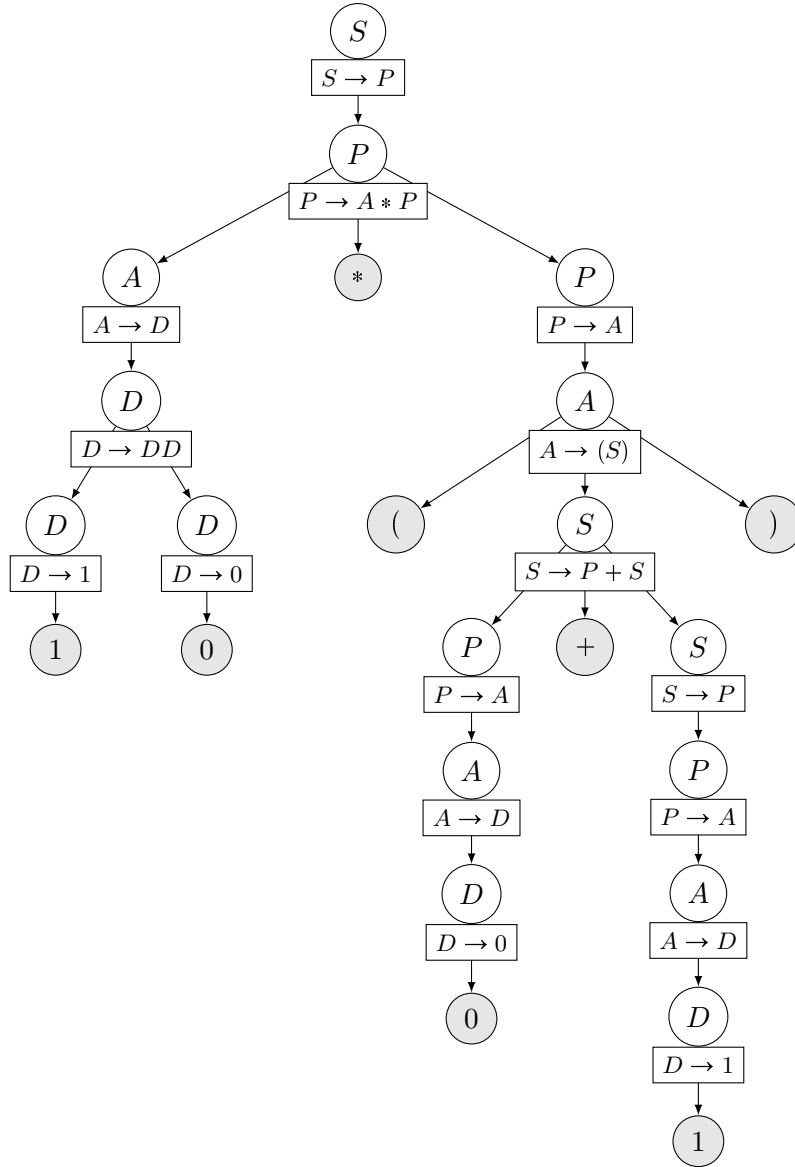
*Das Wort  $10 * (0 + 1)$  ist Teil der von  $G$  erkannten Sprache, die Ableitungen, um aus  $S$  dieses Wort zu erzeugen, sind in Abbildung 3.1 in Baumform dargestellt.*

## 3.2 Lexikalische Analyse

Die lexikalische Analyse erstellt aus einem Eingabewort eine Sequenz von Symbolen, die als Eingabe für den Parser dienen. Die lexikalische Analyse wird von dem sogenannten Lexer durchgeführt, der die Eingabe in logische zusammengehörige Einheiten einteilt, beispielsweise eine Zahl oder eine Zeichenkette.

Der Lexer wird durch die Angabe von Token und dazugehörigen regulären Sprachen spezifiziert. Die Token geben dabei die möglichen Typen der Symbole an, in die die Eingabe zerlegt werden soll. Diese Teile der Eingabe werden als Lexeme bezeichnet. Ein Symbol besteht aus einem Token und einem Lexem, wobei das Lexem in der zum Token gehörigen Sprache enthalten sein muss. Die Aufgabe des Lexers ist es, eine Sequenz von Symbolen zu finden, deren Lexeme konkateniert genau das Eingabewort ergeben.

Abbildung 3.1: Ableitungen für das Wort  $10 * (0 + 1)$



**Definition 3.7** (Token, Symbol, Lexem). *Die Menge der Token wird mit  $T$  bezeichnet. Jedem Token  $t \in T$  ist eine reguläre Sprache zugeordnet, geschrieben  $L(t)$ . Lexeme bezeichnen Teile des Eingabewortes. Ein Symbol ist dann ein Tupel  $S = (t, l)$  mit einem Lexem  $l \in L(t)$ . Eine Zerlegung eines Eingabewortes  $w$  bezeichnet eine Sequenz von Symbolen  $S_1 \dots S_n$  mit  $S_i = (t_i, l_i)$  und  $l_1 \cdot \dots \cdot l_n = w$ .*

Ein solcher Lexer wird mit einem endlichen Automaten umgesetzt. Sind für alle Token  $t_i$  die regulären Sprachen  $L(t_i)$  gegeben, so lassen sich daraus deterministische endliche Automaten  $\mathfrak{A}_i = (\Sigma, Q_{\mathfrak{A}_i}, F_{\mathfrak{A}_i}, q_{\mathfrak{A}_i}, \delta_{\mathfrak{A}_i})$  mit  $L(\mathfrak{A}_i) = L(t_i)$  konstruieren. Sei nun  $\mathfrak{A} = (\Sigma, Q, F, q_0, \delta)$  mit dem Eingabealphabet  $\Sigma$ , den Zuständen  $Q = \bigcup_i Q_{\mathfrak{A}_i}$  und Endzuständen  $F = \bigcup_i F_{\mathfrak{A}_i}$  sowie  $\delta = \{(q_0, \varepsilon) \mapsto q_{\mathfrak{A}_i}\} \cup \bigcup_i \delta_{\mathfrak{A}_i}$ . Dieser Automat ist nichtdeterministisch, wird aber mit der Potenzmengenkonstruktion [HMU02, Kapitel 2.3.4] determinisiert.

Der Lexer beginnt nun mit der Simulation des Eingabewortes auf dem Automaten  $\mathfrak{A}$ . Befindet er sich nach dem Lesen eines Teilwortes in einem Endzustand, so kann das gelesene Teilwort als Lexem gespeichert werden. In diesem Fall wird der Automat auf den Anfangszustand zurückgesetzt und die Simulation fortgesetzt. Die Zerlegung ist vollständig, wenn das Eingabewort vollständig verarbeitet wurde.

### 3.2.1 Eindeutigkeit der Zerlegung

Die Zerlegung eines beliebigen Eingabewortes ist allerdings nicht von vorneherein eindeutig. Ein Lexem kann zum einen in den Sprachen zweier verschiedener Token vorkommen, zum anderen könnten zwei aufeinanderfolgende Lexeme zusammen ein anderes, gültiges Lexem bilden. Da eine Eindeutigkeit jedoch wünschenswert ist, wird sie durch zwei weitere Bedingungen an einer Zerlegung erzwungen.

Haben zwei aufeinanderfolgende Symbole einer Zerlegung dasselbe Token, so ist es in der Regel sinnvoll, die beiden Symbole zu vereinen. Die Zahl 1234 kann beispielsweise in die Symbole (INT, 12), (INT, 34) zerlegt werden, wobei INT das Token für eine Zahl ist. Intuitiv ist klar, dass eine Zerlegung (INT, 1234) eher der Intention der Eingabe entspricht. Daher wird stets gefordert, das längstmögliche Lexem zu wählen.

Die Zugehörigkeit eines Lexems zu Sprachen verschiedener Token kann auf verschiedene Weisen gelöst werden. Wären alle Sprachen paarweise disjunkt, wäre das Problem gelöst. Dies würde in der Praxis allerdings große Unannehmlichkeiten bedeuten, beispielsweise bei der Unterscheidung zwischen Schlüsselwörtern wie `for` oder `class` von Variablennamen, die üblicherweise durch beliebige Zeichenketten bezeichnet werden. Die Angabe der Sprache aller Variablennamen ohne einzelne Schlüsselwörter in Form eines regulären Ausdrucks ist in der Praxis jedoch kaum möglich.

Daher wird in der Regel auf eine gegebene Ordnung der Token zurückgegriffen. Die Token werden in der Reihenfolge ihrer Definition innerhalb der Spezifikation geordnet. Kann ein Lexem zwei Token zugeordnet werden, so wird das kleinere, also vorher definierte, Token gewählt.

### 3.2.2 Konstruktion des Lexers als DEA

Diese beiden Regeln sorgen dafür, dass der Vorgang des Zerlegens nun eindeutig ist, es ergibt sich ein deterministischer Automat der wie im Folgenden beschrieben konstruiert wird. Die Endzustände des Automaten, der die Vereinigung der Sprachen aller Token erkennt, werden so partitioniert, dass jedem Endzustand ein Token zugewiesen wird. Erreicht der Automat mit einem Lexem einen Endzustand, so ist das Lexem in der Sprache dieses Tokens.

Zusätzlich wird der Automat zu einem Backtracking-Automaten erweitert, indem stets der letzte Zustand gespeichert wird, in dem sich der Automat in einem Endzustand befand. Sobald der Automat keinen weiteren Endzustand mehr erreichen kann, wird auf diesen gespeicherten Zustand zurückgegriffen, um stets das längste Lexem zu finden.

Algorithmus 3.1: Konstruktion eines Lexers als Backtracking-Automat

- 1 Erstelle DEA  $\mathfrak{A}_i$  für alle Token  $t_i \in T = \{t_1, \dots, t_n\}$  so dass  $L(t_i) = L(\mathfrak{A}_i)$
- 2 Erstelle Produktautomat  $\mathfrak{A}$  mit  $L(\mathfrak{A}) = \bigcup_{i=1}^n L(\mathfrak{A}_i)$
- 3 Partitioniere Endzustände, weise jedem  $q \in F$  ein  $t_i$  zu
- 4 Erweitere  $\mathfrak{A}$  zu Backtracking-Automaten

Der Produktautomat  $\mathfrak{A}$  hat durch seine Konstruktion Zustände der Form  $q \in Q_1 \times \dots \times Q_n$ . Einem Endzustand  $q \in F \subseteq Q$  soll nun genau dann das Token  $T(q) = t_i \in T$  zugewiesen werden, wenn  $L(t_i)$  die Sprache mit minimalem  $i$  ist, für die ein Wort  $w \in L(t_i)$  existiert, so dass  $q_0 \xrightarrow{w} q$ .

Formaler ausgedrückt wird  $T(q) = t_i$  definiert, falls  $q \in Q_1 \setminus F_1 \times \dots \times Q_{i-1} \setminus F_{i-1} \times F_i \times Q_{i+1} \times \dots \times Q_n$ . Der Index  $i$  ist also so gewählt, dass für alle  $k < i$  das Lexem  $l$  nicht in der Sprache des Tokens  $t_k$  ist,  $t_i$  somit das kleinste Token bezüglich der definierten Reihenfolge mit  $l \in L(t_i)$ .

Auf Basis dieser Partitionierung wird nun der Backtracking-Automat formal eingeführt und erklärt. Dieses Modell wird dann alle Anforderungen an einen Lexer erfüllen. Er akzeptiert eine Eingabe  $w \in \Sigma^*$  und generiert daraus eine Sequenz von Token  $t_i \in T$  unter den bereits genannten Randbedingungen.

**Definition 3.8** (Backtracking-Automaten). *Seien ein deterministischer endlicher Automat  $\mathfrak{A} = (\Sigma, Q, F, q_0, \delta)$  und die Menge der Token  $T$  und Sprachen  $L(t), t \in T$  gegeben. Ein Backtracking-Automat  $\mathfrak{B}$  ist definiert als  $\mathfrak{B} = (\Sigma, Q, F, q_0, M, I, O, \tau)$  mit der Menge der Modi  $M = \{N\} \cup T$ , dem Eingabeband  $vqw \in \Sigma^* \times Q \times \Sigma^*$ , der Ausgabe  $W \in T^* \cdot \{\varepsilon, ERROR\}$  und den Transitionen  $\tau$ .*

*Eine Konfiguration hat die Form  $(m, vqw, W) \in M \times I \times O$ . Die Anfangskonfiguration für eine Eingabe  $w \in \Sigma^*$  ist definiert als  $(N, q_0w, \varepsilon)$ .*

*Die Transitionen  $\tau$  sind wie folgt mit  $q' = \delta(q, a)$  definiert mit  $P$  der Menge der produktiven Zustände.*

$$\begin{array}{lcl}
(N, qaw, W) & \mapsto & \left\{ \begin{array}{ll} (T(q'), q'w, W) & \text{falls } q' \in F \\ (N, q'w, W) & \text{falls } q' \in P \setminus F \\ \mathbf{output: } W \cdot \mathit{ERROR} & \text{falls } q' \notin P \end{array} \right. \\
(T, vqaw, W) & \mapsto & \left\{ \begin{array}{ll} (T(q'), q'w, W) & \text{falls } q' \in F \\ (T, vaq'w, W) & \text{falls } q' \in P \setminus F \\ (N, q_0vaw, W) & \text{falls } q' \notin P \end{array} \right. \\
(T, q, W) & \mapsto & \mathbf{output: } W \cdot T \quad \text{falls } q \in F \\
(N, q, W) & \mapsto & \mathbf{output: } W \cdot \mathit{ERROR} \quad \text{falls } q \in P \setminus F \\
(T, vaq, W) & \mapsto & (N, q_0va, WT) \quad \text{falls } q \in P \setminus F
\end{array}$$

Die verschiedenen Modi speichern das Token des zuletzt besuchten Endzustands und damit das Token des längsten bisher gefundenen Lexems. Der Modus  $m$  ist entweder  $m = N$ , falls noch kein Token gefunden wurde, oder  $m = t_i \in T$  falls  $t_i$  das zuletzt gefundene Token ist.

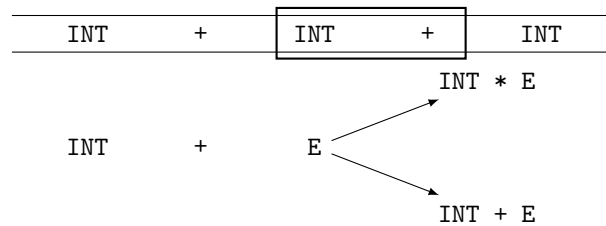
Das Eingabeband  $I$  enthält immer den aktuellen Zustand  $q$ , den noch zu lesenden Teil der Eingabe  $w$  und die schon gelesene Eingabe  $v$ , die im Falle eines Backtracking-Schritts wiederhergestellt werden muss. Die Ausgabe ist die Sequenz der erkannten Token. Tritt ein Fehler auf, so wird der Ausgabe zusätzlich das spezielle *ERROR*-Token angehängt.

Der erste Block von Transitionen ausgehend von der Konfiguration  $(N, qaw, W)$  bezeichnet einen Schritt, indem ein  $a \in \Sigma$  eingelesen wird, während noch kein Token erkannt wurde. Ist der neue Zustand ein Endzustand wird das erkannte Token gespeichert, indem der Modus auf  $T(q')$  gesetzt wird. Ist der neue Zustand zwar produktiv, aber kein Endzustand, so wird lediglich der Zustand entsprechend der durchgeführten  $a$ -Transition geändert. Ist der neue Zustand nicht produktiv, ist es nicht gelungen ein Token zu erkennen und es wird ein Fehler ausgegeben.

Die darauf folgenden Transitionen umfassen die Schritte, in denen ein  $a$  eingelesen wird, nachdem bereits ein Token erkannt wurde. Sollte außer der Senke kein weiterer Zustand erreicht werden können, muss ein Backtracking-Schritt ausgeführt werden.

Ist der neue Zustand ein Endzustand, so kann das bisher gespeicherte Token mit  $T(q')$  überschrieben und  $v$  gelöscht werden, da stets nach dem längsten Lexem gesucht wird. Ist der neue Zustand produktiv, aber kein Endzustand, so wird  $a$  an  $v$  angehängt. Ist der neue Zustand nicht produktiv, so wird ein Backtracking-Schritt ausgeführt: Das gespeicherte Token wird an die Ausgabe angehängt, der bereits gelesene Teil der Eingabe wieder zur Eingabe hinzugefügt und in den Anfangszustand gewechselt.

Die letzten Transitionen behandeln das Ende der Eingabe. Ist der aktuelle Zustand am Ende der Eingabe ein Endzustand, so wird das Token an die Ausgabe angehängt und diese ausgegeben. Ist der aktuelle Zustand kein Endzustand und es wurde noch kein Token erkannt, so wird ein Fehler ausgegeben. Wurde hingegen bereits ein Token erkannt, wird auch hier ein Backtracking-Schritt durchgeführt.

Abbildung 3.2: Parsingsituation für die Eingabe  $\text{INT} + \text{INT} + \text{INT}$ 

Tatsächlich müssen neben den Token auch die jeweiligen Lexeme gespeichert werden, so dass sich eine Sequenz von Symbolen ergibt. Da dies für das Verständnis jedoch unwesentlich ist, wird hier darauf verzichtet.

### 3.3 Syntaktische Analyse

Die syntaktische Analyse soll aus einer Sequenz von Symbolen eine Baumstruktur erstellen, die den Ableitungen des Eingabewortes in der gegebenen Grammatik entspricht. Als Ansatz wird hier das Top-Down-Parsing, auch *LL*-Parsing, verwendet.

Ein *LL*-Parser verwendet stets die Idee des rekursiven Abstiegs. Beginnend mit dem Startsymbol  $S$  der kontextfreien Grammatik werden nichtdeterministisch Linksableitungen vorgenommen bis das Wort aus der Grammatik den Token aus der Zerlegung des Lexers entspricht.

Um dieses Vorgehen deterministisch und damit praktisch durchführbar zu machen, muss der Parser in der Lage sein, weitere Informationen über die Regeln zu nutzen, um sich deterministisch für eine Regel zu entscheiden. Hierzu verwendet man einen sogenannten Lookahead, der es erlaubt, eine gewisse Anzahl an Symbolen voraus zu schauen, wie das folgende Beispiel 3.2 veranschaulicht. Die Größe des Lookaheads wird hierbei mit  $k$  bezeichnet.

**Beispiel 3.2.** In Abbildung 3.2 ist eine Situation während des Parsens der im oberen Teil angegebenen Eingabe  $\text{INT} + \text{INT} + \text{INT}$  dargestellt. Das Startsymbol wurde bereits zu der im unteren Teil angegebenen Sequenz  $\text{INT} + E$  abgeleitet, der Parser muss sich für das Nichtterminal  $E$  nun für eine der beiden Regeln  $E \rightarrow \text{INT} * E$  oder  $E \rightarrow \text{INT} + E$  entscheiden.

Der Lookahead der Größe  $k = 2$  ist durch den Kasten über der Eingabe gekennzeichnet und enthält die Symbole  $\text{INT}$  und  $+$ . Mithilfe des Lookaheads kann der Parser nun feststellen, dass die erste Regel nicht in Frage kommt, da das zweite Symbol der Regel  $*$  nicht zur Eingabe passt. Somit muss der Parser die zweite Regel wählen. Ein Lookahead von  $k = 1$  wäre hier nicht ausreichend, da das erste Symbol  $\text{INT}$  auf beide Regeln passen würde.

### 3 Parsen

Um den Lookahead systematisch zu nutzen, werden für jede Regel all diejenigen Terminalwörter bestimmt, die an der Stelle in der Eingabe stehen können, an der die Regel angewendet werden soll. Die Menge dieser Wörter wird Lookahead-Menge genannt.

Zunächst wird für jede rechte Regelseite  $\alpha$  eine sogenannte First-Menge  $fi_k(\alpha)$  berechnet. Diese enthält alle Wörter  $w \in T^*$  aus Terminalen mit  $|w| \leq k$ , die Präfix eines aus  $\alpha$  ableitbaren Wortes sind. Zusätzlich wird für jedes Nichtterminal  $X$  eine Follow-Menge  $fo_k(X)$  berechnet, die alle Wörter aus Terminalen der Länge maximal  $k$  enthält, die hinter einem Vorkommen von  $X$  ableitbar sind.

Diese Mengen werden zu Lookahead-Mengen  $LA_k(X \rightarrow \alpha)$  zusammengeführt, indem für jede Regel  $X \rightarrow \alpha$  die First-Menge  $fi_k(\alpha \cdot fo_k(X))$  berechnet wird.

**Definition 3.9** (Lookahead-Mengen). *Sei  $X$  ein Nichtterminal einer kontextfreien Grammatik  $G = (N, T, S, R)$  mit Startsymbol  $S$  und Terminalen  $T$  und sei  $k \in \mathbb{N}$  die Größe des Lookheads.*

*Für ein Wort  $\alpha$  ist die First-Menge als  $fi_k(\alpha) = \{w \in T^* \mid \alpha \rightarrow^* w\beta, |w| \leq k\}$  definiert. Die Follow-Menge  $fo_k(X)$  ist analog für jedes Nichtterminal  $X$  definiert als  $fo_k(X) = \{w \in T^* \mid S \rightarrow^* \alpha X w \beta, |w| \leq k\}$ . Die Lookahead-Menge  $LA_k(X \rightarrow \alpha)$  ist dann definiert als  $LA_k(X \rightarrow \alpha) = \{w \mid w \in fi_k(\alpha \cdot fo_k(X)) \wedge |w| \leq k\}$ .*

*Sind die Lookahead-Mengen für alle Regeln eines Nichtterminals paarweise disjunkt, so ist ein deterministisches Parsen mit einem Lookahead von  $k$  möglich. Die Menge aller Grammatiken, für die dies gilt, wird mit  $G \in LL(k)$  bezeichnet.*

Hierbei ist zu beachten, dass es Grammatiken gibt, die für kein  $k \in \mathbb{N}$  von einem  $LL(k)$ -Parser verarbeitet werden können. Einige dieser Grammatiken können mithilfe eines  $LL(*)$ -Parser verarbeitet werden. Diese Technik wird in Abschnitt 3.3.2 genauer vorgestellt. Zunächst wird aber davon ausgegangen, dass die Grammatik für ein  $k$  in  $LL(k)$  enthalten ist.

Sind die Lookahead-Mengen einmal berechnet, so kann der Parser das Eingabewort verarbeiten. Als Informationen stehen ihm jederzeit die Eingabe  $w = uv$  und das aktuell abgeleitete Wort  $\alpha = uX\beta$  zur Verfügung. Der Parser führt nun die in Algorithmus 3.2 beschriebenen Schritte so lange aus, bis  $\alpha = w$ .

#### Algorithmus 3.2: $LL$ -Parsing auf Basis von Lookahead-Mengen

- 1 Sei  $x$  das Präfix von  $v$  der Länge  $k$
- 2 Sei  $\alpha = uX\beta$
- 3 Suche Regel  $X \rightarrow \gamma$  mit  $x \in LA_k(X \rightarrow \gamma)$
- 4 Regel existiert:  $\alpha = u\gamma\beta$
- 5 Sonst  $w \notin L(G)$ , breche ab

Als Ergebnis erhält der Parser also eine Sequenz von Linksableitungen. Der Baum dieser Ableitungen stellt die syntaktische Struktur des Eingabewortes dar und kann zu weiteren semantischen Analysen der Eingabe verwendet werden.

### 3.3.1 LL(1) vs. LL(2)

Für den Parser ist es wichtig zu wissen, für welches  $k$  eine Grammatik in  $LL(k)$  enthalten ist, da die Lookahead-Mengen abhängig von  $k$  berechnet werden müssen. Es soll nun beleuchtet werden, worin sich die Klassen  $LL(k)$  für verschiedene  $k$  unterscheiden.

Die Unterscheidung von Grammatiken in die Klassen  $LL(k)$  für verschiedene  $k$  beschreibt vor allem die Mächtigkeit des Parser, die notwendig ist um eine solche Grammatik zu verwenden. Da mit einem größeren Lookahead prinzipiell auch alle Grammatiken verarbeitet werden können, die nur einen kleineren Lookahead benötigen, ergibt sich direkt, dass  $LL(k) \subseteq LL(k + 1)$  für alle  $k$ . Es gilt sogar  $LL(k) \subsetneq LL(k + 1)$  für alle  $k$ .

Für die Klassen  $LL(0)$ ,  $LL(1)$  und  $LL(2)$  soll nun eine kurze Vorstellung der Ausdrucksstärke folgen. Da die später vorgestellte Grammatik in  $LL(2)$  enthalten ist, werden die höheren Klassen nicht mehr betrachtet.

Da  $k$  die Größe des Lookheads beschreibt, enthält  $LL(0)$  all diejenigen Grammatiken, die sich ohne Lookahead mit einem LL-Parser analysieren lassen. Die Entscheidung, welche Ableitungsregel verwendet wird, muss also auf Basis von Lookahead-Mengen mit  $k = 0$  getroffen werden.

Da eine Lookahead-Menge nur Wörter der Länge maximal  $k = 0$  enthält, muss für jedes Nichtterminal  $N$  gelten, dass  $LA_0(N) = \emptyset$ . Die Auswahl der Regel muss also ohne Information möglich sein. Das ist nur möglich, wenn es genau eine Regel gibt. Ist dies der Fall, ist jeder Ableitungsprozess ohnehin eindeutig, die Sprache der Grammatik enthält also nur ein Wort. Es liegt nahe, solche Grammatiken als uninteressant zu deklarieren.

Die Klasse  $LL(1)$  hingegen erlaubt einen Lookahead der Größe eins, der für die meisten Anwendungen ausreicht. Beispielsweise können viele Programmiersprachen durch eine  $LL(1)$ -Grammatik definiert werden, etwa *PL/0* [BA95] oder *Modula-2* [Kow10].

Häufig kann eine Grammatik  $G \in LL(k)$  so umgeformt werden, dass die Sprache erhalten bleibt, aber ein Parser einen geringeren Lookahead benötigt. Dies ist jedoch schon deshalb nicht immer möglich, da  $LL(k) \subsetneq LL(k + 1)$  gilt. Bei der Umformung verändert sich allerdings in der Regel der syntaktische Aufbau. Daher haben solche Änderungen Konsequenzen für die anschließende Weiterverarbeitung der Ableitungen.

Als Beispiel seien nun zwei Grammatiken gegeben, die beide arithmetische Ausdrücke beschreiben. Während die Grammatik aus Beispiel 3.3 mit einem Lookahead von  $k = 1$  nicht auskommt – tatsächlich ist die Grammatik für kein  $k$  in  $LL(k)$  und nicht einmal in  $LL(*)$  – kann die Grammatik aus Beispiel 3.4 von einem  $LL(1)$ -Parser verarbeitet werden.

**Beispiel 3.3.** Sei die Grammatik  $G = (\{S, T, F\}, \{(\,), +, *, INT\}, S, R)$  mit Regeln  $R$ . Dann ergeben sich untenstehende First-, Follow- und Lookahead-Mengen.

### 3 Parsen

Regeln $R$	$N$	$fi_1(N)$	$fo_1(N)$	$R : N \rightarrow \alpha$	$LA_1(N \rightarrow \alpha)$
$S \rightarrow T \mid T + S$	$S$	$\{(\cdot, INT)\}$	$\{\varepsilon, \cdot\}$	$S \rightarrow T + S$	$\{(\cdot, INT)\}$
$T \rightarrow F \mid T * F$	$T$	$\{(\cdot, INT)\}$	$\{\varepsilon, \cdot, +, *\}$	$S \rightarrow T$	$\{(\cdot, INT)\}$
$F \rightarrow (S) \mid INT$	$F$	$\{(\cdot, INT)\}$	$\{\varepsilon, \cdot, +, *\}$	$T \rightarrow T * F$	$\{(\cdot, INT)\}$
				$T \rightarrow F$	$\{(\cdot, INT)\}$
				$F \rightarrow (S)$	$\{(\cdot)\}$
				$F \rightarrow INT$	$\{INT\}$

Der Schnitt der Lookahead-Mengen verschiedener Regeln für die Nichtterminale  $S$  und  $T$  ist nicht leer, die Grammatik ist daher nicht in  $LL(1)$ . Für das Wort  $INT + INT$  ist bei einem Lookahead von  $k = 1$  nicht entscheidbar, welche Regel vom Startsymbol  $S$  ausgehend gewählt werden soll.

**Beispiel 3.4.** Sei die Grammatik  $G = (\{S, S', T, T', F\}, \{(\cdot), +, *, INT\}, S, R)$  mit Regeln  $R$ . Dann ergeben sich die folgenden First-, Follow- und Lookahead-Mengen.

Regeln $R$	$N$	$fi_1(N)$	$fo_1(N)$	$R : N \rightarrow \alpha$	$LA_1(N \rightarrow \alpha)$
$S \rightarrow TS'$	$S$	$\{(\cdot, INT)\}$	$\{\varepsilon, \cdot\}$	$S \rightarrow TS'$	$\{(\cdot, INT)\}$
$S' \rightarrow +S \mid \varepsilon$	$S'$	$\{\varepsilon, +\}$	$\{\varepsilon, \cdot\}$	$S' \rightarrow +S$	$\{+\}$
$T \rightarrow FT'$	$T$	$\{(\cdot, INT)\}$	$\{\varepsilon, \cdot, +\}$	$S' \rightarrow \varepsilon$	$\{\varepsilon, \cdot\}$
$T' \rightarrow *FT' \mid \varepsilon$	$T'$	$\{\varepsilon, *\}$	$\{\varepsilon, \cdot, +\}$	$T \rightarrow FT'$	$\{(\cdot, INT)\}$
$F \rightarrow (S) \mid INT$	$F$	$\{(\cdot, INT)\}$	$\{\varepsilon, \cdot, +, *\}$	$T' \rightarrow *FT'$	$\{*\}$
				$T' \rightarrow \varepsilon$	$\{\varepsilon, \cdot, +\}$
				$F \rightarrow (S)$	$\{(\cdot)\}$
				$F \rightarrow INT$	$\{INT\}$

Für Regeln eines Nichtterminals sind alle Lookahead-Mengen disjunkt. Daher gilt im Gegensatz zum vorherigen Beispiel  $G \in LL(1)$ . Für das Wort  $INT + INT$  ist hier anhand der Lookahead-Mengen klar, dass das Wort wie folgt abgeleitet werden muss:

$$\begin{aligned}
 S &\rightarrow T S' && \rightarrow F T' S' && \rightarrow INT T' S' \\
 &\rightarrow INT S' && \rightarrow INT + S && \rightarrow INT + T S' \\
 &\rightarrow INT + F T' S' && \rightarrow INT + INT T' S' && \rightarrow INT + INT S' \\
 &\rightarrow INT + INT && && 
 \end{aligned}$$

Zwei wichtige Heuristiken, die in den beiden gezeigten Beispielen bereits Anwendung gefunden haben, um den benötigten Lookahead zu reduzieren, sollen nun vorgestellt werden. Zum einen können mit der Linksfaktorisierung Regeln mit identischem Präfix umgestellt werden, zum anderen können direkte Linksrekursionen – Regeln der Form  $X \rightarrow X\alpha$  – entfernt werden.

Bei der Linksfaktorisierung werden Regeln der Form  $A \rightarrow \alpha\beta \mid \alpha\gamma$  durch die Regeln  $A \rightarrow \alpha A'$ ,  $A' \rightarrow \beta \mid \gamma$  ersetzt. Der benötigte Lookahead wird dadurch verringert, dass die Entscheidung, welche Regel angewendet werden soll, herausgezögert wird bis  $\alpha$  bereits verarbeitet wurde. Diese Heuristik wurde auf die Regel  $S \rightarrow T + S \mid T$  aus Beispiel 3.4 angewandt.

Grammatiken mit Linksrekursionen sind generell für kein  $k$  in  $LL(k)$  enthalten. Direkte Linksrekursionen können entfernt werden, indem Regeln der Form  $A \rightarrow A\alpha_1 \mid \dots \mid A\alpha_n \mid \beta_1 \mid \dots \mid \beta_m$  durch die Regeln  $A \rightarrow \beta_1 A' \mid \dots \mid \beta_m A'$ ,  $A' \rightarrow \alpha_1 A' \mid \dots \mid \alpha_n A' \mid \varepsilon$  ersetzt werden.

Das Entfernen von indirekten Linksrekursionen hingegen ist deutlich aufwendiger, da dazu die Grammatik zunächst in Greibach Normalform umgewandelt werden muss. Dabei gehen nahezu alle syntaktischen Strukturen innerhalb der Grammatik verloren, was die darauffolgende semantische Analyse deutlich erschwert.

Die Linksfaktorisierung wurde auf die Regeln  $S \rightarrow T + S \mid T$  aus Beispiel 3.3 angewendet, in der Regel  $T \rightarrow T + F \mid F$  wurde die Linksrekursion entfernt. Wird dies nach den vorgestellten Heuristiken getan, entsteht genau die Grammatik aus Beispiel 3.4.

### 3.3.2 $LL(*)$

Tatsächlich implementiert ANTLR weder einen Parser für  $LL(1)$  noch für  $LL(2)$  sondern verwendet das erweiterte Konzept des  $LL(*)$ -Parsens [PF11]. Der Stern verdeutlicht dabei, dass diese Parsingstrategie mächtiger ist als  $LL(k)$ -Parsing, egal für welches  $k \in \mathbb{N}$ .  $LL(*)$  erlaubt nicht nur das Parsen von Grammatiken in  $LL(k)$  für ein beliebiges  $k \in \mathbb{N}$ , sondern auch gewisser anderer Grammatiken, wie das folgende Beispiel zeigt.

**Beispiel 3.5.** *Sei die folgende, verallgemeinerte Definition einer Klasse oder eines Interfaces in Java gegeben. Vor den Schlüsselwörtern `class` und `interface` sind jeweils Schlüsselwörter wie `public` oder `abstract` erlaubt. Eine naheliegende Grammatik für diese Definitionen wäre also die in Algorithmus 3.3 angegebene mit dem Startsymbol `def`.*

Algorithmus 3.3: Grammatik  $G \in LL(*)$

```

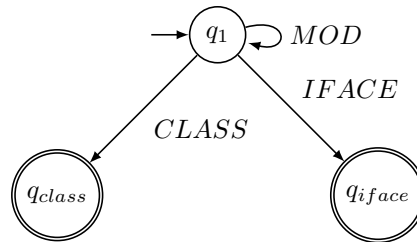
1 def   → class | iface
2 class → MOD class | CLASS
3 iface → MOD iface | IFACE

```

*Die Wahl der passenden Regel für `def` bei jeder beliebigen Eingabe ist für kein  $k \in \mathbb{N}$  deterministisch möglich. Sind mehr als  $k$  Modifier angegeben, so kann ein  $LL(k)$ -Parser nicht mehr zwischen `class` und `iface` unterscheiden.*

ANTLR behebt diese Schwachstelle, indem mit  $LL(*)$  statt endlichen Lookahead-Mengen reguläre Lookahead-Mengen betrachtet werden. ANTLR erstellt für eine solche Entscheidung zwischen mehreren Regeln einen endlichen deterministischen Automaten. In diesem gibt es für jede mögliche Regel einen Endzustand. Für obige Grammatik würde sich der in Abbildung 3.3 dargestellte Automat ergeben.

Abbildung 3.3: Automat für den Lookahead von def



Solch ein Automat implementiert einen beliebig großen Lookahead, so dass ANTLR damit auch obige Grammatik parsen kann. Gleichzeitig ist diese Herangehensweise für Grammatiken in  $LL(k)$  mit großem  $k$  sehr effizient. Statt aufwendige und oft sehr große Lookahead-Mengen zu berechnen, bleiben die Automaten vergleichsweise klein.

### 3.4 Erweiterung von kontextfreien Grammatiken

Aus dem Ansatz von ANTLR, der Nutzung von  $LL(*)$ , ergibt sich eine Erweiterung in der Spezifikation von kontextfreien Grammatiken. Während eine Regel wie  $S \rightarrow R^*T$  für  $LL(k)$ -Parser ein unüberwindbares Hindernis darstellt, kann ein  $LL(*)$ -Parser diese ohne Mühe verarbeiten.

Konsequenterweise wird diese Operation, der endliche Abschluss, neben einigen anderen Konstrukten von ANTLR erlaubt.

**Definition 3.10** (Erweiterte kontextfreie Grammatiken). *Eine erweiterte kontextfreie Grammatik  $G$  ist analog zu herkömmlichen kontextfreien Grammatiken definiert als ein Tupel  $G = (N, T, S, R)$  mit einer Menge von Nichtterminalsymbolen  $N$ , einer Menge von Terminalsymbolen  $T$ , einem Startsymbol  $S \in N$  und einer Menge von Regeln  $R$ .*

*Eine Regel bildet ein Nichtterminal auf einen regulären Ausdruck über den Terminalsymbolen und Nichtterminalsymbolen ab, also  $R = \{A \rightarrow r \mid A \in N, r \in \mathfrak{R}(N \cup T)\}$ .*

*Eine Ableitung ist eine Anwendung einer solchen Regel  $A \rightarrow r$  auf eine Symbolsequenz  $\beta = \beta_1 A \beta_2$ , geschrieben  $\beta \rightarrow_G \beta_1 \gamma \beta_2$ , wobei  $\gamma \in L(r)$ . Linksableitungen, Rechtsableitungen und wiederholte Ableitungen sind wie in Definition 3.5 definiert, ebenso die Sprache der Grammatik, geschrieben  $L(G)$ .*

Im Folgenden wird diese erweiterte Version der kontextfreien Grammatik verwendet. Die Wahl des  $\gamma \in L(r)$  ist Aufgabe des Parsers, die durch Anwendung des  $LL(*)$ -Parsings gelöst wird.

## 4 Fehlerüberprüfung

Die textbasierte Beschreibung einer Heapabstraktionsgrammatik wird zunächst durch den Parser einer syntaktischen Analyse unterzogen. Anschließend wird die semantische Analyse durchgeführt, es müssen also weitere benötigte Informationen berechnet und die Grammatik auf weitere Fehler überprüft werden. Diese Analyse soll in diesem Kapitel vorgestellt werden.

Zur Vereinfachung für den Benutzer werden einige Informationen über die Grammatik nicht explizit verlangt, sondern werden aus der Grammatik bestimmt. So wird automatisch zwischen Terminalsymbolen und Nichtterminalsymbolen sowie Reduktionstentakeln und Nichtreduktionstentakeln unterschieden. Zudem kann eine Grammatik automatisch normalisiert werden und die Ersetzungsregeln als Grafik im DOT-Format [Res] exportiert werden.

Ist eine Grammatik gegeben, so können grundlegende Eigenschaften einer Hyperkantenersetzungsgrammatik verletzt sein: Der Hypergraph könnte nicht zusammenhängend sein oder die Typen von Knoten könnten sich innerhalb einer Regel unterscheiden.

Weiterhin können die Bedingungen für eine Heapabstraktionsgrammatik aus Definition 2.18 verletzt sein: Die Grammatik muss produktiv, wachsend und korrekt typisiert sein. Da im Juggernaut-Framework die gegebene Grammatik in eine Grammatik in lokaler Greibach Normalform [JHKN11] umgewandelt wird, ist diese dann zudem lokal konkretisierbar.

Die in diesem Kapitel vorgestellten Möglichkeiten Fehler zu erkennen werden in Abschnitt 5.5 implementiert.

### 4.1 Zusätzliche Informationen

Vor der Überprüfung der formal notwendigen Eigenschaften der Grammatik, können weitere Daten berechnet werden, die ansonsten vom Benutzer manuell angegeben werden müssten.

Die Unterscheidung zwischen Terminalsymbolen und Nichtterminalsymbolen kann anhand der vorhandenen Regeln gemacht werden. Jedes Symbol, für das eine Regel existiert, wird als Nichtterminalsymbol angenommen, alle anderen sind Terminalsymbole. Terminalsymbole haben jedoch stets genau zwei externe Knoten. Gibt es Symbole ohne zugehörige Regeln mit nicht genau zwei externen Knoten, so liegt ein Fehler vor.

Deutlich aufwendiger ist das Erkennen von Reduktionstentakeln. Für einige Tentakel können diese direkt bestimmt werden: Liegen an einem externen Knoten nur eingehende Terminalkanten an, so ist das zugehörige Tentakel ein Reduktionstentakel. Wird

## 4 Fehlerüberprüfung

eine vom externen Knoten ausgehende Terminalkante erzeugt, so ist das Tentakel kein Reduktionstentakel.

Die Klassifizierung aller weiteren Tentakel muss auf Basis dieser Informationen gefolgert werden: Ist das Tentakel eines externen Knotens ein Reduktionstentakel, so müssen auch alle an diesem Knoten anliegenden Kanten über ein Reduktionstentakel mit diesem Knoten verbunden sein. Liegt umgekehrt an einem externen Knoten eine Kante mit einem Nichtreduktionstentakel an, so kann auch das Tentakel des externen Knotens kein Reduktionstentakel sein.

Als Information für den Benutzer kann zudem die eingelesene Grammatik grafisch dargestellt werden. Die einzelnen Regeln werden dazu als Graphen im DOT-Format [Res] gespeichert.

### 4.2 Hyperkantenersetzungsgrammatik

Nachdem der Parser aus der Eingabe einen Syntaxbaum erstellt hat, werden zunächst einige Eigenschaften der potentiellen Grammatik geprüft. So ist für eine Hyperkantenersetzungsgrammatik nach Definition 2.10 eine konsistente Typisierung der Kanten ebenso gefordert wie eine Angabe der Knotentypen.

Da Knoten in der Grammatik über Namen identifiziert werden, wird ebenfalls eine einheitliche Benennung der externen Knoten in verschiedenen Regeln eines Nichtterminals überprüft. Diese Forderung ist zwar formal nicht erforderlich, ist aber keine echte Einschränkung und erleichtert die weitere Verarbeitung in Juggernaut.

In Definition 2.10 wird zudem gefordert, dass der Hypergraphen einer jeden Regel zusammenhängend ist. Damit wird sichergestellt, Datenstrukturen weder getrennt noch zusammengeführt werden können.

Die Typisierung der Knoten wird in zwei Schritten geprüft. Da der jeweilige Knotentyp wie in Abschnitt 5.1 beschrieben bei jeder Verwendung eines Knoten optional angegeben werden kann, muss zunächst für jede Regel geprüft werden, ob für einen Knoten ein einheitlicher Typ gegeben ist. Ist kein Typ gegeben oder widersprechen sich die Angaben, so ist die Grammatik ungültig.

Ist für jeden Knoten einer jeden Regel ein Typ gefunden, so müssen die Typen der externen Knoten geprüft werden. Eine Grammatik weist gültige Knotentypen genau dann auf, wenn die in den Regeln angegebenen Typen der externen Knoten und die Typen der Knoten, die an einer solchen Nichtterminalkante anliegen, übereinstimmen.

### 4.3 Heapabstraktionsgrammatik

Ist einmal sichergestellt, dass die eingegebene Grammatik eine Hyperkantenersetzungsgrammatik ist, so müssen nur noch die in Definition 2.18 genannten Kriterien überprüft werden.

### 4.3.1 Produktivität

Ein Nichtterminal  $X$  einer Hyperkantenersetzungsgrammatik  $G$  ist produktiv, falls mit  $G$  aus  $X^\bullet$  eine konkrete Heapkonfiguration ableitbar ist. Die Grammatik  $G$  ist dann produktiv, wenn alle Nichtterminale aus  $G$  produktiv sind.

Ein Fehler hingegen liegt vor, wenn es nicht möglich ist, eine Nichtterminalkante zu einem Hypergraphen abzuleiten, der nur Terminalkanten oder produktive Nichtterminalkanten enthält. Diese Charakterisierung legt bereits eine Herangehensweise zur Erkennung von unproduktiven Nichtterminalen nahe. Diese ist in Algorithmus 4.1 beschrieben.

Algorithmus 4.1: Verfahren um Produktivität zu erkennen

```

1  $K_0 = \Sigma$ 
2  $i = 0$ 
3 repeat
4    $i = i + 1$ 
5    $K_i = K_{i-1}$ 
6   for  $(A \rightarrow H)$  in  $G$ 
7     if  $\forall e \in E_H : label(e) \in K_i$ 
8        $K_i = K_i \cup A$ 
9 until  $K_i = K_{i-1}$ 
10 return  $N \setminus K_i = \emptyset$ 

```

Ausgehend von allen Terminalen  $K_0$  werden iterativ alle Kanten  $A \rightarrow H$  durchgegangen. Enthält  $H$  nur Kanten  $e$ , für die  $label(e)$  ein Terminalsymbol ist oder bereits als produktiv gekennzeichnet ist, so wird  $A$  zu  $K_i$  hinzugefügt. Ein Nichtterminal ist als produktiv gekennzeichnet, falls es in  $K_i$  enthalten ist. Die Grammatik ist schließlich genau dann produktiv, falls kein Nichtterminal in  $N$  existiert, das auch im letzten  $K_i$  enthalten ist.

### 4.3.2 Wachstum

Eine Grammatik ist wachsend, falls jede Regel dieser Grammatik wachsend ist. Dies ist gegeben, falls der entstehende Hypergraph jeder Regel mehr als eine Kante enthält oder nur Terminalkanten enthält. Ein Hypergraph muss durch die Anwendung einer solchen Regel die Datenstruktur detaillierter beschreiben: Entweder wird die Nichtterminalkante in mindestens zwei Kanten unterteilt oder sogar an dieser Stelle durch Terminalkanten konkretisiert.

Andersherum liegt ein Fehler genau dann vor, wenn eine Regel ein Nichtterminal auf einen Hypergraphen abbildet, der genau eine Nichtterminalkante enthält. Eine solche Regel ändert nichts an der Heapkonfiguration, außer einer möglichen Umbenennung einer einzelnen Kante. Die neue Heapkonfiguration ist dann isomorph zur Vorherigen, es ergibt sich also keine Änderung.

Um für eine Regel zu überprüfen, ob diese wachsend ist, wird die Anzahl der Kanten im erzeugten Hypergraphen gezählt und geprüft, ob eine Terminalkante im erzeugten

Hypergraphen vorhanden ist. Eine Grammatik ist daher genau dann wachsend, falls  $\forall(A \rightarrow H) \in G : \exists e \in E_H : label(e) \in \Sigma \vee |E_H| > 1$ .

### 4.3.3 Typisierung

Die Typisierung einer Grammatik ist eine wichtige Eigenschaft, um mit komplexen Datentypen in der zu verifizierenden Programmiersprache umzugehen. Sie ist eine striktere Variante der Typsicherheit, die beispielsweise von Java implementiert wird, da Typengleichheit gefordert wird.

Bei der Anwendung einer Ersetzungsregel müssen die Knotentypen der zur Kante inzidenten Knoten und der externen Knoten des Hypergraphen identisch sein. Weiterhin muss es mit jeder Regel möglich sein alle Selektoren für einen Knotentyp zu erzeugen, entweder durch das Erzeugen von ausgehenden Terminalkanten oder die Weitergabe des Knoten an eine weitere Nichtterminalkante an einem Nichtreduktionstentakel.

Da jede Nichtterminalkante alle Selektoren erzeugen können muss, dürfen die beiden oben vorgestellten Möglichkeiten nicht vermischt werden. Ist ein Knoten durch ein Nichtreduktionstentakel mit einer Nichtterminalkante verbunden und wird gleichzeitig eine ausgehende Terminalkante erzeugt, so könnten die entsprechenden Selektoren doppelt erzeugt werden, was den Anforderungen an eine Heapkonfiguration widerspricht.

Ist eine ausgehende Terminalkante an einem Knoten vorhanden, so müssen daher in dieser Regel alle ausgehenden Terminalkanten für diesen Knoten erzeugt werden. Wäre dies nicht der Fall, könnte diese Regel nicht alle Terminalkanten erzeugen.

Um die Typisierung einer konkreten Grammatik zu überprüfen sind also mehrere Schritte nötig. Sind mehrere Regeln für ein Nichtterminal  $X \rightarrow H_1 \mid \dots \mid H_n$  gegeben, so müssen die Typen der externen Knoten der Hypergraphen  $H_1, \dots, H_n$  übereinstimmen. Weiterhin müssen die zu einer Nichtterminalkante  $e$  mit  $label(e) = X$  inzidenten Knoten dieselben Typen wie die externen Knoten der Hypergraphen haben.

Anschließend muss in jeder Regel jeder Knoten auf seine ausgehenden Kanten geprüft werden. Eine Regel ist genau dann zulässig, wenn an jedem Knoten nur entweder eine Nichtterminalkante an einem Nichtreduktionstentakel oder mindestens eine Terminalkante anliegen.

Zum Schluss wird für alle diese Knoten geprüft, ob die ausgehenden Terminalkanten stets dieselben sind. Können zwei Kanten unterschiedliche Terminalkanten erzeugen, so ist die Regel ebenfalls nicht gültig.

## 5 Implementierung

Ein wesentlicher Bestandteil dieser Arbeit ist die Umsetzung der vorgestellten Konzepte in Form eines lauffähigen Übersetzers. Dieser liest eine textbasierte Repräsentation einer Hyperkantenersetzungsgrammatik ein, analysiert sie und generiert daraus schließlich eine auf Fehler geprüfte Grammatik samt weiterer Eigenschaften.

Der Übersetzer besteht aus einem Lexer, einem Parser und verschiedenen Baumparsern, einer komfortablen Möglichkeit, Ableitungsbäume mit ANTLR zu verarbeiten.

Zunächst soll eine Sprache entworfen werden, in der auf möglichst einfache Weise Hyperkantenersetzungsgrammatiken spezifiziert werden können. Anschließend werden für diese Sprache ein Lexer und ein Parser in ANTLR konstruiert. Das eigentliche Erstellen von Lexer und Parser übernimmt ANTLR, wobei das bereits vorgestellte Konzept des  $LL(*)$ -Parsings verwendet wird.

Anschließend werden die mit einem Baumparser durchgeführten Fehlerüberprüfungen näher erläutert. Hierbei handelt es sich um die Implementierungen der in Kapitel 4 vorgestellten Verfahren. Die abschließende Übersetzung des Syntaxbaums in die finale Datenstruktur ist technischer Natur und wird daher nur kurz angesprochen.

Die gezeigten Quelltexte oder Grammatiken sind in der Regel nur Ausschnitte der wesentlichen Inhalte der jeweiligen Dateien, nicht relevante Kopf- und Metadaten oder Hilfsmethoden wurden entfernt. Die kompletten Dateien finden sich auf der beiliegenden CD oder unter <http://moves.rwth-aachen.de/i2/juggrnaut/>.

### 5.1 Definition der Grammatik

Die vorrangigen Ziele beim Entwurf der Sprache sind eine kompakte Darstellung der beschriebenen und der spezifizierenden Grammatik. Daraus ergeben sich unmittelbar weitere Vorteile, beispielsweise eine kleinere Menge an syntaktischen Konstrukten und damit einhergehend eine leichtere Verständlichkeit.

Daher wurde auf eine explizite Deklaration der Knoten verzichtet, die Typinformationen können an jeder beliebigen Stelle, an der dieser Knoten verwendet wird, angegeben werden. Dieser Verzicht erlaubt zwar Fehler, beispielsweise die inkonsistente Angabe von Knotentypen, die jedoch im Nachhinein ohne großen Aufwand geprüft werden können.

Zur einfacheren Handhabung ist die Formatierung für die Syntax irrelevant, Leerzeichen und Zeilenumbrüche können nach Belieben eingefügt werden. Außerdem soll die Möglichkeit gegeben werden, Kommentare einzufügen. Kommentare sind an jeder Stelle

## 5 Implementierung

möglich. Die Zeichen `'//'` leiten einen Kommentar bis zum Zeilenende ein, die Zeichen `'/*'` bis zum darauf folgenden `'*/'`.

Die gesamte Grammatik besteht aus einer Sequenz von Regeln. Jede Regel wiederum besteht aus einem Nichtterminalsymbol, sowie einer Sequenz von Kanten, die von der Regel erstellt werden. Jede Kante besteht aus einem Kantennamen und einer Liste von externen Knoten.

Die Knoten können durch ihren Namen sowie optional ihren Typ gegeben sein. Zusätzlich kann der spezielle Knoten `NULL` angegeben werden. Für diese Sprache ergibt sich die in Abbildung 5.1 angegebene kontextfreie Grammatik.

Abbildung 5.1: Kontextfreie Grammatik für die Sprache der HAGs

```
grammar → ( rule )*
rule → ID ':' tuple ( '->' ID tuple )*
tuple → '(' node ( ',' node )* ')'
node → ID | ID ID | NULL
```

Die einzigen verwendeten Terminalsymbole außer den Zeichen `':'`, `'->'`, `'('`, `','`, `','`, `','`, `)'` sind die Symbole `ID` und `NULL`. Letzteres erkennt die Zeichenkette `'null'` in beliebiger Groß- und Kleinschreibung.

Hinzu kommen zwei weitere Terminalsymbole, die allerdings vom Parser ignoriert werden: `WHITESPACE` enthält alle Leerzeichen und Zeilenumbrüche, `COMMENT` behandelt Kommentare. Die Definition dieser Terminale geschieht wie in Abbildung 5.2 angegeben.

Abbildung 5.2: Definition der nichttrivialen Token

```
ID → ( 'a' ... 'z' | 'A' ... 'Z' | '0' ... '9' )+
COMMENT → ( '/*' .* '*/' | '//'.* '\n'
```

Die Definition von `COMMENT` macht sich eine Besonderheit von ANTLR zu Nutze. Bei der Verwendung der Konstruktion `'.*'` schaltet der Lexer automatisch in einen nicht-gierigen Modus um, so dass hier wirklich nur die Kommentare erfasst werden, nicht aber der Text zwischen zwei Kommentaren.

Bei der Einordnung der Grammatik in  $LL(k)$  wird klar, dass die Grammatik mit einem  $LL(1)$ -Parser in der in Abbildung 5.1 gezeigten Form nicht realisierbar ist, da sie nicht mit einem Lookahead der Größe eins zu parsen ist. Seien zunächst die First-, Follow- und Lookahead-Mengen in Abbildung 5.3 und Abbildung 5.4 gegeben.

Abbildung 5.3: First- und Follow-Mengen der gegebenen Grammatik

$N$	$fi_1(N)$	$fo_1(N)$
grammar	{ ID }	{ $\varepsilon$ }
rule	{ ID }	{ $\varepsilon$ , ID }
tuple	{ '(' }	{ $\varepsilon$ , -> }
node	{ ID, NULL }	{ ,, ')'

Abbildung 5.4: Lookahead-Mengen der gegebenen Grammatik

$R: N \rightarrow \alpha$	$LA_1(N \rightarrow \alpha)$
grammar $\rightarrow$ rule*	{ ID }
rule $\rightarrow$ ID : tuple ( -> ID tuple)*	{ ID }
tuple $\rightarrow$ '(' node (, node)* ')'	{ '(' }
node $\rightarrow$ ID	{ ID }
node $\rightarrow$ ID ID	{ ID }
node $\rightarrow$ NULL	{ NULL }

Da die Lookahead-Mengen  $LA_1(\text{node} \rightarrow \text{ID})$  und  $LA_1(\text{node} \rightarrow \text{ID ID})$  nicht disjunkt sind, liegt ein Konflikt vor und die Grammatik ist nicht in  $LL(1)$  enthalten. Wird der Lookahead jedoch auf  $k = 2$  erhöht, so ergeben sich die folgenden Lookahead-Mengen.

Abbildung 5.5: Lookahead-Mengen für  $k = 2$ 

$$LA_2(\text{node} \rightarrow \text{ID}) = \{ \text{ID } , , \text{ID } ')'\}$$

$$LA_2(\text{node} \rightarrow \text{ID ID}) = \{ \text{ID ID} \}$$

Die Grammatik ist daher in  $LL(2)$  enthalten. Sie könnte durch Linksfaktorisierung so umgeformt werden, dass ein Lookahead von  $k = 1$  genügen würde. Da ANTLR jedoch das Parsen von  $LL(2)$ -Grammatiken unterstützt und die Grammatik dadurch weniger lesbar würde, wurde hier darauf verzichtet.

Tatsächlich genügt die Grammatik sogar den Anforderungen an eine reguläre Grammatik [ASU86, Kapitel 3.3.] und könnte daher mit weniger aufwendigen Methoden analysiert werden. Es wurde dennoch ANTLR verwendet, da ANTLR direkt eine komfortable Fehlerbehandlung mitbringt sowie mit Baumparsern eine elegante Möglichkeit zur Weiterverarbeitung beinhaltet.

## 5.2 Spezifikation des Lexers

Die beschriebenen Terminalsymbole können fast ohne Änderungen aus der Definition übernommen werden, um die Token für den Lexer zu spezifizieren. Um Token eindeutig als solche zu identifizieren fordert ANTLR, dass die Namen der Token mit einem Großbuchstaben beginnen.

## 5 Implementierung

Da Lexer und Parser prinzipiell in einer Datei spezifiziert werden, werden hier nur die für den Lexer relevanten Ausschnitte dieser Datei beschrieben.

Algorithmus 5.1: Grammatik des Lexers

```
1 COLON      : ':' ;
2 TUPEL_START : '(' ;
3 TUPEL_SEP  : ',' ;
4 TUPEL_END  : ')' ;
5 RESULT_MARK : '->' ;
6 NULL      : ('n'|'N') ('u'|'U') ('l'|'L') ('l'|'L') ;
7
8 ID        : ('a'..'z' | 'A'..'Z' | '0'..'9')+ ;
9 WHITESPACE : (' '|'\r'|\t'|\n')+ { $channel = HIDDEN; } ;
10 COMMENT   : ('/*' .* '*/' | '//' .* ('\n'|\r')+ ) { $channel
    = HIDDEN; } ;
11 REST     : . ;
```

Gegenüber der Beschreibung in Algorithmus 5.2 gibt es einige Unterschiede. Zunächst wurde für Zeilenumbrüche statt `\n` auch das zum Teil verwendete `\r` berücksichtigt. Außerdem wurde ein weiteres Token `REST` eingeführt. Damit wird garantiert, dass stets eine Zerlegung der Eingabe existiert, die lexikalische Analyse kann damit nicht fehlschlagen. Ungültige Token können stattdessen auf einheitliche Art und Weise im Parser abgefangen werden.

ANTLR erlaubt es, Token in verschiedene Kanäle einzuteilen. Üblicherweise gibt es einen normalen und einen versteckten Kanal. Dieser kann dafür genutzt werden, Token zu ignorieren. In bestimmten Situationen, zum Beispiel für Fehlermeldungen, kann es jedoch sinnvoll sein, auch die bis dahin ignorierten Token auszugeben. Dies ist mit diesem Konzept problemlos möglich.

Token werden mit dem Befehl `$channel = HIDDEN;` in den versteckten Kanal eingeteilt. Dieser wird in geschweiften Klammern hinter dem jeweiligen Token notiert, hier bei den Token `WHITESPACE` und `COMMENT`.

Als Beispiel wird nun die Grammatik aus Algorithmus 5.2 betrachtet, die eine verkettete Liste beschreibt. Sie könnte als Eingabe wie folgt aussehen.

Algorithmus 5.2: Beispielgrammatik einer verketteten Liste

```
1 L: (List start, List end)
2   -> next(start, inner)
3   -> L(inner, end)
4 L: (List start, List end)
5   -> L(start, inner)
6   -> L(inner, end)
7 L: (List start, List end)
8   -> next(start, end)
```

Der Lexer erzeugt aus der ersten Regel dieser Grammatik die in Algorithmus 5.3 folgenden Symbole im normalen Kanal, der versteckte Kanal wird hier nicht betrachtet.

Algorithmus 5.3: Liste der Token für die erste Regel aus Algorithmus 5.2

```

1 (ID, 'L'), (COLON, ':'), (TUPEL_START, '('), (ID, 'List'), (ID, 'start'),
2 (TUPEL_SEP, ','), (ID, 'List'), (ID, 'end'), (TUPEL_END, ')'),
3 (RESULT_MARK, '->'), (ID, 'next'), (TUPEL_START, '('), (ID, 'start'),
4 (TUPEL_SEP, ','), (ID, 'inner'), (TUPEL_END, ')'),
5 (RESULT_MARK, '->'), (ID, 'L'), (TUPEL_START, '('), (ID, 'inner'),
6 (TUPEL_SEP, ','), (ID, 'end'), (TUPEL_END, ')')

```

## 5.3 Spezifikation des Parsers

Zur Spezifikation des Parsers genügt es ANTLR, die zu erkennende kontextfreie Grammatik anzugeben. Als Terminalsymbole dienen hier die vom Lexer erzeugten Token.

Die gewünschte Ausgabe, der Baum aller Ableitungen, oder Syntaxbaum, muss jedoch zusätzlich definiert werden. Dies löst ANTLR sehr elegant, indem zu jeder Regel ein Fragment des Baums in einer Baumbeschreibungssprache angegeben wird. Diese Fragmente werden beim Parsen dann zu einem Baum zusammengesetzt. Um einen Knoten mit zwei Kindknoten zu erzeugen genügt das Fragment  $\wedge(\text{PARENT CHILD CHILD})$ .

Der Operator  $\wedge$  erhält eine Liste von Bäumen, wobei das erste Argument ein einfaches Token sein muss. Das Ergebnis ist ein Baum mit dem ersten Argument als Wurzelknoten und allen anderen Bäumen als Kinder.

Dieser Baumoperator bietet die Möglichkeit syntaktisch wichtige Symbole ohne semantische Bedeutung nicht in den Syntaxbaum aufzunehmen. So können in der vorgestellten Grammatik beispielsweise Klammern und Kommata weggelassen werden, da die von ihnen gegebene Struktur bereits durch die Baumstruktur vorgegeben wird.

Die oben definierte Grammatik mit den zusätzlichen Baumfragmenten wird in ANTLR wie in Algorithmus 5.4 definiert.

Algorithmus 5.4: Grammatik des Parsers

```

1 grammar
2   : rule* EOF
3   -> (^( RULE rule ))*
4   ;
5
6 rule
7   : ID COLON tuple (RESULT_MARK ID tuple)*
8   -> ^(DEFINITION ^(TYPE ID) tuple)
9       (^(EDGE ^(TYPE ID) tuple))*
10  ;
11
12 tuple
13  : TUPEL_START node ( TUPEL_SEP node )* TUPEL_END
14  -> (^( NODE node ))+
15  ;

```

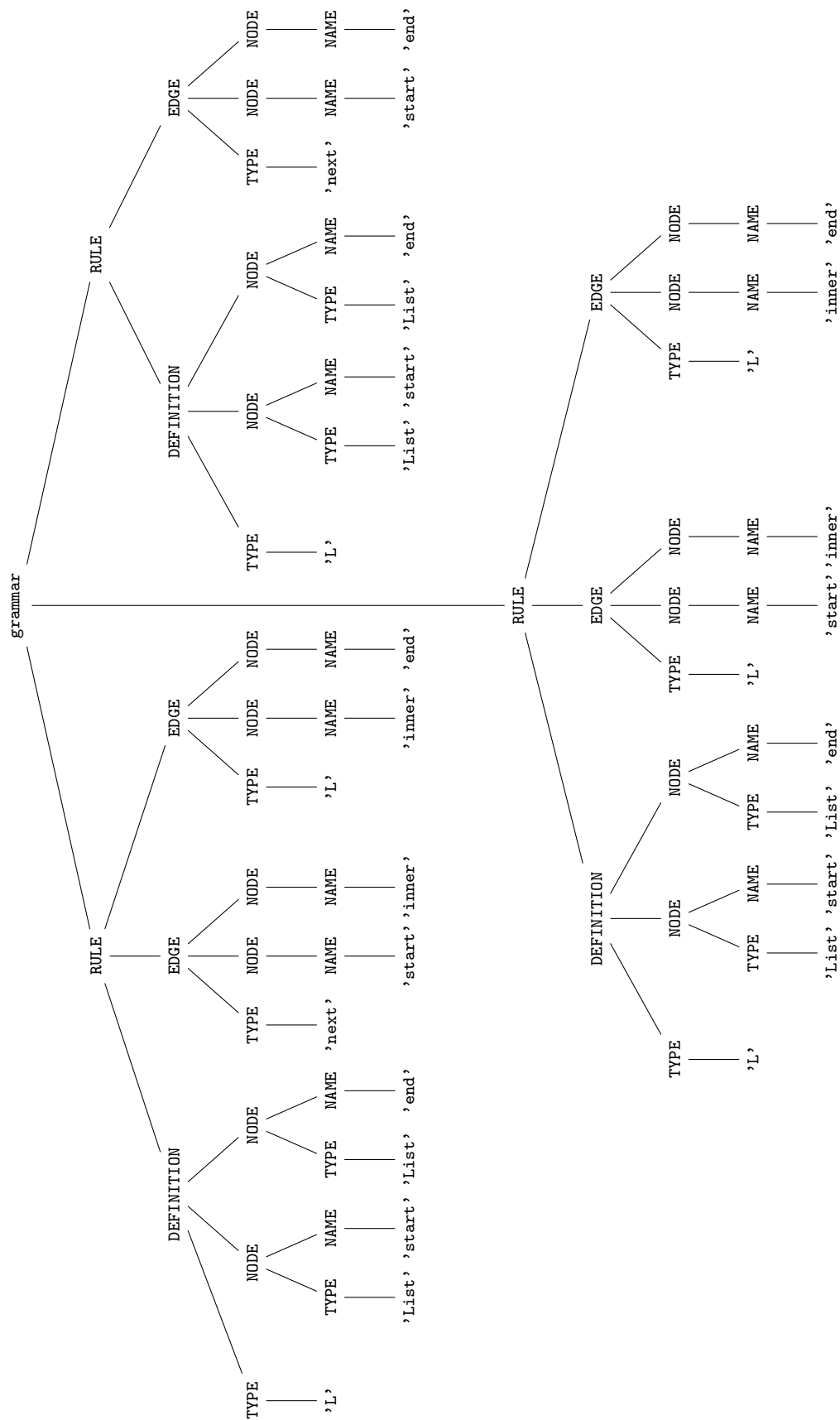
## 5 Implementierung

### Fortsetzung Algorithmus 5.4

```
16 node
17   : ID
18   -> ^(NAME ID)
19   | ID ID
20   -> ^(TYPE ID) ^(NAME ID)
21   | NULL
22   -> NULL
23 ;
```

Als Beispiel dient wiederum die Grammatik für eine verkettete Liste. Der Parser erstellt aus den vom Lexer erzeugten Symbolen den auf der nächsten Seite in Abbildung 5.6 dargestellten Syntaxbaum. Zur besseren Übersichtlichkeit ist bei Symbolen mit dem Token ID nur das entsprechende Lexem angegeben.

Abbildung 5.6: Syntaxbaum für die Grammatik einer verketteten Liste



## 5.4 Spezifikation des Baumparsers

Sobald der Parser aus der Eingabe den Syntaxbaum erstellt hat, kann dieser auf mögliche Fehler in der Eingabegrammatik überprüft und schließlich eine von Juggernaut nutzbare Form der Grammatik erstellt werden.

Syntaxbäume können mit ANTLR mithilfe von Baumparsern analysiert und verarbeitet werden. Solche Baumparser werden durch Baumgrammatiken spezifiziert, die analog zu normalen Grammatiken angegeben werden. Statt der Folge der Eingabetoken werden als Regeln Baumfragmente verwendet.

Als Baumfragmente können genau diejenigen Fragmente verwendet werden, mit denen der Syntaxbaum in Abschnitt 5.3 erstellt wurde. So kann das Gerüst der Baumgrammatik unmittelbar aus der normalen Grammatik übernommen werden.

Jede zu überprüfende Eigenschaft wird in einer eigenen Klasse in Form eines Observers gekapselt, die einige Methoden implementieren muss. Der Baumparser ruft nun jede dieser Methoden auf: `init()` vor Beginn des Parsens, `startRule(E, V)` bei Beginn einer neuen Regel, `edge(E, V)` bei jeder zu erstellenden Kante in einer Regel, `endRule(E)` zu Ende jeder Regel und `validate()` nach Ende des Parsens. Jede der Methoden gibt einen Wahrheitswert über den Erfolg des jeweiligen Tests zurück. Die genaue Bedeutung der Methoden und ihrer Parameter wird in Abschnitt 5.5 erläutert.

Die Grammatik des Baumparsers wird in ANTLR wie in Algorithmus 5.5 angegeben. Die eigentlichen Regeln sind identisch zu den Baumfragmenten aus der normalen Grammatik, hinzugekommen sind einzelne Java-Befehle.

Hinter jeder Regel kann eine Liste von Rückgabewerten angegeben werden, geschrieben als `returns [ Typ Variable ]`. Zusätzlich können mithilfe der `@init`-Klausel Befehle zu Anfang einer Regel ausgeführt werden.

Die Variable `observer` ist eines der oben beschriebenen `Observer`-Objekte, die jeweils eine bestimmte Eigenschaft der Grammatik prüfen.

Algorithmus 5.5: Grammatik des Baumparsers

```

1 rules returns [ boolean res ]
2   : { $res = observer.doInit(); }
3     ^( RULE rule { $res &= $rule.res; } ))*
4     { $res &= observer.isValid(); }
5   ;
6
7 rule returns [ boolean res ]
8   : ^(DEFINITION ^(TYPE name=ID) nodes=tuple)
9     { $res = observer.startRule($name.token, $nodes.nodes); }
10  (
11    ^(EDGE ^(TYPE name=ID) nodes=tuple)
12      { $res &= observer.edge($name.token, $nodes.nodes); }
13  )*
14    { $res &= observer.endRule($name.token); }
15  ;
40

```

## Fortsetzung Algorithmus 5.5

```

16 tuple returns [ List<Definition> nodes ]
17 @init { $nodes = new LinkedList<Definition>(); }
18   : (^ (NODE cur=node
19     { $nodes.add(new Definition($cur.name, $cur.type)); }
20     ))+
21 ;
22
23 node returns [ Token name, Token type ]
24   : ^ (NAME n=ID)
25     { $name = $n.token; $type = null; }
26   | ^ (TYPE t=ID) ^ (NAME n=ID)
27     { $name = $n.token; $type = $t.token; }
28   | NULL
29     { $name = null; $type = null; }
30 ;

```

Die Regel `node` wird immer dann angewandt, wenn in der Grammatik ein Knoten angegeben werden muss. Zurückgegeben wird der Name und der Typ des Knotens. Mit `tuple` werden alle Knoten in einem Tupel zu einer Liste zusammengefasst. Der Typ `Definition` enthält den Namen und Typen eines Knoten.

Eine Regel  $X \rightarrow H$  der Heapabstraktionsgrammatik mit  $X \in \Sigma_N$  und  $H$  einem Hypergraphen wird dann durch `rule` bearbeitet. Zunächst wird `observer.startRule()` mit dem Nichtterminal  $X$  und der Liste der externen Knoten von  $X$  aufgerufen. Der Hypergraph  $H$  wird durch eine Liste von Kanten angegeben. Für jede dieser Kanten wird anschließend `observer.startRule()` mit der jeweiligen Kante und der Liste der anliegenden Knoten aufgerufen. Sobald die Regel abgeschlossen ist wird `observer.endRule()` mit dem Nichtterminal  $X$  aufgerufen.

Sollte eine der Methoden `False` zurückgeben, wird dies durch `rules` zurückgegeben. Dort kann der Wert ausgelesen werden und eine entsprechende Fehlermeldung erzeugt werden. Eine Fehlermeldung mit der genauen Fehlerbeschreibung wird von dem jeweiligen `Observer`-Objekt erzeugt.

## 5.5 Implementierung der Fehlerüberprüfungen

In diesem Abschnitt sollen die in Kapitel 4 beschriebenen Verfahren zur Fehlerüberprüfung durch `Observer`-Objekte implementiert werden. Für jedes `Observer`-Objekt sind hier die wesentlichen Quelltextabschnitte angeben, aufgrund der kompakteren Schreibweise in einem Python-nahen Pseudocode.

Informationen wie die Anzahl der Knoten einer Kante sind in der Regel als Eigenschaften des jeweiligen Objekts umgesetzt, falls es für das Vorgehen keine Rolle spielt, woher diese Information stammt. Häufig wird diese Information jedoch von anderen `Observer`-Objekten bezogen. Die dadurch entstehenden Abhängigkeiten werden in Abschnitt 5.6.2 genauer beleuchtet.

## 5 Implementierung

Jede **Observer**-Klasse muss mindestens die vom Baumparser benötigten Methoden `init()`, `startRule()`, `edge()`, `endRule()` und `validate()` implementieren. Werden einzelne dieser Methoden nicht benötigt, so werden diese hier nicht aufgeführt. Entfernt wurden zudem Details der Klassenhierarchie und Datenkapselung sowie Abschnitte rein technischer Natur und einige syntaktische Elemente wie Semikolons. Listen und Arrays beginnen hier stets mit dem Index 1.

Die Implementierung ist in Java geschrieben und kann auf der beiliegenden CD oder unter <http://moves.rwth-aachen.de/i2/juggernaut/> eingesehen werden. Die Klassen sind entsprechend der Konventionen für Quellcode in Java in eigenen, entsprechend benannten Dateien gespeichert.

### 5.5.1 Terminal- und Nichtterminalsymbole

Das Alphabet  $\Sigma_N$  besteht nach Definition 2.5 aus Terminalen  $\Sigma$  und Nichtterminalen  $N$ . Die Klasse `NonTerminalCollector` übernimmt, wie in Algorithmus 5.6 angegeben, die Aufgabe, die verwendeten Symbole einzulesen und zwischen Terminalsymbolen und Nichtterminalsymbolen zu unterscheiden.

Algorithmus 5.6: Quelltext der Klasse `NonTerminalCollector`

```
1 terminals
2 nonterminals
3
4 ruleStart(nonterminal, nodes):
5     if terminals.contains(nonterminal):
6         terminals.remove(nonterminal)
7         nonterminals.add(nonterminal)
8
9 edge(edge, nodes):
10    if not nonterminals.contains(edge):
11        terminals.add(edge)
12
13 validate():
14    for terminal in terminals:
15        if terminal.nodes() != 2:
16            raise error
```

Die Variablen `terminals` und `nonterminals` sind beide Mengen von Symbolen. Am Ende sollen sie alle Terminalsymbole beziehungsweise Nichtterminalsymbole enthalten.

Zu Beginn jeder Regel wird das Nichtterminal zu `nonterminals` hinzugefügt und gegebenenfalls aus `terminals` entfernt. Somit sind alle Nichtterminale in `nonterminals` enthalten. Für jede neue Kante innerhalb der Regel wird das Symbol der Kante zu `terminals` hinzugefügt, falls es nicht bereits den Nichtterminalen zugeordnet wurde.

Zum Schluss wird geprüft, ob unter den Terminalsymbolen ein Symbol existiert, dessen Typsequenz nicht aus genau zwei Typen besteht, da dies durch die Interpretation von Terminalkanten als einfache Zeiger gefordert wird. Ist dies der Fall, wird eine entsprechende Fehlermeldung ausgegeben.

### 5.5.2 Graphzusammenhang

Wie in Definition 2.10 gefordert, müssen die Hypergraphen auf der rechten Regelseite stets zusammenhängend sein. Dadurch werden ungewünschte Operationen wie das Trennen einer Datenstruktur oder das Zusammenführen zweier vorher getrennter Datenstrukturen vermieden, wie Beispiel 2.6 zeigt.

Die Klasse `RuleConnectedCheck` überprüft die rechte Regelseite jeder Regel darauf, ob der gegebene Hypergraph zusammenhängend ist. Das Vorgehen ist in Algorithmus 5.7 beschrieben.

Algorithmus 5.7: Quelltext der Klasse `RuleConnectedCheck`

```

1 groups
2
3 ruleStart(nonterminal, nodes):
4     groups.clear()
5     for node in nodes:
6         groups.add([node])

```

In der Variablen `groups` werden die Zusammenhangskomponenten der aktuellen Regel gespeichert, also Mengen von Knoten, die bereits durch Kanten miteinander verbunden wurden. Am Anfang jeder Regel werden diese zunächst gelöscht, um anschließend alle Knoten einzeln einzufügen.

Fortsetzung Algorithmus 5.7

```

7 edge(edge, nodes):
8     newgrp = []
9     for node in nodes:
10        for grp in groups:
11            if not groups.contains(node):
12                continue
13            newgrp.addAll(grp)
14            groups.remove(grp)
15        groups.add(newgrp)
16
17 ruleEnd(nonterminal):
18     if groups.size() > 1:
19         raise error

```

Für jede neue Kante wird dann über alle anliegenden Knoten iteriert und alle Zusammenhangskomponenten, in denen einer dieser Knoten enthalten ist, zu einer neuen Menge zusammengefasst. Ist der Hypergraph zusammenhängend, so bleibt am Ende einer Regel genau eine Menge übrig. Sind jedoch mindestens zwei Mengen vorhanden, so gab es keine Kante, die einen Knoten der einen Zusammenhangskomponente mit einem Knoten der Anderen verbunden hat. Der Hypergraph ist dann nicht zusammenhängend und es wird eine Fehlermeldung ausgegeben.

### 5.5.3 Knotennamen

Knoten werden in Juggernaut grundsätzlich über ihre Namen identifiziert. Eine einheitliche Benennung der externen Knoten aller Regeln eines Nichtterminals ist daher sehr hilfreich bei der späteren Nutzung der Grammatik.

Die Klasse `NodeNameCheck` prüft daher, ob diese Knoten stets gleich benannt sind. Dies wird in Algorithmus 5.8 dargestellt.

Algorithmus 5.8: Quelltext der Klasse `NodeNameCheck`

```
1  nodenames
2
3  ruleStart(nonterminal, nodes):
4      if nodenames.contains(nonterminal):
5          for i = 1 to nodes.size():
6              if nodenames[nonterminal[i] != node:
7                  raise error
8      else:
9          nodenames[nonterminal] = nodes
```

Da stets nur die externen Knoten betrachtet werden müssen, wird neben der Variable `nodenames`, in der für jedes Nichtterminal die Namen der externen Knoten gespeichert werden, nur die Methode `ruleStart()` benötigt.

Ist für das Nichtterminal der aktuellen Regel bereits eine Benennung bekannt, so werden alle externen Knoten mit der Benennung verglichen. Sind Knoten unterschiedlich benannt, so wird ein Fehler ausgegeben. Sollte jedoch noch keine Benennung bekannt sein, so werden die aktuellen externen Knoten als Benennung gespeichert.

### 5.5.4 Knotentypen

Für die weitergehende Analyse werden die Knotentypen benötigt, die jedoch an einer beliebigen Stelle innerhalb einer Regel angegeben werden können. Alle Knoten werden daher von der Klasse `NodeTypeCollector` darauf überprüft, ob für sie ein einheitlicher Typ angegeben ist. Anschließend werden die Typen für weitere Analysen zur Verfügung gestellt.

Algorithmus 5.9: Quelltext der Klasse `NodeTypeCollector`

```
1  nodetypes
2  types
3
4  ruleStart(nonterminal, nodes):
5      types = []
6      edge(nonterminal, nodes)
```

Die Variable `nodetypes` soll am Ende für jede Regel und jeden Knoten in der jeweiligen Regel einen Typen enthalten. `types` hingegen speichert nur die Typen der Knoten der aktuell behandelten Regel.

Zu Beginn einer Regel wird zunächst `types` geleert. Anschließend müssen die Typen der externen Knoten in `types` eingetragen werden. Dies muss sowohl für die externen Knoten des Hypergraphen, als auch für alle Knoten jeder Kante in diesem Hypergraphen geschehen. Dies wird von der Methode `edge()` implementiert, die daher hier aufgerufen wird.

Fortsetzung Algorithmus 5.9

```

7 edge(edge, nodes):
8   for node in nodes:
9     if node.type == null:
10      if not types.contains(node):
11        types[node] = null
12      else if types[node] != null:
13        if node.type != types[node]:
14          raise error
15      else:
16        types[node] = node.type

```

Für jeden Knoten der aktuellen Kante wird nun der Typ geprüft. Ist der Typ nicht gegeben, so wird ein Eintrag `null` in `types` angelegt, falls er nicht bereits existiert. Ist der Typ gegeben und bereits in `types` abgelegt, so wird geprüft ob die beiden Typen identisch sind – falls nicht, wird ein Fehler ausgegeben. Ist der Typ gegeben, aber noch nicht in `types` vorhanden, so wird er dort gespeichert.

Es ist nun also sichergestellt, dass für jeden Knoten nie zwei unterschiedliche Typen angegeben werden.

Fortsetzung Algorithmus 5.9

```

17 ruleEnd(nonterminal):
18   for type in types:
19     if type == null:
20       raise error
21   nodetypes.add(types)

```

Ist die Regel beendet, so werden alle Knoten daraufhin überprüft, ob sie einen Typen zugewiesen bekommen haben. Ist dies nicht der Fall, so wird ein Fehler erzeugt. Waren alle Typen konsistent, so wird `types` der Liste `nodetypes` hinzugefügt.

### 5.5.5 Knotentypisierung

Nachdem für alle Knoten jeder Regel ein eindeutiger Typ festgestellt wurde, muss nun geprüft werden, ob die Typen der externen Knoten für alle Kanten und Regeln eines Nichtterminals übereinstimmen. Für jedes Symbol müssen also die Knotentypen bei jeder Verwendung identisch sein, bei Nichtterminalen zusätzlich auch die externen Knoten aller Regeln. Diese Knotentypisierung wird von der Klasse `NodeTypeCheck` wie in Algorithmus 5.10 geprüft.

Algorithmus 5.10: Quelltext der Klasse `NodeTypeCheck`

## 5 Implementierung

```
1 types
2
3 ruleStart(nonterminal, nodes):
4     if types.contains(nonterminal):
5         if nodes != types[nonterminal]:
6             raise error
7     else:
8         list = []
9         for node in nodes:
10            list.add(node.type)
11            types[nonterminal] = list
12
13 edge(edge, nodes):
14     ruleStart(edgename, nodes)
```

Als einzige Variable wird `types` benötigt. Hier wird für jedes Symbol die Liste der Typen hinterlegt. Da für die Liste der externen Knoten ebenso die Typen geprüft werden müssen, wie für eine zu erzeugende Kante, wird in der Methode `edge()` die Methode `startRule()` aufgerufen.

Dort wird zunächst unterschieden, ob die Typen für das gegebene Symbol bereits in `types` gespeichert ist. Ist dies der Fall, werden die Typen verglichen. Sollten die Listen der Typen unterschiedlich sein, wird ein Fehlermeldung ausgegeben.

Sind die Typen noch nicht in `types` vorhanden, so wird eine Liste der entsprechenden Typen erzeugt und diese zu `types` hinzugefügt. Am Ende ist somit sichergestellt, dass die Typen der externen Knoten mit denen übereinstimmen, die an der jeweiligen Nichtterminalkante anliegen. Bei jeder neuen Regel und jeder neuen Benutzung eines Terminals werden somit identische Typen verwendet.

### 5.5.6 Reduktionstentakel

Die Identifikation aller Reduktionstentakel gemäß Definition 2.17 wird für die weitere Nutzung der Grammatik in Juggernaut benötigt. Die explizite Angabe von Reduktionstentakeln ist aber nicht nur aufwendig, sondern auch fehleranfällig, falls der Benutzer nicht mehr mit den Details der Implementierung vertraut ist. Die Klasse `TentacleReductionCollector` ermittelt diese Eigenschaft der Tentakel automatisch aus einer gegebenen Grammatik wie in Algorithmus 5.11 beschrieben.

Sie erstellt dafür ein System von möglichen Folgerungen und identifiziert einige Tentakel, für die klar ist, ob sie Reduktionstentakel sind oder nicht. Dieses System wird anschließend von einem in Abschnitt 5.6.1 beschriebenen Löser gelöst. Ein Beispiel für ein solches System ist in Beispiel 5.1 gegeben.

Algorithmus 5.11: Quelltext der Klasse `TentacleReductionCollector`

```
1 reduction
2 solver
3 mapping
4 candidates
```

Die Variable `reduction` speichert zu jedem Nichtterminal eine Liste von Wahrheitswerten. Für jedes Tentakel wird gespeichert, ob es ein Reduktionstentakel ist. Der Löser für das System von Folgerungen wird in der Variable `solver` abgelegt.

Die beiden Variablen `mapping` und `candidates` sind jeweils für eine Regel gültig. Zu Beginn einer neuen Regel werden sie geleert. `mapping` enthält für jeden externen Knoten seine Position in der Sequenz der externen Knoten, während `candidates` eine List der Tentakel enthält, die noch als Reduktionstentakel in Frage kommen.

Fortsetzung Algorithmus 5.11

```

5 ruleStart(nonterminal, nodes):
6     mapping = []
7     candidates = []
8     for i = 1 to nodes.size():
9         mapping[nodes[i]] = i
10        candidates.add((nonterminal, i))

```

Wie bereits erwähnt, werden am Anfang einer Regel `mapping` und `candidates` geleert. `mapping` wird anschließend mit den externen Knoten der neuen Regel wieder gefüllt. Da zu Beginn noch kein Tentakel durch eine anliegende Kante als potentiell Nichtreduktionstentakel identifiziert werden kann, sind alle Tentakel Kandidaten für Reduktionstentakel und werden in `candidates` eingefügt.

Fortsetzung Algorithmus 5.11

```

11 edge(edge, nodes):
12     if edge.terminal():
13         if mapping.containsKey(nodes[1]):
14             external = mapping[nodes[1]]
15             solver.add(null, (edge, 1), False)
16             solver.add((edge, 1), (currentRule, external), False)
17             candidates.remove((currentRule, external))
18     else:
19         for i = 1 to nodes.size():
20             if not mapping.containsKey(nodes[i]): continue
21             external = mapping[nodes[i]]
22             solver.add((edge, i), (currentRule, external), False)
23             solver.add((currentRule, external), (edge, i), True)
24             candidates.remove((currentRule, i))

```

Bei jeder neuen Kante in einer Regel werden alle externen Knoten, an denen ausgehende Kanten erzeugt werden können, geprüft. Ist die neue Kante eine Terminalkante, so ist dies nur für den ersten Knoten der neuen Kante der Fall. Da das erste Tentakel der Terminalkante kein Reduktionstentakel ist, wird diese Information in Zeile 15 an den Löser weitergegeben. Durch die in Zeile 16 hinzugefügte Information kann der Löser folgern, dass, falls das erste Tentakel der Terminalkante kein Reduktionstentakel ist, auch das Tentakel an diesem externen Knoten der Regel kein Reduktionstentakel ist. Ein Kandidat für ein Reduktionstentakel ist dieses Tentakel dann auch nicht mehr.

## 5 Implementierung

Ist die aktuelle Kante jedoch eine Nichtterminalkante, so muss jeder anliegende Knoten separat durchgegangen werden. Falls der jeweilige Knoten ein externer Knoten ist, so kann durch die Zeilen 22 und 23 gefolgt werden: Liegt der jeweilige Knoten an der aktuellen Kante an einem Nichtreduktionstentakel an, so ist auch das Tentakel der aktuellen Regel kein Reduktionstentakel. Ist das Tentakel der aktuellen Regel jedoch ein Reduktionstentakel, so muss auch das Tentakel der aktuellen Kante eines sein. Damit ist es auch kein sofort erkennbares Reduktionstentakel mehr.

Fortsetzung Algorithmus 5.11

```
25 ruleEnd(nonterminal):
26     for reductionTentacle in candidates:
27         reductionTentacle.reduction = True
28         solver.add(null, reductionTentacle, True)
```

Sind alle Kanten in einer Regel abgearbeitet, so sind die verbliebenen Kandidaten Reduktionstentakel. An ihnen lag weder eine Nichtterminalkante an, noch hatten sie ausgehende Terminalkanten. Dem Löser können sie daher als sichere Reduktionstentakel mitgeteilt werden.

Fortsetzung Algorithmus 5.11

```
29 validate():
30     if solver.solve():
31         for t in solver.objects():
32             if t == null: continue
33             if t.nonterminal.terminal(): continue
34             reduction[t.nonterminal][t.selector] = t.reduction
35     else:
36         raise error
```

Die Hauptarbeit, sofern die Grammatik nicht sehr einfach strukturiert ist und alle Tentakel bereits bestimmt sind, obliegt nun dem Löser. Nachdem dieser das System gelöst hat werden die Ergebnisse in `reduction` eingetragen. Das Lösungsverfahren wird in Abschnitt 5.6.1 genauer beschrieben.

### 5.5.7 Produktivität

Die erste Eigenschaft einer Heapabstraktionsgrammatik gemäß Definition 2.18 ist die Produktivität. Es wird gefordert, dass aus jedem Nichtterminal eine konkrete Heapkonfiguration ableitbar ist. Jedes Nichtterminal muss so ableitbar sein, dass lediglich produktive Kanten erzeugt werden, wobei Terminalkanten von vorneherein als produktiv gelten.

Algorithmus 5.12: Quelltext der Klasse `ProductivityCheck`

```
1 productive
2 unproductive
3
4 dependencies
```

```

5  curEdges
6
7  init():
8    productive = terminals

```

Die Variable `productive` enthält zu jeder Zeit die Symbole, die bereits als produktiv erkannt wurden, `unproductive` enthält alle übrigen. Nach Abschluss der Berechnungen sollten alle Symbole in `productive` enthalten sein, `unproductive` sollte dementsprechend leer sein, da sonst unproduktive Symbole vorhanden sind.

Da die Produktivität von Nichtterminalen in der Regel von der Produktivität anderer Symbole abhängt, enthält `dependencies` für jedes Nichtterminal und jede Regel des entsprechenden Nichtterminals eine Menge von Symbolen. Sind für ein Nichtterminal und eine Regel alle Symbole in dieser Menge produktiv, so ist auch das Nichtterminal produktiv.

In `curEdges` werden die Kanten für die aktuelle Regel gesammelt. Zu Beginn werden alle Terminalsymbole als produktiv gespeichert.

Fortsetzung Algorithmus 5.12

```

9  ruleStart(nonterminal, nodes):
10   curEdges = []
11
12  edge(edge, nodes):
13   curEdges.add(edge)
14
15  ruleEnd(nonterminal):
16   dependencies[currentRule].add(curEdges)

```

Am Anfang jeder Regel muss `curEdges` geleert werden, damit nur die in der jeweiligen Regel vorhandenen Kanten enthalten sind. Jede Kante wird dann in `curEdges` gespeichert und schließlich in `dependencies` zur entsprechenden Regel hinzugefügt. Die so entstandenen Abhängigkeiten werden dann wie im Folgenden beschrieben aufgelöst.

Fortsetzung Algorithmus 5.12

```

17  validate():
18   changed = True
19   while changed:
20     changed = False
21     for nonterminal in dependencies.keys:
22       for dep in dependencies[nonterminal]:
23         resolved = True
24         for s in dep:
25           resolved &= productive.contains(s)
26         if resolved:
27           productive.add(nonterminal)
28           dependencies.remove(nonterminal)
29           changed = True
30         break

```

Fortsetzung Algorithmus 5.12

```

31 unproductive = dependencies.keys()
32 if not unproductive.empty():
33     raise error

```

Die Methode `validate()` erhält als Informationen lediglich die Abhängigkeiten in `dependencies` sowie die bereits als produktiv deklarierten Terminalsymbole. Es werden nun solange weitere Nichtterminals als produktiv markiert, bis das erste Mal keine weitere Markierung möglich ist, also keine Änderung vorgenommen wurde.

In jeder Iteration werden in Zeile 21 alle Nichtterminale durchgegangen. Für jede Regel des Nichtterminals werden in Zeile 24 alle von dieser Regel erzeugten Kanten durchgegangen. Sind alle Kanten produktiv, so wird das Nichtterminal zu `productive` hinzugefügt. Außerdem werden sämtliche Abhängigkeiten dieses Nichtterminals gelöscht und eine Änderung angezeigt.

Ist keine Änderung mehr möglich, so muss geprüft werden, ob alle Nichtterminale als produktiv markiert sind. Da die Abhängigkeiten der produktiven Nichtterminals aus `dependencies` gelöscht wurden, darf diese Variable keine Einträge mehr enthalten. Ist dies dennoch der Fall, wird eine Fehlermeldung ausgegeben.

### 5.5.8 Wachstum

Eine weitere Anforderung an jede einzelne Regel ist, dass sie wachsend ist. Diese Eigenschaft wurde in Definition 2.18 vorgestellt und sorgt dafür, dass jede Anwendung einer Regel eine Änderung am Hypergraphen bewirkt. Dafür muss jede Regel mindestens eine Terminalkante oder mehr als eine Nichtterminalkante erzeugen. Diese Eigenschaft wird von der Klasse `IncreasingCheck` getestet, wie in Algorithmus 5.13 gezeigt.

Algorithmus 5.13: Quelltext der Klasse `IncreasingCheck`

```

1 containsTerminal
2 edgeCount
3
4 ruleStart(nonterminal, nodes):
5     containsTerminal = False
6     edgeCount = 0

```

In den Variablen `containsTerminal` und `edgeCount` wird für die jeweils aktuelle Regel festgehalten, ob bereits eine Terminalkante gefunden wurde und wieviele Kanten insgesamt Teil der Regel sind.

Fortsetzung Algorithmus 5.13

```

7 edge(edge, nodes):
8     edgeCount++
9     if edge.terminal():
10        containsTerminal = True

```

Für jede Kante in der Regel wird dann zunächst `edgeCount` erhöht. Ist die Kante eine Terminalkante, wird zudem `containsTerminal` auf `True` gesetzt.

Fortsetzung Algorithmus 5.13

```

11 ruleEnd(nonterminal):
12     if not containsTerminal:
13         if edgeCount < 2:
14             raise error

```

Zum Ende jeder Regel wird schließlich geprüft, ob eine Terminalkante oder mindestens zwei Kanten vorhanden waren. Ist beides nicht der Fall, so ist die entsprechende Regel nicht wachsend und ein Fehler wird ausgegeben.

### 5.5.9 Typisierung

Die abschließende Prüfung der Typisierung nach Definition 2.18 wird von der Klasse `TypingCheck` übernommen. Hierbei wird für jeden Typen bestimmt, welche ausgehenden Kanten erzeugt werden können. Dabei wird für jeden Knoten dieses Typs geprüft, ob dessen ausgehende Kanten entweder direkt erzeugt oder von einer einzelnen anliegenden Nichtterminalkante erzeugt werden.

Algorithmus 5.14: Quelltext der Klasse `TypingCheck`

```

1  nodeout
2  typeout
3
4  ruleStart(nonterminal, nodes):
5      nodeout = []

```

Die beiden Variablen `nodeout` und `typeout` speichern jeweils die ausgehenden Kanten, einmal für jeden Knoten und einmal für alle Knotentypen. Da in `nodeout` nur die Knoten der aktuellen Regel gespeichert werden sollen, wird diese Variable zu Beginn jeder Regel geleert.

Zusätzlich zu den bekannten Methoden werden einige Hilfsfunktionen benötigt, die die Kompatibilität von Typen prüfen oder Kanten zu Knotentypen hinzufügen. Diese Methoden gehen grundsätzlich davon aus, dass Reduktionstentakel nicht betrachtet werden.

Fortsetzung Algorithmus 5.14

```

6  addTerminal(type, edge):
7      if type.hasNonterminal():
8          raise error
9      else if type.terminals.contains(edge):
10         raise error
11     else:
12         type.terminals.add(edge)

```

Fortsetzung Algorithmus 5.14

```

13 addNonterminal(type, edge):
14     if type.hasTerminals():
15         raise error
16     else if type.hasNonterminal():
17         raise error
18     else:
19         type.nonterminal = edge
20
21 compatible(type1, type2):
22     if type1.hasNonterminal():
23         return True
24     else:
25         if type2.hasNonterminal():
26             return True
27         else:
28             return type1.terminals == type2.terminals
29
30 merge(type1, type2)
31     if not type1.hasTerminals():
32         type1.nonterminal = null
33         type1.terminals = type2.terminals

```

Die Methode `addTerminal()` fügt zu einem gegebenen Typen ein neues Terminal hinzu. Wurde dem Typ bereits ein Nichtterminal zugewiesen, werden also bereits ausgehende Kanten von einer Nichtterminalkante erzeugt, so wird eine Fehlermeldung erzeugt. Ein Fehler liegt ebenso vor, wenn der Knoten bereits eine ausgehende Kante dieses Terminals besitzt.

Das Nichtterminal eines Typen wird von `addNonterminal()` gesetzt. Hat der Typ bereits ausgehende Terminalkanten oder ein Nichtterminal, so wird wiederum eine Fehlermeldung ausgegeben.

Die Frage, ob zwei Typen kompatibel sind, beantwortet `compatible()`. Dies ist der Fall, falls mindestens an einem der beiden Typen ein Nichtterminal – und damit keine Terminale – anliegt, oder die Terminalkanten beider Typen identisch sind.

Um zwei solcher Typen zusammenzuführen und dabei den konkreteren von beiden zu erhalten, wird `merge()` genutzt. Hat einer der beiden Typen Terminalkanten, so werden diese zum ersten Typ hinzugefügt.

Fortsetzung Algorithmus 5.14

```

34 edge(edge, nodes):
35     if edge.terminal():
36         if not nodeout.contains(nodes[1]):
37             nodeout[nodes[1]] = []
38         addTerminal(nodeout[node], edge)
39     else:

```

Fortsetzung Algorithmus 5.14

```

40     for i = 1 to nodes.size():
41         if not reduction(edge, i):
42             if not nodeout.contains(nodes[i]):
43                 nodeout[nodes[i]] = []
44                 addNonterminal(nodeout[nodes[i]], edge)

```

Für jede von der Regel erzeugte Kante wird diese Kante zunächst den anliegenden Knoten hinzugefügt. Falls es eine Terminalkante ist, so wird der Eintrag für diesen Knoten in `nodeout` erzeugt und die Terminalkante diesem Eintrag hinzugefügt. Ist die aktuelle Kante eine Nichtterminalkante, so wird die Kante für alle Knoten dieser Kante, die nicht an einem Reduktionstentakel anliegen, dem Eintrag in `nodeout` hinzugefügt.

Dabei werden die Hilfsmethoden `addTerminal()` und `addNonterminal()` verwendet, um mögliche Fehler wie oben beschrieben zu erkennen.

Fortsetzung Algorithmus 5.14

```

45 ruleEnd(nonterminal):
46     for node in nodeout.keys():
47         if typeout.contains(node.type):
48             if not compatible(typeout[node.type], nodeout[node]):
49                 raise error
50             else:
51                 merge(typeout[node.type], nodeout[node])
52         else:
53             typeout[node.type] = nodeout[node]

```

Am Ende jeder Regel werden die Typen aller Knoten mit den globalen Typen verglichen. Es wird also für jeden Knoten in `nodeout` geprüft, ob sein Typ bereits in `typeout` vorhanden ist. Ist dies der Fall, so wird die Kompatibilität der beiden geprüft. Sind sie kompatibel, so werden die beiden mittels `merge()` zusammengeführt, ansonsten liegt ein Fehler vor. Ist der Typ noch nicht in `typeout` vorhanden, so wird er aus `nodeout` übernommen.

## 5.6 Weitere Klassen

Neben den bereits beschriebenen Klassen zur Fehlerüberprüfung und den von ANTLR erstellten Klassen gibt es noch einige weitere Bausteine. So wurde für die Identifikation der Reduktionstentakel in Abschnitt 5.5.6 ein Löser für Systeme von Folgerungen benötigt, der nun vorgestellt werden soll. Zudem bestehen Abhängigkeiten zwischen den verschiedenen Klassen, so dass diese in einer gewissen Reihenfolge ausgeführt werden müssen. Außerdem wird eine Klasse benötigt, die alle vorgestellten Funktionen bündelt und dem Benutzer eine möglichst einfache Schnittstelle zur Verfügung stellt.

### 5.6.1 Löser für Systeme von Folgerungen

Wie bereits in Abschnitt 5.5.6 erwähnt, werden Reduktionstentakel durch das Lösen eines Systems von Folgerungen ermittelt. Ein solches System von Folgerungen wird

## 5 Implementierung

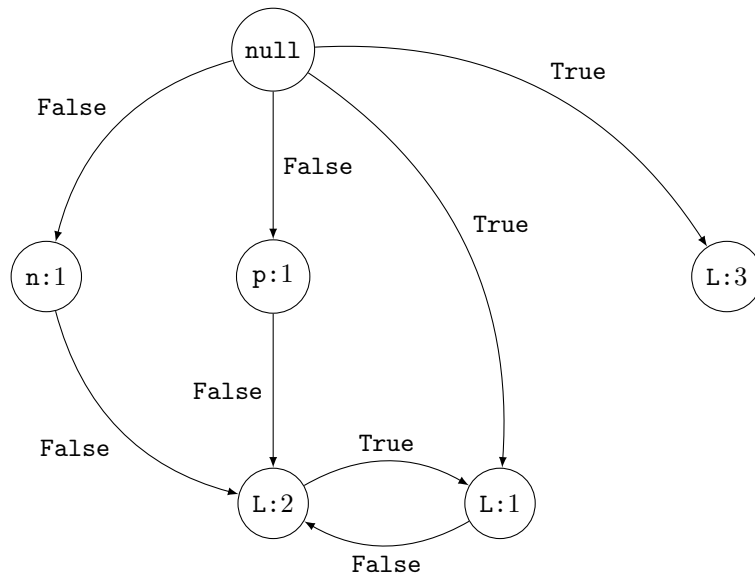
stets als Graph angegeben. Knoten entsprechen dabei Tentakel, für die ermittelt werden soll, ob es Reduktionstentakel sind, Kanten möglichen Folgerungen zwischen den Tentakeln. Die Folgerungen werden im Folgenden als **True**- oder **False**-Folgerungen bezeichnet.

Eine **True**-Folgerung von einem Tentakel  $T_1$ , dem Antezedens, zu einem Objekt  $T_2$ , dem Sukzedens, sagt aus, dass  $T_2$  ein Reduktionstentakel ist, falls  $T_1$  eines ist. Im Gegensatz dazu besagt die **False**-Folgerung, dass die  $T_2$  kein Reduktionstentakel ist, falls  $T_1$  ebenfalls keines ist.

Um gewisse Anfangsinformationen zu modellieren, existiert ein ausgezeichnete Knoten **null**. Eine **True**-Folgerung von **null** aus besagt, dass das Sukzedens ein Reduktionstentakel ist, eine **False**-Folgerung entsprechend, dass es keines ist. Der Aufbau eines solchen Systems von Folgerungen ist in Beispiel 5.1 dargestellt.

**Beispiel 5.1.** Sei die Grammatik für das Nichtterminal  $L$  aus Beispiel 2.4 gegeben. Es wird ein System von Folgerungen aufgestellt, um Reduktionstentakel zu identifizieren. Es ergibt sich der Graph aus Abbildung 5.7.

Abbildung 5.7: System von Folgerungen für Beispiel 2.4



Die Anfangsinformationen ergeben sich gemäß dem Vorgehen in Abschnitt 5.5.6 aus den Terminalkanten. Für den ersten Knoten der Terminalkanten  $n$  und  $p$  kann offensichtlich mindestens eine ausgehende Kante erzeugt werden. Da dieser Knoten identisch mit dem zweiten externen Knoten von  $L$  ist, ergeben sich die Folgerungen von  $n:1$  und  $p:1$  nach  $L:2$ .

Es ist jedoch nicht möglich, eine vom ersten oder dritten externen Knoten ausgehende Kante zu erzeugen, daher existieren zwei **True**-Folgerungen von **null** zu  $L:1$  und  $L:3$ .

Da das erste Tentakel von  $L$  auch am zweiten externen Knoten anliegt, ergeben sich die Folgerungen zwischen  $L:1$  und  $L:2$ : Eine *False-Folgerung* von  $L:1$  nach  $L:2$  und eine *True-Folgerung* von  $L:2$  nach  $L:1$ .

Ein solches System kann nun durch das in Algorithmus 5.15 angegebene Verfahren gelöst werden.

Algorithmus 5.15: Quelltext der Methode zum Lösen der Klasse `InferenceSolver`

```

1  inferences
2
3  solve():
4      if inferences[null] == null:
5          raise error
6
7      todo = inferred = inferences[null]
8
9      while not todo.empty():
10         item = todo.pop()
11
12         for inf in inferences[item]:
13             if inferred.contains(inf.consequence):
14                 inf.consequence.check(item, inf)
15             else:
16                 if inf.consequence.infer(item, inf):
17                     inferred.add(inf.consequence)
18                     todo.add(inf.consequence)
19
20     for objects in inferences:
21         if not inferred.contains(objects):
22             raise error

```

Die aus der Grammatik gewonnenen Folgerungen sind in `inferences` abgelegt. Für jedes Tentakel ist hier eine Menge von Folgerungen, die jeweils aus einem Tentakel und einer Information über die Folgerung bestehen, abgelegt. Die Information ist in diesem Fall, ob es sich um eine *True-Folgerung* oder eine *False-Folgerung* handelt.

Falls es keine Anfangsinformationen gibt ist dies ein Fehler und es wird abgebrochen. Prinzipiell könnten die Folgerungen unter gewissen Umständen auch ohne Anfangsinformationen durch das Finden von bedingten Widersprüchen aufgelöst werden, für das Ermitteln von Reduktionstentakeln reicht das dargestellte Vorgehen jedoch auch.

Die Liste `inferred` enthält alle Tentakel, für die die gesuchte Eigenschaft bereits ermittelt wurde, `todo` enthält alle Tentakel, deren Eigenschaft ermittelt wurde, aber deren Folgerungen noch nicht weiter betrachtet wurden. Zu Beginn der Methode `solve()` sind in beiden Listen alle Knoten enthalten, für die Anfangsinformationen vorliegen.

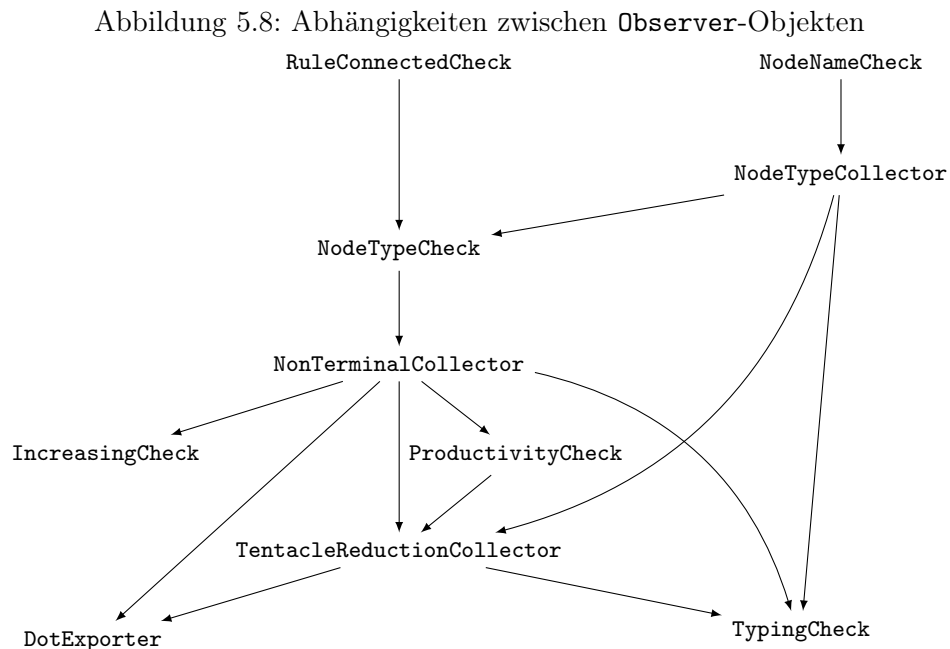
Es wird nun so lange nach Folgerungen gesucht, bis `todo` keine weiteren Tentakel mehr enthält. Für jedes Tentakel in `todo` werden dabei alle hinterlegten Folgerungen durchgegangen. Wurde die Eigenschaft des Sukzedens bereits ermittelt, so wird diese

lediglich nochmal überprüft, um mögliche Widersprüche zu ermitteln. Ansonsten wird sie erstmals ermittelt und, falls dies möglich war, das Sukzedens zu `inferred` und `todo` hinzugefügt. Eine Folgerung kann nicht möglich sein, falls die Eigenschaft des Antezedens `True` ist, aber eine `False`-Folgerung vorliegt.

Zum Schluss werden alle Tentakel daraufhin überprüft, ob die gesuchte Eigenschaft ermittelt werden konnte. Ist dies nicht der Fall wird ein Fehler ausgegeben.

### 5.6.2 Abhängigkeiten zwischen Observer-Objekten

Zwischen den beschriebenen `Observer`-Objekten existieren eine ganze Reihe von Abhängigkeiten. So muss beispielsweise zwischen Terminalen und Nichtterminalen unterschieden werden, bevor die Grammatik auf Produktivität getestet werden kann, welche wiederum Voraussetzung für das Ermitteln der Reduktionstentakel ist. Auch wenn die einzelnen Abhängigkeiten nicht im Detail erläutert werden sollen, ist in Abbildung 5.8 ein Graph der bestehenden Abhängigkeiten angegeben.



Da die bestehenden Abhängigkeiten keine zyklischen Abhängigkeiten enthalten, ergibt sich ein azyklischer Graph, der mit dem bereits für den `ProductivityCheck` genutzten Verfahren aufgelöst werden kann. Zyklische Abhängigkeiten können hierbei ausgeschlossen werden, da sie auch bei einer manuellen Ausführung zwangsläufig in eine Sackgasse führen müssen.

Um die Abhängigkeiten aufzulösen, existiert die Klasse `DependencyScheduler`. Jedes der vorgestellten `Observer`-Objekte liefert von sich aus Informationen darüber, von

welchen anderen Objekten es abhängig ist. Die Ausführung aller **Observer**-Objekte in einer korrekten Reihenfolge funktioniert dann wie in Algorithmus 5.16 angegeben.

Aufgrund der weiteren Verwendung ist die Klasse als **Iterator** implementiert. Bei jedem Aufruf der Methode `next()` wird dabei eine Menge von Objekten generiert, die nun ausgeführt werden können: Sie wurden weder bereits ausgeführt, noch sind sie noch von Objekten abhängig. Die Abhängigkeiten dieser Objekte werden dabei entfernt.

Algorithmus 5.16: Quelltext der Klasse `DependencyScheduler`

```

1 remaining
2 dependencies
3
4 reset():
5     remaining = dependencies
6
7 next():
8     cur = []
9     for obj in remaining:
10        if remaining[obj].empty():
11            cur.add(obj)
12
13    if cur.empty():
14        raise error
15
16    for obj in cur:
17        remaining.remove(obj)
18        for o in remaining:
19            remaining[o].remove(obj)
20
21    return cur

```

Sämtliche Abhängigkeiten werden zu Beginn in `dependencies` eingefügt. Für jedes Objekt ist hier die Menge der Objekte gespeichert, von denen dieses abhängig ist. Wird der **Iterator** mittels `reset()` initialisiert, so werden alle Abhängigkeiten nach `remaining` kopiert. Diese Variable enthält während eines Laufs alle übrigen Abhängigkeiten.

Wird die nächste Liste von Objekten mittels `next()` angefordert, so wird zunächst eine leere Liste angelegt. Alle Objekte, die in `remaining` keine Abhängigkeiten mehr haben, werden in diese Liste eingefügt. Sind keine Objekte ohne Abhängigkeiten mehr vorhanden, so wird die Methode abgebrochen. Dies kann sowohl das erfolgreiche Ende eines Laufs als auch unerfüllbare Abhängigkeiten signalisieren, abhängig davon ob noch Objekte in `remaining` vorhanden sind.

Anschließend werden alle durch die ausgewählten Objekte gelösten Abhängigkeiten aus `remaining` entfernt, es werden also die Einträge der gewählten Objekte selber, als auch die Objekte aus den Einträgen anderer Objekte gelöscht. Zurückgegeben wird die am Anfang bestimmte Menge von Objekten `cur`.

Um die verschiedenen **Observer**-Objekte auszuführen nutzt die Klasse `PhaseHandler` die Klasse `DependencyScheduler`.

### 5.6.3 Hinzufügen zum Grammatikobjekt

Falls eine Eingabegrammatik fehlerfrei ist, so muss sie in ein von Juggernaut verwertbares Format gebracht werden. Intern werden dazu die Klassen `Alphabet` und `HRGrammar` verwendet, um das Alphabet und die Grammatik zu speichern.

Das Einfügen der eingegebenen Grammatik in das `HRGrammar`-Objekt wird dabei in zwei Phasen abgewickelt. Zunächst werden alle benötigten Informationen über die Nichtterminale in Objekten der Klasse `NonterminalInfo` gesammelt. Dann werden mithilfe dieser Informationen die Nichtterminale in das Alphabet und die Regeln in die Grammatik eingefügt.

Algorithmus 5.17: Quelltext der Klasse `NonterminalInfo`

```

1 class NonterminalInfo:
2     String label
3     NodeType[] typeSequence
4     boolean[] entrance
5     Nonterminal nonterminal

```

In dieser Klasse sind neben dem Nichtterminal selber und seinem Namen auch der an jedem Tentakel anliegenden Knotentyp sowie die Information, ob es sich um ein Reduktionstentakel handelt, enthalten.

Die in Zeile 8 von Algorithmus 5.18 verwendete Regel `rule` liefert als Ergebnis das Nichtterminal und den Hypergraphen der entsprechenden Regel. Die Konstruktion dieses Hypergraphen ist rein technischer Natur und wird daher nicht weiter behandelt.

Algorithmus 5.18: Quelltext der Klasse `HRGCollectorGrammar`

```

1 rules
2 @init {
3     for nt in nonterminals:
4         nI = NonterminalInfo()
5         nI.label = nt
6         nonterminalMap[nt] = nI
7 }
8 : (^ ( RULE rule {
9     nI = nonterminalMap[$rule.nonterminal]
10    if nI.typeSequence == null:
11        nI.typeSequence = NodeType[$rule.graph.externalCount()]
12        for i = 1 to nI.typeSequence.length():
13            nI.typeSequence[i] = $rule.graph.external(i).type()
14    if nI.entrance == null:
15        nI.entrance = Boolean[$rule.graph.externalCount()]
16        for i = 1 to nI.typeSequence.length():
17            nI.entrance[i] = not (
18                $rule.graph.external(i).tentacle.reduction() or
19                $rule.graph.external(i).isNull()
20            )
21    } ))*
22 ;

```

In der ersten Phase wird zunächst jedes Nichtterminal mit seinem Namen erstellt und in der Variablen `nonterminalMap` gespeichert. Für jede Regel werden nun die Typen der externen Knoten des entsprechenden Nichtterminals in der Variablen `typeSequence` vermerkt und die Reduktionstentakel in `entrance` markiert. Danach wird die zweite Phase durchgeführt.

Algorithmus 5.19: Quelltext der Klasse `HRGBuilderGrammar`

```

1 rules
2 @init {
3     for nI in nonterminalMap:
4         nI.nonterminal = alphabet.addNonTerminal(nI)
5 }
6 : (^ ( RULE rule {
7     nI = nonterminalMap[$rule.nonterminal]
8     grammar.addRule(nI.nonterminal, $rule.graph)
9 } ))*
10 ;

```

Zunächst werden alle Nichtterminale in das Alphabet eingefügt. Anschließend wird das `HRGrammar`-Objekt um die einzelnen Regeln ergänzt.

#### 5.6.4 Schnittstelle zum Nutzer

Eine wichtige Eigenschaft eines Programms oder einer Bibliothek ist auch die Benutzbarkeit. Auf die entscheidenden Funktionen sollte möglichst einfach zugegriffen werden können, ohne dass dabei auf interne Details eingegangen werden muss.

Die Kernaufgabe der vorgestellten Klassen ist es, eine Grammatik auf Fehler zu überprüfen, weitere Eigenschaften zu ermitteln und in das von Juggernaut gewünschte Format zu bringen. Die Methode `readGrammar()` wird von der Klasse `HRGGrammarReader` implementiert und erfüllt diese beiden Aufgaben, so dass sie eine Grammatik aus einer Datei in ein gegebenes Grammatikobjekt einfügt. Dafür werden Lexer, Parser, Baumparser und alle `Observer`-Objekte erstellt, alle Fehlerüberprüfungen durchgeführt und die finale Datenstruktur zur Grammatik hinzugefügt.

Algorithmus 5.20: Quelltext der Klasse `HRGGrammarReader`

```

1 readGrammar(filename, grammar):
2     lexer = HRGLexer(ANTLRFileStream(filename))
3     tokens = CommonTokenStream(lexer)
4
5     parser = HRGParser(tokens)
6     tree = parser.rules().getTree()
7     nodestream = CommonTreeNodeStream(tree)
8
9     treeparser = HRGWalkerGrammar(nodestream)

```

Zunächst wird der Lexer aus Abschnitt 5.2 erstellt und erhält die Eingabedatei. Der Lexer erzeugt als Ausgabe einen `CommonTokenStream`, der dem damit erstellten Parser

## 5 Implementierung

als Eingabe dient. Der Befehl `parser.rules()` führt nun das Parsen durch und erzeugt einen Syntaxbaum, der dem Baumparser übergeben wird.

Fortsetzung Algorithmus 5.20

```
10 phases = PhaseHandler(treeparser)
11
12 register(NodeTypeCheck(treeparser))
13 register(NodeNameCheck(treeparser))
14 register(RuleConnectedCheck(treeparser))
15 register(NodeTypeCollector(treeparser))
16 register(NonTerminalCollector(treeparser))
17 register(ProductivityCheck(treeparser))
18 register(TentacleReductionCollector(treeparser))
19 register(IncreasingCheck(treeparser))
20 register(TypingCheck(treeparser))
21
22 if not dotFolder == null:
23     register(DotExporter(treeparser, dotFolder, dotPrefix))
24 if not normalizedGrammar == null:
25     register(GrammarNormalizer(treeparser, normalizedGrammar))
26
27 phases.prepare()
28
29 while not phases.done():
30     nodestream.reset()
31     treeparser.setObserver(phases)
32     if not treeparser.rules():
33         raise error
```

Nun müssen alle Fehlerüberprüfungen anhand der `Observer`-Objekte durchgeführt werden. Dazu wird ein `PhaseHandler` erstellt und alle `Observer`-Objekte werden an zentraler Stelle registriert. Die Variablen `dotFolder` und `normalizedGrammar` stellen dabei Konfigurationen dar, die bestimmen, ob Graphen in Form von Dateien im DOT-Format gespeichert werden oder eine normalisierte Grammatik ausgegeben wird.

Anschließend wird mit `phases.prepare()` die Auflösung der Abhängigkeiten zwischen den `Observer`-Objekten gestartet. Solange dann noch Objekte abzuarbeiten sind, wird der `treeparser` wiederholt mit den jeweiligen Objekten aufgerufen. Bei der Auswahl der Objekte wird intern auf `DependencyScheduler.next()` zurückgegriffen.

Falls ein Fehler erkannt wird, so gibt `treeparser.rules()` diesen zurück und die Ausführung von `readGrammar()` entsprechend abgebrochen.

Fortsetzung Algorithmus 5.20

```
34 nodestream.reset()
35 collector = HRGCollectorGrammar(nodestream, grammar)
36 collector.rules()
```

Fortsetzung Algorithmus 5.20

```
37  nodestream.reset()
38  builder = HRGBuilderGrammar(nodestream, grammar)
39  builder.rules()
```

Ist das Verfahren ohne einen Fehler zu finden verlaufen, werden als letztes die in Abschnitt 5.6.3 vorgestellten Klassen ausgeführt, die die eingelesene Grammatik dem Objekt `grammar` hinzufügen.



## 6 Fazit

Zum Abschluss dieser Arbeit sollen die Inhalte und Hauptergebnisse kurz zusammengefasst werden. Anschließend folgt ein Ausblick auf mögliche Erweiterungen von Juggernaut und deren Auswirkungen auf die vorgestellten Konzepte.

### 6.1 Zusammenfassung

Zu Beginn dieser Arbeit wurde das formale Modell vorgestellt, auf dem das Verifikationstool Juggernaut aufbaut. Ausgehend von Hypergraphen wurden Kantenersetzungen auf Hypergraphen und darauf aufbauende Grammatiken vorgestellt.

Anschließend wurden sowohl reguläre also auch kontextfreie Sprachen eingeführt, sowie deren Benutzung zur lexikalischen und syntaktischen Analyse. Dabei wurde als Parsingstrategie Top-Down-Parsing betrachtet, insbesondere die von ANTLR implementierte  $LL(*)$ -Strategie.

Darauf aufbauend wurde die semantische Analyse erläutert. Dort wurden zunächst Nichtterminalsymbole sowie Reduktionstentakel automatisch berechnet, um dem Benutzer die manuelle Eingabe zu ersparen. Im Folgenden wurde die Konsistenz der Grammatik bezüglich der Benennung und der einheitlichen Typisierung ihrer Knoten geprüft und getestet, ob die Hypergraphen zusammenhängend sind. Schließlich wurden die drei Eigenschaften überprüft, die neben der lokalen Konkretisierbarkeit für eine Heapabstraktionsgrammatik notwendig sind: Die Grammatik muss produktiv, wachsend und typisiert sein.

Abschließend hat diese Arbeit die Implementierung betrachtet. Zunächst wurden Lexer und Parser mithilfe des vorgestellten Konzepts des  $LL(*)$ -Parsing mit ANTLR realisiert, darüber hinaus wurde die semantische Analyse durch einen mit ANTLR erstellten Baumparser vorbereitet. Die einzelnen Schritte der semantischen Analyse wurden dann als einzelne Klassen in Java implementiert und hier in Pseudocode vorgestellt. Ebenso wurden ausgewählte, konzeptuell interessante, Hilfsklassen erläutert, sowie das letztendliche Zusammenspiel all dieser Komponenten dargestellt.

### 6.2 Ausblick

Da die Arbeit an Juggernaut und der zugrundeliegenden Theorie und Formalisierung keineswegs abgeschlossen ist, ist auch mit weiteren Änderungen der Anforderungen an eine Grammatik zu rechnen. Neben der Funktionalität sollen daher auch neue Anforderungen flexibel in das bestehende System integriert werden können.

Die Aufteilung der einzelnen Funktionen in separate `Observer`-Objekte bietet genau den Vorteil, dass neue Überprüfungen durch ein oder mehrere neue `Observer`-Objekte realisiert werden können, ohne etwas am bestehenden System ändern zu müssen. Nicht mehr benötigte Tests können idealerweise durch das Entfernen einer einzigen Zeile entfernt werden.

Eine geplante Änderung, die in dieser Arbeit nicht betrachtet wurde, ist eine Erweiterung der Knotentypen um eine Hierarchie. Da das Ziel von Juggernaut die Verifikation von Java-Programmen ist, liegt es nahe, für Knoten eine zu Java ähnliche Typhierarchie einzuführen. Eine theoretische Integration einer Typhierarchie in die vorgestellte Formalisierung existiert bereits [HJ11]. Neben der Typhierarchie selber müssten sämtliche Vergleiche zwischen Knotentypen durch Vergleiche bezüglich dieser Hierarchie ersetzt werden, der Großteil der Objekte müsste aber nicht geändert werden.

Eine andere Erweiterung betrifft das direkte Feedback an den Benutzer. Die jetzige Lösung prüft die Grammatik auf Gültigkeit, kann aber nicht die Intention des Benutzers erkennen. Ein Tippfehler bei der Verwendung eines Nichtterminals von Rang zwei in einer Regel kann beispielsweise nicht erkannt werden. Die als Nichtterminal gedachte Kante würde in diesem Fall als Terminal gespeichert. Der Fehler würde frühestens beim Einlesen des Java-Programms auffallen, da die Terminalsymbole exakt wie die Selektoren der Java-Klasse heißen müssen, dies aber durch den Tippfehler nicht gegeben sein kann.

Dieses Problem könnte durch eine direkte grafische Darstellung der Grammatik vermieden werden. Während in der vorliegenden Implementierung von einem separaten, hier nicht detailliert vorgestellten `Observer`-Objekt alle Regeln als Grafiken im DOT-Format ausgegeben werden, könnte diese Information direkt als Grafik an den Benutzer weitergegeben werden, bevor mit der Verifikation begonnen wird. Der Unterschied zwischen einem Terminalsymbol und einem Nichtterminalsymbol würde in dieser Darstellung sofort auffallen.

Ein weiterer Blick in die Zukunft zeigt die Möglichkeit, Heapabstraktionsgrammatiken direkt aus einer implementierten Datenstruktur zu generieren [Jan10]. Dieser Ansatz hat den Vorteil, dass sich der Benutzer nicht mehr mit dem konkreten Design einer Grammatik beschäftigen muss, sondern diese nur noch prüfen muss. Leider entstehen bei dem involvierten Lernprozess bisher nur wenig intuitive Grammatiken.

Insgesamt existieren sowohl für den theoretischen Ansatz zur Verifikation als auch für Juggernaut noch diverse Möglichkeiten zur Erweiterung und Verbesserung. Diese Arbeit ist dabei vor allem eine Verbesserung in Richtung Nutzbarkeit und Flexibilität.

# Literaturverzeichnis

- [AB03] A. Asteroth and C. Baier. *Theoretische Informatik*. Informatik - Pearson Studium. Pearson Studium, 2003.
- [ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.
- [BA95] Peter Baeumle and Heinz Alenfelder. *Compilerbau, eine praxisorientierte Einführung*. S+W Steuer- und Wirtschaftsverlag, 1995.
- [BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.
- [Bü60] J. R. Büchi. Weak second-order arithmetic and finite automata. *Mathematical Logic Quarterly*, 6(1-6):66–92, 1960.
- [Elg61] C.C. Elgot. Decision problems of finite automata design and related arithmetics. *Trans. Amer. Math. Soc.*, 98:21–51, 1961.
- [HJ11] Jonathan Heinen and Christina Jansen. Juggernaut - An Abstract Jvm. Technical Report AIB-2011-21, RWTH Aachen, September 2011.
- [HMU02] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullmann. *Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie*. Pearson Studium, München, 2., überarbeitete aufl. edition, 2002.
- [HNR10] Jonathan Heinen, Thomas Noll, and Stefan Rieger. Juggernaut: Graph grammar abstraction for unbounded heap structures. *Electronic Notes in Theoretical Computer Science*, 266:93 – 107, 2010. Proceedings of the 3rd International Workshop on Harnessing Theories for Tool Support in Software (TTSS).
- [Jan10] Christina Jansen. Konstruktion und Inferenz von Heapabstraktionsgrammatiken. Diplomarbeit, RWTH Aachen, Informatik, 2010.
- [JHKN11] Christina Jansen, Jonathan Heinen, Joost-Pieter Katoen, and Thomas Noll. A Local Greibach Normal Form for Hyperedge Replacement Grammars. Technical Report AIB-2011-04, RWTH Aachen, January 2011.
- [Kow10] Benjamin Kowarsch. Antlr v3 grammar for objective modula-2. <http://modula2.net/>, <http://objective.modula2.net/sources/objm2.g>, 2010. aufgerufen am 19. September 2011.

*Literaturverzeichnis*

- [Par07] Terence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Programmers. Pragmatic Bookshelf, first edition, May 2007.
- [PF11] Terence Parr and Kathleen Fisher. LL(\*): The Foundation of the ANTLR Parser Generator. Technical report, University of San Francisco, 2011.
- [Res] AT&T Research. The dot language. <http://www.graphviz.org/content/dot-language>. aufgerufen am 31. August 2011.
- [Roz97] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific, 1997.
- [Tra57] B.A. Trakhtenbrot. On operators realizable in logical nets. *Dokl. Akad. Nauk. SSSR*, 112:1005–1007, 1957.