Institut für Softwaretechnik

UNIVERSITÄT KOBLENZ · LANDAU

Fachbereich 4: Informatik

Extractor Description Language

Eine Beschreibungssprache für Extraktoren zur Erzeugung von TGraphen

A description language for extractors to generate TGraphs

Masterarbeit

im Studiengang Informatik

vorgelegt von:

B.Sc. Daniel Dominik Janke

danijank@uni-koblenz.de

(Matr.-Nr.: 206210018)

Gutachter:	Prof. Dr. Jürgen Ebert, Institut für Softwaretechnik, Fachbereich 4
	Dr. Volker Riediger, Institut für Softwaretechnik, Fachbereich 4
zus. Betreuer:	Dr. Daniel Bildhauer, Institut für Softwaretechnik, Fachbereich 4

Koblenz, im Oktober 2012

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Mit der Einstellung dieser Arbeit in die Bibliothek bin ich einverstanden.

Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.

Koblenz, den . . . .

Abstract

In a software reengineering task legacy systems are adapted computer-aided to new requirements. For this an efficient representation of all data and information is needed. TGraphs are a suitable representation because all vertices and edges are typed and may have attributes. Further more there exists a global sequence of all graph elements and for each vertex exists a sequence of all incidences.

In this thesis the „Extractor Description Language" (EDL) was developed. It can be used to generate an extractor out of a syntax description, which is extended by semantic actions. The generated extractor can be used to create a TGraph representation of the input data. In contrast to classical parser generators EDL support ambiguous grammars, modularization, symbol table stacks and island grammars. These features simplify the creation of the syntax description.

The collected requirements for EDL are used to determine an existing parser generator which is suitable to realize the requirements. After that the syntax and semantics of EDL are described and implemented using the suitable parser generator. Following two extractors one for XML and one for Java are created with help of EDL. Finally the time they need to process some input data is measured.

Zusammenfassung

Um Altsysteme beim sogenannten Software-Reengineering an neue Anforderungen mittels Computerunterstützung anpassen zu können, wird eine effiziente Repräsentation der vorliegenden und im Laufe des Prozesses anfallenden Daten und Informationen benötigt. Als geeignete Repräsentationsform haben sich TGraphen herausgestellt, da sowohl Knoten als auch gerichtete Kanten typisiert sind und über Attribute verfügen können. Darüber hinaus besteht eine globale Anordnung aller Elemente des Graphen sowie eine Reihenfolge der Inzidenzen jedes Knoten.

In dieser Arbeit wurde die „Extractor Description Language“ (EDL) entwickelt, um die Syntax der Eingabedaten zu beschreiben und mittels frei definierbarer semantischer Aktionen einen TGraphen aufzubauen. Im Gegensatz zu klassischen Parsergeneratoren wie ANTLR werden mehrdeutige Grammatiken, Modularisierung, Inselgrammatiken und Symboltabellen-Stacks von EDL unterstützt, wodurch die Erstellung der Syntaxbeschreibung vereinfacht wird.

Nach der Erhebung der Anforderungen an EDL, werden zunächst die existierenden Parsergeneratoren daraufhin untersucht, welcher zur Realisierung der Anforderungen am geeignetsten ist. Im Anschluss wird die Syntax sowie die Semantik von EDL beschrieben und unter Nutzung des geeigneten Parsergenerators implementiert. Anhand von zwei mittels EDL generierten exemplarischen Extraktoren für XML und Java wird der zeitliche Aufwand zum Verarbeiten der Eingabe gemessen.

Danksagung

An dieser Stelle möchte ich all jenen danken, die mir durch ihre fachliche und persönliche Unterstützung bei der Erstellung dieser Masterarbeit geholfen haben.

Mein erster Dank geht an meine Betreuer Prof. Dr. Jürgen Ebert, Dr. Volker Riediger und Dr. Daniel Bildhauer. Sie haben mir ermöglicht, meine Masterarbeit über ein spannendes Thema zu schreiben. Durch ihre persönliche Betreuung halfen sie mir bei der Verwirklichung dieser Arbeit. Durch gezielte konstruktive Kritik und Diskussionen brachten sie mir das wissenschaftliche Arbeiten näher.

Meinem Bruder Andreas Janke möchte ich für die Verbesserungsvorschläge und Formulierungshilfen bei meiner Masterarbeit danken. Zusammen mit meiner Familie und Vitali Keppel war er eine wertvolle moralische Stütze.

Mein letzter Dank geht an meine Eltern, die mich stets unterstützt und mir mein Studium ermöglicht haben.

Inhaltsverzeichnis

1 Einleitung **1**

2 TGraphen **5**

3 Syntaktische Analyse **9**

3.1 Formale Definition von Sprachen 9
3.2 Metasprachen 12
3.2.1 Backus-Naur-Form (BNF) 13
3.2.2 Erweiterte Backus-Naur-Form (EBNF) 15
3.2.3 Syntax Definition Formalism (SDF) 19
3.3 Top-Down-Algorithmen 31
3.3.1 LL(k) Parsing 31
3.3.2 GLL Parsing 37
3.4 Bottom-Up-Algorithmen 54
3.4.1 LR(k) Parsing 54
3.4.2 GLR Parsing 57

4 Anforderungen **101**

4.1 Funktionale Anforderungen 101
4.1.1 Grammatikdefinition 102
4.1.2 Semantische Aktionen 102
4.1.3 Parsing 104
4.1.4 Debugging 104
4.2 Nicht-funktionale Anforderungen 105

5 Parsergeneratoren **107**

5.1 Übersicht über Parsergeneratoren 107
5.2 Stratego/XT 110
5.2.1 Allgemeine Funktionsweise 110
5.2.2 Eingabegrammatik 112
5.2.3 Erzeugung der Parse-Tabelle 113

5.2.4 Interner Parse-Forest 117
5.2.5 `ITreeBuilder` 119

6 Extractor Description Language (EDL) 127
6.1 Beispiel 127
6.1.1 TGraph-Schema 127
6.1.2 EDL-Grammatik 132
6.1.3 Erzeugung des Graphen 140
6.2 Semantische Aktionen 143
6.2.1 Zugriff auf Elemente einer Regel 144
6.2.2 Generalisierungskonzept 149
6.2.3 Schema-spezifische semantische Aktionen 151
6.2.4 Symboltabellen 155
6.2.5 Nutzerspezifische semantische Aktionen 159
6.3 Inselgrammatik 160

7 Konzeptioneller Systementwurf 161
7.1 Vorverarbeitung 162
7.2 Erzeugung des Graphen aus der Eingabedatei 163

8 Parser 167
8.1 Beispiel 169
8.2 Repräsentation einer Regel 192
8.3 Realisierung des Stacks 196
8.4 TreeTraverser 207
8.5 Debug-Ausgaben 210
8.6 Realisierung des Symboltabellen-Stacks 215
8.7 Inselgrammatiken 217
8.8 Ausführung der semantischen Aktionen 218

9 EDL-Processor 227
9.1 EDL2EDLGraph 228
9.1.1 EDL-Grammatik 229
9.1.2 EDL-Schema 235
9.1.3 Implementation der `EDL2EDLGraph`-Komponente 239
9.2 SDF-Generator 243
9.3 GraphBuilder-Generator 244
9.3.1 Desugarer 245
9.3.2 CodeGenerator 256

10 Beispielanwendungen **271**
10.1 XML 271
10.2 Java 280

11 Zeitmessung **291**
11.1 Messverfahren 291
11.2 Zeitmessung mithilfe einer XML-Grammatik 292
11.3 Zeitmessung mithilfe einer Java-Grammatik 294
11.4 Zusammenfassung der Ergebnisse 296

12 Zusammenfassung und Ausblick **299**
12.1 Zusammenfassung 299
12.2 Ausblick 300
12.3 Fazit 301

Literaturverzeichnis **303**

A EDL-Schema **III**

B Inhalt der CD **XXI**

Abbildungsverzeichnis **XXIII**

Tabellenverzeichnis **XXVI**

Listings **XXVII**

Index **XXXI**

1 Einleitung

In der Wirtschaft werden heutzutage Softwaresysteme eingesetzt, die die Arbeit in nahezu allen Bereichen unterstützen und vereinfachen soll. Im Laufe der Zeit können sich jedoch die Anforderungen an ein Softwaresystem verändern. So ist beispielsweise mit der steigenden Verbreitung des Internets die Möglichkeit entstanden, Produkte und Dienstleistungen weltweit anzubieten. Dies hat zur Konsequenz, dass die bereits existierenden Softwaresysteme angepasst werden müssen, um die Einbindung ins Web zu ermöglichen.

Aus der Notwendigkeit Altsysteme anzupassen hat sich das Software-Reengineering entwickelt. Nach [Bor91] umfasst Reengineering alle Aktivitäten, deren Ziel die qualitative Verbesserung und Aufbereitung von Software ist. Wie in Abbildung 1.1 zu erkennen, muss die bestehende Implementation durch ein sogenanntes „Reverse Engineering“ abstrahiert werden. Bei diesem Schritt wird aus der Implementation beispielsweise die zugrunde liegende Softwarearchitektur extrahiert und die realisierten Anforderungen bestimmt. Im Anschluss werden die bei der Abstraktion entstandenen Artefakte angepasst, bevor beim sogenannten „Forward Engineering“ die Anpassungen auf allen Abstraktionsebenen bis hin zur Implementation vorgenommen werden.

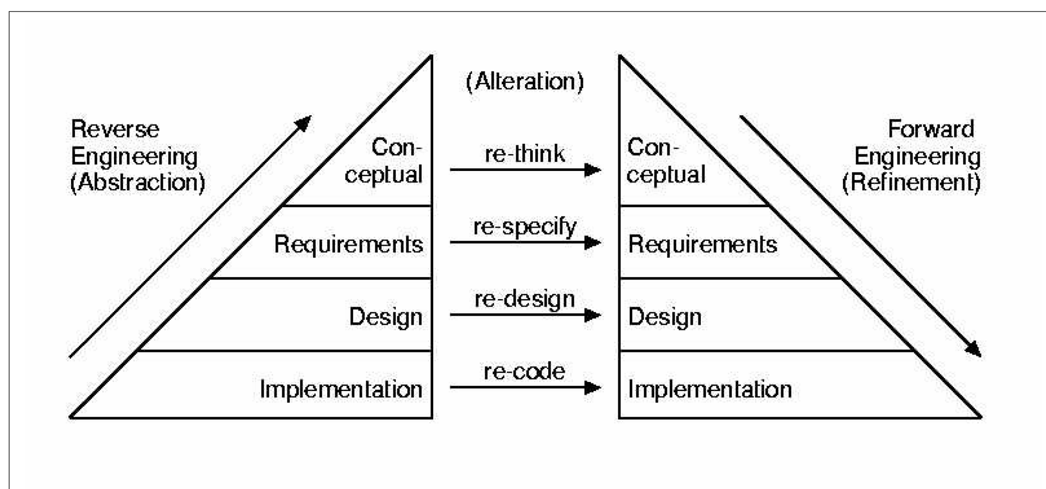


Abbildung 1.1: Software Reengineering nach Byrne. Quelle: [Byr92]

Motivation

Um das Software-Reengineering mit Computerunterstützung durchführen zu können, wird eine Repräsentation der vorliegenden und im Laufe des Prozesses anfallenden Daten und Informationen benötigt. Als geeignete Repräsentationsformen haben sich neben relationalen Datenbanken und Ontologien auch Graphen herausgestellt. Insbesondere die am Institut für Softwaretechnik der Universität Koblenz-Landau entwickelten *TGraphen* eignen sich zur effizienten Darstellung von Daten und Beziehungen, da sowohl Knoten und Kanten als auch der Graph selbst typisiert sind und über Attribute verfügen können. Darüber hinaus besteht eine globale Anordnung aller Knoten und Kanten im Graphen sowie eine Reihenfolge der Inzidenzen an Knoten. Jede Kante besitzt zwar eine Richtung, kann aber auch ungerichtet betrachtet werden.

Zur automatisierten Extraktion eines Graphen aus den vorliegenden Daten und Informationen können *Parsergeneratoren* genutzt werden. Diese benötigen zunächst eine formale Beschreibung der Eingabe-Syntax, mit deren Hilfe die Eingabe erkannt werden kann. Die Beschreibung wird im Anschluss um semantische Aktionen ergänzt, deren Ziel der Aufbau des gewünschten TGraphen ist. Aus der erweiterten Syntaxbeschreibung wird schließlich ein Parser generiert, mit dessen Hilfe die Eingabe verarbeitet werden kann. Die Verwendung der gängigen Parsergeneratoren hat den Nachteil, dass die verwendeten Algorithmen nicht mächtig genug sind, um die Syntax aller Eingabedateien zu parsen. Ein weiterer Nachteil besteht darin, dass die gesamte um semantische Aktionen erweiterte Syntaxbeschreibung in einer einzelnen Datei definiert sein muss. Diese wird aufgrund ihrer Größe schnell so unübersichtlich, dass ihre Wartung schwierig wird. Sollten bei der Ausführung des generierten Parsers Fehler auftreten, so ist mit der von den Parsergeneratoren angebotenen Unterstützung nur schwer möglich, die verursachende semantische Aktion zu identifizieren.

Zielsetzung

Im Rahmen dieser Arbeit wurde eine „Extractor Description Language“ (EDL) entwickelt, mit deren Hilfe die Syntax der zu repräsentierenden Daten beschrieben wird. Diese kann um semantische Aktionen erweitert werden, die zum Aufbau des benötigten TGraphen führen. Um die Wartbarkeit zu erhöhen, ist die erstellte Syntaxbeschreibung modularisierbar. Dies hat den Vorteil, dass ein einzelnes Modul auf die Beschreibung eines einzelnen Aspekts der Syntax wie beispielsweise der Definition von Literalen beschränkt und damit übersichtlicher und verständlicher ist. Darüber hinaus kann ein einzelnes

Modul unter Umständen auch für andere Syntaxbeschreibungen wiederverwendet werden. Die zum Aufbau eines TGraphen häufig benötigten Anweisungen sind durch EDL-Befehle leicht zu definieren und zu verstehen. Da nicht immer die gesamten Daten repräsentiert werden kann, wurde eine Einschränkung auf die relevanten Bestandteile ermöglicht.

Aus der EDL-Beschreibung der Syntax soll ein Extraktor erzeugt werden, der Parsing-Algorithmen verwendet, mit deren Hilfe eine größere Menge von Sprachen erkannt werden kann als durch die klassischen Parsergeneratoren wie yacc oder ANTLR. Darüber hinaus soll das Debuggen unterstützt werden, indem im Falle eines Fehlers dem Nutzer mitgeteilt wird, welche semantische Aktion in der Syntaxbeschreibung diesen verursacht hat.

Gliederung der Arbeit

Um diese Ziele zu erreichen, werden im Rahmen dieser Arbeit zunächst die Grundlagen vermittelt. Diese bestehen zunächst aus der Definition von TGraphen (Kapitel 2). Im Anschluss wird im Kapitel 3 erläutert, wie die Syntax einer Eingabe beschrieben und die Eingabedaten erkannt werden können.

Nachdem die Grundlagen geklärt sind, wird untersucht, welche Anforderungen an die EDL bestehen. Sie werden im Kapitel 4 aufgelistet. Im Anschluss wird untersucht, welche Parsergeneratoren existieren und inwiefern sie für die gestellten Anforderungen ausreichend sind. Das Ergebnis dieser Untersuchung und die Auswahl des geeigneten Parsergenerators ist in Kapitel 5 dargestellt.

Im Folgenden wird die „Extractor Description Language“ definiert, mit deren Hilfe die Syntax der Eingabe beschrieben und die auszuführenden semantischen Aktionen festgelegt werden können (siehe Kapitel 6). Aus einer in EDL verfassten Syntaxbeschreibung muss ein lauffähiger Extraktor generiert werden. Um dies zu leisten, wird zunächst ein System entworfen (Kapitel 7) und im Anschluss implementiert (Kapitel 8 und 9).

Um die Nutzung von EDL zu veranschaulichen, wird im Kapitel 10 beschrieben, wie je ein exemplarischer Extraktor für XML- und .java-Dateien erzeugt werden kann. Zur Abschätzung ihres Aufwands wird die zur Extraktion von abstrakten Syntaxgraphen aus verschiedenen Eingabedateien benötigte Zeit gemessen und in Kapitel 11 dargestellt. Zum Abschluss der Arbeit wird das Ergebnis dieser Arbeit zusammengefasst und ein Ausblick gegeben, wie die EDL erweitert werden kann (Kapitel 12).

2 TGraphen

TGraphen wurden am Institut für Softwaretechnik der Universität Koblenz-Landau entwickelt. Bei ihnen handelt es sich um gerichtete, angeordnete, typisierte und attributierte Graphen. Die Bedeutung dieser Eigenschaften soll im Folgenden anhand eines Beispielgraphen verdeutlicht werden.

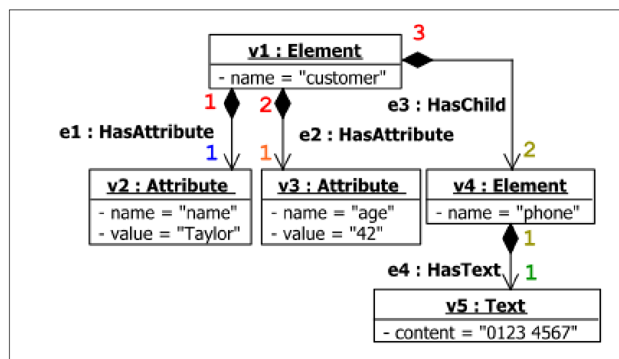


Abbildung 2.1: Eine TGraph-Instanz zur Repräsentation eines XML-Elements.

Abbildung 2.1 zeigt einen TGraphen, der das folgende XML-Element repräsentiert:

```
<customer name="Taylor" age="42">
   <phone>0123 4567</phone>
</customer>
```

Der dargestellte Graph besteht aus den Knoten `v1` bis `v5` sowie den Kanten `e1` bis `e4`.

Die in diesem Graphen vorkommenden Kanten sind *gerichtet*. Dies bedeutet, dass jede Kante einen Start- und einen Endknoten besitzt. So beginnt die Kante `e1` bei dem Knoten `v1` und endet bei `v2`. Die Richtung wird in dieser Abbildung durch die Pfeilspitze ausgedrückt.

Darüber hinaus sind TGraphen *angeordnet*. Zum einen gibt es eine Sequenz aller im Graphen vorkommenden Knoten sowie eine Sequenz aller Kanten. Diese Sequenz entspricht im vorliegenden Graphen der Zahl hinter `v` bzw. `e`. Zum anderen gibt es für jeden Knoten eine Reihenfolge aller seiner Inzidenzen. Diese sogenannte Inzidenzreihenfolge ist im dargestellten Graphen durch die farbigen Zahlen neben den Inzidenzen visualisiert.

TGraphen sind *typisiert*, da jeder Knoten und jede Kante sowie der Graph selbst einen Typ haben muss. Im angegebenen Beispielgraphen ist der Typ der Graphelemente hinter einem Doppelpunkt angegeben. So hat der Knoten `v1` beispielsweise den Typ `Element`.

Des Weiteren können Knoten, Kanten und auch Graphen Attribute besitzen, wodurch TGraphen *attributiert* sind. Ein Attribut ist ein Paar aus einem Bezeichner und einem Wert. So hat der Knoten `v1` ein Attribut mit Namen `name` dessen Wert `"customer"` ist.

Die Typisierung und Attributierung der TGraphen wird durch ein Schema definiert. In diesem Schema wird unter anderem festgelegt, welche Typen die Knoten, die Kanten sowie der Graph besitzen und welche Beziehungen zwischen den einzelnen Typen existieren. Zur Notation des Schemas wird grUML genutzt, welches eine echte Teilmenge der Klassendiagramme aus UML ist.

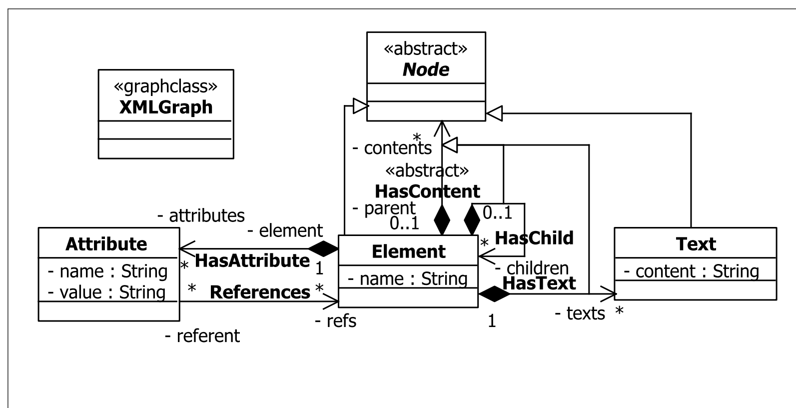


Abbildung 2.2: Das Schema des Graphen aus Abbildung 2.1.

Das für den Beispielgraphen definierte XML-Schema ist in Abbildung 2.2 dargestellt[1]. Der Typ des Graphen wird durch die Klasse `XMLGraph` ausgedrückt. Die einzelnen XML-Elemente werden durch Knoten vom Typ `Element` repräsentiert. Der jeweilige Name wird im Attribut `name` persistiert, dessen Werte aus der String-Domäne stammen. Die in einem Element geschachtelten XML-Bestandteile werden durch Instanzen des abstrakten Knotentyps `Node` dargestellt und über Kanten vom abstrakten Typ `HasContent` verbunden. Im Falle von XML-Elementen bedeutet dies, dass `Element`-Instanzen über `HasChild`-Kanten angebunden werden. Alle anderen geschachtelten Bestandteile von XML werden durch `Text`-Knoten repräsentiert und über `HasText`-Kanten mit dem Elternelement verbunden. Die beiden Kantenklassen `HasChild` und `HasText` sind dabei von `HasContent` abgeleitet.

[1] Das XML-Schema ist im „Java Graph Laboratory" (JGraLab) enthalten, welches eine Java-Implementation der TGraphen und unter `https://github.com/jgralab/jgralab` zu finden ist.

Die Attribute eines XML-Elements werden durch Instanzen der Knotenklasse `Attribute` dargestellt und über Kanten vom Typ `HasAttribute` angebunden. Der Name wird im Attribut `name` und der Wert im Attribut `value` persistiert. Sollte der Name `idref` oder `idrefs` lauten, so enthält dieses Attribut die Ids der referenzierten XML-Elemente als Wert. Sie werden über `References`-Kanten mit dem referenzierenden `Attribute`-Knoten verbunden.

Definition von TGraphen

TGraphen besitzen eine formale Definition, wie von Steffen Kahle auf Seite 11 seiner Diplomarbeit [Kah06] angegeben:

Definition (TGraph)
Seien
typeID eine endliche Menge von **Typbezeichnern**,
attrID eine endliche Menge von **Attributbezeichnern**,
Value eine endliche Menge von **Attributwerten**,
V eine endliche Menge von **Knoten** und
E eine endliche Menge von **Kanten**.
Dann ist
$G = (Vseq, Eseq, \Lambda seq, type, value)$
ein **TGraph**, falls gilt:

- $Vseq \in \text{seq } V$ ist eine Anordnung von V,
- $Eseq \in \text{seq } E$ ist eine Anordnung von E,
- $\Lambda seq : V \rightarrow \text{seq}(E \times \{in,out\})$ ist eine **Inzidenzabbildung** für die gilt:
  $\forall\, e \in E\ \exists!\, v, w \in V :$
  $(e,out) \in \text{ran } Iseq(v) \wedge (e,in) \in \text{ran } \Lambda seq(w)$,
- $type : V \cup E \rightarrow typeID$ ist eine **Typisierung** und
- $value : V \cup E \rightarrow (attrID \twoheadrightarrow\!\!\!\!\!\!\!\!\;\;\; Value)$ ist eine **Attributierung**.

Da die Attribute einer Kante oder eines Knotens vom jeweiligen Schematyp abhängen, gilt weiterhin:

- $attributes : typeID \rightarrow (attrID \twoheadrightarrow \mathbb{P}(Value))$

Außerdem müssen folgende Bedingungen erfüllt sein:

- $\forall\, e \in V \cup E$, mit $(\kappa,X) = attributes(type(e)) : \exists(\kappa,\lambda) \in value(e),\ \lambda \in X$
  und
- $\forall\, e \in V \cup E$, mit $(\kappa,\lambda) = value(e) : \exists(\kappa,X) \in attributes(type(e)),\ \lambda \in X$
  [Mar06]

Darüber hinaus wird in dieser Arbeit der Begriff des Pfads benötigt. In [Ebe11] ist er wie folgt definiert:

Definition (Pfad)
Eine alternierende Folge
$C =< v_0, e_1, v_1, \ldots, e_k, v_k >, k \geq 0$
heißt (gerichteter) Pfad von v_0 nach v_k falls gilt:
$\forall\, i : N \mid 1 \leq i \leq k : \alpha(e_i) = v_{i-1} \wedge \omega(e_i) = v_i$
wobei $\alpha(e)$ den Startknoten und $\omega(e)$ den Endknoten von e angibt.

3 Syntaktische Analyse

Dieses Kapitel befasst sich mit der syntaktischen Analyse, bei der überprüft wird, ob eine Folge von Zeichen der Syntax einer Sprache wie beispielsweise Java entspricht. Daher wird in Abschnitt 3.1 zunächst formal beschrieben, was eine Sprache ist und welche unterschiedlichen Sprachklassen es gibt. Im Anschluss werden im Abschnitt 3.2 verschiedene Metasprachen vorgestellt, mit deren Hilfe die Syntax einer Sprache definiert werden kann. Um mit dieser Definition die Zeichenfolge der Eingabe überprüfen zu können, gibt es zwei verschiedene grundsätzliche Arbeitsweisen von Algorithmen. Die einen arbeiten nach dem Top-Down-Prinzip und versuchen ausgehend von einem Startsymbol die Eingabe zu erkennen (Abschnitt 3.3). Die anderen arbeiten nach dem Bottom-Up-Prinzip und versuchen anhand der Eingabezeichen die anzuwendenden Regeln zu identifizieren (Abschnitt 3.4).

3.1 Formale Definition von Sprachen

Bevor formal definiert werden kann, was eine Sprache ist, werden zunächst ihre Bestandteile beschrieben. Die angegebenen Definitionen entstammen [EP08].

Definition (Alphabet, Wort)
Ein **Alphabet** Σ ist eine endliche nicht-leere Menge. Ihre Elemente heißen **Buchstaben**. Eine endliche möglicherweise leere Folge von Buchstaben wird als **Wort** w bezeichnet. Die **Länge eines Worts** $|w|$ gibt die Länge der Buchstabenfolge wieder, d.h. die Anzahl der im Wort enthaltenen Buchstaben.

Um die für einen Buchstaben stehenden Variablen in dieser Arbeit leicht identifizieren zu können, werden sie mit $a, b, c, \ldots$ bezeichnet. Ein Wort wird als eine Folge von Buchstaben der Form $a_0 a_1 \ldots a_n$ notiert. Das leere Wort, das aus keinen Buchstaben besteht, wird durch ε angegeben. Um die für ein Wort stehenden Variablen von denen unterscheiden zu können, die für einen Buchstaben stehen, erhalten sie die Bezeichner $u, v, w, \ldots$.

Ein Beispiel für ein Alphabet Σ_{bin} wäre beispielsweise $\{0, 1, +\}$. Daraus ließen sich die folgenden Wörter bilden: 1001, 101+1, 1+1+0

Definition (Konkatenation, Σ^*, Sprache)
Seien $v = a_0 \dots a_n$ und $w = b_0 \dots b_m$ Wörter. Dann ist $v \circ w = vw = a_0 \dots a_n b_0 \dots b_m$ die **Konkatenation der Wörter** v und w.
Σ^* ist die kleinste Menge an Wörtern, die Σ sowie ε enthält und gegen die Konkatenation von Wörtern abgeschlossen ist.
Eine **Sprache** L ist eine Teilmenge von Σ^*, d.h. $L \subseteq \Sigma^*$. $\varnothing$ ist die leere Sprache.
Seien L_1 und L_2 Sprachen, dann ist $L_1 \circ L_2 = L_1 L_2 = \{uv \mid u \in L_1, v \in L_2\}$ die **Konkatenation der Sprachen** L_1 und L_2.

Nach dieser Definition ist eine Sprache eine endliche Menge von Wörtern. Manche dieser Sprachen lassen sich formal durch eine Grammatik definieren.

Definition (Grammatik)
Eine **Grammatik** G ist ein Tupel $G = (V, T, R, S)$. Dabei ist

- V eine endliche Menge von syntaktischen Variablen.
- T eine endliche Menge von Terminalen, die von den Variablen verschieden sein müssen.
- R eine endliche Menge von Regeln. Eine Regel ist ein Element (P, Q) aus $((V \cup T)^* \, V \, (V \cup T)^*) \times (V \cup T)^*$. P heißt Regelrumpf und Q Regelkopf. Eine Regel lässt sich als $P \to Q$ schreiben.
- $S \in V$ das Startsymbol.

Um syntaktische Variablen von Terminalzeichen unterscheiden zu können, beginnen erstere mit einem Großbuchstaben. Variablen, die für Wörter über dem Alphabet $T \cup V$ definiert sind, werden mit $\alpha, \beta, \gamma, \dots$ bezeichnet. Um diese formale Definition besser verstehen zu können, wird mit der folgenden Grammatik G_{bin} die Addition von Binärzahlen definiert:

$G_{bin} = \left(\{N, N_{suff}, E\}, \Sigma_{bin}, R, E\right)$ mit
$R =$

$$
\begin{aligned}
\{E + E &\to E \\
N &\to E \\
0 &\to N \\
1 &\to N \\
1\ N_{suff} &\to N \\
\varepsilon &\to N_{suff} \\
0\ N_{suff} &\to N_{suff} \\
1\ N_{suff} &\to N_{suff}\}
\end{aligned}
$$

Um mithilfe einer Grammatik ein Wort erzeugen zu können, muss zunächst der Begriff der Ableitung definiert werden.

Definition (Ableitung)
Sei $G = (V, T, R, S)$ eine Grammatik und seien α, α' Wörter aus $(V \cup T)^*$, so gilt $\alpha \Longrightarrow \alpha'$, falls gilt:

$$\exists \beta, \gamma \in (V \cup T)^* \, \exists P \to Q \in R \; (\alpha = \beta Q \gamma \wedge \alpha' = \beta P \gamma)$$

Falls es Wörter $\alpha_0, \ldots, \alpha_n \in (V \cup T)^*$ gibt mit $\alpha = \alpha_0$, $\alpha' = \alpha_n$ und $\alpha_i \Longrightarrow \alpha_{i+1}$ für $0 \leq i < n$, so kann dies abkürzend durch $\alpha \Longrightarrow^* \alpha'$ notiert werden. Die Folge $\alpha_0, \ldots, \alpha_n$ heißt **Ableitung** der Länge n.

Mithilfe der Beispielgrammatik G_{bin} ließe sich das Wort 1+1+0 aus dem Startsymbol E wie folgt ableiten:

$$E \Longrightarrow E\text{+}E \Longrightarrow N\text{+}E \Longrightarrow 1\text{+}E \Longrightarrow 1\text{+}E\text{+}E \Longrightarrow 1\text{+}N\text{+}E \Longrightarrow 1\text{+}1\text{+}E \Longrightarrow 1\text{+}1\text{+}N \Longrightarrow 1\text{+}1\text{+}0$$

Definition (von einer Grammatik erzeugte Sprache, Linksableitung, Mehrdeutigkeit)
Die von einer Grammatik erzeugte Sprache $L(G)$ ist definiert als

$$L(G) := \{w \in T^* \mid S \Longrightarrow^* w\}$$

wobei zur Ableitung von w nur Regeln der Grammatik G genutzt werden dürfen.
Eine Ableitung $\alpha_0 \Leftrightarrow \alpha_1 \Longrightarrow \ldots \Longrightarrow \alpha_n$ heißt **Linksableitung**, falls für alle $i < n$ α_{i+1} durch Ersetzen der linkesten syntaktischen Variable in α_i entsteht.
Eine Grammatik G heißt **mehrdeutig** gdw. es ein Wort $w \in L(G)$ gibt, so dass G zwei verschiedene Linksableitungen zu w besitzt.

Wie zu erkennen ist, wurde in der weiter oben aufgelisteten Ableitung des Worts 1+1+0 mithilfe der Grammatik G_{bin} immer die linkeste syntaktische Variable ersetzt. Daher ist dies eine Linksableitung. Eine alternative Linksableitung wäre:

$$E \Longrightarrow E\text{+}E \Longrightarrow E\text{+}E\text{+}E \Longrightarrow N\text{+}E\text{+}E \Longrightarrow 1\text{+}E\text{+}E \Longrightarrow 1\text{+}N\text{+}E \Longrightarrow 1\text{+}1\text{+}E \Longrightarrow 1\text{+}1\text{+}N \Longrightarrow 1\text{+}1\text{+}0$$

Da es nun zwei verschiedene Linksableitungen für das Wort 1+1+0 gibt, ist G_{bin} mehrdeutig.

Chomsky-Hierarchie

Wird die Form der für eine Grammatik zulässigen Regeln beschränkt, so lassen sich die diversen Grammatiken in Typen von verschiedenen Schwierigkeitsgraden einordnen. Da jede Grammatik eine Sprache erzeugt, werden sie Sprachklassen genannt. Nach [EP08]

gehört die von einer Grammatik $G = (V, T, R, S)$ erzeugte Sprache zu

$\mathcal{L}_3$, falls G rechtslinear ist, d.h. $\forall P \to Q \in R : P \in T^* \cup T^+ V \wedge Q \in V$.

$\mathcal{L}_2$, falls G kontextfrei (cf) ist, d.h $\forall P \to Q \in R : P \in (T \cup V)^* \wedge Q \in V$.

$\mathcal{L}_1$, falls G kontextsensitiv (cs) ist, d.h $\forall P \to Q \in R : \exists u, v, \alpha \in (T \cup V)^* : \exists A \in V : Q = uAv \wedge P = u\alpha v$ mit $|\alpha| \geq 1$ oder die Regel hat die Form $\varepsilon \to S$, wobei S in keinem Regelrumpf vorkommen darf.

$\mathcal{L}_0$, falls G beschränkt ist, d.h $\forall P \to Q \in R : |P| \geq |Q|$ oder die Regel hat die Form $\varepsilon \to S$, wobei S in keinem Regelrumpf vorkommen darf.

Die Sprachklasse aller Sprachen ist $\mathcal{L}$.

3.2 Metasprachen

Die bekanntesten *Metasprachen für kontextfreie Sprachen* sind die Backus-Naur-Form (Abschnitt 3.2.1), die erweiterte Backus-Naur-Form (Abschnitt 3.2.2) und der Syntax Definition Formalism (Abschnitt 3.2.3).

Um die Verständlichkeit und Vergleichbarkeit zu erhöhen, werden sie anhand einer *Beispielsprache* beschrieben. Ein Wort dieser Sprache ist in Listing 3.1 zu sehen.

```
1  a = -5;
2  b = (a + 2) * 3 - 1;
```

Listing 3.1: Ein Wort einer einfache Beispielsprache

Diese einfache Beispielsprache besitzt nur einen ganzzahligen Datentyp und nur ein einziges zulässiges *Statement*: die Zuweisung. Wie in Zeile 1 zu sehen ist, besteht die linke Seite aus einer Variablen und die rechte Seite aus einem mathematischen Ausdruck. Abgeschlossen wird jedes Statement durch ein Semikolon. Durch die erste Zuweisung eines Werts an eine Variable wird sie deklariert. Sie bleibt für alle folgenden Statements gültig. Mit kontextfreien Sprachen ist es jedoch nicht möglich, zu erkennen, dass es sich bei der Verwendung der Variable a in Zeile 2 um dieselbe Variable handelt, die in Zeile 1 deklariert wurde. Daher wird mit einer Metasprache dieses Abschnitts nur beschrieben, dass eine Variable verwendet wird, ohne einen Bezug zu ihren weiteren Verwendungen oder ihrer Deklaration auszudrücken.

Die Ausdrücke auf der rechten Seite einer Zuweisung bestehen aus einem *mathematischen Ausdruck*. Im Falle dieser Beispielsprache ist ein Variablenzugriff oder eine ganze Zahl auch schon ein mathematischer Ausdruck. Er kann geklammert oder mit den Vorzeichen + oder - versehen werden. Verschiedene mathematische Ausdrücke können mittels der

binären Operatoren +, -, * und / zu einem neuen mathematischen Ausdruck kombiniert werden, wie in Zeile 2 zu sehen ist. Dabei haben * und / die höheren Prioritäten.

3.2.1 Backus-Naur-Form (BNF)

Die erste Vorgestellte Metasprache heißt *Backus-Naur-Form (BNF)* und ist nach ihren Urhebern John Backus und Peter Naur benannt worden. Beide nutzten diese Metasprache, um die Sprache ALGOL 60 zu definieren [BNF11], welche im Paper [BBJ+63] veröffentlicht wurde. Neben der Syntaxbeschreibung von ALGOL 60 ist in diesem Paper auch die BNF definiert. Bevor die Syntax der BNF vorgestellt wird, folgt zunächst ein Beispiel, in dem die am Anfang dieses Abschnitts eingeführte Beispielsprache definiert wird.

Beispiel

Um die Beschreibung der BNF-Syntax zu veranschaulichen, wird zunächst ein Beispiel angegeben. Hierzu ist in Listing 3.2 die *BNF-Definition der* zu Beginn dieses Abschnitts eingeführten einfachen *Beispielsprache* angegeben.

In der ersten Zeile wird ein Programm als eine *Liste von Statements* definiert. Die Statementliste besteht entweder aus einer einzigen Zuweisung oder einer Liste von Zuweisungen (Zeile 2). Jede von ihnen besteht aus einem Identifier gefolgt von einem Gleichheitszeichen, einer Expression und einem Semikolon (Zeile 3).

Um die unterschiedlichen Prioritäten der Operationen einer *Expression* auszudrücken, wird die Definition der Expression auf vier BNF-Regeln verteilt. Die erste Regel „expression“ repräsentiert die binären Operatoren + und -, die die geringste Priorität haben (Zeile 5-7). Gefolgt wird sie von „multExp“, das die Multiplikation * und die Ganzzahldivision / definiert (Zeile 8-10). Durch die Linksrekursion in den beiden Regeln werden die binären Operatoren als linksassoziativ definiert. Im Anschluss werden die Vorzeichen mithilfe der Regel „unaryOpExp“ festgelegt (Zeile 11). Zum Schluss folgen die Definition der Variablenidentifier, der ganzen Zahlen und der geklammerten Expressions als „element“ (Zeile 12).

In den Zeilen 14 bis 17 werden die *Variablenbezeichner* näher definiert. Dabei muss das erste Zeichen ein Buchstabe sein (Zeile 14), gefolgt von einer beliebigen Folge von Ziffern oder Buchstaben (Zeile 15 und 16). In den letzten vier Zeilen des Listings werden die *Zahlen* so definiert, dass sie keine führenden Nullen haben dürfen (Zeile 19). Die drei Punkte in Zeile 17 und 22 sind kein Bestandteil der BNF sondern dienen lediglich der Schreibabkürzung.

```
1  <program> ::= <statementList>
2  <statementList> ::= <assignment> | <assignment> <statementList>
3  <assignment> ::= <identifier> = <expression> ;
4
5  <expression> ::= <expression> + <multExp>
6                 | <expression> - <multExp>
7                 | <multExp>
8  <multExp> ::= <multExp> * <unaryOpExp>
9              | <multExp> / <unaryOpExp>
10             | <unaryOpExp>
11 <unaryOpExp> ::= + <element> | - <element> | <element>
12 <element> ::= <identifier> | <number> | ( <expression> )
13
14 <identifier> ::= <letter> | <letter> <charsequence>
15 <charsequence> ::= <char> | <char> <charsequence>
16 <char> ::= <letter> | <digit>
17 <letter> ::= a | ... | z | A | ... | Z
18
19 <number> ::= 0 | <one2nine> | <one2nine> <digits>
20 <digits> ::= <digit> | <digit> <digits>
21 <digit> ::= 0 | <one2nine>
22 <one2nine> ::= 1 | ... | 9
```

Listing 3.2: BNF-Definition der einfachen Beispielsprache

Definition der Metasprache

Die Syntax der Backus-Naur-Form, deren Anwendung im vorangegangenen Abschnitt anhand eines Beispiels verdeutlicht wurde, wurde in [BBJ+63] definiert. Eine kontextfreie Sprache wird durch eine *Menge von BNF-Regeln* beschrieben. Der Aufbau einer BNF-Regel wird durch Formel (3.1) verdeutlicht.

$$\langle \mathit{Head} \rangle ::= \langle \mathit{Body} \rangle \tag{3.1}$$

Der *Head* einer BNF-Regel besteht aus einer syntaktischen Variablen. Sie setzt sich aus einer in $\langle\rangle$ eingefassten Zeichen-Folge zusammen. So besteht beispielsweise der Head der BNF-Regel in Zeile 3 des Listings 3.2 aus der syntaktischen Variable $\langle \mathit{assignment} \rangle$.

Der *Body* einer Regel, wie sie in Formel (3.2) definiert ist, besteht aus einer *Sequence*. Existieren mehrere Regeln mit gleichem *Head* so können sie zu einer einzigen Regel zusammengefasst werden, indem die *Body*s mit | verodert werden, wie in Formel (3.3) zu sehen. Dabei steht $\langle \mathit{Or} \rangle$ für das |-Zeichen. Die einzelnen alternativen *SubSequences* kön-

nen aus einer beliebig langen Folge von *TerminalSymbol*s und Variables bestehen, wie sie Formel (3.4) zeigt. Der *Body* der „assignment"-Regel $\langle identifier \rangle$ = $\langle expression \rangle$; besteht beispielsweise aus einer einzigen *SubSequence*. Sie enthält die beiden *TerminalSymols* = und ; sowie die beiden Variables $\langle identifier \rangle$ und $\langle expression \rangle$.

$$\langle Body \rangle ::= \langle Sequence \rangle \tag{3.2}$$

$$\langle Sequence \rangle ::= \langle SubSequence \rangle \mid \langle Sequence \rangle \langle Or \rangle \langle SubSequence \rangle \tag{3.3}$$

$$\langle SubSequence \rangle ::= \langle TerminalSymbol \rangle \mid \langle TerminalSymbol \rangle \langle SubSequence \rangle \mid \langle Variable \rangle \mid \langle Variable \rangle \langle SubSequence \rangle \mid \varepsilon \tag{3.4}$$

3.2.2 Erweiterte Backus-Naur-Form (EBNF)

Die im vorangegangenen Abschnitt beschriebene Backus-Naur-Form hat zum einen den Nachteil, dass man für optionale oder sich beliebig häufig wiederholende Elemente mehrere Regeln braucht. Zum anderen verwendet die BNF keine gesonderte Kennzeichnung von Terminalzeichen der beschriebenen kontextfreien Sprache. Dadurch können die von der BNF-Syntax verwendeten Zeichen wie beispielsweise | durch eine BNF-Regel nicht beschrieben werden. Daher wurde 1996 von der International Organisation for Standardisation (ISO) die *erweiterte Backus-Naur-Form (EBNF)* als ISO/IEC 14977:1996(E) *standardisiert* [Sta96]. Bevor die Metasprache definiert wird, folgt zur Veranschaulichung zunächst ein Beispiel einer EBNF-Beschreibung.

Beispiel

```
1  program = assignment {assignment};
2  assignment = identifier "=" expression ";";
3
4  expression = [expression ("+" | "-")] multExp;
5  multExp = [multExp ("*" | "/")] unaryOpExp;
6  unaryOpExp = [("+" | "-")] element;
7  element = identifier | number | "(" expression ")";
8
9  identifier = letter {letter | digit};
10 letter = "a" | ... | "z" | "A" | ... | "Z";
11
12 number = "0" | digit - "0", {digit};
13 digit = "0" | "1" | ... | "9";
```

Listing 3.3: EBNF-Definition der einfachen Beispielsprache

Um die EBNF-Definition zu verdeutlichen, wurde die zu Beginn dieses Abschnitts vorgestellte Beispielsprache in Listing 3.3 definiert.

Definition der Metasprache

Wie bei der BNF besteht eine *EBNF-Beschreibung aus einer Menge von Regeln*. Den Aufbau einer Regel beschreibt die Formel (3.5).

$$Head = Body \text{ ";" } ; \tag{3.5}$$

Anders als bei der BNF ist das Trennzeichen zwischen dem *Head* und dem *Body* nur ein Gleichheitszeichen. Des Weiteren muss jede Regel mit einem Semikolon beendet werden. Ein Beispiel für eine EBNF-Regel ist die Zeile 2 in Listing 3.3. Anhand dieses Beispiels lassen sich zwei weitere *Unterschiede zur BNF feststellen*. Zum einen werden Terminale von zwei " oder von zwei ' umschlossen. Zum anderen werden syntaktische Variablen ohne spitze Klammern geschrieben, da in der EBNF die Terminale in Anführungszeichen stehen und somit eine Verwechslung ausgeschlossen ist.

Die *Alternativ-Schreibweise* mit | für verschiedene Sequenzen von syntaktischen Variablen und Terminalen im Body einer Regel wurde beibehalten, wie in Zeile 10 des Listings 3.3 zu sehen ist. Um die Beschreibung von kontextfreien Sprachen mittels EBNF zu vereinfachen, wurden auch *neue Sprachkonstrukte* eingeführt:

Gruppierung. Bei der Gruppierung wird eine Sequenz von syntaktischen Variablen und Terminalen mit () umschlossen, wie in der Formel (3.6) zu sehen ist.

$$V = \alpha\,(\beta)\,\gamma; \tag{3.6}$$

Semantisch ist eine Gruppierung äquivalent mit den beiden BNF-Regeln in den Formeln (3.7) und (3.8). Dabei ist W eine neue Variable.

$$V \quad ::= \quad \alpha W \gamma \tag{3.7}$$

$$W \quad ::= \quad \beta \tag{3.8}$$

Ein Beispiel einer Gruppierung ist in Zeile 4 des Listings 3.3 zu finden, in der die Alternative zwischen "+"und "-"gruppiert wurde.

Option. Eine Option ist eine von [ ] umschlossene Sequenz, wie in Formel (3.9) gezeigt.

$$V = \alpha\,[\beta]\,\gamma; \tag{3.9}$$

Semantisch bedeutet dies, dass β höchstens einmal vorkommt. Die äquivalente BNF-Regel zeigt Formel (3.10).

$$V ::= \alpha\gamma \mid \alpha\beta\gamma \tag{3.10}$$

Zeile 6 des Listings 3.3 zeigt eine mögliche Anwendung für die Option. In der Regel wurden die Vorzeichen "+"und "-"als optional definiert, wodurch ein „element" auch ohne Vorzeichen vorkommen kann.

Wiederholung. Die Wiederholung beschreibt das beliebig häufige Vorkommen der mit {} umschlossenen Sequenz, wie in Formel (3.11) zu sehen.

$$V = \alpha\,\{\beta\}\,\gamma; \tag{3.11}$$

Die äquivalenten BNF-Regeln sind in den Formeln (3.12) und (3.13) zu sehen. Dabei ist W eine neue Variable.

$$V \quad ::= \quad \alpha W\gamma \tag{3.12}$$
$$W \quad ::= \quad \varepsilon \mid \beta W \tag{3.13}$$

Ein Beispiel für die Wiederholung ist in Zeile 1 des Listings 3.3 zu sehen, in der ein Programm aus einer Zuweisung gefolgt von beliebig vielen weiteren Zuweisungen besteht.

Ausnahme. Formel (3.14) zeigt, wie Ausnahmen in der EBNF definiert werden.

$$V = \alpha\ \beta\text{-}\gamma,\ \delta; \tag{3.14}$$

Dabei repräsentieren β-γ Worte einer kontextfreien Sprache, die durch β beschrieben sind, aber nicht durch γ. Das Komma markiert das Ende der Sequenz γ. Kommt hinter γ eine schließende Klammer oder ein Semikolon, so kann das Komma auch wegfallen.

Mit Hilfe der Ausnahme lassen sich mehr Sprachen als nur kontextfreie Sprachen beschreiben. So wäre es beispielsweise möglich ein Paradoxon ähnlich dem von Russel[1] zu erzeugen: V = "A"-V; . Bei einer Sprache, die durch diese Regel be-

[1] Russels Paradoxon beschreibt eine Menge R, die jede Menge enthält die nicht in R enthalten ist: $R = \{M \mid M \notin R\}$. Es ist unter [Rus09] genauer erläutert.

schrieben wird, ist es nicht möglich zu entscheiden, ob "A"ein Wort dieser Sprache ist oder nicht. Aus diesem Grund schränkt der ISO-Standard die Verwendung der Ausnahme so ein, dass die durch β-γ beschriebene Sprache kontextfrei sein muss. Daher gibt es auch eine äquivalente EBNF-Beschreibung ohne die Verwendung der Ausnahme.

Die Anwendung einer Ausnahme ist in Zeile 12 des Listings 3.3 zu sehen. In dieser Zeile wird definiert, dass eine "0"eine „number“ ist. Da „digit“ für die Terminale "0"bis "9"steht, repräsentiert $digit-"0"$ die Ziffern eins bis neun. Daher ist die zweite definierte Alternative einer „number“ eine beliebig lange Folge von Ziffern, die nicht mit einer "0"beginnen darf.

Faktor. Wie ein Faktor in der EBNF definiert wird, zeigt Formel (3.15).

$$V = \alpha \ \ n * \beta, \gamma; \tag{3.15}$$

n steht dabei für eine natürliche Zahl[2]. Das Komma stellt die Endbegrenzung der Sequenz β dar. Die semantische BNF-Äquivalenz ist in Formel (3.16) zu sehen.

$$V ::= \alpha\beta_0 \dots \beta_{n-1}\gamma \tag{3.16}$$

Kommentar. Kommentare werden, wie in Formel (3.17) gezeigt, angegeben.

$$(* \text{ Ein Kommentar } *) \tag{3.17}$$

Kommentare haben keine semantische Bedeutung.

Spezialsequenz. Formel (3.18) zeigt die Anwendung einer Spezialsequenz.

$$V = \alpha \ \ ?\text{Spezialsequenz}? \ \ \gamma; \tag{3.18}$$

Die Semantik und der innere syntaktische Aufbau einer Spezialsequenz sind in [Sta96] nicht näher definiert. Sie stellt eine Möglichkeit dar, die EBNF nutzerspezifisch zu erweitern. Eine mögliche Erweiterung wäre die Nutzung der Spezialsequenz, um semantische Aktionen zu definieren.

Der *Body* jeder EBNF-Regel stellt einen regulären Ausdruck dar, der um Rekursion ergänzt wurde. Die *Backus-Naur-Form ist äquivalent zur erweiterten Backus-Naur-Form*. Das heißt, dass jede Sprache, die durch BNF-Regeln beschrieben werden kann, auch durch EBNF-Regeln definiert werden kann und umgekehrt.

[2] Die 0 wird auch als natürliche Zahl angesehen.

3.2.3 Syntax Definition Formalism (SDF)

Der erste Entwurf des *Syntax Definition Formalism (SDF)* wurde 1986 von Jan Heering und Paul Klint vorgestellt ([HK86]). Die Metasprache wurde entwickelt, um insbesondere komplexe kontextfreie Sprachen zu beschreiben. Dies wird vor allem durch das Erlauben von Mehrdeutigkeiten und von Modularität, die 1997 von Eelco Visser eingeführt wurde ([Vis97]), erreicht. Bevor auf die in [BKV07] beschriebene Syntax von SDF näher eingegangen wird, folgt zunächst die Definition der zu Beginn dieses Abschnitts eingeführten Beispielsprache.

Beispiel

Anders als bei den zuvor beschriebenen Metasprachen ist die Beschreibung der Beispielsprache aus Gründen des „Separation of Concerns“-Prinzips in fünf Module aufgeteilt. Das erste Modul in Listing 3.4 definiert, was die zu ignorierenden Zeichen sind. Danach zeigt Listing 3.5 ein Modul, das Identifier beschreibt. Mit dem Aufbau der Numbers befasst sich das in Listing 3.6 gezeigte Modul. Das vorletzte Modul in Listing 3.7 definiert Expressions und schließlich zeigt Listing 3.8 das Modul, das den Aufbau eines Programms beschreibt.

```
1 module Whitespace
2 exports
3    lexical syntax
4       [\ \n\r\t]+ -> LAYOUT
5    context-free restrictions
6       LAYOUT -/- [\ \n\r\t]
```

Listing 3.4: SDF-Definition von Whitespace

Listing 3.4 zeigt das *Modul Whitespace* (Zeile 1). Das Schlüsselwort „exports“ in Zeile 2 bedeutet, dass die in diesem Block angegebenen Definitionen (Zeile 3-6) auch in Modulen sichtbar sind, die dieses Modul importieren. Zeile 4 definiert eine nicht-leere Sequenz von Leerzeichen, Zeilenumbrüchen, Wagenrücklaufzeichen oder Tabulatorzeichen als LAYOUT. LAYOUT ist eine spezielle vordefinierte syntaktische Variable, deren Bedeutung später erklärt wird. Um die „longest match“-Strategie zu realisieren, wird in Zeile 6 definiert, dass das erste Zeichen hinter einem LAYOUT-Wort kein weiteres LAYOUT-Zeichen sein darf. Die Bedeutung der in den Zeilen 4 und 6 verwendeten in [ ] eingeschlossenen CharacterClasses wird bei der Beschreibung der Syntax näher erläutert.

```
1 module Identifier
2 exports
3    sorts Identifier
4    lexical syntax
5       [a-zA-Z] [a-zA-Z0-9]* -> Identifier
6    lexical restrictions
7       Identifier -/- [a-zA-Z0-9]
```

Listing 3.5: SDF-Definition von Identifier

Das zweite Modul, das sich mit der Definition von *Identifiern* beschäftigt, ist in Listing 3.5 zu sehen. In Zeile 4 werden alle in diesem Modul definierten syntaktischen Variablen aufgelistet. Zeile 6 beschreibt einen Identifier als einen kleinen oder großen Buchstaben gefolgt von einer Sequenz von Buchstaben oder Ziffern. Damit der Identifier *ab* nicht als die beiden Identifier *a* und *b* erkannt wird, definiert Zeile 8, dass das erste Zeichen hinter einem Identifier kein Buchstabe und keine Ziffer sein darf.

```
1 module Number
2 exports
3    sorts Number UnsignedNumber
4    lexical syntax
5       [0] | ([1-9][0-9]*) -> UnsignedNumber
6       [\+\-]? UnsignedNumber -> Number
7    lexical restrictions
8       Number -/- [0-9]
```

Listing 3.6: SDF-Definition von Number

Im nächsten Modul (Listing 3.6) wird definiert, was eine *Number* ist. Hierzu wird zunächst in Zeile 5 festgelegt, dass eine 0 oder eine Ziffernfolge, die nicht mit einer 0 beginnt, eine UnsignedNumber ist. Die Klammern um die zweite Alternative sind notwendig, da die Priorität von `|` ansonsten $([0] \mid [1-9])\,[0-9]^*$ implizieren würde. Eine UnsignedNumber mit einem optionalen vorangestellten Vorzeichen wird in Zeile 6 als Number definiert. Die Strategie des „longest match“ wird für die Erkennung von Number in Zeile 8 festgelegt.

Listing 3.7 zeigt das Modul, welches den Aufbau von *Expression*s beschreibt. Zunächst werden in den Zeilen 2 bis 4 die Definitionen der Module Whitespace, Identifier und Number importiert. In Zeile 8 werden Identifier und in Zeile 9 Number als Expression definiert. Die Kombination von zwei Expressions mittels der binären Infix-Operatoren "+", "-", "*" und "/" wird in den Zeilen 10 und 11 beschrieben. Die Angabe von {*left*}

hinter beiden Regeln weist sie als links-assoziativ aus. In Zeile 12 werden schließlich geklammerte Expressions definiert. Auf die Bedeutung von {*bracket*} wird bei der Beschreibung der Syntax näher eingegangen. In den Zeilen 14 bis 16 wird festgelegt, dass die Operatoren "*" und "/" eine höhere Priorität haben als "+" und "-".

```
1  module Expression
2  imports Whitespace
3  imports Identifier
4  imports Number
5  exports
6     sorts Expression
7     context-free syntax
8        Identifier -> Expression
9        Number -> Expression
10       Expression ("+"|"-") Expression  -> Expression {left}
11       Expression ("*"|"/") Expression  -> Expression {left}
12       "(" Expression ")" -> Expression {bracket}
13    context-free priorities
14       Expression ("*"|"/") Expression  -> Expression {left}
15       >
16       Expression ("+"|"-") Expression  -> Expression {left}
```

Listing 3.7: SDF-Definition von Expression

Im Gegensatz zu den zuvor beschriebenen Modulen werden die Regeln in einem Block mit dem Namen *„context-free syntax"* definiert. Dies hat zur Konsequenz, dass zwischen den syntaktischen Variablen und Konstanten auf der linken Seite jeder Regel das in Listing 3.4 definierte *LAYOUT* implizit ergänzt wird. Zeile 12 sähe demnach beispielsweise so aus:

"(" *LAYOUT*? *Expression LAYOUT*? ")" -> *Expression* {*bracket*}.

Das Fragezeichen weist die ergänzten *LAYOUT* als optional aus. Durch die Ergänzung werden die definierten Zeichen wie das Leerzeichen zwischen den auf der linken Seite einer Regel definierten Elementen ignoriert. Das Ergänzen von *LAYOUT*? vor und hinter der linken Seite einer Regel ist nicht notwendig, da eine Regel nur verwendet werden kann, wenn sie von einer anderen Regel verwendet wird oder das Startsymbol definiert. Im ersten Fall werden die umschließenden *LAYOUT*? von der aufrufenden Regel gesetzt. Im zweiten Fall werden bei der Deklaration der „context-free start-symbols" vor und hinter jedem Startsymbol *LAYOUT*? ergänzt.

```
module Program
imports Expression
exports
  context-free start-symbols Program
  sorts Program Assignment
  context-free syntax
    Identifier "=" Expression ";" -> Assignment
    Assignment+ -> Program
```

Listing 3.8: SDF-Definition von Program

Das in Listing 3.8 gezeigte Modul beschreibt den Aufbau eines *Program*s. Hierfür wird zunächst in Zeile 7 definiert, was eine Assignment ist. Im Anschluss wird ein Program als eine nicht-leere Sequenz von Assignments beschrieben (Zeile 8). Damit ein Parser weiß, mit welcher Regel er beginnen soll, wird in Zeile 4 Program als Startsymbol festgelegt. „context-free“ bedeutet, dass vor und hinter dem Startsymbol *LAYOUT*? ergänzt wird.

Definition der Metasprache

Da die Syntax des SDF sehr umfangreich ist, wie im zuvor angegebenen Beispiel zu sehen, wird sie zur besseren Verständlichkeit in drei Abschnitte unterteilt. Der erste Abschnitt beschreibt den Aufbau eines Moduls. Die Grammatikdefinitionen, die in einem Modul enthalten sein können, werden im zweiten Abschnitt beschrieben. Bei der Definition der Grammatik werden Regeln angegeben, die Terme benutzen. Was die in SDF zulässigen Terme sind, wird schließlich im letzten Abschnitt beschrieben.

Definition eines Moduls

Eine SDF-Beschreibung einer kontextfreien Sprache besteht aus mindestens einem Modul. Wie ein Modul formal aufgebaut ist, zeigt die EBNF-Definition von Module in Formel (3.19).

$$Module = ModuleDeclaration\ ImportSection\ \{ExportSection \mid HiddenSection\}; \tag{3.19}$$

Die einzelnen Bestandteile der Definition sind:

ModuleDeclaration. Jedes Modul beginnt mit einer Zeile, in der das Modul benannt wird. Optional kann es auch mit formalen Parametern versehen werden, wie in den Formeln (3.20) bis (3.22) gezeigt.

(3.20) *ModuleDeclaration* = "module" *ModuleName* [*Parameter*];

(3.21) *Parameter* = "[" *Type* {" " *Type*} "]";

(3.22) *Type* = *Variable* ["[[" *Variable* {"," *Variable*} "]]"];

Dabei steht *ModuleName* in Formel (3.21) für den Namen des Moduls. Befindet sich das Modul in einer eigenen .sdf-Datei, so entspricht der Modulname dem Dateinamen mit dem relativen Pfad separiert durch / als Präfix. *Variable* in Formel (3.22) steht für den Namen einer syntaktischen Variablen. In SDF müssen Variable mit einem Großbuchstaben beginnen.

```
1 module List[X]
2 imports Whitespace
3 exports
4   context-free start-symbols List
5   sorts List X
6   context-free syntax
7     "[" (X ("," X)*)? "]" -> List
```

Listing 3.9: Ein Beispiel für die Verwendung von formalen Parametern

Ein sinnvoller Einsatz für die formalen Parameter, wie sie in der Formel (3.21) definiert sind, ist in Listing 3.9 zu sehen. In dem angegebenen Modul wird eine mit Kommas separierte generische Liste definiert. Wie die Elemente der Liste syntaktisch aufgebaut sind, muss beim Importieren dieses Moduls als aktueller Parameter übergeben werden.

ImportSection. Nach der ModulDeclaration werden die „top-level“-Importe angegeben. Die Importe an dieser Stelle sind für Module, die dieses importieren, sichtbar. Ihr formaler Aufbau wird in den Formeln (3.23) und (3.24) gezeigt. Die Parameter hinter dem Modulnamen in der Formel (3.24) dienen der Übergabe der aktuellen Parameter beim Import von generischen Modulen.

(3.23) *ImportSection* = {*Import*};

(3.24) *Import* = "imports" *ModuleName* [*Parameter*];

Da beim Importieren alle syntaktischen Variablen im aktuellen Modul sichtbar werden, kann es zu *Namenskollisionen* kommen. Daher wurde die Möglichkeit gegeben, die aktuellen Parameters um Umbenennung zu erweitern. Die Syntax der Umbenennung ist in Formel (3.25) gezeigt. Dabei wird der importierte Typ, der links steht, in dem aktuellen Modul in den Namen der rechts steht umbenannt.

$$\textit{Renaming} \quad = \quad \textit{Type}\ \text{"=>"}\ \textit{Type}; \tag{3.25}$$

ExportSection. Alle Angaben in der ExportSection sind für Module sichtbar, die dieses importieren. Der Aufbau dieser Sektion ist in Formel (3.26) zu sehen.

$$\textit{ExportSection} \quad = \quad \text{"exports"}\ \textit{Grammar}\ \{\textit{Grammar}\}; \tag{3.26}$$

HiddenSection. Die Definitionen in der HiddenSection sind nur in dem aktuellen Modul sichtbar. Ihr Aufbau wird in Formel (3.27) gezeigt.

$$\textit{HiddenSection} \quad = \quad \text{"hiddens"}\ \textit{Grammar}\ \{\textit{Grammar}\}; \tag{3.27}$$

Comment. Kommentare können an beliebigen Stellen in der Modulbeschreibung vorkommen. Einzeilige Kommentare werden mit %% eingeleitet. Kommentare innerhalb einer Zeile werden mit % umschlossen.

Definition der Grammatik

Die ExportSection und die HiddenSection, wie sie im vorangegangenen Abschnitt beschrieben wurden, beinhalten die Beschreibung der Grammatik. Der *Aufbau einer Grammatik* ist in Formel (3.28) zu sehen.

$$\begin{aligned} \textit{Grammar} \quad = \quad & \textit{ImportSection} \mid \textit{Aliases} \mid \textit{Sorts} \mid \textit{StartSymbols} \mid \textit{SyntaxDeclaration} \\ & \mid \textit{Priorities} \mid \textit{Restrictions} \end{aligned} \tag{3.28}$$

Die einzelnen Bestandteile sind:

ImportSection. Die ImportSection in der Grammatik ist vom syntaktischen Aufbau identisch mit der „top-level“-ImportSection. Der einzige semantische Unterschied ist, dass die ImportSection in einer HiddenSection nur für das aktuelle Modul sichtbar ist.

Aliases. Aliases dienen der Schreibabkürzung für komplexere Ausdrücke. Der syntaktische Aufbau ist in Formel (3.29) gezeigt.

(3.29) *Aliases* = "aliases" *Term* "->" *Term*;

Ein Beispiel wäre: *aliases* [*a-zA-Z*] -> *Letter*.

Sorts. In dieser Rubrik werden alle syntaktische Variablen aufgeführt, die in dem Modul neu definiert werden. Dazu zählen auch die formalen Parameter. Formel (3.30) zeigt ihren syntaktischen Aufbau.

(3.30) *Sorts* = "sorts" {*Type*};

StartSymbols. Bei StartSymbols werden alle syntaktischen Variablen angegeben, von denen aus alle zur beschriebenen kontextfreien Sprache gehörenden Wörter hergeleitet werden können. Den Aufbau zeigt Formel (3.31).

(3.31) *StartSymbols* = ("lexical" | "context-free") "start-symbols" {*Type*};

Bei „lexical start-symbols" dürfen nur syntaktische Variablen aufgeführt werden, die in einer Syntaxdeklaration der Form „lexical syntax" definiert wurden. Analoges gilt für „context-free start-symbols". Bei kontextfreien Startsymbolen werden darüber hinaus führende und folgende Sequenzen von zu ignorierenden Zeichen übersprungen. Welche Zeichen zu ignorieren sind, kann mit der syntaktischen Variablen *LAYOUT* angegeben werden.

SyntaxDeclaration. Als SyntaxDeclaration versteht man die Angabe der Regeln, mit deren Hilfe die Worte der kontextfreien Sprache hergeleitet werden können. Ihr syntaktischer Aufbau ist in den Formeln (3.32) bis (3.34) beschrieben.

(3.32) *SyntaxDeclaration* = ("lexical" | "context-free") "syntax" {*Rule*};

(3.33) *Rule* = {*Term*} "->" *Term* [*Attributes*];

(3.34) *Attributes* = "{" *Attribute* {"," *Attribute*} "}";

Der Unterschied zwischen „lexical syntax" und „context-free syntax" besteht darin, dass bei Letzterem zwischen allen Elementen der linken Seite jeder angegebenen Regel *LAYOUT*? ergänzt wird. Des Weiteren haben beide SyntaxDeclarations eigene Namensräume. Dies bedeutet, dass alle bei „lexical syntax" definierten syntaktischen Variablen von denen verschieden sind, die in „context-free syntax" deklariert wurden, selbst wenn sie gleich heißen.

Sollte es in der Regelmenge zwei Regeln geben, deren linke und rechte Seite identisch sind, so werden die Attributmengen beider Regeln vereinigt und eine der beiden Regeln wird entfernt.

Die wichtigsten Attribute, die eine Regel haben kann, sind in Formel (3.35) zu sehen. Ihre Bedeutung ist in Tabelle 3.1 gezeigt.

$$\begin{aligned} \textit{Attribute} &= \text{"bracket"} \mid \text{"prefer"} \mid \text{"avoid"} \mid \text{"reject"} \\ &\quad \mid \text{"left"} \mid \text{"right"} \mid \text{"assoc"} \mid \text{"non-assoc"}; \end{aligned} \tag{3.35}$$

Priorities. Mithilfe dieser Definition können Prioritäten für Regeln vergeben werden. Die Syntax zur Angabe von Prioritäten ist in den Formeln (3.36) bis (3.40) gezeigt. Unter „lexical priorities“ werden die Prioritäten zwischen den „lexical syntax“-Regeln festgelegt. Analoges gilt für die Angaben bei „context-free priorities“. Im Allgemeinen sind die angegebenen Prioritäten transitiv. Sollte dies nicht gewünscht sein, so muss an Stelle von ">"".>"verwendet werden.

$$\begin{aligned} \textit{Priorities} &= (\text{"lexical"} \mid \text{"context-free"})\ \text{"priorities"} \\ &\quad \textit{Priority}\ \{\text{","}\ \textit{Priority}\}; \end{aligned} \tag{3.36}$$

$$\begin{aligned} \textit{Priority} &= \textit{RuleSet}\ [\textit{ArgIndex}]\ [\text{"."}]\ \text{">"}\ \textit{RuleSet} \\ &\quad \{[\textit{ArgIndex}]\ [\text{"."}]\ \text{">"}\ \textit{RuleSet}\}; \end{aligned} \tag{3.37}$$

$$\textit{RuleSet} = \textit{Rule} \mid \text{"\{"}[\textit{AssocLabel}]\ \textit{Rule}\ \{\textit{Rule}\}\ \text{"\}"}; \tag{3.38}$$

$$\textit{ArgIndex} = \text{"}\langle\text{"}\ \textit{NatNumber}\ \{\text{","}\ \textit{NatNumber}\}\ \text{"}\rangle\text{"}; \tag{3.39}$$

$$\textit{AssocLabel} = (\text{"left"} \mid \text{"right"} \mid \text{"assoc"} \mid \text{"non-assoc"})\ \text{":"}; \tag{3.40}$$

```
context-free priorities
    E "[" E "]" -> E
    <0> >
    E "+" E -> E {left}
```

Listing 3.10: Ein Beispiel für die Verwendung von Argumentenlisten in Priorities (Quelle: [BKV07])

Um die Bedeutung der Argumentenliste (Formel (3.39)) vor ">"zu verdeutlichen, folgt in Listing 3.10 ein Beispiel. Die Angabe des Indexes 0 bedeutet, dass im Falle von $3 + 4\ [\ 5\ ]$ die Regel in Zeile 1 Vorrang hätte $(3 + (4\ [\ 5\ ]))$. Bei einer Eingabe der Form $3\ [4 + 5]$ hat allerdings die Regel in Zeile 3 Vorrang $(3\ [(4 + 5)])$.

Attribut	Bedeutung
prefer	Sollte es an einer Stelle der Ableitung neben der mit diesem Attribut versehenen Regel eine weitere anwendbare Regel geben, so wird die „prefer"-Regel genommen.
avoid	Gibt es bei der Ableitung zwei anwendbare Regeln, so wird der Regel ohne dem „avoid"-Attribut der Vorrang gegeben.
reject	Ableitungsbäume, die eine mit „reject" attributierte Regel verwenden müssen, werden verworfen.
bracket	Dieses Attribut hat nur eine semantische Bedeutung für Programme, die SDF implementieren. So kann dieses Attribut beispielsweise von Programmen verwendet werden, die Parsebäume miteinander vergleichen. Knoten des Parsebaums, die durch bracket-attributierte Regeln erzeugt wurden, können beim Vergleich ignoriert werden, da sie keine semantische Bedeutung haben. Hierfür müssen Regeln der Form *aVa*->V mit dem bracket-Attribut versehen werden.
left	Indem dieses Attribut angegeben wird, ist die Regel als links-assoziativ deklariert.
right	Durch Angabe dieses Attributs wird die Regel als rechts-assoziativ interpretiert.
assoc	Die Bedeutung dieses Attributs ist identisch mit dem Attribut „left".
non-assoc	Dieses Attribut bedeutet, dass für keine syntaktische Variable der linken Seite diese Regel erneut angewendet werden darf. Ein Beispiel für die Verwendung dieses Attributes wäre die Erweiterung der Expression-Defintion in Listing 3.7 um die Potenz. Da beispielsweise bei a^{b^c} nicht definiert ist, ob es als $a^{(b^c)}$ oder als $(a^b)^c$ zu interpretieren wäre, soll a^{b^c} verboten werden. Dies wird erreicht, indem man folgende Regel ergänzt: *Expression* "^" *Expression* -> *Expression* {**non-assoc**} Darüber hinaus müsste noch angegeben werden, dass die Potenz eine höhere Priorität als die Multiplikation und Ganzzahldivision hat, was an dieser Stelle aber nicht gezeigt wird.

Tabelle 3.1: Die vordefinierten Regelattribute in SDF.

Restrictions. Wie Restrictions aufgebaut sind, zeigen die Formeln (3.41) bis (3.43). Eine Restriktion bedeutet, dass hinter keinem aus der linken Seite herleitbarem Wort Zeichen der rechten Seite vorkommen dürfen.

(3.41) *Restrictions* = ("lexical" | "context-free") "restrictions" {*Restriction*};

(3.42) *Restriction* = *Term* {*Term*} "-/-" *Lookahead*;

(3.43) *Lookahead* = *CharacterClass* {"." *CharacterClass*};

Bei lexikalischen Restriktionen dürfen auf der linken Seite nur Literale oder lexikalische Variablen vorkommen. Im Falle von kontextfreien Restriktionen sind nur syntaktische Variablen und Typen, wie sie in Formel (3.22) definiert sind, erlaubt.

Beschreibung eines Terms

Die zuvor beschriebene Grammatik verwendet an mehreren Stellen Terme. Was in SDF *zulässige Terme*[3] sind, zeigt Formel (3.44).

(3.44) *Term* = *Literal* | *Type* | *CharacterClass* | *Option* | *Sequence*
| *Repetition* | *List* | *Alternative* | *LabeledTerm*
| *Tuple* | *FunctionTerm* | *Lifting* | *PrefixFunction* | *LAYOUT*;

Die elementaren Terme sind:

Literal. Unter einem Literal ist in SDF die explizite Angabe einer Character-Folge zu verstehen. Es gibt zwei Arten von Literalen: "*literal*" und '*literal*'. Im ersten Fall wird die Groß- und Kleinschreibung beachtet, im zweiten Fall ignoriert. Um Anführungszeichen und Apostrophen in Literalen verwenden zu können, müssen sie mit einem vorangestellten Backslash versehen werden.

Type. Ein Type hat die Form, wie sie in der Formel (3.22) angegeben ist und steht für eine syntaktische Variable. Es werden lexikalische und kontextfreie Types unterschieden.

[3] Die in der SDF-Beschreibung definierten Symbole, werden in dieser Masterarbeit als Terme bezeichnet, da diese Bezeichnung besser verständlich ist.

CharacterClass. Eine CharacterClass ist eine Menge von Zeichen. Zur Definition einer CharacterClass gibt es verschiedene Alternativen:

- explizite Angabe der Zeichen z.B. [abc]
- Angabe einer Menge von kleinen Buchstaben z.B. [a-i]
- Angabe einer Menge von großen Buchstaben z.B. [A-X]
- Angabe einer Menge von Ziffern z.B. [3-7]
- Angabe einer Menge von Zeichen anhand ihres Dezimalwerts z.B. [\100-\200]

Zur Darstellung von Sonderzeichen wie + in einer CharacterClass, müssen sie ebenfalls mit einem vorangestellten Backslash versehen werden. In der abkürzenden Schreibweise mit einem Bindestrich muss das erste Zeichen lexikalisch vor dem zweiten Zeichen kommen. Des Weiteren gibt es noch einige Operatoren für CharacterClass:

- ~C: Komplement der CharacterClass C
- C/D: Mengendifferenz der beiden CharacterClasses C und D
- C/\D: Durchschnitt der beiden CharacterClasses C und D
- C\/D: Vereinigung der beiden CharacterClasses C und D

Die zusammengesetzten Terme sind:

Option. Die Option repräsentiert einen Term, der höchstens einmal vorkommen kann. Ihren Aufbau definiert Formel (3.45).

$$Option \;=\; Term \text{ "?"}; \tag{3.45}$$

Sequence. Eine Sequence beschreibt eine Gruppierung von mindestens zwei Termen[4], wie sie in Formel (3.46) definiert ist.

$$Sequenz \;=\; \text{"("}\; Term\; Term\; \{Term\}\; \text{")"}; \tag{3.46}$$

Repetition. Eine Repetition ist das mehrfache Vorkommen eines Terms und ist, wie in Formel (3.47) gezeigt, definiert. Eine Folge mit mindestens einem Term wird mit einem + dargestellt. Eine Folge mit einer beliebigen Anzahl von Termen wird durch ein * symbolisiert.

$$Repetition \;=\; Term\; (\text{"+"} \mid \text{"*"}); \tag{3.47}$$

[4] Eine leere Sequence wird in SDF entfernt und eine Sequence, die nur den Term T enthält, wird durch T ersetzt.

List. Eine List repräsentiert das mehrfache Vorkommen eines Terms, wobei zwischen zwei Termen ein Separatorzeichen steht. Der syntaktische Aufbau ist in Formel (3.48) zu sehen. Der zweite Term in der Formel steht für das Separatorzeichen.

$$\textit{List} \quad = \quad \text{"\{"}\ \textit{Term Term}\ \text{"\}"("+" | "*")}; \tag{3.48}$$

Alternative. Die Syntax einer Alternative zwischen zwei Termen ist in Formel (3.49) zu sehen.

$$\textit{Alternative} \quad = \quad \textit{Term}\ \text{"|"}\ \textit{Term}; \tag{3.49}$$

LabeledTerm. Ein Label hat keine semantische Bedeutung. Es dient lediglich der besseren Lesbarkeit oder wird beispielsweise von API-Generatoren genutzt, um Methoden zu benennen. Die Syntax ist in Formel (3.50) definiert.

$$\textit{LabeledTerm} \quad = \quad \textit{Label}\ \text{":"}\ \textit{Term}; \tag{3.50}$$

Tuple. Tuples stellen eine Schreibabkürzung für die Definition einer Sequenz mit einer festen Länge dar. Das gesamte Tuple muss dabei in spitzen Klammern stehen und die einzelnen Elemente müssen mit Kommas separiert sein. Die Syntax eines Tuples ist, wie in Formel (3.51) beschrieben, definiert.

$$\textit{Tuple} \quad = \quad \text{"}\langle\text{"}\ \textit{Term}\ \{\text{","}\ \textit{Term}\}\ \text{"}\rangle\text{"}; \tag{3.51}$$

FunctionTerm. FunctionTerms haben die Form, wie sie in Formel (3.52) gezeigt ist. Für welche Syntax sie in der beschriebenen kontextfreien Sprache stehen, muss explizit angegeben werden.

$$\textit{FunctionTerm} \quad = \quad \text{"("}\ \{\textit{Term}\}\ \text{"=>"}\ \textit{Term}\ \text{")"}; \tag{3.52}$$

Lifting. Lifitings haben die in Formel (3.53) gezeigte Form. Dabei wird die in "`" eingeschlossene Zeichenfolge durch eine Folge von Literalen ersetzt. So wird beispielsweise *`ein Lifting`* ersetzt durch *"ein" "Lifting"*.

$$\textit{Lifting} \quad = \quad \text{"`"}\ \textit{CharacterSequence}\ \text{"`"}; \tag{3.53}$$

PrefixFunction. Eine PrefixFunction ist ein spezieller Term, der nur auf der linken Seite einer Regel vorkommen kann. Der syntaktische Aufbau ist in Formel (3.54) zu sehen. *FunctionName* steht dabei für ein Literal ohne umschließende Anführungs-

zeichen und Apostrophe. Dieser Termtyp dient lediglich der Schreibabkürzung. So wird beispielsweise $plus(E, E)$ in die Termfolge "plus(" E "," E ")" übersetzt.

(3.54) *PrefixFunction* = *FunctionName* "(" *Type* {"," *Type*} ")";

LAYOUT. LAYOUT stellt eine vordefinierte syntaktische Variable dar und repräsentiert Sequenzen von Zeichen, die ignoriert werden sollen.

3.3 Top-Down-Algorithmen

Wie in [SLA08] beschrieben, wird bei den Top-Down-Algorithmen versucht ausgehend vom Startsymbol eine Ableitung für eine gegebene Eingabe zu finden. Zunächst wird in Abschnitt 3.3.1 der klassische LL-Algorithmus kurz vorgestellt, bevor in Abschnitt 3.3.2 der GLL-Algorithmus erläutert wird.

3.3.1 LL(k) Parsing

Beim LL(k) Parsing wird das Eingabewort von links nach rechts verarbeitet und dabei eine Linksableitung erstellt. Bei jedem Schritt wird durch das vorausschauende Betrachten der folgenden k Eingabebuchstaben versucht zu bestimmen, welche Regel anzuwenden ist. Dadurch wird laut [SLA08] linearer Zeitaufwand erreicht. Werden die vorgestellten Algorithmen um Backtracking erweitert, werden sie LL(*) genannt. Dieser Abschnitt beruht auf [SLA08] und genauso wie im Buch wird hier nur der Fall von $k = 1$ betrachtet. Um die vorgestellten Algorithmen besser verstehen zu können, wird eine Grammatik zur Erkennung von Binärzahlen angegeben:

$G_{bindig} = \left(\{N, N_{suff}\}, \{0, 1\}, R, N\right)$ mit
$R =$

$$
\begin{aligned}
\{0 &\rightarrow N \\
1\ N_{suff} &\rightarrow N \\
\varepsilon &\rightarrow N_{suff} \\
0\ N_{suff} &\rightarrow N_{suff} \\
1\ N_{suff} &\rightarrow N_{suff}\}
\end{aligned}
$$

Beschränkung der zulässigen Grammatiken

Bevor die Algorithmen vorgestellt werden können, muss zunächst geklärt werden, welche Grammatiken zulässig sind. Hierfür werden die Definitionen der *FIRST*- und der *FOLLOW*-Mengen benötigt. Bei ersterer handelt es sich um die Menge aller initialen Terminale oder ε, die sich aus einem Terminal oder einer lexikalischen Variable herleiten lassen.

Definition (*FIRST*)
Sei V die Menge der lexikalischen Variablen, T die Menge der Terminalzeichen und R die Menge der Regeln, dann ist die Funktion
$FIRST : (V \cup T \cup R) \rightarrow 2^{T \cup \{\varepsilon\}}$
definiert als:

- $FIRST(a) = \{a\}$,
  falls $a \in T$
- $FIRST(X) = \bigcup_{\alpha \rightarrow X \in R} FIRST(\alpha \rightarrow X)$,
  falls $X \in V$ und $\alpha \in (V \cup T \cup \{\varepsilon\})*$
- $FIRST(\varepsilon \rightarrow X) = \{\varepsilon\}$
- $FIRST(\beta_0 \dots \beta_k \rightarrow X) = FIRST(\beta_0) \cup \dots \cup FIRST(\beta_i)$,
  falls $\exists\, i \leq k : \varepsilon \notin FIRST(\beta_i) \wedge \forall\, j < i : \varepsilon \in FIRST(\beta_j)$
- $FIRST(\beta_0 \dots \beta_k \rightarrow X) = FIRST(\beta_0) \cup \dots \cup FIRST(\beta_k) \cup \{\varepsilon\}$,
  falls $\forall\, i < k : \varepsilon \in FIRST(\beta_i)$

Wird nach dieser Definition die *FIRST*-Menge von N_{suff} gebildet, so ist:

$$
\begin{aligned}
& FIRST(N_{suff}) \\
&= FIRST(\varepsilon \rightarrow N_{suff}) \cup FIRST(0\ N_{suff} \rightarrow N_{suff}) \cup FIRST(1\ N_{suff} \rightarrow N_{suff}) \\
&= \{\varepsilon\} \cup FIRST(0) \cup FIRST(1) \\
&= \{\varepsilon, 0, 1\}
\end{aligned}
$$

Die *FOLLOW*-Menge einer lexikalischen Variable X gibt die Menge aus Terminalen bzw. \$ an, die direkt auf das durch X erkannte Lexem folgen können. \$ ist dabei die Endmarke der Eingabe.

Definition (*FOLLOW*)
Sei V die Menge der lexikalischen Variablen, T die Menge der Terminalzeichen und R die Menge der Regeln, dann enthält die Ergebnismenge des Aufrufs $FOLLOW(X)$ der Funktion

$FOLLOW : V \rightarrow 2^{T \cup \{\$\}}$

- $\$$,
  falls X das Startsymbol ist.
- $FIRST(\beta) \backslash \{\varepsilon\}$,
  für jede Regel der Form $\alpha X \beta \rightarrow A \in R$ mit $\alpha, \beta \in (V \cup T)*$ und $\beta \neq \varepsilon$.
- $(FOLLOW\,(A) \cup FIRST\,(\beta)) \backslash \{\varepsilon\}$,
  für jede Regel der Form $\alpha X \rightarrow A \in R$ oder $\alpha X \beta \rightarrow A \in R$
  mit $A \neq X$, $\alpha, \beta \in (V \cup T)*$, $\beta \neq \varepsilon$ und $\varepsilon \in FIRST(\beta)$.

Um LL(1)-Algorithmen zum Parsen verwenden zu können, muss die verwendete Grammatik eindeutig und ohne Linksrekursion sein. Da bei lexikalischen Variablen in der Grammatik anhand des ersten folgenden Eingabebuchstabens eindeutig entschieden werden muss, welche Regel anzuwenden ist, müssen für zwei unterschiedliche Regeln $\alpha \rightarrow A$ und $\beta \rightarrow A$ der Grammatik folgende Bedingungen erfüllt sein:

1. $FIRST(\alpha) \cap FIRST(\beta) = \varnothing$
2. $\varepsilon \in FIRST(\beta) \Rightarrow FIRST(\alpha) \cap FOLLOW(A) = \varnothing$
3. $\varepsilon \in FIRST(\alpha) \Rightarrow FIRST(\beta) \cap FOLLOW(A) = \varnothing$

Nutzung einer Parse-Tabelle

Der erste vorgestellte LL(1) Parsing-Algorithmus benötigt eine Parse-Tabelle, die aus der Grammatik generiert wird[5]. Diese Tabelle gibt für die aktuelle lexikalische Variable an, welche Regel bei welchem Eingabebuchstaben verwendet werden soll. Für die zu Beginn dieses Abschnitts vorgestellte Grammatik G_{bindig} sieht die Parse-Tabelle wie in Tabelle 3.2 gezeigt aus.

lexikalische Variable	**Eingabesymbol**		
	0	1	$
N	$0 \rightarrow N$	$1\ N_{suff} \rightarrow N$	*error*
N_{suff}	$0\ N_{suff} \rightarrow N_{suff}$	$1\ N_{suff} \rightarrow N_{suff}$	$\varepsilon \rightarrow N_{suff}$

Tabelle 3.2: Die aus der Grammatik G_{bindig} erzeugte LL-Parse-Tabelle.

[5]Wie die Erzeugung der Parse-Tabelle vonstatten geht, ist auf Seite 271 in [SLA08] nachzulesen

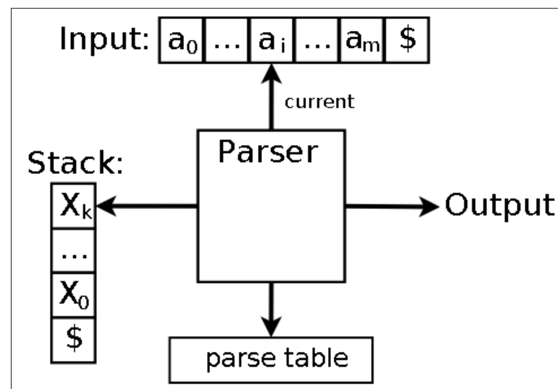


Abbildung 3.1: Schematische Darstellung eines LL-Parsers. Quelle: [SLA08]

Abbildung 3.1 zeigt die schematische Darstellung eines LL-Parsers. Als Eingabe erhält er das um den Endmarker $ erweiterte Eingabewort. Darüber hinaus besitzt er einen Stack, auf dem initial das Startsymbol der Grammatik liegt.

In jedem Arbeitsschritt schaut der Parser das oberste Element auf dem Stack an. Handelt es sich um ein Terminalzeichen t, so kontrolliert er, ob es identisch mit dem aktuellen Eingabebuchstaben ist. Im positiven Fall, entfernt er t vom Stack und verschiebt den Zeiger (*current*) auf den Folgebuchstaben des Eingabeworts. Ansonsten gibt er aus, dass die Eingabe nicht erkannt wurde. Sollte das oberste Stack-Element eine lexikalische Variable sein, so schaut der Parser in der Parse-Tabelle nach, welche Regel bei der aktuellen Variable und dem aktuellen Eingabebuchstaben anzuwenden ist. Sollte eine Regel eingetragen sein, so entfernt er das Top-Element vom Stack und legt die Elemente des Regelrumpfs in umgekehrter Reihenfolge auf den Stack. Sollte in der Tabelle *error* eingetragen sein, so meldet der Parser, dass das Eingabewort nicht erkannt werden konnte. Nur in dem Fall, dass der Stack leer ist und *current* auf den Endmarker der Eingabe zeigt, wird die erfolgreiche Erkennung der Eingabe ausgegeben.

Übereinstimmung	**Stack**	**Eingabe**	**Aktion**
	N $	101$	Anwendung von $1\ N_{suff} \to N$
	$1\ N_{suff}$ $	101$	Übereinstimmung mit 1
1	N_{suff} $	01$	Anwendung von $0\ N_{suff} \to N_{suff}$
1	$0\ N_{suff}$ $	01$	Übereinstimmung mit 0
10	N_{suff} $	1$	Anwendung von $1\ N_{suff} \to N_{suff}$
10	$1\ N_{suff}$ $	1$	Übereinstimmung mit 1
101	N_{suff} $	$	Anwendung von $\varepsilon \to N_{suff}$
101	$	$	Ausgabe: Eingabe erkannt

Tabelle 3.3: Der LL-Parse-Vorgang für die Eingabe 101 mithilfe der Parse-Tabelle 3.2.

Die Tabelle 3.3 zeigt die Arbeitsweise eines LL(1)-Parsers, der mithilfe der obigen Parse-Tabelle das Eingabewort 101 erkennt.

Nutzung des rekursiven Abstiegs

Als Alternative zur Verwendung einer Parse-Tabelle kann auch der rekursive Abstieg genutzt werden. Dabei wird ausgenutzt, dass anhand des nächsten Buchstabens in der Eingabe die anzuwendende Regel eindeutig identifiziert wird. Die Idee des rekursiven Abstiegs besteht darin, dass für alle Regeln mit dem gleichen Regelkopf eine Methode implementiert sein muss, die anhand des nächsten Eingabebuchstabens entscheidet, welche Methode auszuführen ist. Für die Regeln mit Kopf N in der zu Beginn dieses Abschnitts vorgestellten Grammatik G_{bindig} würde die in Listing 3.11 gezeigte Methode benötigt.

```
1  public boolean N() {
2     if (lookahead ∈ {0}) {
3        if (lookahead != '0') {
4           return false;
5        }
6        lookahead = getNextLookahead();
7        return true;
8     } else if (lookahead ∈ {1}) {
9        if (lookahead != '1') {
10          return false;
11       }
12       lookahead = getNextLookahead();
13       if (!N_suff()) {
14          return false;
15       }
16       return true;
17    } else {
18       return false;
19    }
20 }
```

Listing 3.11: Die für den rekursiven Abstieg der N-Regeln benötigte Methode.

Allgemein wird für jede lexikalische Variable A der Grammatik eine Methode benötigt. Sollten in einer Grammatik nur die Regeln $R_0 = \alpha_0 \to A$ bis $R_n = \alpha_n \to A$ über den Kopf A verfügen, so würde für sie die in Listing 3.12 gezeigte Methode erzeugt werden.

```
public boolean N() {
    if (lookahead ∈ firstTerminals(R_0)) {
        code(α_0)
        return true;
    } else if (lookahead ∈ firstTerminals(R_1)) {
        code(α_1)
        return true;
    }
    ...
    else if (lookahead ∈ firstTerminals(R_n)) {
        code(α_n)
        return true;
    } else {
        return false;
    }
}
```

Listing 3.12: Die für die Regeln R_0 bis R_n mit Kopf A erzeugte Methode.

Dabei ist $\mathit{firstTerminals}(\alpha \to A) =$

- $\mathit{FIRST}(\alpha)$, falls $\varepsilon \notin \mathit{FIRST}(\alpha)$.
- $\mathit{FIRST}(\alpha) \cup \mathit{FOLLOW}(A)$, sonst.

Die Funktion $\mathit{code} : (V \cup T)^* \to \mathit{String}$ ist wie folgt definiert:

- $\mathit{code}(\alpha) = \mathit{code}(X_0) \circ \cdots \circ \mathit{code}(X_n)$, falls $\alpha = X_0 \ldots X_n$ und $\circ$ die Konkatenation von Strings ist.
- $\mathit{code}(a)$ erzeugt den in Listing 3.13 gezeigten Code, falls a ein Terminal ist.
- $\mathit{code}(Z)$ erzeugt den in Listing 3.14 gezeigten Code, falls Z eine syntaktische Variable ist.

```
if (lookahead != 'a') {
    return false;
}
lookahead = getNextLookahead();
```

Listing 3.13: Der für das Terminal a erzeugte Code.

```
if (!Z()) {
    return false;
}
```

Listing 3.14: Der für die syntaktische Variable Z erzeugte Code.

3.3.2 GLL Parsing

Im Gegensatz zu den zuvor beschriebenen Algorithmen LL(k) und LL(*) wurde der *Generalized LL Parsing Algorithmus* entwickelt, um auch *mehrdeutige und links-rekursive* Grammatiken erkennen zu können. Er wurde 2010 von Elizabeth Scott und Adrian Johnstone in [SJ10] veröffentlicht.

Die Idee hinter GLL Parsing ist ähnlich dem des *rekursiven Abstiegs*, wie er bei LL(k) im Abschnitt 3.3.1 beschrieben wurde. Im Gegensatz zum herkömmlichen rekursiven Abstieg, werden bei GLL die einzelnen BNF-Regeln *in Code überführt*, der mit bedingten *Verzweigungen und goto-Sprüngen* zu gelabelten Codestellen arbeitet, wie im Abschnitt „Erzeugung eines Recognizers[6]" erläutert wird. Die schematische Darstellung des erzeugten Recognizers ist in Abbildung 3.2 zu sehen.

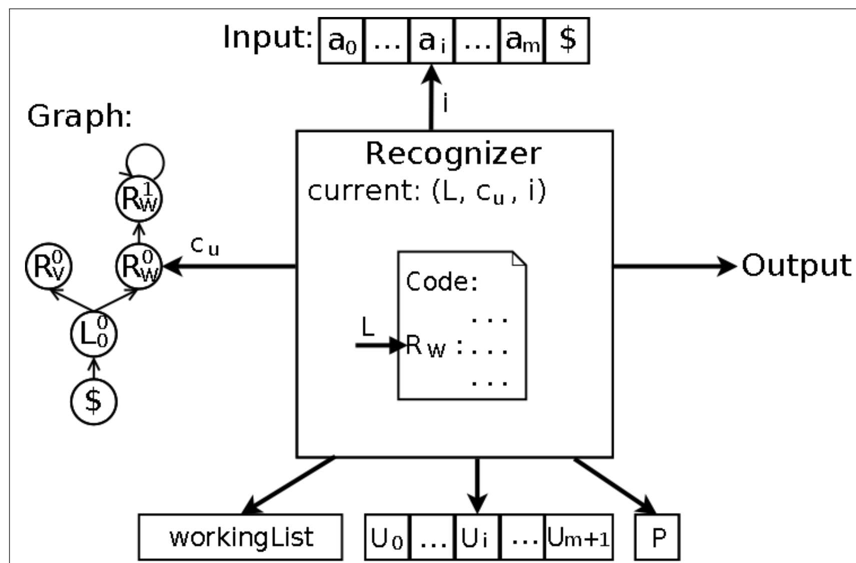


Abbildung 3.2: Schematische Darstellung des GLL-Recognizers

Schematische Darstellung

Die *Eingabe* des Recognizers befindet sich in einem Array und besteht aus einem Wort, das um die Schlussmarke $ ergänzt wurde.

Durch den *Verzicht auf Methoden* wird erreicht, dass *an Stelle des Call-Stacks ein gerichteter Graph* zum Merken der aktuellen Arbeitspositionen im generierten Code genutzt werden kann. Die Datenstruktur des Graphen ist notwendig, da bei Mehrdeutigkeit eine

[6]Ein Recognizer entscheidet, ob ein Wort zu einer Sprache gehört oder nicht.

Verzweigung möglich ist und so *verschiedene Stacks gleichzeitig simuliert* werden können. Alle simulierte Stacks haben eine gemeinsame Wurzel, die durch den Knoten $ ausgedrückt wird. Ein Stack besteht dann aus den Knoten eines Pfads der Form:
$\langle \$, e_1, v_1, \ldots v_{k-1}, e_k, v_k \rangle, k \geq 0$
Das Top-Element v_k des aktuellen Stacks wird durch den Zeiger c_u referenziert. Wird dieses Element mittels *pop* vom simulierten Stack entfernt, so wird v_k nicht gelöscht, sondern nur der Zeiger c_u auf den Vorgängerknoten v_{k-1} gesetzt.

Bei der Arbeit des Recognizers gibt es einen aktuellen *Arbeitszustand*. In der Abbildung 3.2 wird er als „current" bezeichnet und besteht aus dem Tripel (L, c_u, i). L ist das Label der Codeposition, an der weitergearbeitet wird. c_u ist der Knoten, der das aktuelle Top-Element des Stacks markiert und i ist der Index des zu erkennenden Eingabezeichens. Da beispielsweise bei indeterministischen Grammatiken nicht eindeutig bestimmt werden kann, mit welcher Regel weitergearbeitet werden soll, wird für jede alternative Abarbeitung ein Arbeitszustand in die WorkingList eingefügt. Die Elemente der WorkingList werden im Folgenden bearbeitet. Die Reihenfolge, in der die einzelnen Arbeitszustände bearbeitet werden, hat auf das Ergebnis des Algorithmus keine Auswirkungen. Wie dies im Detail geschieht, ist im Abschnitt „Parsen eines Worts" beschrieben.

Für jeden Buchstaben des Eingabeworts existiert eine Menge U_j, die alle Arbeitszustände merkt, die bei der Erkennung des Eingabezeichens mit Index i, bearbeitet wurden. Die Menge $\mathcal{P}$ enthält Tupel (u, i), die bedeuten, dass bei der Erkennung des Eingabezeichens mit Index i das Top-Element c_u von einem simulierten Stack entfernt wurde. Die Bedeutung dieser Mengen wird auch im Abschnitt „Parsen eines Worts" näher erläutert.

Eine Bewertung des Aufwands von GLL wird schließlich im letzten Abschnitt „Aufwandsbewertung" vorgenommen.

Erzeugung eines Recognizers

Bei der *Erzeugung eines Recognizers* wird eine kontextfreie Grammatik, die in der BNF-Notation definiert ist, *schrittweise in Code überführt*. Die BNF-Regeln müssen dabei in der Form vorliegen, dass *jedes Nonterminal durch genau eine Regel* definiert wird und es *genau eine Startregel* gibt. Um die formale Beschreibung zu veranschaulichen, wird im Folgenden parallel der Code für die in Formel (3.55) angegebene nur aus einer Regel bestehenden *Beispielgrammatik* erzeugt. Die Listings, die die formale Beschreibung zeigen, stammen bis auf Umbenennung der Bezeichner aus [SJ10]. Diese BNF-Regel beschreibt Worte, die aus einer Folge von mindestens einem a bestehen. Sie ist indeterministisch, da die

FIRST-Mengen[7] beider Alternativen auf der rechten Seite aus der Menge $\{a\}$ bestehen. Darüber hinaus besitzt die angegebene BNF-Regel durch *Sa* auch eine Linksrekursion.

$$S = S\,a \mid a \tag{3.55}$$

Um den generierten Code besser verstehen zu können, ist es notwendig zu wissen, dass der generierte Recognizer über eine *Workinglist* iteriert. Dies bedeutet, dass er nacheinander jedes Element der Workinglist bearbeitet, bis diese leer ist. Die Elemente der Workinglist sind *Tripel der Form* (L, v, i), wobei L für eine mit dem Label L markierte Position im Code, v für einen Knoten des gerichteten Graphen und damit für einen Stack stehen. i ist der Index des zu bearbeitenden Buchstabens des Eingabeworts.

Bei der Codegenerierung werden *drei verschiedene Arten von Labels* verwendet, die als Sprungmarken dienen. Der erste Labeltyp hat die Form L_V und markiert den *Codeanfang einer Regel*, die das Nonterminal V definiert. L_0 steht dabei für das fest vorgegebene Label, das die Stelle im Code markiert, die für die Iteration über die Workinglist zuständig ist. Besteht der Body einer Regel mit dem Kopf V aus den verschiedenen Alternativen $V_1 \mid V_2 \mid ... \mid V_n$ mit $n \in \mathbb{N}$ so wird der Anfang jeder Regel mit den Labeln L_{V_1} *bis* L_{V_n} versehen. Damit die Labels für die Rücksprungadressen eindeutig sind, müssen zunächst alle Nonterminale in den Rümpfen aller Regeln mit einem eindeutigen Index versehen werden. Kommt in einer Alternative V_k mit $1 \leq k \leq n$ ein Nonterminal W_i vor, so wird die Rücksprungadresse nach Abarbeitung des Codes von W_i mit dem Label R_{W_i} markiert. Formel (3.56) zeigt die Position der bei der Codegenerierung erzeugten Labels in der zuvor beschriebenen Beispielgrammatik.

$$L_S\, S = L_{S_1}\, S_1\, R_{S_1}\, a \mid L_{S_2}\, a \tag{3.56}$$

Bei der Codegenerierung werden *Aufrufe der vier Methoden test, add, pop und create* erzeugt. Ihre Arbeitsweise wird im Abschnitt „Parsen eines Worts“ definiert.

Die Beschreibung der Codegenerierung wird in vier Bereiche untergliedert. Zunächst wird die *Erzeugung des Coderumpfs* erläutert. Im Anschluss wird die Codeerzeugung für eine BNF-Regel beschrieben. Danach wird der generierte Code für jede Alternative im Rumpf einer Regel erklärt. Zum Schluss wird erläutert, wie der erzeugte Code für die Terminale und Nonterminale aussieht, die auf das erste Elemente einer Alternative folgen.

[7] Die FIRST-Menge von α beschreibt die Menge der ersten Terminalzeichen, die sich aus α herleiten lassen. Die genaue Definition ist in Abschnitt 3.3.1 angegeben.

Erzeugung des Coderumpfs

```
1      read the input into I and set I[m]:=$, i:=0
2      create graph nodes u_1 = L_0^0, u_o =$ and an edge from u_1 to u_0
3      c_u:=u_1
4      for (0≤ j ≤ m) { U_j:=∅}
5      workingList:=∅, P:=∅
6
7      if(I[0] ∈ FIRST(S$))){ goto L_S } else { report failure }
8
9  L_0:if (workingList ≠ ∅){
10         remove a triple (L,u,j) from workingList
11         c_u:=u, i:=j, goto L
12     } else if ((L_0,u_0) ∈ U_m){
13         report success
14     } else{
15         report failure
16     }
17 L_A:code(A,i)
18     ...
19 L_X:code(X,i)
```

Listing 3.15: Der erzeugte Coderumpf.

Listing 3.15 zeigt den generierten *Coderumpf des Recognizers*. Die Zeilen 1 bis 5 dienen der Initialisierung. Zunächst wird in Zeile 1 das Eingabewort in ein Array mit dem Namen I kopiert. Im Anschluss wird hinter das Eingabewort ein \$ angehängt. m steht für die Länge des Eingabeworts. Zum Schluss wird i, der Index des zu erkennenden Worts, auf das erste Zeichen gesetzt.

In der zweiten Zeile wird der gerichtete *Graph initialisiert*, indem die beiden Knoten u_0 und u_1 erzeugt und mit einer Kante verbunden werden. Da der Graph einen möglicherweise verzweigten Call-Stack repräsentiert, ist $u_0 = \$$ als das unterste Element des Stacks zu verstehen. Es dient der Erkennung, ob ein Stack abgearbeitet wurde oder nicht. Seine genaue Bedeutung wird im Abschnitt „Parsen eines Worts“ ersichtlich werden. Der Knoten $u_1 = L_0^0$ repräsentiert den Startaufruf im Call-Stack. Dabei steht L_0^0 für die Abarbeitung des Labels L_0 und 0 für das Eingabezeichen, das als nächstes bearbeitet werden muss. In Zeile 3 wird u_1 als der aktuelle Graphknoten, der in c_u gemerkt wird, festgelegt.

In den Zeilen 4 und 5 werden die Mengen *workingList*, $\mathcal{P}$ sowie alle U-Mengen als leere Menge initialisiert. Die Bedeutung der einzelnen Mengen wird im Abschnitt „Parsen eines Worts“ erläutert.

Die *Erkennung des Eingabeworts* wird in Zeile 7 angestoßen. Hier wird zunächst überprüft, ob das erste Zeichen $I[0]$ aus dem Startsymbol S herleitbar ist. Falls dem so ist, beginnt die Abarbeitung der Regel S, indem zu der Coderepräsentation dieser Regel, deren erste Zeile mit dem Label L_S markiert ist, gesprungen wird. Ansonsten wird ausgegeben, dass das Eingabewort nicht erkannt werden kann. Falls das Startsymbol der BNF-Regeln nicht S heißt, so muss anstelle von S der Name des Startsymbols angegeben werden und das Label L_S angepasst werden.

Die Zeilen 9 bis 16 repräsentieren die *Iteration über die workingList*. Falls es noch Elemente in der *workingList* gibt, wird eins entfernt. In welcher Reihenfolge die Elemente entfernt werden, spielt für den Algorithmus keine Rolle. Das soeben entnommene Element (L, u, j) repräsentiert einen gespeicherten Zustand bei der Erkennung des Eingabeworts. L steht für das Label, bei dem weitergearbeitet werden soll. u steht für einen Knoten des Graphen, der das Top-Element des Call-Stacks der gespeicherten Abarbeitung ist. j ist der Index des nächsten zu erkennenden Zeichens des Eingabeworts. In Zeile 11 wird schließlich der gespeicherte Zustand wiederhergestellt und durch den Sprung zum Label L die Arbeit fortgesetzt. Die Zeilen 12 bis 16 dienen der Erkennung, ob ein Wort erfolgreich erkannt wurde oder nicht. Die Bedeutung der Bedingung $(L_0, u_0) \in U_m$ in Zeile 12 wird im Abschnitt „Parsen eines Worts“ erklärt.

In den Zeilen 17 bis 19 werden schließlich *alle Regeln der BNF-Definition* (A bis X) mithilfe der Funktion *code* in Code übersetzt. Wie dies genau geschieht, ist im Abschnitt „Parsen eines Worts“ zu lesen. Im Falle der zuvor eingeführten Beispielgrammatik würde an dieser Stelle nur L_S : *code*(S, i) stehen.

Codeerzeugung einer BNF-Regel

Die Codeerzeugung für eine *Regel der Form* $A ::= \alpha_1 \mid \ldots \mid \alpha_n$ unterscheidet sich je nachdem, ob sich anhand des aktuellen Zeichens der Eingabe eindeutig bestimmen lässt, welche der Alternativen im Rumpf der Regel genommen wird, oder nicht. Im Falle der Eindeutigkeit ist *code*(A, j), wie in Listing 3.16 zu sehen, definiert. Ansonsten zeigt Listing 3.17 die Definition von *code*(A, j). Der formale Parameter A von *code* steht für die Regel, die in Code übersetzt werden soll. Mithilfe des zweiten formalen Parameters j kann der Name der Variable übergeben werden, die den Index des aktuellen Eingabezeichens enthält.

```
1  code(A, j)=
2      if (test(I[j], A, α_1) {  goto L_{A_1}  }
3      ...
4      if (test(I[j], A, α_n) {  goto L_{A_n}  }
5  L_{A_1} : code(A ::= α_1, j)
6      ...
7  L_{A_n} : code(A ::= α_n, j)
```

Listing 3.16: Die Codeerzeugung einer Regel mit eindeutigem Rumpf.

In den Zeilen 2 bis 4 des Listings 3.16 wird für jede Alternative im Rumpf der Regel A nacheinander mit der *test*-Methode überprüft, ob das Eingabezeichen $I[j]$ das ersten Terminalzeichen ist, das aus der Alternative α_k herleitbar ist. Falls der Test positiv ist, so wird zur ersten Codezeile, die aus der entsprechenden Alternative generiert wurde, gesprungen und direkt abgearbeitet. Die Codegenerierung aus den einzelnen Zeilen geschieht in den Zeilen 5 bis 7.

```
1  code(A, j)=
2      if (test(I[j], A, α_1) {  add(L_{A_1}, c_u, j)  }
3      ...
4      if (test(I[j], A, α_n) {  add(L_{A_n}, c_u, j)  }
5      goto L_0
6  L_{A_1} : code(A ::= α_1, j)
7      ...
8  L_{A_n} : code(A ::= α_n, j)
```

Listing 3.17: Die Codeerzeugung einer Regel mit mehrdeutigem Rumpf.

Listing 3.17 zeigt, in welchen Code eine Regel *A mit mehrdeutigem Rumpf* überführt wird. In den Zeilen 2 bis 4 wird für jede Alternative der rechten Seite überprüft, ob sich das aktuelle Eingabezeichen $I[j]$ als erstes Zeichen aus der Alternative α_k herleiten lässt. Falls dem so ist, wird für jede der möglichen Alternativen mithilfe der *add*-Funktion ein Arbeitszustand (L, n, j) in die *workingList* eingefügt. L steht dabei für das Label, das die erste Codezeile der möglichen Alternative markiert. Durch dieses Vorgehen wird erreicht, dass im Falle der Mehrdeutigkeit mit jeder möglichen Alternative weitergearbeitet werden kann. Mit dem Sprung nach L_0 in Zeile 5 wird mit dem nächsten Arbeitszustand aus *workingList* weitergearbeitet. In den Zeilen 6 bis 8 wird schließlich der Code für die einzelnen Alternativen erzeugt.

```
1 code(S, i)=
2     if (test(I[i], S, Sa) { add(L_{S_1}, c_u, i) }
3     if (test(I[i], S, a) { add(L_{S_2}, c_u, i) }
4     goto L_0
5 L_{S_1}: code(S ::= S_1 a, i)
6 L_{S_2}: code(S ::= a, i)
```

Listing 3.18: Die Codeerzeugung für die Regel S der Beispielgrammatik.

Da die Regel S der **Beispielgrammatik** mehrdeutig ist, sieht der für S generierte Code wie in Listing 3.18 gezeigt aus.

Codeerzeugung einer Alternative

Der erzeugte *Code für die Alternativen* α_k im Rumpf einer Regel variiert je nach den Vorkommen von Terminalen und Nonterminalen. Falls $alpha_k$ aus dem leeren Wort ε besteht, zeigt Listing 3.19 wie $code(A ::= \varepsilon, j)$ definiert ist.

```
1 code(A ::= ε, j)=
2     pop(c_u, j), goto L_0
```

Listing 3.19: Die Codeerzeugung einer leeren Alternative.

Im Falle einer *leeren Alternative* ist die Regel $A = \varepsilon$ abgearbeitet und das oberste Element vom Call-Stack kann durch den Aufruf von *pop* entfernt werden. Zum Abschluss kann durch einen Sprung zu L_0 mit dem nächsten Arbeitszustand aus der *workingList* weitergearbeitet werden.

Ist die *Alternative* α_k *nicht leer*, so besteht sie aus einer Folge von Terminalen und Nonterminalen $x_1 x_2 \ldots x_l$. Falls x_1 ein Terminal ist, zeigt Listing 3.20 wie $code(A ::= \alpha_k, j)$ definiert ist. Ansonsten ist x_1 das Nonterminal V_p und Listing 3.21 stellt die Definition von $code(A ::= \alpha_k, j)$ dar.

Der Code einer *Alternative, die mit einem Terminal beginnt*, wird nur erreicht, wenn das aktuelle Zeichen der Eingabe mit diesem Terminal übereinstimmt (siehe hierzu die ersten drei Zeilen der Listings 3.16 und 3.17). Daher kann in Zeile 2 des Listings 3.20 der Index des aktuellen Zeichens inkrementiert werden. Im Anschluss folgt in den Zeilen 3 bis 6 der Code für die weiteren Terminale oder Nonterminale dieser Alternative. Wurde schließlich die Alternative vollständig abgearbeitet, so kann durch Aufruf von *pop* in Zeile 7 der oberste Eintrag des Call-Stacks entfernt werden. Im Anschluss wird zurück nach L_0 gesprungen, wodurch das nächste Element der *workingList* bearbeitet werden kann.

```
1  code(A ::= α_k, j)=
2        j := j + 1
3        code(x_2 ... x_l, j, A)
4        code(x_3 ... x_l, j, A)
5        ...
6        code(x_l, j, A)
7        pop(c_u, j), goto L_0
```

Listing 3.20: Die Codeerzeugung einer Alternative mit initialem Terminal.

```
1  code(A ::= α_k, j)=
2        c_u := create(R_{V_p}, c_u, j), goto L_V
3  R_{V_p} : code(x_2 ... x_l, j, A)
4        code(x_3 ... x_l, j, A)
5        ...
6        code(x_l, j, A)
7        pop(c_u, j), goto L_0
```

Listing 3.21: Die Codeerzeugung einer Alternative mit initialem Nonterminal.

Listing 3.21 zeigt, wie der generierte Code für eine *Alternative, die mit einem Nonterminal beginnt*, aussieht. Durch den Aufruf der *create*-Methode in Zeile 2 wird ein neuer Knoten im Graphen erzeugt und durch eine Kante mit dem aktuellen Knoten c_u verbunden. Dies entspricht dem „Pushen" eines neuen Elements auf den Call-Stack. Im Anschluss wird der mit dem Label L_V markierte Code, der für die Abarbeitung des Nonterminals V zuständig ist, ausgeführt. Das Label R_{V_p} in Zeile 3 markiert die Zeile, an der weitergearbeitet werden muss, nachdem der Code von V abgearbeitet wurde. Die Zeilen 3 bis 6 entsprechen der Abarbeitung der folgenden Terminale und Nonterminale der Alternative α_k. In der letzten Zeile wird durch den Aufruf von *pop* das Top-Element des Call-Stacks entfernt und durch einen Sprung nach L_0 kann mit dem nächsten Element der *workingList* fortgefahren werden.

```
1  code(S ::= S_1 a, i)=
2        c_u := create(R_{S_1}, c_u, i), goto L_S
3  R_{S_1} : code(a, i, S)
4        pop(c_u, i), goto L_0
```

Listing 3.22: Die Codeerzeugung für die erste Alternative *Sa* der Beispielgrammatik.

Da die Alternative *aS* (Zeile 6 in Listing 3.18) der **Beispielgrammatik** mit einem Nonterminal beginnt, sieht der Code dieser Alternative, wie in Listing 3.22 gezeigt, aus.

```
1  code(S ::= a, i)=
2      i := i + 1
3      pop(c_u, i), goto L_0
```

Listing 3.23: Die Codeerzeugung für die zweite Alternative *a* der Beispielgrammatik.

Die zweite Alternative *a* (Zeile 7 in Listing 3.18) der **Beispielgrammatik** beginnt mit einem Terminal. Der Code dieser Alternative sieht, wie in Listing 3.23 gezeigt, aus.

Codeerzeugung für folgende Terminale und Nonterminale

Die *Codeerzeugung ab dem zweiten Element einer Alternative* unterscheidet sich je nachdem, ob es sich um ein Terminal (Listing 3.24) oder Nonterminal handelt (Listing 3.25).

```
1  code(aβ, j, V)=
2      if (I[j] = a) {  j := j + 1  }
3      else {  goto L_0  }
```

Listing 3.24: Die Codeerzeugung eines folgenden Terminals.

Im Falle eines *Terminals* wird in Zeile 2 des Listings 3.24 zunächst überprüft, ob das aktuelle Zeichen der Eingabe mit dem in der Regel definierten Terminal übereinstimmt. Ist dies der Fall, wird der Index des aktuellen Eingabezeichens inkrementiert. Ansonsten ist die aktuelle Herleitung falsch und muss nicht mehr fortgesetzt werden. Dies wird erreicht, indem in Zeile 3 direkt nach L_0 gesprungen wird, ohne dass der aktuelle Arbeitszustand in die *workingList* eingefügt wird.

```
1  code(W_kβ, j, V)=
2      if (test(I[j], V, W_kβ)) {
3          c_u := create(R_{W_k}, c_u, j),  goto L_W
4      } else {  goto L_0  }
5  R_{W_k}:
```

Listing 3.25: Die Codeerzeugung eines folgenden Nonterminals.

Im Falle eines *Nonterminals* W_k, wie in Listing 3.25 zu sehen, wird zunächst überprüft, ob das erste aus $W_k\beta$ herleitbare Terminal mit dem aktuellen Eingabezeichen $I[j]$ übereinstimmt. Falls dem so ist, wird, wie in Zeile 3 zu sehen, ein neuer Knoten im Graphen erzeugt und per Kante mit dem aktuellen Knoten c_u verbunden. Dadurch wird ein neuer Eintrag auf den aktuellen Call-Stack gelegt. Im Anschluss wird durch einen Sprung zu L_W

mit der Abarbeitung der Regel für das Nonterminal W begonnen. Sobald sie abgeschlossen ist, wird der Folgecode nach einem Sprung zum Label R_{W_k} in Zeile 5 ausgeführt. Falls sich das aktuelle Eingabezeichen nicht aus $W_k\beta$ herleiten lässt, ist die aktuelle Herleitung falsch und wird durch einen Sprung zu L_0 in Zeile 4 verworfen.

Bei der **Beispielgrammatik** muss noch der Code für das Terminal a (Zeile 3 in Listing 3.22) erzeugt werden. Wie der Code aussieht, ist in Listing 3.26 zu sehen.

```
1  code(a, i, S)=
2      if (I[i] = a) {  i := i + 1  }
3      else { goto L_0 }
```

Listing 3.26: Der erzeugte Code für das Terminal a der Beispielgrammatik.

Parsen eines Worts

```
1      read the input into I and set I[m]:=$, i:=0
2      create graph nodes u_1 = L_0^0, u_o =$ and an edge from u_1 to u_0
3      c_u:=u_1
4      for (0≤ j ≤ m) { U_j:=∅}
5      workingList:=∅, P:=∅
6
7      if(I[0] ∈ FIRST(S$))){ goto L_S } else { report failure }
8  L_0:if (workingList ≠ ∅){
9         remove a triple (L,u,j) from workingList
10        c_u:=u, i:=j, goto L
11     } else if ((L_0,u_0) ∈ U_m){
12        report success
13     } else{
14        report failure
15     }
16
17 L_S:if(test(I[i],S,Sa) { add(L_S1,c_u,i) }
18     if(test(I[i],S,a) { add(L_S2,c_u,i) }
19     goto L_0
20
21 L_S1:c_u := create(R_S1,c_u,i), goto L_S
22 R_S1:if(I[i] = a) { i := i + 1 }
23     else { goto L_0 }
24     pop(c_u,i), goto L_0
25
26 L_S2:i := i + 1
27     pop(c_u,i), goto L_0
```

Listing 3.27: Der Code des Recognizers der Beispielgrammatik.

Bevor die Semantik der Methoden *test*, *add*, *pop* und *create* sowie der Mengen U_k und $\mathcal{P}$ näher erläutert wird, folgt zunächst ein Beispiel, in dem mithilfe des erstellten Recognizers der Beispielgrammatik in Listing 3.27 ein Eingabewort erkannt wird.

Erkennung eines Beispielworts

Listing 3.27 zeigt den Recognizer, der aus der Beispielgrammatik in Formel (3.55) generiert wurde. Im Folgenden wird die *Erkennung des Beispielworts aaa* gezeigt. Um die Arbeitsweise besser verstehen zu können, arbeitet die *workingList* wie ein Stack nach dem *LIFO-Prinzip*. Dadurch arbeitet der Recognizer ähnlich dem rekursiven Abstieg, bei dem im Falle von Mehrdeutigkeit eine mögliche Alternative zunächst bis zum Ende abgearbeitet wird, bevor an einer anderen Alternative weitergearbeitet wird. Aus Gründen der Übersichtlichkeit werden die Mengen U_k nicht betrachtet. Sie haben den Zweck sicherzustellen, dass, falls ein Arbeitszustand (L, u, k) bereits einmal abgearbeitet wurde, er nicht erneut abgearbeitet wird.

Nach der *Initialisierung* in den Zeilen 1 bis 5 sind die Variablen folgendermaßen belegt:

$I = [a, a, a, \$]$
$i = 0$
$workingList = \varnothing$
$\mathcal{P} = \varnothing$
$c_u = L_0^0$
Graph bzw. Call-Stack = $(\$) \leftarrow (L_0^0)$

Da der erste Buchstabe des Eingabeworts durch den Aufruf der Startregel S erzeugt werden kann, beginnt die Abarbeitung mit dem Code der Startregel, die am Label L_0 beginnt (Zeile 7).

Bei der *Abarbeitung der Regel S* wird in den Zeilen 17 und 18 festgestellt, dass sich das erste Zeichen des Eingabeworts a aus beiden Alternativen der Regel herleiten lässt. Daher werden $(L_{S_1}, L_0^0, 0)$ und $(L_{S_2}, L_0^0, 0)$ zur *workingList* hinzugefügt.

Graph = $(\$) \leftarrow (L_0^0)$
$workingList = \{(L_{S_1}, L_0^0, 0), (L_{S_2}, L_0^0, 0)\}$
$\mathcal{P} = \varnothing$

Im Anschluss wird durch einen Sprung zum Label L_0 in Zeile 8 bewirkt, dass der Arbeitszustand $(L_{S_2}, L_0^0, 0)$ aus der *workingList* entnommen und mit dem aktuellen Knoten L_0^0 sowie dem aktuellen Index 0 des Eingabeworts an der Stelle L_{S_2} weitergearbeitet wird.

Da die Alternative S_2 mit dem Terminal a beginnt und bei der Abarbeitung der Regel S in Zeile 18 bereits geprüft wurde, ob das aktuelle Eingabezeichen a als erstes Terminal der Alternative S_3 herleitbar ist, kann in Zeile 26 *der aktuelle Index inkrementiert* werden. Da nun die Alternative S_2 komplett abgearbeitet wurde, kann das oberste Element des Call-Stacks entfernt werden. Beim Aufruf der Methode $pop(L_0^0, 1)$ wird zunächst $(L_0^0, 1)$ in $\mathcal{P}$ eingefügt. Im Anschluss wird das Entfernen des Top-Elements des Call-Stacks simuliert, indem der *Arbeitszustand* $(L_0, \$, 1)$in die *workingList* eingefügt wird. Der Graph bleibt dabei unverändert. L_0 ist dabei die gespeicherte Codeposition, die sich zuvor im obersten Knoten des Call-Stacks L_0^0 befunden hat. \$ ist der Vorgängerknoten im Call-Stack. Zum Schluss wird durch einen Sprung zum Label L_0 in Zeile 27 die Bearbeitung des nächsten Arbeitszustands angestoßen.

Graph = $(\$) \leftarrow (L_0^0)$
$workingList = \{(L_{S_1}, L_0^0, 0), (L_0, \$, 1)\}$
$\mathcal{P} = \{(L_0^0, 1)\}$

Der nächste zu bearbeitende Arbeitszustand ist $(L_0, \$, 1)$. \$ als oberstes Element des Call-Stacks signalisiert, dass der Call-Stack leer ist. Die an dieser Stelle beendete Herleitung ist: $S\$ \vdash a\$$. Das Label L_0 bewirkt, dass mit dem nächsten Arbeitszustand weitergearbeitet werden kann, da die aktuelle Abarbeitung abgeschlossen ist.

Graph = $(\$) \leftarrow (L_0^0)$
$workingList = \{(L_{S_1}, L_0^0, 0)\}$
$\mathcal{P} = \{(L_0^0, 1)\}$

Bei der Ausführung vom *Code bei* L_{S_1} in Zeile 21 muss das *Nonterminal* S abgearbeitet werden. Hierzu muss zunächst ein neuer Eintrag auf den Call-Stack gelegt werden. Dies geschieht durch den Methodenaufruf $create(R_{S_1}, L_0^0, 0)$. Der Aufruf bewirkt, dass im Graphen ein neuer Knoten $R_{S_1}^0$ erzeugt und durch eine Kante mit L_0^0 verbunden wird. Im Anschluss wird nach einem Sprung zum Label L_S weitergearbeitet. Da das aktuelle Eingabezeichen a aus beiden Alternativen herleitbar ist, werden die beiden Arbeitszustände $((L_{S_1}, R_{S_1}^0, 0))$ und $((L_{S_2}, R_{S_1}^0, 0))$ zur *workingList* hinzugefügt.

Graph = $(\$) \leftarrow (L_0^0) \leftarrow (R_{S_1}^0)$
$workingList = \{(L_{S_1}, R_{S_1}^0, 0), (L_{S_2}, R_{S_1}^0, 0)\}$
$\mathcal{P} = \{(L_0^0, 1)\}$

Im nächsten Schritt wird der *Code bei* L_{S_2} abgearbeitet. Zunächst wird der Zähler des aktuellen Eingabezeichens inkrementiert und im Anschluss das oberste Element des Call-Stacks durch einen Aufruf von $pop(R_{S_1}^0, 1)$ entfernt, da die Alternative a der S-Regel vollständig erkannt wurde.

Graph = $\$ \leftarrow L_0^0 \leftarrow R_{S1}^0$
workingList $= \{(L_{S_1}, R_{S_1}^0, 0), (R_{S_1}, L_0^0, 1)\}$
$\mathcal{P} = \{(L_0^0, 1), (R_{S_1}^0, 1)\}$

Da nun das S in der Alternative Sa abgearbeitet wurde, wird festgestellt, dass das aktuelle Eingabezeichen ein a ist (Zeile 22) und der Index i inkrementiert. Die Erkennung der Alternative ist nun abgeschlossen, weshalb in Zeile 24 durch einen Aufruf von $pop(L_0^0, 2)$ das oberste Element vom Call-Stack entfernt werden kann. $(L_0^0, 2)$ wird dabei in $\mathcal{P}$ eingefügt.

Graph = $\$ \leftarrow L_0^0 \leftarrow R_{S1}^0$
workingList $= \{(L_{S_1}, R_{S_1}^0, 0), (L_0, \$, 1)\}$
$\mathcal{P} = \{(L_0^0, 1), (R_{S_1}^0, 1), (L_0^0, 2)\}$

Der letzte Arbeitszustand der *workingList* kann entfernt werden, da sein Call-Stack leer ist. Der Zustand entspricht der Herleitung $S\$ \vdash Sa\$ \vdash aa\$$.

Graph = $\$ \leftarrow L_0^0 \leftarrow R_{S1}^0$
workingList $= \{(L_{S_1}, R_{S_1}^0, 0)\}$
$\mathcal{P} = \{(L_0^0, 1), (R_{S_1}^0, 1), (L_0^0, 2)\}$

Bei der Abarbeitung des aktuellen Arbeitszustands wird in Zeile 21 zunächst die Methode $create(R_{S_1}, R_{S_1}^0, 0)$ aufgerufen. Da der Knoten $R_{S_1}^0$ bereits existiert, wird kein neuer Knoten angelegt. Allerdings wird die noch nicht existente *Schlinge* an $R_{S_1}^0$ erzeugt, da er sein eigener Vorgänger im Call-Stack ist. Darüber hinaus erkennt die *create*-Methode anhand der Menge $\mathcal{P}$, dass der Knoten R_{S_1} bereits einmal von einem Call-Stack entfernt worden ist. Das dazugehörige Tupel in $\mathcal{P}$ ist $(R_{S_1}^0, 1)$. Es ist so zu interpretieren, dass in der Vergangenheit eine Herleitung abgebrochen wurde, die den aktuellen Knoten R_{S_1} auf dem Call-Stack und das erste Zeichen der Eingabe bereits erkannt hatte. Durch das Erzeugen der Schlinge im aktuellen Arbeitszustand, gibt es für die damals abgebrochene Herleitung eine Fortsetzung, die nun als $(R_{S_1}, R_{S_1}^0, 1)$ in die *workingList* eingefügt wird. Der geschilderte Sachverhalt kann auch mit Herleitungen veranschaulicht werden
($\cdot$ signalisiert die Position der Abarbeitung):
Die zuvor abgebrochene Herleitung war : $\cdot S\$ \vdash \cdot Sa\$ \vdash a \cdot a\$ \vdash aa \cdot \$$
Die aktuelle Herleitung ist: $\cdot S\$ \vdash \cdot Sa\$ \vdash \cdot SSa\$$
Kann kombiniert werden zu: $\cdot S\$ \vdash \cdot Sa\$ \vdash \cdot SSa\$ \vdash aa \cdot a\$$

Nachdem der *create*-Aufruf beendet ist, wird das S in der Alternative Sa abgearbeitet. Da das aktuelle Eingabezeichen aus beiden Alternativen der S-Regel herleitbar ist, müsste

die *workingList* um $(L_{S_1}, R^0_{S_1}, 0)$ und $(L_{S_2}, R^0_{S_1}, 0)$ ergänzt werden. Da aber identische Arbeitszustände bereits zuvor in die *workingList* eingefügt wurden, müssen sie nicht erneut abgearbeitet werden. Dadurch werden *endlose Herleitungen bei Linksrekursion vermieden*.

Graph bzw. Call-Stack =
workingList $= \{(R_{S_1}, R^0_{S_1}, 1)\}$
$\mathcal{P} = \{(L^0_0, 1), (R^0_{S_1}, 1), (L^0_0, 2)\}$

Bei dem aktuellen Arbeitszustand wurde das S in der Alternative Sa bereits erkannt. Daher folgt nun in Zeile 22 der Test, ob das aktuelle Eingabezeichen ein a ist. Da dies der Fall ist, kann i inkrementiert werden. Dadurch wird das dritte a zum aktuellen Eingabezeichen. Die Alternative Sa wurde nun vollständig erkannt und das oberste Element $R^0_{S_1}$ des Call-Stack kann entfernt werden. Da $R^0_{S_1}$ die beiden Vorgänger $R^0_{S_1}$ und L^0_0 hat, werden die beiden Arbeitszustände $(R_{S_1}, R^0_{S_1}, 2)$ und $(R_{S_1}, L^0_0, 2)$ in die *workingList* eingefügt.

Graph bzw. Call-Stack =
workingList $= \{(R_{S_1}, R^0_{S_1}, 2), (R_{S_1}, L^0_0, 2)\}$
$\mathcal{P} = \{(L^0_0, 1), (R^0_{S_1}, 1), (L^0_0, 2), (R^0_{S_1}, 2)\}$

Mit $(R_{S_1}, L^0_0, 2)$ wird nun in Zeile 22 überprüft, ob das aktuelle Eingabezeichen ein a ist. Da dem so ist, wird i inkrementiert, wodurch die Alternative Sa vollständig abgearbeitet wurde. Mit dem Aufruf von $pop(L^0_0, 3)$ wird das oberste Element des Call-Stacks entfernt und $(L_0, \$, 3)$in die *workingList* eingefügt. Darüber hinaus wird $\mathcal{P}$ um $(L^0_0, 3)$ ergänzt.

Graph bzw. Call-Stack =
workingList $= \{(R_{S_1}, R^0_{S_1}, 2), (L_0, \$, 3)\}$
$\mathcal{P} = \{(L^0_0, 1), (R^0_{S_1}, 1), (L^0_0, 2), (R^0_{S_1}, 2), (L^0_0, 3)\}$

Da der Call-Stack von $(L_0, \$, 3)$ leer ist, wird es aus der *workingList* entfernt und mit dem nächsten Arbeitszustand fortgefahren. Die beendete Herleitung hat die Form:
$S\$ \vdash Sa\$ \vdash SSa\$ \vdash aSa\$ \vdash aaa\$$

Graph bzw. Call-Stack =
workingList $= \{(R_{S_1}, R^0_{S_1}, 2)\}$
$\mathcal{P} = \{(L^0_0, 1), (R^0_{S_1}, 1), (L^0_0, 2), (R^0_{S_1}, 2), (L^0_0, 3)\}$

Da der Test bei Label R_{S_1} erfolgreich ist, wird der Index des aktuellen Eingabezeichens erhöht. Da nun die Alternative Sa abgearbeitet wurde, muss das oberste Element des Call-Stacks entfernt werden. Beim Aufruf von $pop(R^0_{S_1}, 3)$ wird $(R^0_{S_1}, 3)$ in $\mathcal{P}$ eingefügt und die *workingList* um die beiden Vorgänger $(R_{S_1}, R^0_{S_1}, 3)$ und $(R_{S_1}, L^0_0, 3)$ von $R^0_{S_1}$ ergänzt.

Graph bzw. Call-Stack =
workingList $= \{(R_{S_1}, R_{S_1}^0, 3), (R_{S_1}, L_0^0, 3)\}$
$\mathcal{P} = \{(L_0^0, 1), (R_{S_1}^0, 1), (L_0^0, 2), (R_{S_1}^0, 2), (L_0^0, 3), (R_{S_1}^0, 3)\}$

Da das dritte Eingabezeichen \$ kein *a* ist, wie es von der ersten Alternative *Sa* gefordert wird, wird die aktuelle Herleitung durch den Sprungbefehl nach L_0 in Zeile 23 verworfen.

Graph bzw. Call-Stack =
workingList $= \{(R_{S_1}, R_{S_1}^0, 3)\}$
$\mathcal{P} = \{(L_0^0, 1), (R_{S_1}^0, 1), (L_0^0, 2), (R_{S_1}^0, 2), (L_0^0, 3), (R_{S_1}^0, 3)\}$

$(R_{S_1}, R_{S_1}^0, 3)$ wird aus demselben Grund wie der zuvor bearbeitete Arbeitszustand verworfen, indem nach L_0 gesprungen wird.

Graph bzw. Call-Stack =
workingList $= \varnothing$
$\mathcal{P} = \{(L_0^0, 1), (R_{S_1}^0, 1), (L_0^0, 2), (R_{S_1}^0, 2), (L_0^0, 3), (R_{S_1}^0, 3)\}$

Da die *workingList nun leer* ist, wird in Zeile 11 $(L_0, \$) \in U_3$ geprüft. Diese Bedingung bedeutet, dass es einen Arbeitszustand $(L_0, \$, 3)$ gegeben haben muss. Dieser Zustand entspricht einer Herleitung, bei der jedes Zeichen des Eingabeworts erkannt wurde und der Call-Stack leer ist. Da dies der Fall war, wird in Zeile 12 die erfolgreiche Erkennung der Eingabe festgestellt.

Erläuterung der Semantik der verwendeten Mengen und Funktionen

Bei der im vorangegangenen Abschnitt beschriebenen Erkennung des Worts *aaa* wurden die *Mengen* U_k nicht betrachtet. Sie dienen der Erkennung, ob ein Arbeitszustand (L, u, i) bereits in der *workingList* eingefügt worden ist. Um bei dem implementierten Recognizer Speicherplatz zu sparen, gibt es für jedes Zeichen der Eingabe eine eigene Menge. So gibt es bei dem um \$ erweiterten Eingabewort *aaa*\$ die Mengen U_0 bis U_3. Da der aktuelle Index durch die Menge, in der sich der Arbeitszustand befindet, ausgedrückt wird, genügt es zum Speichern von (L, u, i) nur noch (L, u) in die Menge U_i einzufügen. Da die Indizes nur monoton wachsen können, kann die Menge U_k gelöscht werden, sobald es in der *workingList* keinen Arbeitszustand mit einem Index $i \leq k$ mehr gibt.

Ein *Arbeitszustand* (L, u, i) besteht aus dem Label L, das die Stelle im Code markiert, wo weitergearbeitet werden muss. u stellt den obersten Knoten des aktuellen Call-Stacks dar und i markiert den Index des Zeichens im Eingabewort, das aktuell erkannt werden

muss. Die Knoten des Call-Stacks haben die Form L^i. L ist wieder ein Label und i der Index eines Eingabezeichens. $R^0_{S_1}$ bedeutet beispielsweise, dass am Label R_{S_1} das erste Zeichen erkannt werden muss. Befindet sich (L, u, i) am Anfang einer *direkten oder indirekten Linksrekursion*, wird irgendwann der identische Arbeitszustand erneut erreicht. Da im Graphen jeder Knoten eindeutig sein muss und die Knoten nur aus dem Label und dem Index des Eingabeworts bestehen, entsteht ein Kreis. Damit eine Endlositeration in dem Kreis vermieden wird, werden die Mengen U_k benötigt.

Die *Menge* $\mathcal{P}$ speichert die Tupel (u, i), falls im Arbeitszustand (L, u, i) das oberste Element u vom Call-Stack entfernt wird. Jedes Element von $\mathcal{P}$ repräsentiert eine Aussage, dass es eine Herleitung der Eingabezeichen mit den Indizes $0 \leq i - 1$ gibt, die mit dem Knoten u als Top-Element des Stacks endet. Welchen Sinn dieses Wissen hat, wird in der Beschreibung der *create*-Methode näher erläutert.

```
1  create(L, u, i)=
2      if (L^i ∉ V) { v=createVertex(L^i) }
3      else { v = getVertex(L^i) }
4      if (v → u ∉ E) {
5          createEdge(v → u)
6          for all ((v, k) ∈ P) { add(L, u, k) }
7      }
8      return v
```

Listing 3.28: Die *create*-Methode. Quelle: [SJ10]

Die Arbeitsweise der *create-Methode* ist in Listing 3.28 dargestellt und entspricht dem *push* des Call-Stacks. Sie wird nur aufgerufen, wenn die Abarbeitung eines Nonterminals im Rumpf einer Regel begonnen wird. Zunächst wird in den Zeilen 2 und 3 der Knoten L^i erzeugt, sofern er zuvor noch nicht existierte. Falls es noch keine Kante $v \rightarrow u$ gibt, so wird diese in Zeile 5 erzeugt. In der nächsten Zeile wird überprüft, ob der aktuelle Knoten v zuvor ein Top-Element eines Call-Stacks gewesen war. Dies geschieht, indem nach einem Tupel (v, k) in $\mathcal{P}$ gesucht wird. Wird ein solches Element gefunden, so wird es mit dem Aufruf von *add*(L, u, k) zur *workingList* hinzugefügt.

Wird beispielsweise *create*$(R_{S_1}, u, 0)$ bei dem Beginn der Abarbeitung der Regel $S = aS$ aufgerufen, so entspricht dies dem Punkt $\cdot Sa\alpha\$$ in der Herleitung. α ist dabei eine beliebig lange Folge von a und S. Zunächst wird der Knoten $v = R^0_{S_1}$ erzeugt, falls er zuvor noch nicht existierte. Da es noch keine Kante $R^0_{S_1} \rightarrow u$ gibt, wird sie in Zeile 5 erzeugt. Bei der Überprüfung in der nächsten Zeile wird festgestellt, dass sich $(R^0_{S_1}, 1)$ in $\mathcal{P}$ befindet. Dieses Tupel ist durch den Aufruf der *pop*-Methode am Ende der Abarbeitung einer Regel in die Menge $\mathcal{P}$ gelangt. Der Index 1 signalisiert, dass das erste Eingabezeichen er-

kannt worden ist. Anhand des Knotens $R^0_{S_1}$ im Tupel ist ersichtlich, dass ein Rücksprung zum Label R_{S_1} erfolgt ist und das nächste abzuarbeitende Zeichen ein a sein muss, da dieses Label in der abzuarbeitenden Regel die Position $S = S\, R_{S_1}\, a$ hat. In der Herleitung entspräche dies dem Zustand: $a \cdot a\beta\$$. β steht dabei für eine beliebig lange Folge aus a und S.

Das *Rücksprunglabel* R_{S_1} des Tupels ist identisch mit dem Label, das durch den *Aufruf create*$(R_{S_1}, u, 0)$ zum Rücksprung vorgesehen ist. Daher kann davon ausgegangen werden, dass das $\cdot Sa$ der aktuellen Herleitung $\cdot Sa\alpha\$$ identisch ist mit $a \cdot a$ aus der „gepopten" Herleitung $a \cdot a\beta\$$ und daraus $a \cdot a\alpha\$$ gefolgert werden kann. Aus diesem Grund wird in Zeile 6 $(R_{S_1}, u, 1)$ in die *workingList* eingefügt. Durch diesen Schritt werden Wörter erkannt, die durch Linksrekursion erzeugt werden, obwohl die direkte Herleitung unter Zuhilfenahme der Mengen U_k abgebrochen wurde, um das Entstehen einer Endlosherleitung zu verhindern.

```
1  add(L, u, i)=
2      if ((L, u) ∉ U_i) {
3          U_i.add((L, u)))
4          workingList.add((L, u, i)))
5      }
```

Listing 3.29: Die *add*-Methode. Quelle: [SJ10]

Listing 3.29 zeigt die Funktionsweise der *add-Methode*. Da der Arbeitszustand (L, u, i) nur dann in *workingList* eingefügt wird, falls er zuvor noch nicht enthalten war. Dies wird mithilfe der für diesen Index verantwortlichen U_i-Menge in Zeile 2 überprüft. In den folgenden beiden Zeilen wird der neue Arbeitszustand in *workingList* und in U_i eingefügt.

```
1  pop(u, i)=
2      if (u ≠ $) {
3          P.add((u, j)))
4          for all (v ← u) {
5              add(L_u, v, i))
6          }
7      }
```

Listing 3.30: Die *pop*-Methode. Quelle: [SJ10]

Die Funktionsweise der *pop-Methode* ist in Listing 3.30 dargestellt. In Zeile 2 wird überprüft, ob das aktuelle Top-Element u des Call-Stacks das unterste Element $\$$ ist, das keinen Vorgänger hat. Falls dem nicht so ist, wird in Zeile 3 (u, j) zu $\mathcal{P}$ hinzugefügt.

Die Zeilen 4 bis 6 dienen dazu, dass für jedes Vorgängerelement v von u ein Arbeitszustand (L_u, v, i) in die *workingList* eingefügt wird. L_u steht dabei für das Label des Knotens $u = L_u^j$.

```
1  test(a, V, α)=
2      return (a ∈ FIRST(α)) || (ε ∈ FIRST(α) && a ∈ FOLLOW(V))
```

Listing 3.31: Die *test*-Methode. Quelle: [SJ10]

Wie die *test-Methode* funktioniert, zeigt Listing 3.31. Die formalen Parameter V und α stellen eine Regel $V = \alpha$ dar. α steht für eine beliebig lange Sequenz aus Terminalen und Nonterminalen. In Zeile 2 wird überprüft, ob sich der Buchstabe a als das erste Terminal aus α herleiten lässt. Falls dies nicht der Fall ist, wird geprüft, ob aus α das leere Wort herleitbar und das erste Terminal hinter V mit a identisch ist. Die formalen Definitionen von *FIRST* und *FOLLOW* stehen im Abschnitt „LL(k) Parsing“ 3.3.1.

Aufwandsbewertung

Ist die zu erkennende Sprache durch eine *LL(1)-Grammatik* beschrieben, so kann an jedem Punkt der Herleitung anhand des nächsten Eingabezeichens deterministisch bestimmt werden, welche Regel bzw. Alternative im Rumpf einer Regel angewendet werden muss. Daher verzweigt sich der Call-Stack nicht und der Recognizer arbeitet mit *linearem Aufwand*. Die Autoren behaupten, dass der Aufwand im *schlechtesten Fall* $\mathcal{O}(m^3)$ beträgt.

3.4 Bottom-Up-Algorithmen

Die Bottom-Up-Algorithmen probieren aus dem Eingabewort das Startsymbol der Grammatik herzuleiten. Sie erreichen dies, indem ein in der Eingabe erkannter Regelrumpf durch den jeweiligen Regelkopf ersetzt wird. Zunächst wird in Abschnitt 3.4.1 der klassische LR-Algorithmus vorgestellt, bevor in Abschnitt 3.4.2 der GLR-Algorithmus erläutert wird.

3.4.1 LR(k) Parsing

Beim LR(k) Parsing wird das Eingabewort von links nach rechts verarbeitet und dabei eine umgekehrte Rechtsableitung erzeugt. Da bei diesem Vorgehen die einzigen Einschränkungen sind, dass die Grammatik eindeutig sein und das Vorkommen eines Regelrumpfs

anhand von k Zeichen erkannt werden muss, können mit diesem Ansatz mehr Grammatiken verarbeitet werden als mit LL(k). Dieser Abschnitt beruht auf [SLA08], wo geschrieben steht, dass der Zeitaufwand von LR(k) linear ist. Zur Veranschaulichung wird das LR(1) Parsing mithilfe der Grammatik G_{bindig} veranschaulicht.

$G_{bindig} = \left(\{N, N_{suff}\}, \{0,1\}, R, N\right)$ mit
$R =$

$$\begin{aligned} \{0 &\rightarrow N \\ 1\ N_{suff} &\rightarrow N \\ \varepsilon &\rightarrow N_{suff} \\ 0\ N_{suff} &\rightarrow N_{suff} \\ 1\ N_{suff} &\rightarrow N_{suff}\} \end{aligned}$$

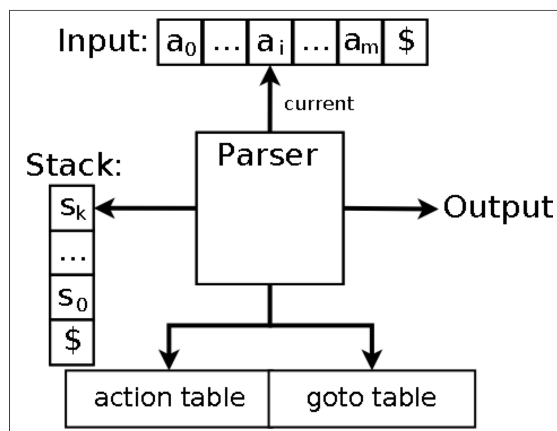


Abbildung 3.3: Schematische Darstellung eines LR-Parsers. Quelle: [SLA08]

Abbildung 3.3 zeigt die schematische Darstellung eines LR-Parsers. Als Eingabe erhält er das um die Endmarke $ erweiterte Eingabewort. Auf dem Stack merkt sich der Parser den aktuellen Zustand. Die Parse-Tabelle besteht aus einer ACTION- und einer GOTO-Tabelle. Erstere definiert, welche Aktion auszuführen ist und letztere, welches der Folgezustand ist. Als Beispiel wurde nach dem in [SLA08] beschriebenen Verfahren aus der Grammatik G_{bindig} die in Tabelle 3.4 gezeigte Parse-Tabelle generiert. Da ihre Größe sehr schnell zunimmt, ist in [SLA08] das sogenannte LALR Parsing beschrieben, bei dem versucht wird, verschiedene Zustände miteinander zu verschmelzen. Dieses Verfahren wird in dieser Arbeit nicht betrachtet.

Zustand	ACTION			GOTO	
	0	1	$	N	N_{suff}
0	*shift* 2	*shift* 3	*error*	1	*error*
1	*error*	*error*	*accept*	*error*	*error*
2	*error*	*error*	*reduce* $0 \to N$	*error*	*error*
3	*shift* 5	*shift* 6	*reduce* $\varepsilon \to N_{suff}$	*error*	4
4	*error*	*error*	*reduce* $1\ N_{suff} \to N$	*error*	*error*
5	*shift* 5	*shift* 6	*reduce* $\varepsilon \to N_{suff}$	*error*	7
6	*shift* 5	*shift* 6	*reduce* $\varepsilon \to N_{suff}$	*error*	8
7	*error*	*error*	*reduce* $0\ N_{suff} \to N_{suff}$	*error*	*error*
8	*error*	*error*	*reduce* $1\ N_{suff} \to N_{suff}$	*error*	*error*

Tabelle 3.4: Die aus der Grammatik G_{bindig} erzeugte LR-Parse-Tabelle.

Die Funktionsweise sowie die Bedeutung der in der Parse-Tabelle vorhandenen Einträge werden anhand des Parse-Vorgangs für das Eingabewort 101$ erläutert. Für jeden Arbeitsschritt wird angegeben, wie der aktuelle Stack aussieht, welche Eingabezeichen noch erkannt werden müssen und welche Aktion ausgeführt wurde. Zum besseren Verständnis werden zusätzlich zu den Zuständen auf dem Stack repräsentierte Symbole in Klammern angegeben.

Zeile	Stack	Eingabe	Aktion
(1)	0($)	101$	*shift* 3
(2)	0($) 3(1)	01$	*shift* 5
(3)	0($) 3(1) 5(0)	1$	*shift* 6
(4)	0($) 3(1) 5(0) 6(1)	$	*reduce* $\varepsilon \to N_{suff}$
(5)	0($) 3(1) 5(0) 6(1) 8($N_{suff}$)	$	*reduce* $1\ N_{suff} \to N_{suff}$
(6)	0($) 3(1) 5(0) 7($N_{suff}$)	$	*reduce* $0\ N_{suff} \to N_{suff}$
(7)	0($) 3(1) 4($N_{suff}$)	$	*reduce* $1\ N_{suff} \to N$
(8)	0($) 1($N$)	$	*accept*

Tabelle 3.5: Der LR-Parse-Vorgang für die Eingabe 101 mithilfe der Parse-Tabelle 3.4.

In Zeile (1) der Tabelle 3.5 ist der initiale Zustand angegeben. In der ACTION-Tabelle ist im Zustand 0 bei Eingabe 1 die Aktion *shift* 3 eingetragen. Dies bedeutet, dass der erste Buchstabe der Eingabe erkannt wurde und der Zustand 3 auf den Stack gelegt wird. In den Zeilen (2) und (3) wird analog verfahren.

In Zeile (4) ist in der ACTION-Tabelle zum ersten Mal die Anweisung *reduce* $\varepsilon \to N_{suff}$ vorhanden. Dadurch werden so viele Zustände vom Stack entfernt, wie Terminale und syntaktische Variablen im Rumpf der angegebenen Regel vorhanden sind. Da es sich um eine ε-Regel handelt, bleibt der Stack zunächst unverändert und das Top-Element ist

Zustand 6. Im Anschluss wird in der GOTO-Tabelle nachgesehen, welcher Folgezustand unter dem Top-Element des Stacks und dem Regelkopf registriert ist. Im aktuellen Beispiel ist unter dem Zustand 6 und der syntaktischen Variable N_{suff} der Folgezustand 8 eingetragen, welcher auf den Stack gelegt wird. Die Reduktionen in den Zeilen (5) bis (7) verlaufen analog.

Bei dem in Zeile (8) vorliegenden Zustand ist in der ACTION-Tabelle die Aktion *accept* eingetragen. Dies bedeutet, das das Eingabewort vollständig erkannt wurde. Stünde dort *error*, wäre die Erkennung fehlgeschlagen.

Würden in einer Zelle der ACTION-Tabelle zwei verschiedene *reduce*-Aktionen eingetragen sein, so wird dies ein Reduce-Reduce-Konflikt genannt. Der Fall einer *shift*- und einer *reduce*-Aktion in einer Zelle heißt Shift-Reduce-Konflikt. Sollte einer dieser Konflikte auftreten, so handelt es sich um eine mehrdeutige Grammatik oder der Parser kann nicht anhand der folgenden k Zeichen entscheiden, nach welcher Regel reduziert werden muss.

3.4.2 GLR Parsing

Der LR(k)-Algorithmus, der im vorangegangenen Abschnitt 3.4.1 beschrieben wurde, hat den Nachteil, dass im Falle von mehrdeutigen Grammatiken Shift-Reduce- bzw. Reduce-Reduce-Konflikte auftreten können. Damit ein Parser trotz Konflikten weiterarbeiten kann, *verallgemeinerte* Masaru Tomita das *LR(k)-Verfahren*. Der so genannte „Generalized LR“-Algorithmus wurde 1986 in [Tom86] veröffentlicht.

Das Problem der Mehrdeutigkeit wurde von Tomita gelöst, indem der Algorithmus jede mögliche Herleitung bearbeitet. Um zu verhindern, dass für jede Herleitung der gesamte Stack kopiert werden müsste, wird ein Graph verwendet, in dem jeder maximale Pfad den Stack einer möglichen Herleitung darstellt.

Im ersten Abschnitt „Schematische Darstellung“ wird ein Überblick über einen GLR Parser gegeben. Dabei werden die vom Parser verwendeten Datenstrukturen näher erläutert, die zum Parsen eines Worts benötigt werden. Dies wird im folgenden Abschnitt „Parsen eines Beispielworts“ veranschaulicht. Nachdem der Ablauf des Parsens verdeutlicht wurde, wird im Abschnitt „Arbeitsweise des Parsers“ anhand von Pseudocode der GLR-Algorithmus beschrieben. Zum Schluss folgt die „Aufwandsbewertung“.

Schematische Darstellung

Die schematische Darstellung eines GLR Parsers ist in Abbildung 3.4 zu sehen. Als Eingabe erhält er das um $ erweiterte zu parsende Wort und als Ausgabe wird der erzeugte Parse-Forest zurückgeliefert.

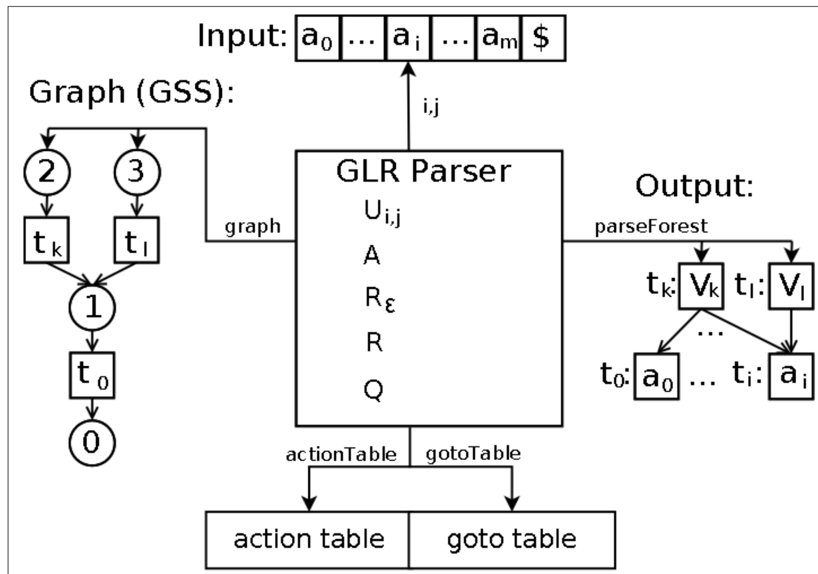


Abbildung 3.4: Schematische Darstellung eines GLR Parsers

Um festzustellen, welches das *aktuelle Eingabezeichen* ist, besitzt der GLR Parser den Index i. Der Index j markiert das j-te ε hinter dem Eingabezeichen a_i. Wird beispielsweise das Wort aa mit der in Formel (3.57) angegeben Grammatik geparst, so wird das Wort $a\varepsilon\varepsilon a$ erkannt. Dabei hat das erste a die Indizes $i = 1$ $j = 0$, das erste ε die Indizes $i = 1$ $j = 1$, das zweite ε die Indizes $i = 1$ $j = 2$ und das abschließende a hat die Indizes $i = 2$ $j = 0$.

$$
\begin{aligned}
S &:= \text{"}a\text{"}EE\text{"}a\text{"}; \\
E &:= \varepsilon;
\end{aligned}
\tag{3.57}
$$

Die vom GLR Parser *verwendete Parse-Tabelle* wird wie beim LR(k)-Algorithmus erzeugt. Sie besteht aus der *Aktionstabelle* und der *Sprungtabelle*. Erstere definiert welche Aktionen in einem Zustand bei einem bestimmten Eingabezeichen ausgeführt werden müssen. Die Sprungtabelle hingegen legt fest, welches der Nachfolgezustand nach einem Reduce ist.

Im Falle von Konflikten kann es vorkommen, dass in der Aktionstabelle mehr als eine Aktion definiert ist. Damit alle möglichen Herleitungen des Eingabeworts abgearbeitet werden können, wird ein *graph-structured Stack (GSS)* verwendet. Dabei handelt es sich

um einen gerichteten azyklischen Graphen mit einem Wurzelknoten, der von allen Knoten des Graphen aus erreichbar ist. Im Graphen der Abbildung 3.4 ist dies der Knoten mit der Nummer 0. Darüber hinaus sind die Knoten jedes Pfads in einem GSS abwechselnd vom Typ *StateVertex* und *SymbolVertex*. Erstere werden in der Abbildung durch einen Kreis symbolisiert. Die Zahl im Kreis steht für einen Zustand. Die Symbolknoten vom Typ SymbolVertex werden durch Quadrate dargestellt. Sie besitzen eine Referenz auf einen Knoten im Parse-Forest.

Die Verwendung eines GSS hat den Vorteil der *Speicherplatzersparnis*. Dies wird zum einen dadurch erreicht, dass in allen Herleitungen so lange dieselben Knoten verwendet werden, bis durch einen Konflikt in der Aktionstabelle eine Verzweigung notwendig wird. Zum anderen können sich verschiedene Herleitungen auch wieder vereinigen, sobald sie in denselben Zustand gelangen.

Während des Parse-Vorgangs wird ein *Parse-Forest* aufgebaut. Ein einziger Parse-Tree ist bei GLR nicht nutzbar, da bei mehrdeutigen Grammatiken alle möglichen Parse-Trees aufgebaut werden. Die Blätter bestehen dabei aus Knoten, die für die Eingabebuchstaben stehen, oder aus Variablen einer ε-Regel. Die inneren Knoten entstehen bei Reduktionen und stehen für Variablen. Jeder Kindknoten entspricht dabei einem Terminal bzw. Nonterminal. Ihre Reihenfolge entspricht der im Rumpf der Regel, nach der reduziert wird.

Um Speicherplatz zu sparen, werden im Parse-Forest nur dann innere Knoten erzeugt werden müssen, wenn es noch keinen äquivalenten inneren Knoten gibt[8]. Gibt es im Parse-Forest zwei Teilbäume mit denselben Blättern und den Wurzeln t_1 und t_2, die mit den gleichen Nonterminal versehen sind, so handelt es sich um eine Mehrdeutigkeit. In einem solchen Fall werden t_1 und t_2 zu einem einzigen Knoten t vereinigt. Durch dieses so genannte *„Local Ambiguity Packing“* wird erreicht, dass für t_1 und t_2 nicht gesonderte Elternknoten erzeugt werden müssen, sondern nur ein Knoten, der t als Kind hat. Um diese Vorteile realisieren zu können, werden in den Symbolknoten des GSS nur die Referenzen auf die Knoten des Parse-Forest gespeichert.

Durch die Verwendung des GSS und eines Parse-Forest ist es möglich, alle möglichen Herleitungen parallel zu erzeugen. Dies hat den Vorteil, dass kein Backtracking durchgeführt werden muss. Der resultierende Nachteil besteht allerdings darin, dass im Parse-Forest auch für Herleitungen Knoten erzeugt werden, die beispielsweise nur ein Teilwort erkennen können und kein Bestandteil einer vollständigen Herleitung sind.

Ähnlich einem LR Parser werden bei GLR *zunächst alle möglichen Reduktionen* durchgeführt, bevor das nächste Eingabezeichen auf den Stack „geshifted“ wird. Um dies zu ge-

[8]Zwei innere Knoten im Parse-Forest werden im Kontext des GLR Parsens als äquivalent angesehen, wenn sie mit dem gleichen Grammatiksymbol versehen sind und die gleichen direkten Kinder haben.

währleisten, besitzt jeder GLR Parser die Mengen $U_{i,j}$, A, R_ε, R und Q. Ihre Bedeutungen sind:

$U_{i,j}$ (Zustandsknotenmenge) enthält alle *Zustandsknoten*, die bei der Abarbeitung des Eingabezeichens a_i bzw. dem j-ten ε hinter a_i erzeugt wurden. Mithilfe dieser Menge wird sichergestellt, dass nur Zustandsknoten zur Vereinigung von zwei Herleitungen genutzt werden können, die bei der Abarbeitung des gleichen Eingabezeichens entstanden sind.

A (Aktionsknotenmenge) besteht aus den *Zustandsknoten*, für die in der Aktionstabelle nachgesehen werden muss, welches die nächsten Aktionen sind. Sollte es sich um eine ε-Reduktion handeln, so wird ein entsprechendes Element in die Menge R_ε eingefügt. Für alle anderen Reduktionen werden Elemente in R und bei Shift-Aktionen in die Menge Q eingefügt.

R_ε (εReduktionsmenge) enthält *Tupel der Form* $\langle v, p \rangle$. Das ist so zu verstehen, dass in der Aktionstabelle bei dem im Zustandsknoten v abgelegten Zustand und bei dem aktuellen Eingabezeichen die Aktion „reduce p“ definiert ist. p ist dabei eine Regel deren rechte Seite aus einem ε besteht.

R (Reduktionsmenge) besteht aus den *Tripeln* $\langle v, x, p \rangle$. v und p haben dabei die gleiche Bedeutung wie bei den Tupeln in der Menge R_ε. p ist jedoch eine Regel mit einer nicht-leeren rechten Seite. Da ein Zustandsknoten über mehrere ausgehende Kanten mit verschiedenen Knoten verbunden sein kann, wird der Symbolknoten x benötigt. Er definiert, dass die Reduktion auf allen Pfaden, die mit $x \leftarrow v$ beginnen, ausgeführt wird.

Q (Shiftmenge) enthält *Tupel der Form* $\langle v, s \rangle$. Dies bedeutet, dass in der Aktionstabelle beim durch den Zustandsknoten v repräsentierten Zustand und beim aktuellen Eingabezeichen die Aktion „shift s“ eingetragen ist. s ist dabei der Folgezustand.

Zu Beginn wird der Graph mit einem Startknoten, der den Zustand 0 repräsentiert, initialisiert. Dieser Knoten wird in die Menge $U_{0,0}$ eingefügt. Danach wird nacheinander jeder Buchstabe des Eingabeworts verarbeitet.

Bei der *Abarbeitung des Buchstabens a_i* wird zunächst $U_{i,j}$ in die Aktionsknotenmenge A kopiert. Im Anschluss wird in der Sprungtabelle nachgesehen, welche Aktionen für die Zustände der in A enthaltenen Knoten vorgesehen sind. Anhand der vorgesehenen Aktion wird entschieden, in welche Menge das aktuelle Element eingefügt wird. Im Falle einer Reduktion nach einer Regel, deren rechte Seite nicht leer ist, wird ein Tripel in die Reduktionsmenge R eingefügt. Bei einer Reduktion nach einer ε-Regel wird die εReduktionsmenge R_ε um ein Tupel ergänzt und bei einem Shift wird ein entsprechendes Element zur Shiftmenge Q hinzugefügt. Sind alle Aktionen in der Aktionstabelle behandelt, so wird der aktuelle Knoten aus A entfernt.

Ist die *Aktionsmenge A leer*, so werden zunächst alle Reduktionen ausgeführt, indem alle Elemente der Reduktionsmenge R verarbeitet werden. Im Anschluss folgen die Tupel der εReduktionsmenge R_ε. Die bei der Reduktion neu erzeugten Zustandsknoten werden in $U_{i,j}$ und die Aktionsmenge A eingefügt. Dadurch wird erreicht, dass die in der Aktionstabelle vorgesehenen Folgeaktionen für die neu erzeugten Zustandsknoten ebenfalls ausgeführt werden.

Sind die Mengen *A, R und R_ε leer*, so wurden alle möglichen Reduktionen ausgeführt. Daher kann $U_{i,j}$ gelöscht und eine neue Menge $U_{i+1,j}$ angelegt werden. Im Anschluss werden die in der Shiftmenge Q gespeicherten Shift-Operationen ausgeführt. Die dabei entstehenden Zustandsknoten werden in $U_{i+1,j}$ eingefügt. Wurden alle in Q enthaltenen Shift-Aktionen ausgeführt sein, so kann der nächste Buchstabe a_{i+1} verarbeitet werden. Sind alle Buchstaben verarbeitet, so werden die Wurzelknoten aller Bäume im Parse-Forest, die durch „local ambiguity packing“ in einem einzigen Knoten geschachtelt werden, als Ergebnis des Parsens zurückgegeben.

Parsen eines Beispielworts

Um die zuvor schematisch dargestellte Arbeitsweise eines GLR Parsers zu veranschaulichen, wird in diesem Abschnitt anhand der aus [Tom86] stammenden *Beispielgrammatik* in Formel (3.58) das Eingabewort *nvanpanpan* geparst. Jede angegebene Regel wurde mit einer führenden Nummer versehen. Diese Zahl dient der Schreibabkürzung bei Reduktionsanweisungen in der Parse-Tabelle.

$$\begin{aligned} (1)\ S &:= N\ V; \\ (2)\ S &:= S\ P; \\ (3)\ N &:= "n"; \\ (4)\ N &:= "a"\ "n"; \\ (5)\ N &:= N\ P; \\ (6)\ P &:= "p"\ N; \\ (7)\ V &:= "v"\ N; \end{aligned} \tag{3.58}$$

Zunächst muss die Grammatik in eine *Parse-Tabelle* überführt werden. Dies geschieht nach dem im Abschnitt 3.4.1 beschriebenen Algorithmus. Die resultierende *Parse-Tabelle* ist in Tabelle 3.6 zu sehen. Der Eintrag „sh 3“ bedeutet, dass das nächste Eingabzeichen auf den Stack „geshifted“ wird und der Folgezustand 3 ist. „re 3“ bedeutet, dass nach der

Regel (3) der Beispielgrammatik in Formel (3.58) reduziert wird. „acc“ signalisiert, dass das Eingabewort vollständig und ohne Fehler erkannt wurde.

Zustand	a	n	v	p	$
0	sh 3	sh 4			
1				sh 6	acc
2			sh 7	sh 6	
3		sh 10			
4			re 3	re 3	re 3
5				re 2	re 2
6	sh 3	sh 4			
7	sh 3	sh 4			
8				re 1	re 1
9			re 5	re 5	re 5
10			re 4	re 4	re 4
11			re 6	re 6, sh 6	re 6
12				re 7, sh 6	re 7

(a) Die Aktionstabelle.

Zustand	N	P	V	S
0	2			1
1		5		
2		9	8	
3				
4				
5				
6	11			
7	12			
8				
9				
10				
11		9		
12		9		

(b) Die Sprungtabelle.

Tabelle 3.6: Die Parse-Tabelle. Quelle: [Tom86]

Während der Abarbeitung des Eingabeworts werden in jedem Schritt die Shiftmenge Q, die Reduktionsmenge R und die Menge $U_{i,0}$, der nächste zu bearbeitende Buchstabe, der GSS sowie der Parse-Forest angegeben. Die Menge $U_{i,0}$ enthält die Zustandsknoten, die bei der Verarbeitung des das Eingabezeichen a_i erzeugt wurden. Da diese Information nur für die Abarbeitung des aktuellen Eingabezeichens a_i notwendig ist, wird auch nur die aktuelle U-Menge angegeben. R_ε braucht nicht betrachtet zu werden, weil es in der Grammatik keine ε-Reduktionen gibt. Es wird ebenfalls auf A verzichtet, da die Elemente, die für die Folgeaktionen der neu erzeugten Zustandsknoten stehen, direkt in die entsprechenden Mengen eingefügt werden. Bei den Elementen der Mengen Q, R und $U_{i,0}$ werden Zustandsknoten des Graphen mit dem Zustand k als v_k, Symbolknoten mit einer Referenz auf den Knoten t_l als t_l, der Folgezustand k als k_s und die Regel p, nach der reduziert werden soll, als p_r notiert. Die Regelnummer p referenziert die Regel mit der Nummer (p) in der Beispielgrammatik in Formel (3.58).

Obwohl im Pseudocode, der im folgenden Abschnitt „Arbeitsweise des Parsers“ angegeben ist, keine Knoten im GSS gelöscht werden, sind Knoten, die für das Parsen irrelevant geworden sind, in den Bildern nicht dargestellt, wodurch diese kleiner werden. Um die Verständlichkeit des Beispiels zu erhöhen, wird zusätzlich hinter jedem Top-Element des GSS angegeben, welche Aktionen als nächstes mit diesem Element ausgeführt werden müssen. Die verwendeten Bilder, die den Zustand des GSS und des Parse-Forest

beschreiben, stammen aus [Tom86]. Sie wurden jedoch an die in diesem Abschnitt verwendete Notations- und Darstellungsform angepasst.

Bei der Initialisierung des GLR Parsers wird ein leerer Parse-Forest und ein Graph mit dem Zustandsknoten 0 erzeugt. Da in der Aktionstabelle im Zustand 0 beim Buchstaben n die Aktion „sh 4“ festgelegt ist, wird in die Shiftmenge Q das Tupel $\langle v_0, 4_s \rangle$ eingefügt.

$U_{0,0} = \{v_0\}$
$R = \emptyset$
$Q = \{\langle v_0, 4_s \rangle\}$
nächster Buchstabe = n
GSS =
0 sh 4
Parse-Forest = $\emptyset$

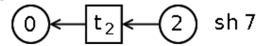
Die Menge R enthält keine Elemente, was bedeutet, dass keine Reduktionen ausgeführt werden müssen. Daher kann die „Shift-Anweisung“ $\langle v_0, 4_s \rangle$ verarbeitet werden. Hierzu wird zunächst im Parse-Forest der Knoten t_1 erzeugt, der für den Buchstaben n steht. Im GSS werden ein Symbolknoten mit Referenz auf t_1 und der Zustandsknoten v_4 erzeugt, der in $U_{1,0}$ eingefügt wird. Da die Aktionstabelle im Zustand 4 beim Eingabezeichen v die Reduktion nach Regel (3) vorschreibt, wird das Tripel $\langle v_4, t_1, 3_r \rangle$ in die Reduktionsmenge R eingefügt.

$U_{1,0} = \{v_4\}$
$R = \{\langle v_4, t_1, 3_r \rangle\}$
$Q = \emptyset$
nächster Buchstabe = v
GSS =
0 ← t1 ← 4 re 3
Parse-Forest =
t1:"n"

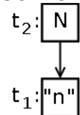
Das aus der Reduktionsmenge R extrahierte Tripel $\langle v_4, t_1, 3_r \rangle$ wird als nächstes verarbeitet. Zunächst wird im Parse-Forest ein neuer Knoten t_2 erzeugt und mit dem Kopf N der Regel (3) $N :=$ "n"; versehen. Da der Rumpf nur aus einem Terminal besteht, ist der durch den Symbolknoten t_1 referenzierte Parse-Forest-Knoten der einzige, der als Elternknoten t_2 erhält. Der nach der Reduktion aktuelle Zustandsknoten ist v_0. In der Sprungtabelle ist im Zustand 0 und beim Nonterminal N der Folgezustand 2 vorgesehen. Daher werden zwei neue GSS-Knoten v_2 und t_2 erzeugt und mit v_0 verbunden.

Im Zustand 2 sieht die Aktionstabelle „sh 7“ vor. Daher wird $\langle v_2, 7_s \rangle$ in die Shiftmenge Q eingefügt.

$U_{1,0} = \{v_4, v_2\}$
$R = \varnothing$
$Q = \{\langle v_2, 7_s \rangle\}$
nächster Buchstabe = v
GSS =



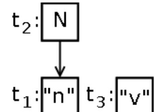
Parse-Forest =



Bei der Ausführung der Shift-Aktion, wird im Parse-Forest der Knoten t_3 angelegt und im GSS die beiden Knoten t_3 und v_7 erzeugt, die mit v_2 verbunden werden. Letzterer wird in die Menge $U_{2,0}$ eingefügt. Aufgrund des in der Aktionstabelle vorgesehenen „sh 3“ wird $\langle v_7, 3_s \rangle$ in die Shiftmenge Q eingefügt.

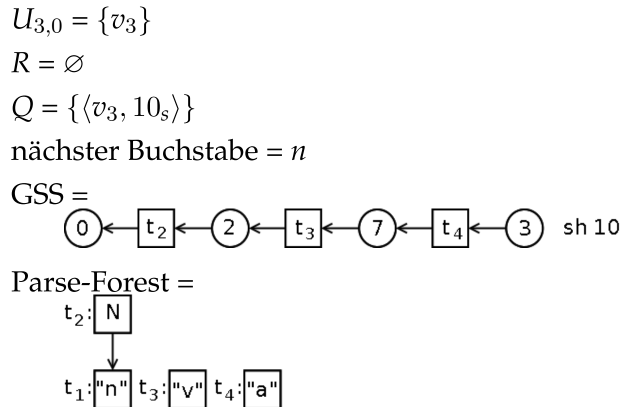
$U_{2,0} = \{v_7\}$
$R = \varnothing$
$Q = \{\langle v_7, 3_s \rangle\}$
nächster Buchstabe = a
GSS =

Parse-Forest =



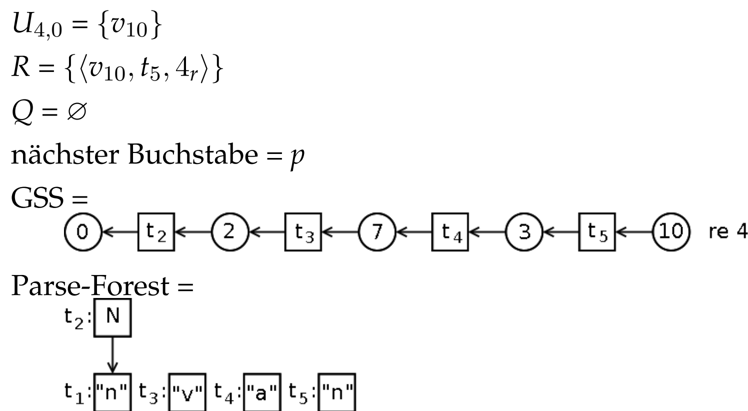
Analog zur soeben beschriebenen Shift-Aktion wird nun $\langle v_7, 3_s \rangle$ abgearbeitet.

$U_{3,0} = \{v_3\}$
$R = \varnothing$
$Q = \{\langle v_3, 10_s \rangle\}$
nächster Buchstabe = n
GSS =
Parse-Forest =



Nachdem $\langle v_3, 10_s \rangle$ bearbeitet wurde, ist in der Aktionstabelle „re 4“ vorgesehen. Daher wird $\langle v_{10}, t_5, 4_r \rangle$ in die Reduktionsmenge R eingefügt.

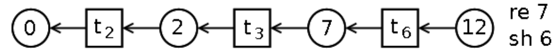
$U_{4,0} = \{v_{10}\}$
$R = \{\langle v_{10}, t_5, 4_r \rangle\}$
$Q = \varnothing$
nächster Buchstabe = p
GSS =
Parse-Forest =



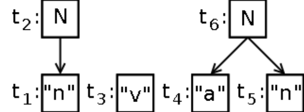
Als nächstes wird das Tripel $\langle v_{10}, t_5, 4_r \rangle$ verarbeitet, welches eine Reduktion nach der Regel (4) $N := "a""n";$ vorsieht. Zunächst wird im Parse-Forest der Knoten t_6 erzeugt, der mit dem Kopf N der Regel versehen wird. Da der Rumpf aus zwei Terminalen besteht, wird nur der nachfolgende Symbolknoten t_4 von t_5 gesucht. Die von beiden Symbolknoten referenzierten Parse-Forest-Knoten werden als Kinder unter t_6 eingefügt. In der Sprungtabelle sind beim Zustand 7, der im Zustandsknoten v_7 vermerkt ist, und beim Nonterminal N der Folgezustand 12 eingetragen. Daher werden im GSS die beiden Knoten t_6 sowie v_{12} erzeugt und mit v_7 verbunden.

In der Aktionstabelle sind im Zustand 12 und dem Terminal p die beiden Aktionen „re 7“ und „sh 6“ eingetragen. Im Gegensatz zu LR Parsern, die in einem solchen Shift-Reduce-Konflikt nicht weiterarbeiten könnten, wird $\langle v_{12}, t_6, 7_r \rangle$ in die Reduktionsmenge R und $\langle v_{12}, 6_s \rangle$ in die Shiftmenge Q eingefügt.

$U_{4,0} = \{v_{10}, v_{12}\}$
$R = \{\langle v_{12}, t_6, 7_r \rangle\}$
$Q = \{\langle v_{12}, 6_s \rangle\}$
nächster Buchstabe = p
GSS =



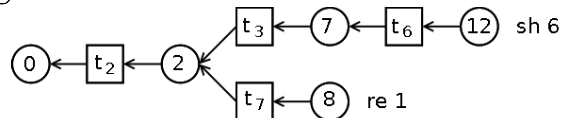
Parse-Forest =



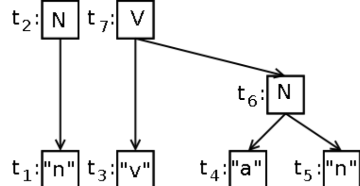
Da sich in der Reduktionsmenge R ein Element befindet, wird zunächst dieses abgearbeitet. Dies geschieht analog zu dem zuvor beschriebenen Reduktionsschritt. Wie an der Regel (7) $V := "v"N;$ zu sehen ist, wird im Parse-Forest der Knoten t_7 erzeugt und mit den beiden Kindern t_3 und t_6 verbunden. Die im Graphen neu erzeugten Knoten t_7 und v_8 werden mit dem Knoten v_2 verbunden. Somit ist durch den im vorangegangen Schritt aufgetretene Shift-Reduce-Konflikt eine Verzweigung an v_2 entstanden.

Als Folgeaktion nach der Reduktion ist „re 1" vorgesehen, weshalb $\langle v_8, t_7, 1_r \rangle$ in R eingefügt wird.

$U_{4,0} = \{v_{10}, v_{12}, v_8\}$
$R = \{\langle v_8, t_7, 1_r \rangle\}$
$Q = \{\langle v_{12}, 6_s \rangle\}$
nächster Buchstabe = p
GSS =

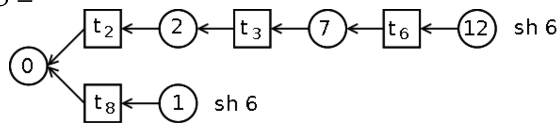


Parse-Forest =

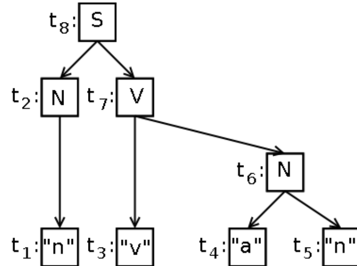


Bei der Reduktion anhand der Regel (1) $S := N\ V;$ wird im Parse-Forest der Knoten t_8 erzeugt und mit den beiden Kindern t_2 und t_7 versehen. Im GSS werden die beiden neu erzeugten Knoten t_8 und v_1 mit dem Knoten v_0 verbunden. Für die in der Aktionstabelle vorgesehene Aktion „sh 6“ wird Q um das Tupel $\langle v_1, 6_s\rangle$ ergänzt.

$U_{4,0} = \{v_{10}, v_{12}, v_8, v_1\}$
$R = \varnothing$
$Q = \{\langle v_{12}, 6_s\rangle, \langle v_1, 6_s\rangle\}$
nächster Buchstabe = p
GSS =

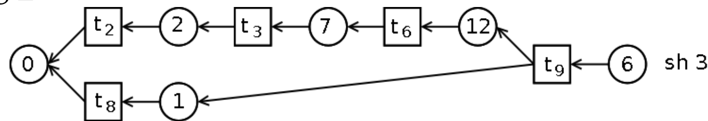


Parse-Forest =

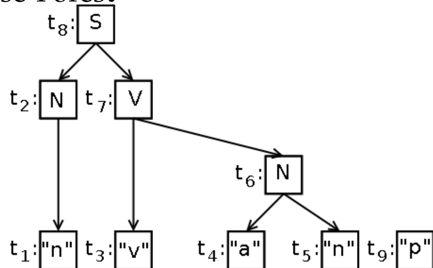


Da die Reduktionsmenge R nun leer ist, werden die Shift-Aktionen ausgeführt. Dazu wird im Parse-Forest ein Knoten für das Eingabezeichen p erzeugt. Da die beiden in der Shiftmenge Q enthaltenen Tupel in den gleichen Folgezustand übergehen, können sie in einem Schritt bearbeitet werden. Hierzu werden im GSS die Knoten t_9 und v_6 erzeugt und ersterer wird durch jeweils eine Kante mit v_{12} und v_1 verbunden. Damit werden die beiden maximalen Pfade im GSS vereinigt. Die Folgeaktion wird als $\langle v_6, 3_s\rangle$ in Q eingefügt.

$U_{5,0} = \{v_6\}$
$R = \varnothing$
$Q = \{\langle v_6, 3_s\rangle\}$
nächster Buchstabe = a
GSS =

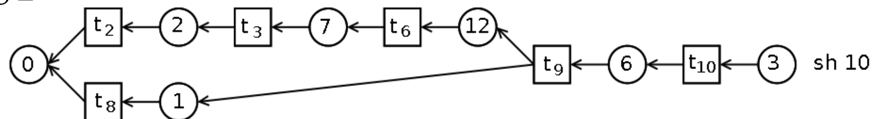


Parse-Forest =

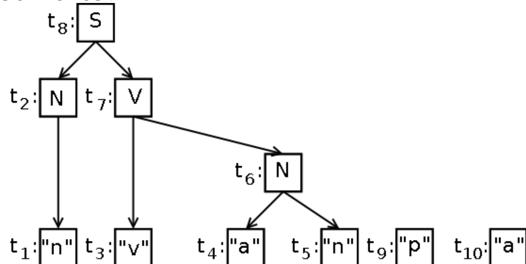


Als nächstes wird die Shift-Aktion $\langle v_6, 3_s\rangle$ verarbeitet. Für das im neuen Zustand 3 durch die Aktionstabelle vorgesehene „sh 10“ wird $\langle v_3, 10_s\rangle$ in die Shiftmenge Q eingefügt.

$U_{6,0} = \{v_3\}$
$R = \varnothing$
$Q = \{\langle v_3, 10_s\rangle\}$
nächster Buchstabe = n
GSS =

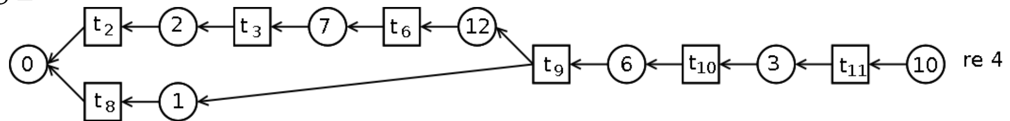


Parse-Forest =

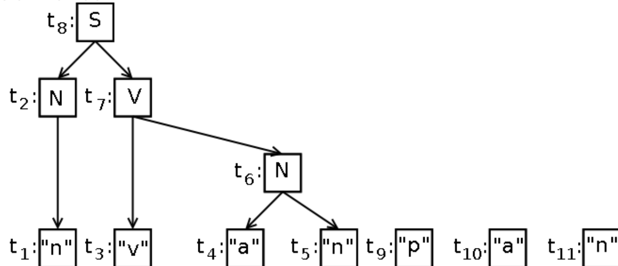


Analog zur letzten Aktion wird $\langle v_3, 10_s\rangle$ bearbeitet. Die Folgeaktion „re 4“ wird als Tripel $\langle v_{10}, t_{11}, 4_r\rangle$ zur Reduktionsmenge R hinzugefügt.

$U_{7,0} = \{v_{10}\}$
$R = \{\langle v_{10}, t_{11}, 4_r \rangle\}$
$Q = \varnothing$
nächster Buchstabe = p
GSS =

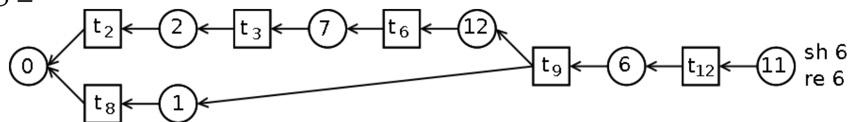


Parse-Forest =

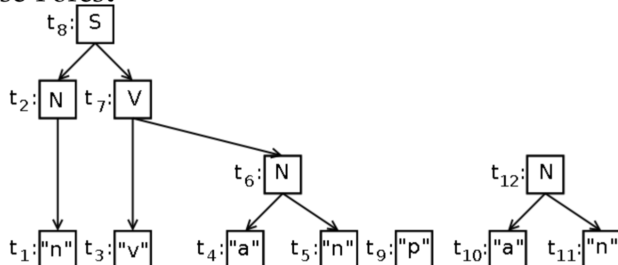


Bei der Abarbeitung der Reduktion $\langle v_{10}, t_{11}, 4_r \rangle$ nach Regel (4) N := "a""n"; wird im Parse-Forest der Knoten t_{12} erzeugt und die t_{10} sowie t_{11} als Kinder angefügt. Im GSS werden die Knoten t_{12} und v_{11} erzeugt, bevor sie mit v_6 verbunden werden. In der Aktionstabelle sind „sh 6“ und „re 6“ eingetragen. Die entsprechenden Elemente werden in die Reduktionsmenge R und Shiftmenge Q eingefügt.

$U_{7,0} = \{v_{10}, v_{11}\}$
$R = \{\langle v_{11}, t_{12}, 6_r \rangle\}$
$Q = \{\langle v_{11}, 6_s \rangle\}$
nächster Buchstabe = p
GSS =

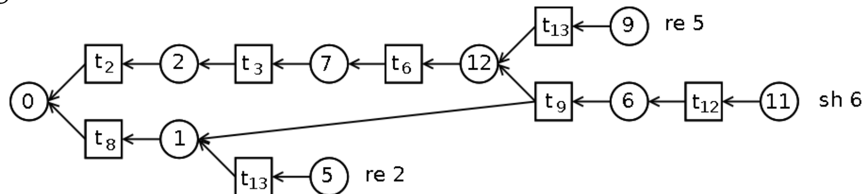


Parse-Forest =

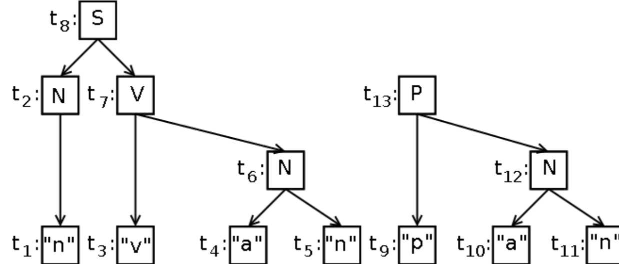


Durch die Reduktion nach Regel (6) P := "p"N; wird im Parse-Forest der Knoten t_{13} erzeugt und mit den beiden Kindern t_9 und t_{12} verbunden. Der Symbolknoten t_9 im GSS hat die beiden Nachfolger v_{12} und v_1. Die Sprungtabelle liefert für den ersten Knoten den Folgezustand 9 und für den zweiten 5. Da dies unterschiedliche Zustände sind, erhält v_{12} die beiden neu erzeugten Vorgänger t_{13} sowie v_9. v_1 wird mit den neu erzeugten Knoten t_{13} und v_5 verbunden. Die Folgeaktionen für die beiden neuen Zustandsknoten sind „re 5“ und „re 2“. Entsprechende Tripel werden in die Reduktionsmenge R eingefügt.

$U_{7,0} = \{v_{10}, v_{11}, v_9, v_5\}$
$R = \{\langle v_5, t_{13}, 2_r\rangle, \langle v_9, t_{13}, 5_r\rangle\}$
$Q = \{\langle v_{11}, 6_s\rangle\}$
nächster Buchstabe = p
GSS =

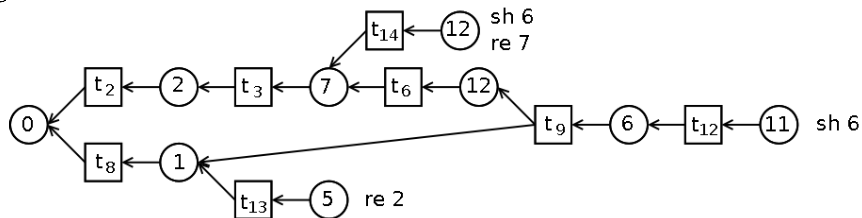


Parse-Forest =

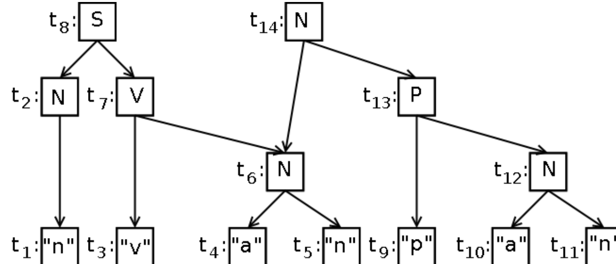


Welche der beiden Reduktionen zuerst ausgeführt wird, spielt für den Algorithmus keine Rolle. Per Zufall wird $\langle v_9, t_{13}, 5_r \rangle$ genommen. Bei der Reduktion nach Regel (5) $N := N\ P$; bekommen im Parse-Forest die beiden Knoten t_6 und t_{13} den neu erzeugten Elternknoten t_{14}. Somit wird der Teilbaum, der t_6 als Wurzelknoten besitzt, in zwei verschiedenen ParseTrees verwendet. Im GSS werden die beiden neuen Knoten t_{14} und v_{12} mit dem Knoten v_7 verbunden. Für die in der Aktionstabelle vorgesehenen Anweisungen „sh 6“ und „re 7“ werden entsprechende Einträge in der Shiftmenge Q und der Reduktionsmenge R ergänzt.

$U_{7,0} = \{v_{10}, v_{11}, v_9, v_5, v_{12}\}$
$R = \{\langle v_5, t_{13}, 2_r \rangle, \langle v_{12}, t_{14}, 7_r \rangle\}$
$Q = \{\langle v_{11}, 6_s \rangle, \langle v_{12}, 6_s \rangle\}$
nächster Buchstabe = p
GSS =

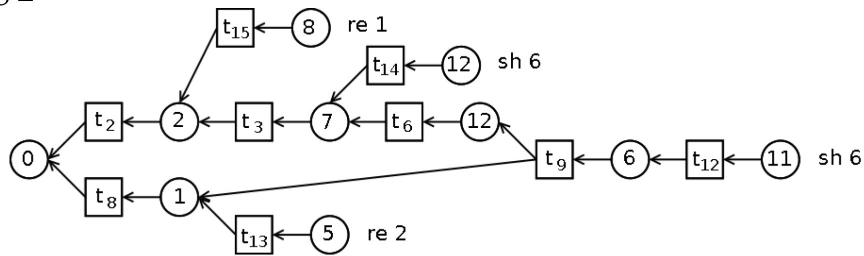


Parse-Forest =

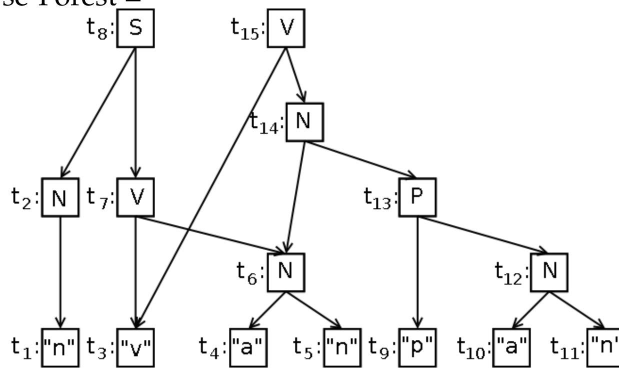


Als nächstes wird die Reduktion $\langle v_{12}, t_{14}, 7_r \rangle$ nach der Regel (7) $V := "v"N;$ ausgeführt. Der Ablauf ist analog zu dem zuvor beschriebenen Reduktionsschritt. Die Folgeaktion „re 1“ wird als Tripel in die Reduktionsmenge R eingefügt.

$U_{7,0} = \{v_{10}, v_{11}, v_9, v_5, v_{12}, v_8\}$
$R = \{\langle v_5, t_{13}, 2_r \rangle, \langle v_8, t_{15}, 1_r \rangle\}$
$Q = \{\langle v_{11}, 6_s \rangle, \langle v_{12}, 6_s \rangle\}$
nächster Buchstabe = p
GSS =

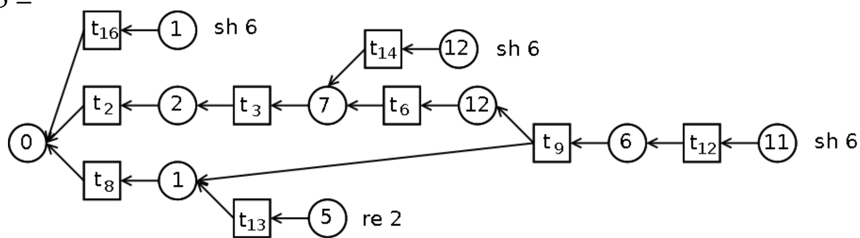


Parse-Forest =

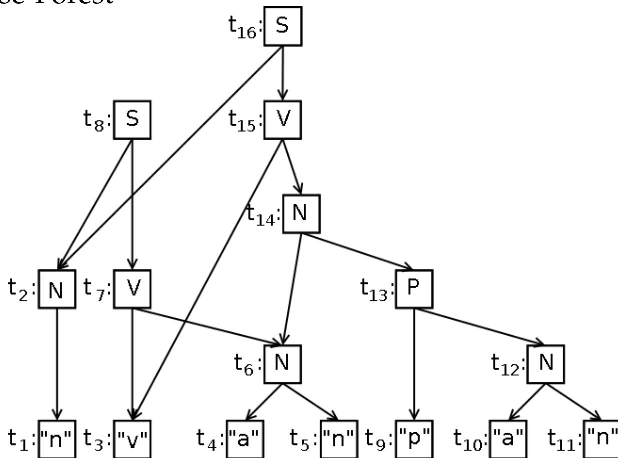


Die Abarbeitung der Reduktion $\langle v_8, t_{15}, 1_r \rangle$ wird anhand der Regel (1) $S := N\ V;$ vorgenommen. Sie verläuft genauso, wie die vorangegangene Aktion. Q wird für das folgende „sh 6“ um $\langle v_1, 6_s \rangle\}$ ergänzt.

$U_{7,0} = \{v_{10}, v_{11}, v_9, v_5, v_{12}, v_8, v_1\}$
$R = \{\langle v_5, t_{13}, 2_r \rangle\}$
$Q = \{\langle v_{11}, 6_s \rangle, \langle v_{12}, 6_s \rangle, \langle v_1, 6_s \rangle\}$
nächster Buchstabe = p
GSS =



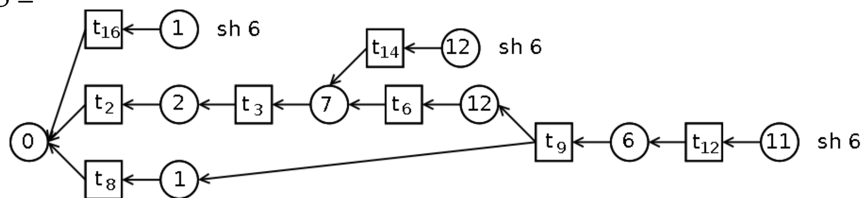
Parse-Forest =



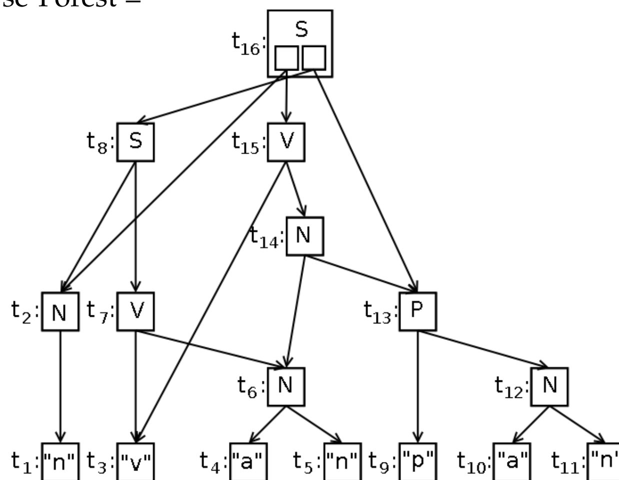
Als letzte Reduktion wird $\langle v_5, t_{13}, 2_r \rangle$ verarbeitet. Die zugehörige Regel (2) hat die Form $S := S\ P;$. In $U_{7,0}$ befindet sich bereits der Knoten v_1 mit dem Folgezustand 1. Der nachfolgende Zustandsknoten v_0 ist derselbe, mit dem die durch „re 2“ entstehenden Knoten verbunden worden wären. Durch diesen Tatbestand ist eine *Mehrdeutigkeit* erkannt worden und der GSS bleibt unverändert. Der zu v_1 gehörende Symbolknoten referenziert den ParseTree-Knoten t_{16}. Das „local ambiguity packing“ wird dadurch realisiert, dass die Referenz auf t_{16} auf einen neuen Knoten gelegt wird. Dieser erhält den alten t_{16} als Unterknoten. Des Weiteren wird ein weiterer Unterknoten erzeugt, der die beiden Knoten t_8 und t_{13} als Kinder erhält.

Der resultierenden Parse-Forest besteht aus zwei Bäumen, die beide den Knoten t_{16} als Wurzelknoten haben. Die beiden alternativen Bäume werden durch die in diesem Knoten enthaltenen Quadrate dargestellt. Der erste Baum hat als direkte Kinder den Knoten t_2, der das Teilwort *n* repräsentiert, und den Knoten t_{15}, der das Teilwort *vanpan* herleitet. Der zweite Baum hat t_8 und t_{13} als Kinder. Ersterer repräsentiert das Teilwort *nvan* und letzterer *pan*.

$U_{7,0} = \{v_{10}, v_{11}, v_9, v_5, v_{12}, v_8, v_1\}$
$R = \varnothing$
$Q = \{\langle v_{11}, 6_s\rangle, \langle v_{12}, 6_s\rangle, \langle v_1, 6_s\rangle\}$
nächster Buchstabe = *p*
GSS =

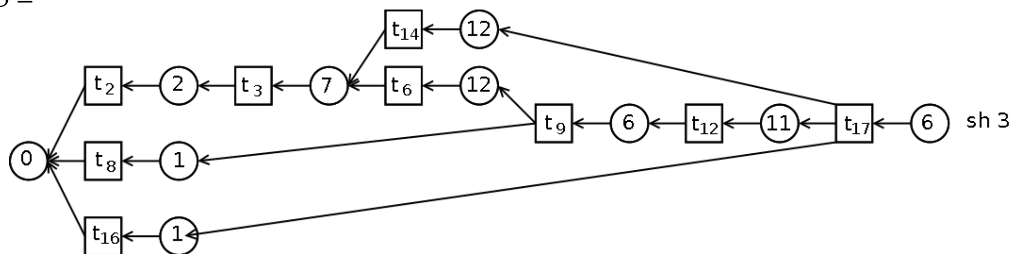


Parse-Forest =

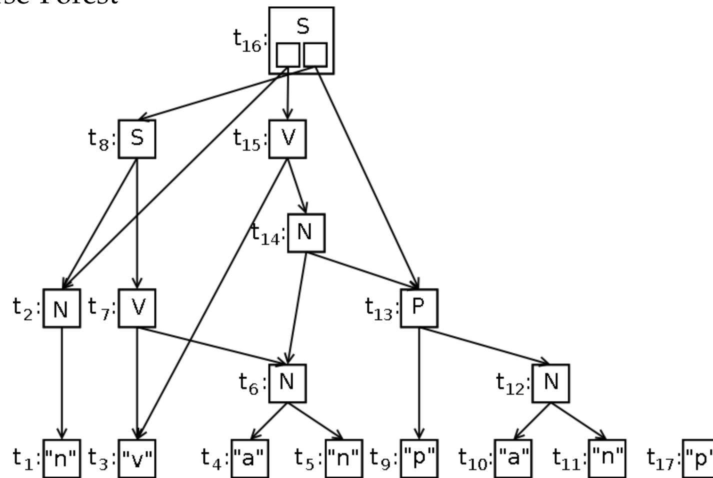


Da es keine weiteren Reduktionen gibt, werden nun die Shift-Operationen ausgeführt. Zunächst wird für den Buchstaben p ein Knoten im Parse-Forest erzeugt. Da alle drei Tripel der Shiftmenge Q den gleichen Folgezustand haben, werden im GSS nur die beiden Knoten t_{17} und v_6 erzeugt. Im Anschluss wird t_{17} mit den in den Tripeln angegebenen Knoten v_{11}, v_{12} und v_1 verbunden. Für die folgende Aktion „sh 3" wird $\langle v_6, 3_s \rangle$ in Q eingefügt.

$U_{8,0} = \{v_6\}$
$R = \varnothing$
$Q = \{\langle v_6, 3_s \rangle\}$
nächster Buchstabe = a
GSS =

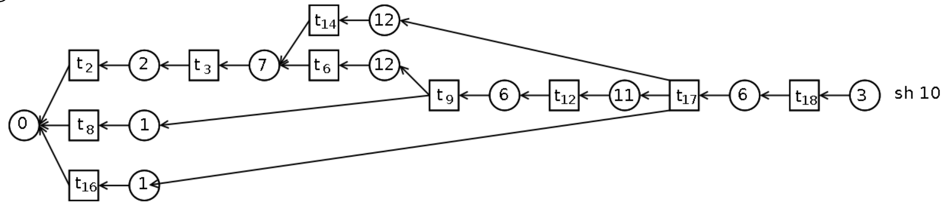


Parse-Forest =

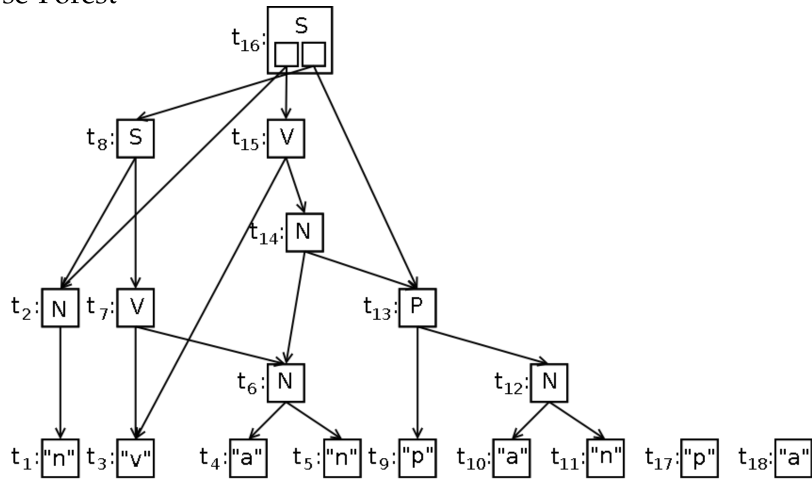


Die Shift-Operation $\langle v_6, 3_s \rangle$ verläuft genauso, wie schon zuvor öfters beschrieben wurde. Die Folgeaktion ist „sh 3“, wodurch die Shiftmenge Q um $\langle v_6, 3_s \rangle$ ergänzt wird.

$U_{9,0} = \{v_3\}$
$R = \varnothing$
$Q = \{\langle v_6, 3_s \rangle\}$
nächster Buchstabe = n
GSS =

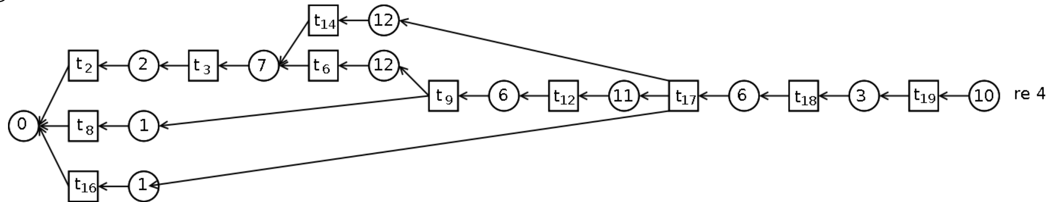


Parse-Forest =

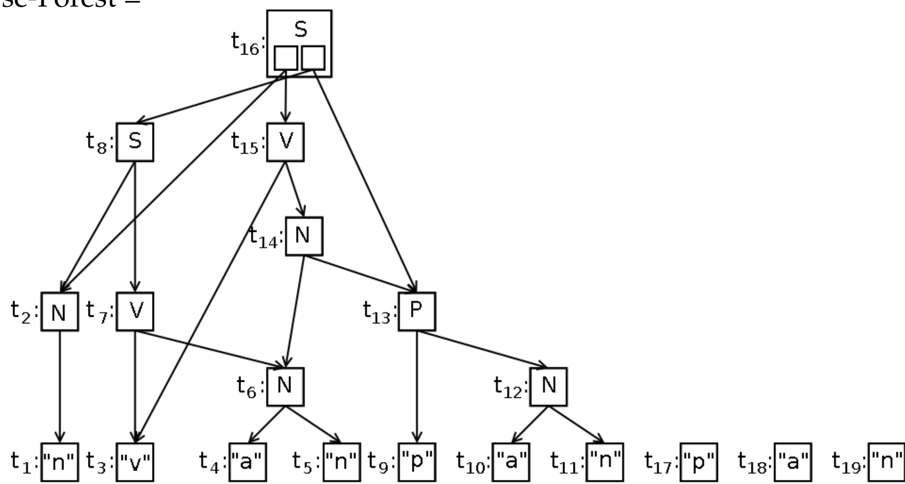


Nach der Abarbeitung von $\langle v_6, 3_s \rangle$ wird für das in der Aktionstabelle angegeben „re 4“ das Tripel $\langle v_{10}, t_{19}, 4_r \rangle$ in die Reduktionsmenge R eingefügt.

$U_{10,0} = \{v_{10}\}$
$R = \{\langle v_{10}, t_{19}, 4_r \rangle\}$
$Q = \varnothing$
nächster Buchstabe = \$
GSS =

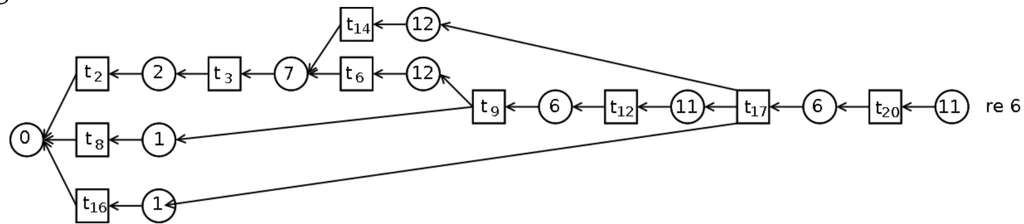


Parse-Forest =

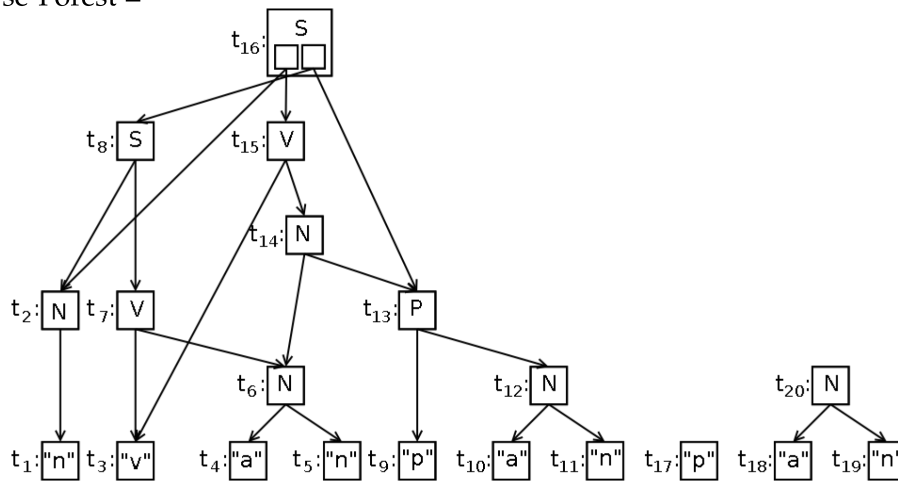


Bei der Reduktion nach Regel (4) $N := "a""n";$ wird im Parse-Forest zunächst der Knoten t_{20} erzeugt und t_{18} sowie t_{19} als Kinder hinzugefügt. Im Anschluss werden im GSS die Knoten t_{20} und v_{11} erzeugt. Die Folgeaktion „re 6“ wird als $\langle v_{11}, t_{20}, 6_r \rangle$ in der Reduktionsmenge R ergänzt.

$U_{10,0} = \{v_{10}, v_{11}\}$
$R = \{\langle v_{11}, t_{20}, 6_r \rangle\}$
$Q = \varnothing$
nächster Buchstabe = $
GSS =



Parse-Forest =

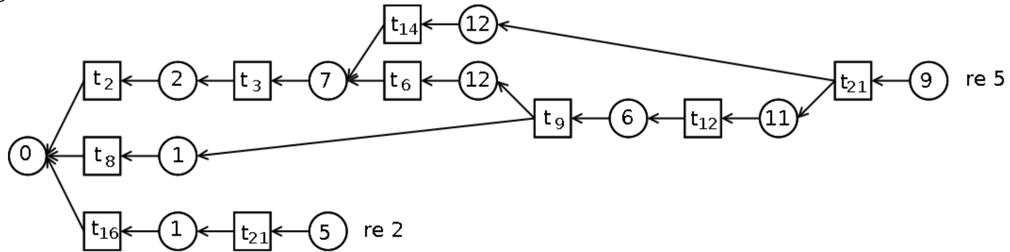


Die Reduktion nach Regel (6) $P := "p"N;$ wird per Zufall zuerst bearbeitet. Der im Parse-Forest neu erzeugte Knoten t_{21} erhält die beiden Kinder t_{17} und t_{20}. Im GSS werden v_{12}, v_{11}[9] und v_1 als die Zustandsknoten identifiziert, die mit den durch die Reduktion entstehenden Knoten verbunden werden müssen. Die Sprungtabelle gibt für die ersten beiden Knoten 9 und für v_1 5 als Folgezustände an. Daher werden die beiden neu erzeugten Knoten t_{21} und v_9 sowohl mit v_{12} als auch mit v_{11} verbunden. Als Folgeaktion wird $\langle v_9, t_{21}, 5_r \rangle$ in die Reduktionsmenge R eingefügt. v_1 wird mit den beiden neu erzeugten Knoten t_{21}

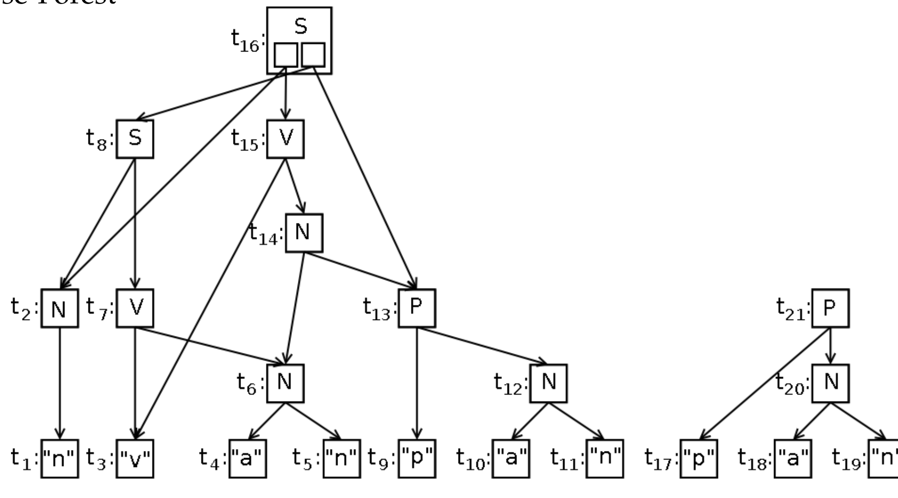
[9] In $U_{10,0}$ befindet sich ein Knoten mit dem Zustand 11. Da es sich dabei aber nicht um denselben Zustandsknoten handelt wie der, der identifiziert wurde, wird er nicht genommen.

und v_5 verbunden. Für das in der Aktionstabelle definierte „re 2“ wird R um $\langle v_5, t_{21}, 2_r \rangle$ ergänzt.

$U_{10,0} = \{v_{10}, v_{11}, v_9, v_5\}$
$R = \{\langle v_5, t_{21}, 2_r \rangle, \langle v_9, t_{21}, 5_r \rangle\}$
$Q = \varnothing$
nächster Buchstabe = \$
GSS =



Parse-Forest =

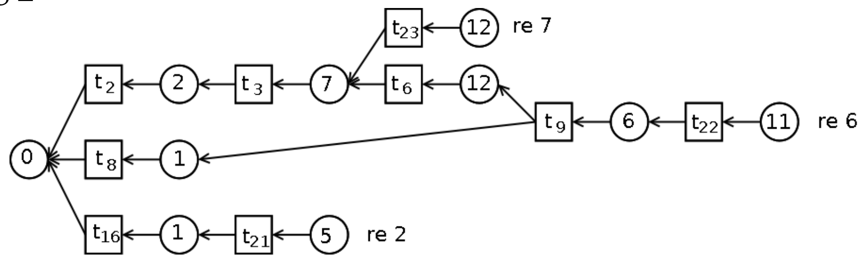


Als nächstes wird $\langle v_9, t_{21}, 5_r \rangle$ verarbeitet. Dabei handelt es sich um eine Reduktion nach der Regel (5) $N\ :=\ N\ P;$. Diese Reduktion kann im GSS *auf zwei verschiedenen Wegen* geschehen, die beide bearbeitet werden müssen:
(1) $v_6 \leftarrow t_{12} \leftarrow v_{11} \leftarrow t_{21} \leftarrow v_9$ und
(2) $v_7 \leftarrow t_{14} \leftarrow v_{12} \leftarrow t_{21} \leftarrow v_9$.

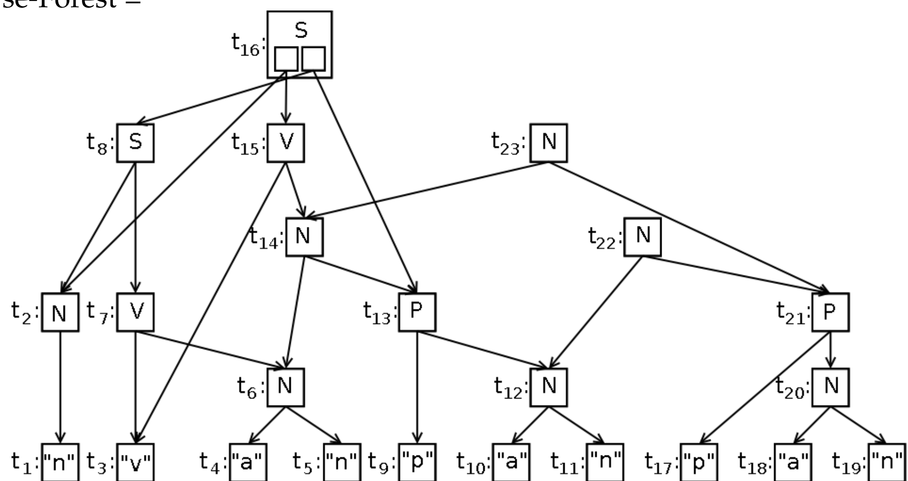
Zunächst betrachten wir den *ersten Fall*. Hierzu wird im Parse-Forest der Knoten t_{22} erzeugt und mit den beiden Kindern t_{12} und t_{21} verbunden. Als Folgezustand gibt die Sprungtabelle 11 an. Da es in $U_{10,0}$ bereits einen Zustandsknoten mit dem Zustand 11 gibt, wird dieser genommen. Im Anschluss wird noch ein Symbolknoten t_{22} erzeugt und mit v_{11} sowie v_6 verbunden. Da die Folgeaktion „re 6“ ist, wird $\langle v_{11}, t_{22}, 6_r \rangle$ in die Reduktionsmenge R eingefügt.

Im *zweiten Fall* wird im Parse-Forest t_{23} erzeugt und zum Vaterknoten von t_{14} und t_{21} gemacht. Im GSS werden die beiden Knoten t_{23} sowie v_{12} erzeugt und mit v_7 verbunden. Für das in der Aktionstabelle definierte „re 7“ wird R um $\langle v_{12}, t_{23}, 7_r \rangle$ ergänzt.

$U_{10,0} = \{v_{10}, v_{11}, v_9, v_5, v_{12}\}$
$R = \{\langle v_5, t_{21}, 2_r \rangle, \langle v_{12}, t_{23}, 7_r \rangle, \langle v_{11}, t_{22}, 6_r \rangle\}$
$Q = \varnothing$
nächster Buchstabe = \$
GSS =



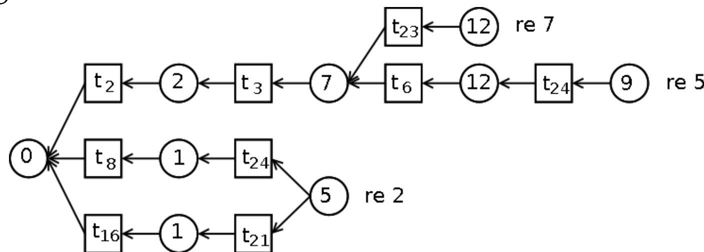
Parse-Forest =



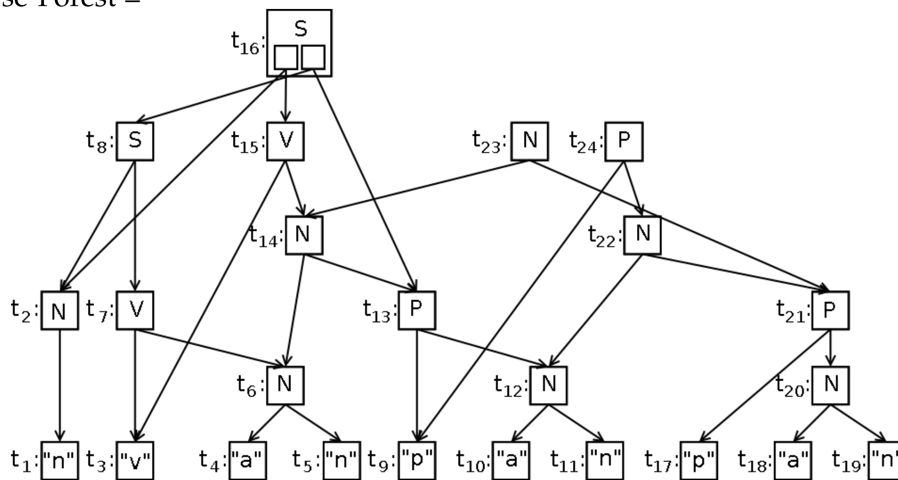
Bei der Reduktion $\langle v_{11}, t_{22}, 6_r \rangle$ nach Regel (6) $P := "p"N;$ wird im Parse-Forest zunächst der Knoten t_{24} erzeugt und mit den Kindknoten t_9 und t_{21} verbunden. Im GSS hat t_9 die beiden Nachfolger v_{12} und v_1. Für Ersteren liefert die Sprungtabelle den Folgezustand 9. Da sich in $U_{10,0}$ bereits ein Knoten v_9 mit diesem Zustand befindet, wird kein neuer Zustandsknoten angelegt. Es wird ein neuer Symbolknoten t_{24} erzeugt und mit v_{12} und v_9 verbunden. Für die in der Aktionstabelle vorgesehene „re 5“ wird $\langle v_9, t_{24}, 5_r \rangle$ in die Reduktionsmenge R eingefügt.

Im Falle von v_1 liefert die Sprungtabelle den Folgezustand 5. Auch für diesen Zustand gibt es in $U_{10,0}$ bereits einen Zustandsknoten. Daher wird ein neuer Symbolknoten t_{24} angelegt und mit v_1 und v_5 verbunden. Für die Folgeaktion „re 2“ wird R um $\langle v_5, t_{24}, 2_r \rangle$ ergänzt.

$U_{10,0} = \{v_{10}, v_{11}, v_9, v_5, v_{12}\}$
$R = \{\langle v_5, t_{21}, 2_r \rangle, \langle v_5, t_{24}, 2_r \rangle, \langle v_{12}, t_{23}, 7_r \rangle, \langle v_9, t_{24}, 5_r \rangle\}$
$Q = \varnothing$
nächster Buchstabe = \$
GSS =



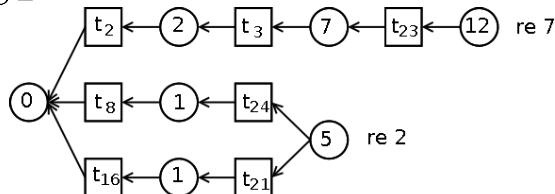
Parse-Forest =



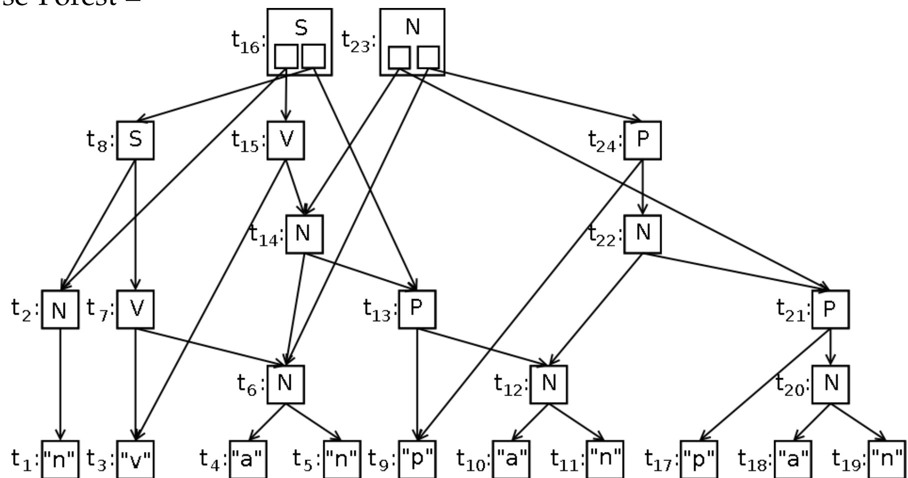
Die nächste durchzuführende Reduktion ist $\langle v_9, t_{24}, 5_r \rangle$. Die zugehörige Regel (5) lautet $N := N\ P;$. Da der Folgezustand 12 der Reduktion bereits durch den Knoten v_{12} in $U_{10,0}$ vertreten ist, muss kein neuer Zustandsknoten angelegt werden. Da der Nachfolger-Zustandsknoten von v_{12} derselbe ist wie der, mit dem der durch die Reduktion entstehende Symbolknoten verbunden worden wäre, ist eine weitere Mehrdeutigkeit festgestellt worden. Daher wird kein neuer Symbolknoten angelegt.

Aufgrund der Mehrdeutigkeit wird im Parse-Forest der durch den v_{12} direkt nachfolgenden Symbolknoten referenzierte Knoten t_{23} modifiziert. Zunächst wird ein neuer Knoten erzeugt, auf den nun alle t_{23}-Referenzen gesetzt werden. Der alte t_{23}-Knoten wird dem neuen unterstellt. Für die aktuelle Reduktion wird ein neuer Unterknoten von t_{23} erstellt, der t_6 und t_{24} als Kinder erhält.

$U_{10,0} = \{v_{10}, v_{11}, v_9, v_5, v_{12}\}$
$R = \{\langle v_5, t_{21}, 2_r \rangle, \langle v_5, t_{24}, 2_r \rangle, \langle v_{12}, t_{23}, 7_r \rangle\}$
$Q = \varnothing$
nächster Buchstabe = \$
GSS =

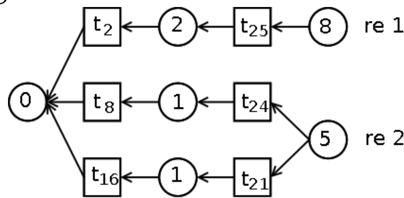


Parse-Forest =

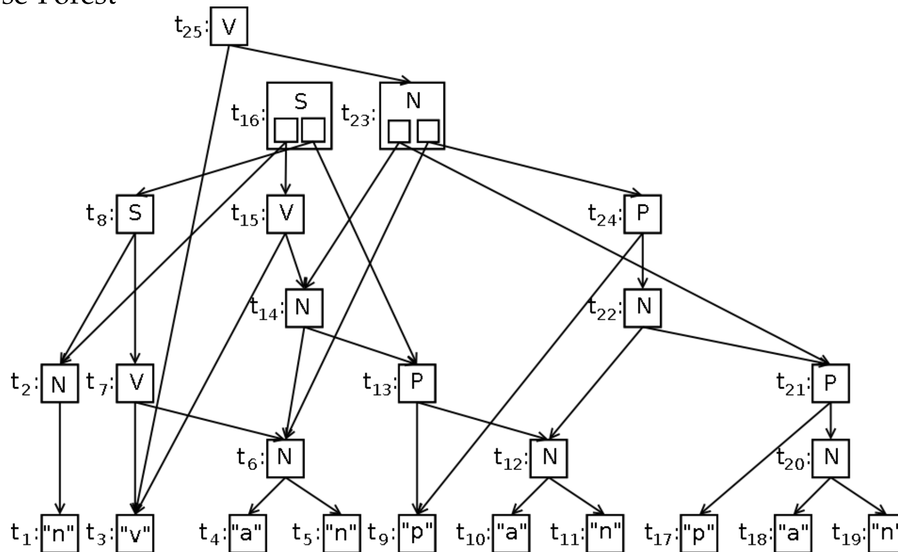


Bei der Reduktion $\langle v_{12}, t_{23}, 7_r \rangle$ nach der Regel (7) V := "v"N; wird im Parse-Forest zunächst der Knoten t_{25} erzeugt. Als Kinder werden ihm t_3 und t_{23} zugewiesen. Im GSS werden die neu erzeugten Knoten t_{25} und v_8 mit dem Knoten v_2 verbunden. Die Folgeaktion „re 1“ wird als $\langle v_8, t_{25}, 1_r \rangle$ in die Reduktionsmenge R eingefügt.

$U_{10,0} = \{v_{10}, v_{11}, v_9, v_5, v_{12}, v_8\}$
$R = \{\langle v_5, t_{21}, 2_r \rangle, \langle v_5, t_{24}, 2_r \rangle, \langle v_8, t_{25}, 1_r \rangle\}$
$Q = \varnothing$
nächster Buchstabe = \$
GSS =

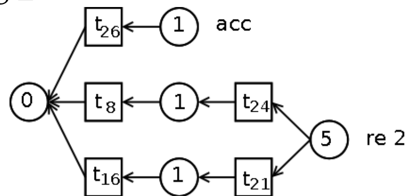


Parse-Forest =

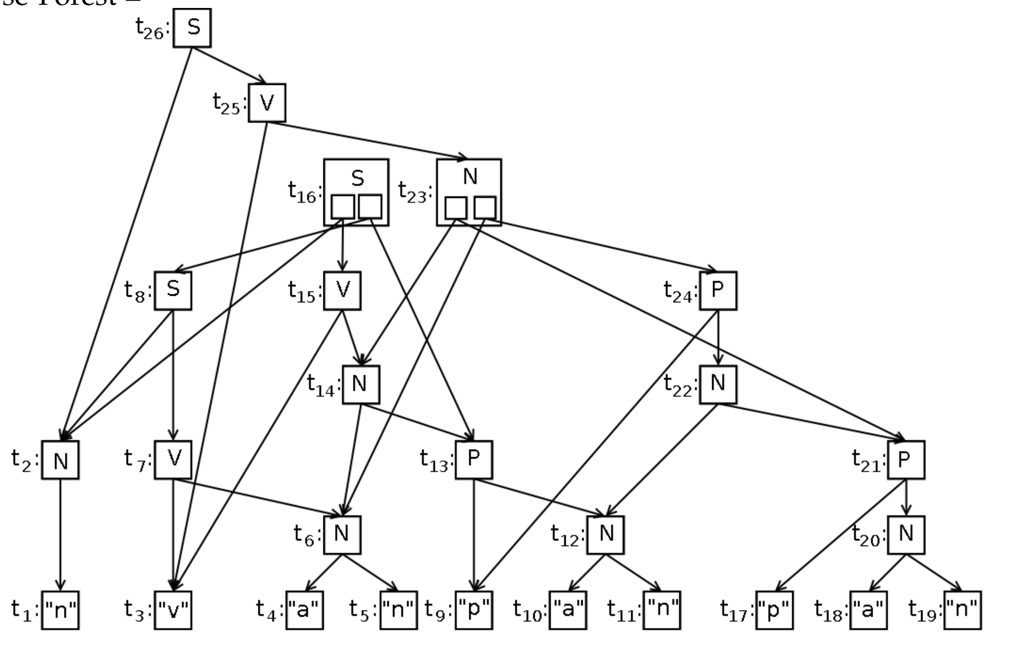


Die Reduktion $\langle v_8, t_{25}, 1_r \rangle$ nach Regel (1) $S := N\ V$; wird als nächstes bearbeitet. Zunächst wird im Parse-Forest der Knoten t_{26} angelegt und erhält t_2 und t_{25} als Kindknoten. t_{25} und v_1 werden im GSS erzeugt und mit v_0 verbunden. Da in der Aktionstabelle im Zustand 1 beim Eingabezeichen \$ „acc“ eingetragen ist, wird der Knoten t_{26} als zurückzulieferndes Ergebnis des Parsevorgangs vorgemerkt und die verbliebenen Elemente der Reduktionsmenge R werden weiter bearbeitet.

$U_{10,0} = \{v_{10}, v_{11}, v_9, v_5, v_{12}, v_8, v_1\}$
$R = \{\langle v_5, t_{21}, 2_r\rangle, \langle v_5, t_{24}, 2_r\rangle\}$
$Q = \varnothing$
nächster Buchstabe = \$
GSS =



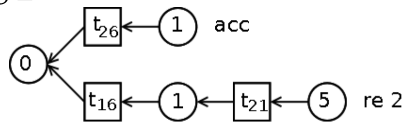
Parse-Forest =



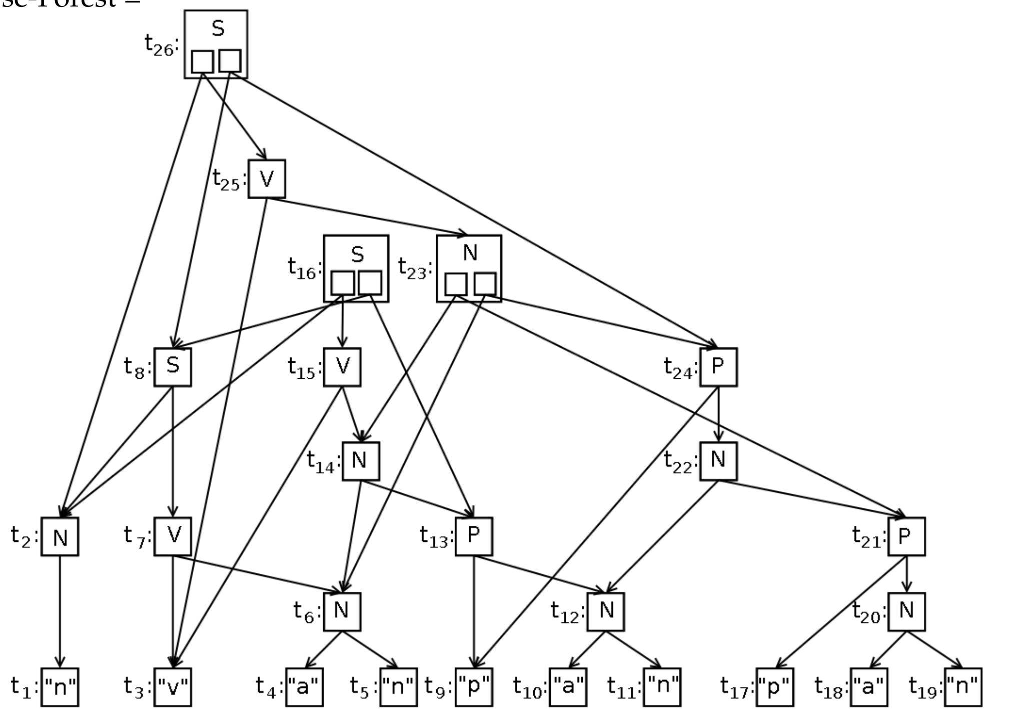
Als nächstes wird die Reduktion $\langle v_5, t_{24}, 2_r\rangle$ nach Regel (2) S := S P; abgearbeitet. Der relevante Weg im GSS ist $v_0 \leftarrow t_8 \leftarrow v_1 \leftarrow t_{24} \leftarrow v_5$. Der Folgezustand nach der Reduktion ist 1. Da sich in $U_{10,0}$ bereits ein Knoten mit diesem Zustand befindet, muss kein neuer Zustandsknoten angelegt werden. Da der v_1 vorangegangene Zustandsknoten v_0 derselbe ist wie der Endknoten des relevanten Pfads, wurde erneut eine Mehrdeutigkeit festgestellt und der GSS muss nicht verändert werden.

Der v_1 direkt nachfolgende Symbolknoten referenziert den Knoten t_{26}. Im Parse-Forest wird ein neuer Knoten angelegt und alle Referenzen auf t_{26} werden auf diesen neuen Knoten umgelegt. Der alte t_{26} wird dem neuen Knoten unterstellt. Des Weiteren wird ein neuer Unterknoten erzeugt, der die Kindknoten t_8 und t_{24} erhält.

$U_{10,0} = \{v_{10}, v_{11}, v_9, v_5, v_{12}, v_8, v_1\}$
$R = \{\langle v_5, t_{21}, 2_r \rangle\}$
$Q = \varnothing$
nächster Buchstabe = \$
GSS =

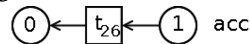


Parse-Forest =

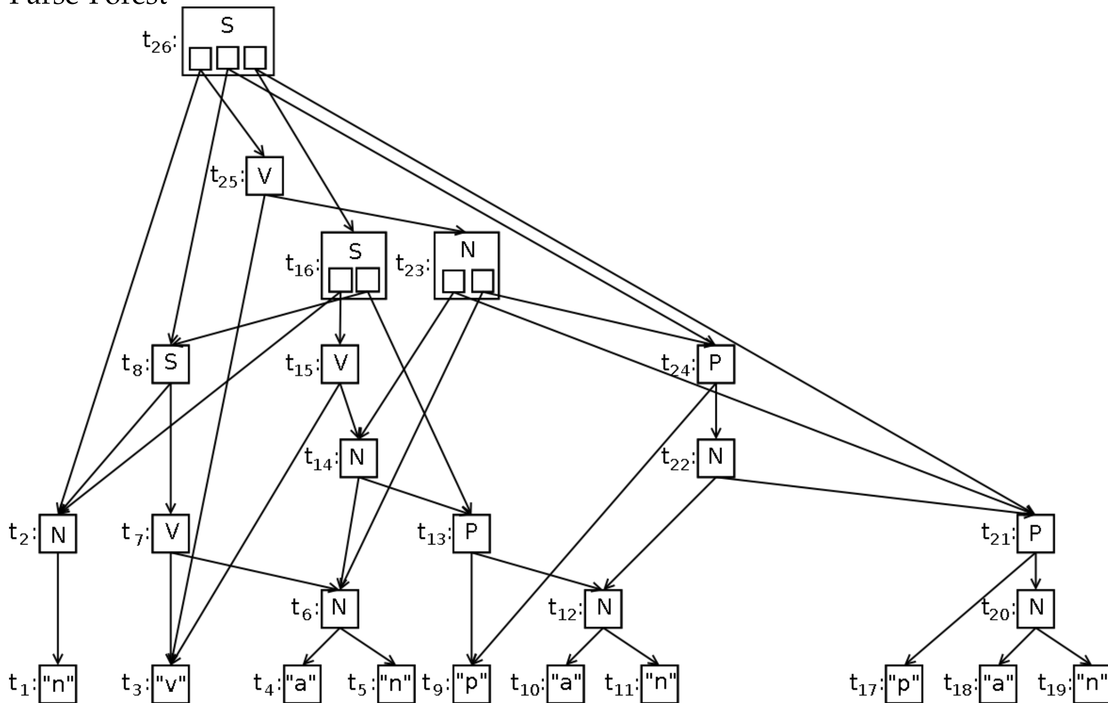


Als letzte verbliebene Aktion wird schließlich die Reduktion $\langle v_5, t_{21}, 2_r \rangle$ nach Regel (2) $S := S\ P;$ bearbeitet. Auch in diesem Fall bleibt der GSS unverändert, weil eine weitere Mehrdeutigkeit festgestellt wurde. Im Parse-Forest wird der Knoten t_{26} um einen weiteren Unterknoten erweitert, der t_{16} und t_{21} als Kinder zugewiesen bekommt.

$U_{10,0} = \{v_{10}, v_{11}, v_9, v_5, v_{12}, v_8, v_1\}$
$R = \varnothing$
$Q = \varnothing$
nächster Buchstabe = \$
GSS =



Parse-Forest =



Da nun sowohl die Reduktionsmenge R als auch die Shiftmenge Q leer sind und es keine weiteren Eingabezeichen gibt, ist der GLR-Algorithmus am Ende angelangt. Da der Knoten t_{26} als Wurzelknoten des Parse-Forest vorgemerkt wurde, war der Parsevorgang erfolgreich und t_{26} *wird als Ergebnis zurückgeliefert*.

Anhand des Wurzelknotens t_{26} ist zunächst erkennbar, dass der Parse-Forest sich aus drei verschiedenen Bäumen zusammensetzt. Der Wurzelknoten des ersten Baumes hat die Kindknoten t_2 und t_{25}. Letzterer hat als zweites Kind den Knoten t_{23}. Da es sich dabei wieder um einen Superknoten handelt, der zwei verschiedene Teilbäume repräsentiert, ist der erste in t_{26} enthaltene Knoten die Wurzel für zwei verschiedenen Bäume:

$$\left({}_{t_{26}}\,({}_{t_2}n)_{t_2}\left({}_{t_{25}}v\left({}_{t_{23}}\,({}_{t_{14}}anpan)_{t_{14}}\,({}_{t_{21}}pan)_{t_{21}}\right)_{t_{23}}\right)_{t_{25}}\right)_{t_{26}} \quad \text{und}$$

$$\left({}_{t_{26}}\,({}_{t_2}n)_{t_2}\left({}_{t_{25}}v\left({}_{t_{23}}\,({}_{t_6}an)_{t_6}\,({}_{t_{24}}panpan)_{t_{24}}\right)_{t_{23}}\right)_{t_{25}}\right)_{t_{26}}.$$

Die zweite in t_{26} enthaltene Wurzel repräsentiert nur einen Baum:

$$\left(_{t_{26}} (_{t_8} nvan)_{t_8} (_{t_{24}} panpan)_{t_{24}}\right)_{t_{26}}$$

Schließlich muss noch die letzte in t_{26} enthaltene Wurzel betrachtet werden. Ihre Kinder sind t_{16} und t_{21}. Ersterer ist wieder ein Superknoten, der für zwei verschiedene Teilbäume steht:

$$\left(_{t_{26}} \left(_{t_{16}} (_{t_2} n)_{t_2} (_{t_{15}} vanpan)_{t_{15}}\right)_{t_{16}} (_{t_{21}} pan)_{t_{21}}\right)_{t_{26}} \text{ und}$$

$$\left(_{t_{26}} \left(_{t_{16}} (_{t_8} nvan)_{t_8} (_{t_{13}} pan)_{t_{13}}\right)_{t_{16}} (_{t_{21}} pan)_{t_{21}}\right)_{t_{26}}.$$

Somit besteht der Parse-Forest aus fünf verschiedenen Bäumen.

Arbeitsweise des Parsers

Die formalere Beschreibung der *Arbeitsweise eines GLR Parsers*, die im vorangegangenen Abschnitt anhand eines Beispiels verdeutlicht wurde, geschieht anhand von *Pseudocode*. Die angegebenen Listings stammen *aus [Tom86]*. Um die Verständlichkeit des Codes zu erhöhen, wurde der von Tomita verwendete Pseudocode in eine Java-ähnliche Syntax überführt, wodurch Variablen einen sprechenden Bezeichner erhielten. Eigenschaften wie das „local ambiguity packing“ wurden von Tomita durch die mathematische Definition des Parse-Forest realisiert. Solche impliziten Eigenschaften wurden im Pseudocode dieses Abschnitts durch explizite Befehle realisiert.

Der angegebene Pseudocode geht von *einigen globalen Feldern* aus. Diese sind zunächst einmal die Mengen, die bereits im Abschnitt „Schematische Darstellung“ vorgestellt wurden. Darüber hinaus gibt es noch folgende Felder:

graph enthält die Referenz auf den GSS.

Parse-Forest verweist auf den Parse-Forest.

actionTable referenziert die Aktionstabelle. Enthaltene Reduktionsanweisungen haben die Form „re p“ wobei p ein Regelobjekt ist, das die entsprechende Grammatikregel repräsentiert.

gotoTable enthält die Referenz auf die Sprungtabelle.

root zeigt am Ende des Algorithmus auf den obersten Knoten des Parse-Forest, falls die Eingabe vollständig erkannt wurde. Ansonsten hat `root` den Wert `null`.

$a_1 \ldots a_m$**$** stellt das um \$ erweiterte Eingabewort der Länge m dar. Seine Buchstaben werden durch die Verwendung von a_i referenziert.

Für lokale Variablen wird folgende Schreibabkürzung verwendet:

Zustandsknoten sind vom Typ StateVertex und werden mit den Buchstaben u, v oder w bezeichnet.

Symbolknoten erhalten die Bezeichner x, y oder z. Ihr Typ ist SymbolVertex.
Regeln werden durch Variablen mit den Namen p oder q notiert.

Der *Pseudocode*, der den GLR Parsingalgorithmus beschreibt, ist analog zum „Divide and Conquer"-Prinzip *in mehrere Methoden aufgeteilt*. Die `parse`*-Methode*, die in Listing 3.32 dargestellt ist, initialisiert die Felder, stößt das Parsen nacheinander für jeden Buchstaben an und liefert am Ende das Ergebnis zurück. Danach folgt die `parseLetter`*-Methode* in Listing 3.33. Sie stößt die Abarbeitung der in den Mengen A, R, R_ε und Q enthaltenen Elemente an. Als nächstes folgt die `act`*-Methode* (Listing 3.34), die für die Zustandsknoten in A die Folgeoperationen in der Aktionstabelle bestimmt und Einträge in den zur korrekten Verarbeitung notwendigen Mengen erstellt.

Die folgende `reduce`*-Methode* ist in drei Listings aufgeteilt. Das erste zeigt die Arbeitsweise für den Fall, dass für den Folgezustand bei der Abarbeitung des i-ten Buchstabens noch kein neuer Zustandsknoten erzeugt wurde (Listing 3.35). Listing 3.36 beschreibt die Ausführung bei Mehrdeutigkeit und Listing 3.37 behandelt schließlich den Fall, dass ein bereits erzeugter Zustandsknoten wiederverwendet wird, ohne dass eine Mehrdeutigkeit vorliegt. Die Behandlung von ε-Reduktionen wird durch die *Methode* `εreduce` beschrieben. Wie Shift-Operationen verarbeitet werden, ist in der in Listing 3.38 gezeigten `shift`*-Methode* festgelegt.

`parse`-Methode

Die `parse`-Methode, wie sie in Listing 3.32 zu sehen ist, stößt das Parsen des Eingabeworts an, welches als aktueller Parameter übergeben wird.

```
1  parse(a_1...a_m)=
2     a_{m+1} = '$';
3     graph = new Graph();
4     v = graph.createStateVertex(0);
5     parseForest = new ParseForest();
6     root = null;
7     U_{0,0} = {v};
8     for(i=0; i<=m; i++){
9        parseLetter(i);
10    }
11    return root;
```

Listing 3.32: Die `parse`-Methode. Quelle: [Tom86]

Zunächst wird in Zeile 2 das Eingabewort $a_1 \dots a_m$ um \$ erweitert. In den folgenden beiden Zeilen wird ein neuer GSS mit Zustandsknoten v_0 erzeugt. In Zeile 5 wird ein neuer

leerer Parse-Forest angelegt und in Zeile 6 `root` mit `null` initialisiert. Im Anschluss wird der zuvor erzeugte Knoten v_0 in die Menge $U_{0,0}$ eingefügt. In den Zeilen 8 bis 10 wird nacheinander jeder Buchstabe durch einen Aufruf der Methode `parseLetter` geparst. Der Index `i` läuft von 0 bis m anstelle von 1 bis $m+1$, wie es die Indizierung der Eingabebuchstaben vermuten ließe, weil beim Parsen immer der als nächstes zu parsende Buchstabe a_{i+1} betrachtet wird. Nachdem die gesamte Eingabe geparst wurde, wird der durch `root` referenzierte Wurzelknoten des Parse-Forest zurückgeliefert. Sollte der Parser nie zu einem „acc“ gekommen sein, so hat `root` den Wert `null`.

`parseLetter`-Methode

Die `parseLetter`-Methode, wie sie in Listing 3.33 zu sehen ist, kümmert sich um das Parsen eines Buchstabens. Als Parameter bekommt sie den Index des Eingabezeichens, das zuletzt „geshifted“ wurde. Beim ersten Aufruf hat `i` den Wert 0, da noch kein Buchstabe verarbeitet wurde.

```
1  parseLetter(i)=
2     j = 0;
3     A = U_{i,0};
4     R = ∅;
5     R_ε = ∅;
6     Q = ∅;
7     do{
8        if(A ≠ ∅){
9           act(A.extractElement(),i);
10       }else if(R ≠ ∅){
11          reduce(R.extractElement(), i, j);
12       }else if(R_ε ≠ ∅){
13          j = εreduce(i, j);
14       }
15    }while(A == ∅ && R == ∅ && R_ε == ∅);
16    shift(i)
```

Listing 3.33: Die `parseLetter`-Methode. Quelle: [Tom86]

In der ersten Zeile wird der Index `j` mit 0 initialisiert, da seit dem „shiften“ des Buchstabens a_i noch kein ε erkannt worden ist. In Zeile 2 werden die Elemente von $U_{i,0}$ in A kopiert, damit für diese Zustandsknoten im späteren Verlauf bestimmt werden kann, welche Folgeaktion auszuführen ist. Zu diesem Zeitpunkt enthält $U_{i,0}$ im Falle des ersten Aufrufs dieser Methode den Knoten v_0 und ansonsten die Zustandsknoten des GSS, die

die Folgezustände nach dem „Shiften" des aktuellen Buchstabens sind. In den Zeilen 4 bis 6 werden die übrigen Mengen mit $\varnothing$ initialisiert.

Die Schleife in den Zeilen 7 bis 15 stellt sicher, dass zunächst alle möglichen Reduktionen ausgeführt werden, bevor schließlich in Zeile 16 die Shift-Operationen ausgeführt werden. Durch die Verzweigung innerhalb der Schleife werden zunächst für alle Zustandsknoten in A die Folgeaktionen bestimmt (Zeile 9). Bevor die ε-Reduktionen behandelt werden (Zeile 13), werden zunächst die anderen Reduktionen verarbeitet (Zeile 11). Wie viele ε bei der Abarbeitung des aktuellen Buchstabens bereits erkannt wurden, wird von der `εreduce`-Methode zurückgeliefert. Die Aufrufe von `extractElement()` bewirken, dass ein beliebiges Element aus der jeweiligen Menge entfernt und zurückgeliefert wird.

`act`-Methode

Die `act`-Methode, wie sie in Listing 3.34 zu sehen ist, bestimmt für den durch v repräsentierten Zustand die Folgeaktion.

```
1  act(v, i)=
2     for(Action α: actionTable.getActions(v.getState(),a_{i+1})){
3        if(α == 'acc'){ ∃x : x ← v; root = x.getTreeNode();}
4        if(α == 'sh s'){ Q.add(⟨v,s⟩);}
5        if(α == 're p' && !p.isεRule()){
6           for(SymbolVertex x: {x | x ← v}){ R.add(⟨v,x,p⟩);}
7        }
8        if(α == 're p' && p.isεRule()){
9           R_ε.add(⟨v,p⟩);
10       }
11    }
```

Listing 3.34: Die `act`-Methode. Quelle: [Tom86]

In Zeile 2 wird zunächst in der Aktionstabelle nachgesehen, welche Menge von Aktionen für den in v gespeicherten Zustand und bei dem als nächstes zu bearbeitenden Buchstaben definiert ist. Die Schleife in den Zeilen 2 bis 10 iteriert über alle Folgeaktionen.

Im Falle von „acc" (Zeile 3) wurde das Eingabewort erkannt und `root` muss auf den in dieser Herleitung erzeugten Wurzelknoten des Parse-Forest gesetzt werden. Dieser Knoten wird durch den Symbolknoten referenziert, der der direkte Nachfolger vom Zustandsknoten v ist.

Falls es sich bei der Aktion um „sh s" handelt (Zeile 4), wird in Q das Tupel $\langle v, s \rangle$ eingefügt. Dabei ist v der aktuelle Zustandsknoten und s der durch die Shift-Aktion definierte Folgezustand.

Bei Reduktionen wird unterschieden, ob es sich um ε-Reduktionen handelt oder nicht. Im ersten Fall wird in R_ε das Tupel $\langle v, p \rangle$, bestehend aus dem aktuellen Zustandsknoten v und der durch die Aktion vorgegebenen Regel p, eingefügt (Zeile 8 bis 10). Im anderen Fall (Zeile 5 bis 7) muss R um Tripel der Form $\langle v, x, p \rangle$ ergänzt werden. Hierbei ist x der direkte Nachfolger-Symbolknoten von v. Die Schleife in Zeile 6 ist notwendig, da ein Zustandsknoten mehrere verschiedene nachfolgende Symbolknoten haben kann, wie in der Abbildung des GSS auf Seite 81 zu sehen war.

`reduce`-Methode im Standardfall

Die `reduce`-Methode in Listing 3.35 beschreibt die Abarbeitung einer Reduktion im Standardfall. Ein solcher ist dann gegeben, wenn bei der bisherigen Verarbeitung des aktuellen Buchstabens der Folgezustand der Reduktion noch nicht erreicht worden ist. Der erste formale Parameter besteht aus dem Tripel $\langle v, x, p \rangle$. v bezeichnet den Zustandsknoten, von dem aus die Reduktion beginnt. Da dieser mehrere direkt nachfolgende Symbolknoten haben kann, legt x fest, in welche Richtung die Reduktion vonstatten geht. p ist das Regel-Objekt, das die Regel repräsentiert, nach der reduziert werden soll.

Zunächst wird in Zeile 2 der Kopf der Regel p extrahiert. Da die Reduktion über verschiedene Pfade vonstatten gehen kann, werden in der Schleife von Zeile 3 bis 34 alle möglichen Pfade nacheinander betrachtet. Ein exemplarischer Fall, in dem es mehr als einen Pfad gibt, ist auf Seite 79 beschrieben. x ist der Symbolknoten, der den Knoten im Parse-Forest referenziert, der das letzte Terminal bzw. Nonterminal des Regelrumpfs von p repräsentiert. Der Symbolknoten y zeigt dabei auf den Repräsentanten des ersten Terminals bzw. Nonterminals des Regelrumpfs von p. Der Pfad von x nach y muss dabei die Länge $2 * \mathit{Rumpflänge} - 2$ haben. Dabei steht *Rumpflänge* für die Anzahl der Terminale und Nonterminale im Rumpf der Regel p. Die Verdoppelung der Rumpflänge und anschließende Reduktion um 2 ist notwendig, da sich zwischen zwei Symbolknoten immer ein Zustandsknoten befindet.

In Zeile 5 werden alle Symbolknoten extrahiert, die an der Reduktion beteiligt sind. Die Reihenfolge der Knoten, in der sie in die Liste `children` eingefügt werden, ist umgekehrt zu ihren Vorkommen im Pfad von x nach y.

Die Schleife von Zeile 7 bis 33 iteriert über alle direkten Nachfolger von y. Diese sind die Zustandsknoten, mit denen die durch die Reduktion erzeugten GSS-Knoten verbunden werden. Darüber hinaus wird der in ihnen gespeicherte Zustand genutzt, um anhand der Sprungtabelle den Nachfolgezustand zu bestimmen (Zeile 8 und 9). Auf Seite 70 ist ein Beispiel für einen solchen Fall beschrieben. In den Zeilen 10 und 11 werden alle direkten

```
reduce(⟨v,x,p⟩, i, j)=
   head = extractHead(p);
   for(SymbolVertex y: {y | y ←^{2*extractBodyLength(p)−2} x}){
      path = y ←^{2*extractBodyLength(p)−2} x;
      children = extractTreeNodesOfSymbolVerticesIn(path);
      // children = [y.getTreeNode(), ..., x.getTreeNode()]
      for(StateVertex w:{w | w ← y}){
         State nextState = gotoTable
               .getNextState(w.getState(),head);
         prevStateVertices = {w' | w' ← y && gotoTable.
               getNextState(w'.getState(),head) == nextState};
         if(∃u ∈ U_{i,j} && u.getState() == nextState){
            ...
         } else {
            // the StateVertex after reduction does not exist
            headNode = parseTree
                  .getTreeNodeWithChildren(head, children);
            if(headNode == null){
               headNode = parseForest.createTreeNode(head);
               for(TreeNode child: children){
                  parseForest.createTreeEdge(headNode→child);
               }
            }
            u = graph.createStateVertex(nextState);
            z = graph.createSymbolVertex(headNode);
            graph.createEdge(z ← u);
            for(StateVertex prevState: prevStateVertices){
               graph.createEdge(prevState← z);
            }
            A.add(u);
            U_{i,j}.add(u);
         }
      }
   }
```

Listing 3.35: Die `reduce`-Methode im Standardfall. Quelle: [Tom86]

Nachfolger-Zustandsknoten von y bestimmt, die nach der Reduktion in den gleichen Folgezustand übergehen.

In Zeile 12 wird überprüft, ob bei der Abarbeitung des aktuellen Eingabezeichens bereits ein Zustandsknoten mit dem benötigten Folgezustand erzeugt wurde. Da dies im dargestellten Standardfall nicht zutrifft, wird der else-Teil ausgeführt (Zeile 14 bis 32).

Zunächst wird in den Zeilen 16 und 17 im Parse-Forest ein Knoten `headNode` gesucht, der mit dem zu erzeugenden äquivalent ist. Falls dem nicht so ist, wird ein entsprechender Knoten erzeugt (Zeile 19) und mit den Kindknoten, die die Terminale und Nonterminale im Regelrumpf repräsentieren, verbunden (Zeile 20 bis 22).

Als nächstes folgt die Anpassung des GSS. In Zeile 24 wird ein Zustandsknoten erzeugt, der den Folgezustand der Reduktion repräsentiert. Anschließend wird ein Symbolknoten angelegt, der eine Referenz auf den Parse-Forest-Knoten `headNode` speichert. Die Verbindung dieser beiden GSS-Knoten mit einer Kante geschieht in Zeile 26. Die Anbindung mit allen Zustandsknoten, die nach der Reduktion den gleichen Folgezustand erfordern, erfolgt in den Zeilen 27 bis 29. Ein Beispiel für einen Fall, in dem die durch eine Reduktion erzeugten Knoten mit mehreren Nachfolgern verbunden werden, ist auf Seite 79 gegeben.

Zum Abschluss wird der neu erzeugte Zustandsknoten zu A hinzugefügt, damit die Folgeaktionen bestimmt werden können (Zeile 30). Das Einfügen in $U_{i,j}$ geschieht, damit bei der Abarbeitung von a_i nicht erneut ein Zustandsknoten mit gleichem Zustand erzeugt wird.

`reduce`-Methode bei Mehrdeutigkeit

Die `reduce`-Methode in Listing 3.36 zeigt, wie die Reduktion bei Mehrdeutigkeit vonstatten geht.

Die Aktionen in den Zeilen 2 bis 11 sind identisch mit den Aktionen der Reduktion im Standardfall. Daher werden sie hier nicht erneut beschrieben. Die Bedingung in Zeile 12 besagt, dass es bei der Bearbeitung des aktuellen Buchstabens bereits einen Zustandsknoten u mit dem geforderten Folgezustand gibt. In Zeile 14 und 15 wird darüber hinaus geprüft, ob die dem existierenden u direkt folgenden Zustandsknoten dieselben sind wie die, die sich in der Menge `prevStateVertices` befinden. Bei diesem Vergleich reicht die Gleichheit der Zustände nicht aus. Sollten beide Bedingungen zutreffen, so wurde eine Mehrdeutigkeit entdeckt.

```
1  reduce(⟨v,x,p⟩, i, j)=
2     head = extractHead(p);
3     for(SymbolVertex y: {y | y ←^{2*extractBodyLength(p)−2} x}){
4        path = y ←^{2*extractBodyLength(p)−2} x;
5        children = extractTreeNodesOfSymbolVerticesIn(path);
6        // children = [y.getTreeNode(), ..., x.getTreeNode()]
7        for(StateVertex w:{w | w ← y}){
8           State nextState = gotoTable
9                 .getNextState(w.getState(),head);
10          prevStateVertices = {w′ | w′ ← y && gotoTable.
11                getNextState(w′.getState(),head) == nextState};
12          if(∃u ∈ U_{i,j} && u.getState() == nextState){
13             // the StateVertex after reduction already exists
14             if(∃z: existsEdge(z ← u)
15                   && {z′ | z′ ← z} == prevStateVertices){
16                // the previous StateVertices are equal, too
17                // thus an ambiguity was detected
18                superNode = z.getTreeNode();
19                if(!superNode.isSuperNode()){
20                   superNode = parseTree.createTreeNode(head);
21                   superNode.addSubNode(z.getTreeNode());
22                   graph.replaceAllReferences(
23                         z.getTreeNode(),superNode);
24                }
25                headNode = parseTree.createTreeNode(head);
26                superNode.addSubNode(headNode);
27                for(TreeNode child: children){
28                   if(!existsEdge(headNode→child)){
29                      parseForest
30                         .createTreeEdge(headNode→child);
31                   }
32                }
33             }else{...}
34          } else {...}
35       }
36    }
```

Listing 3.36: Die `reduce`-Methode im Falle von Mehrdeutigkeit. Quelle: [Tom86]

In einem solchen Fall muss das „local ambiguity packing“ angewendet werden. Hierzu wird zunächst überprüft, ob der durch den u direkt nachfolgende Symbolknoten referenzierte Parse-Forest-Knoten bereits ein SuperKnoten[10] ist (Zeile 18 und 19). Falls dem nicht so ist, wird ein neuer Superknoten angelegt (Zeile 20) und der referenzierte Baumknoten ihm als Unterknoten hinzugefügt (Zeile 21). Im Anschluss werden alle Referenzen auf den alten Knoten auf den neuen Superknoten umgelegt (Zeile 22).

Für die aktuelle Reduktion wird zunächst ein neuer Knoten im Parse-Forest angelegt (Zeile 24), dem Superknoten hinzugefügt (Zeile 25) und mit den entsprechenden Kindknoten verbunden (Zeile 26 bis 31). Ein Beispiel für die Reduktion bei Mehrdeutigkeit ist auf Seite 73 gegeben.

`reduce`-Methode bei Wiederverwendung eines Zustandsknotens

Listing 3.37 verdeutlicht die Arbeitsweise der `reduce`-Methode, wenn bei der Reduktion ein Zustandsknoten, der bei der Bearbeitung des aktuellen Eingabezeichens erzeugt wurde, wiederverwendet wird.

Die Anweisungen in den Zeilen 2 bis 15 wurden bereits in den beiden zuvor behandelten Reduktionsfällen erläutert. In den Zeilen 18 bis 26 werden die notwendigen Änderungen im Parse-Forest vorgenommen. Dies geschieht analog zum Standardfall.

In Zeile 27 wird ein neuer Symbolknoten z angelegt, der eine Referenz auf den aktuellen Parse-Forest-Knoten erhält. Da der Zustandsknoten u wiederverwendet werden soll, wird er durch eine Kante mit z verbunden (Zeile 28) und im Anschluss werden die entsprechenden Nachfolgerknoten gesetzt (Zeile 29 bis 31).

Falls die Repräsentanten der Folgeaktionen für den Zustandsknoten u bereits in die entsprechenden Mengen eingefügt worden sind, werden die Zeilen 33 bis 38 ausgeführt. Für alle in der Aktionstabelle definierten Reduktionen, die einen nicht-leeren Rumpf haben, wird ein Tripel $\langle u, z, q \rangle$ in R eingefügt. Dabei ist u der wiederverwendete Zustandsknoten, z der neu erzeugte Symbolknoten und q die Regel, nach der reduziert werden soll. Alle anderen definierten Aktionen brauchen nicht berücksichtigt zu werden, da der Knoten u erst bei der Bearbeitung des aktuellen Eingabezeichens bzw. dem aktuellen ε erzeugt wurde. Definierte ε-Reduktionen oder Shift-Aktionen sind daher noch durch entsprechende Tupel in den Mengen vertreten, da bevor sie ausgeführt werden, zunächst alle Reduktionen verarbeitet werden. Ein Beispiel ist auf Seite 81 zu sehen.

[10] Als Superknoten wird ein Parse-Forest-Knoten bezeichnet, der Unterknoten besitzt.

```
1  reduce(⟨v,x,p⟩, i, j)=
2     head = extractHead(p);
3     for(SymbolVertex y: {y | y ←^{2*extractBodyLength(p)−2} x}){
4        path = y ←^{2*extractBodyLength(p)−2} x;
5        children = extractTreeNodesOfSymbolVerticesIn(path);
6        // children = [y.getTreeNode(), ..., x.getTreeNode()]
7        for(StateVertex w:{w | w ← y}){
8           State nextState = gotoTable
9                 .getNextState(w.getState(),head);
10          prevStateVertices = {w' | w' ← y && gotoTable.
11                getNextState(w'.getState(),head) == nextState};
12          if(∃u ∈ U_{i,j} && u.getState() == nextState){
13             // the StateVertex after reduction already exists
14             if(∃z : existsEdge(z ← u)
15                   && {z' | z' ← z} == prevStateVertices){ ...
16             }else{
17                // reuse the existing StateVertex
18                headNode = parseTree
19                      .getTreeNodeWithChildren(head, children);
20                if(headNode == null){
21                   headNode = parseForest.createTreeNode(head);
22                   for(TreeNode child: children){
23                      parseForest
24                         .createTreeEdge(headNode→child);
25                   }
26                }
27                z = graph.createSymbolVertex(headNode);
28                graph.createEdge(z ← u);
29                for(StateVertex prevState: prevStateVertices){
30                   graph.createEdge(prevState← z);
31                }
32                if(u ∉ A){
33                   for('reduce q' : actionTable
34                         .getActions(u.getState(),a_{i+1})){
35                      if(!q.isεRule()){
36                         R.add(⟨u,z,q⟩);
37                      }
38                   }
39                }
40             }
41          } else {...}
42       }
43    }
```

Listing 3.37: Die `reduce`-Methode im Falle eines wiederverwendeten Zustandes. Quelle: [Tom86]

εreduce-Methode

Die εreduce-Methode in Listing 3.38 beschreibt das Vorgehen bei der Reduktion bei einer ε-Regel.

```
1  εreduce(i, j)=
2     U_{i,j+1} = ∅;
3     alreadySeenStates = ∅;
4     for(⟨v,p⟩: R_ε){
5        head = extractHead(p);
6        nextState = gotoTable.getNextState(v.getState(), head);
7        if(nextState ∉ alreadySeenStates){
8           leaf = parseForest.createTreeNode(head);
9           w = graph.createStateVertex(nextState);
10          x = graph.createSymbolVertex(leaf);
11          graph.createEdge(x ← w);
12          for(⟨v′,p′⟩: R_ε){
13             if(nextState == gotoTable.
14                   getNextState(v′.getState(), extractHead(p′))){
15                graph.createEdge(v′ ← x);
16             }
17          }
18          U_{i,j+1}.add(w);
19          alreadySeenStates.add(nextState);
20       }
21    }
22    R_ε = ∅;
23    A = U_{i,j+1};
24    return j + 1;
```

Listing 3.38: Die εreduce-Methode. Quelle: [Tom86]

Zunächst wird in Zeile 2 die Menge $U_{i,j+1}$ initialisiert. Die Schleife in den Zeilen 4 bis 21 iteriert über alle auszuführenden ε-Reduktionen. Zunächst wird der Kopf der anzuwendenden Regel extrahiert (Zeile 5) und im Anschluss der Folgezustand bestimmt (Zeile 6).

Im Parse-Forest wird ein Blatt erzeugt, das mit den Kopf der Regel p versehen ist (Zeile 8). Anschließend werden im GSS ein Zustandsknoten für den Folgezustand (Zeile 9) und ein Symbolknoten mit einer Referenz auf den Blattknoten (Zeile 10) erzeugt. Beide werden in Zeile 11 mit einer Kante verbunden. Als Nachfolger werden sie mit allen Zustandsknoten verbunden, für die eine ε-Reduktion mit dem gleichen Folgezustand vorgesehen ist. Um zu verhindern, dass diese bereits verarbeiteten Tripel erneut bearbeitet werden, gibt es

die in Zeile 3 definierte Menge `alreadySeenStates` sowie die Bedingung in Zeile 7. In Zeile 18 wird der neu erzeugte Zustandsknoten in $U_{i,j+1}$ eingefügt.

Nachdem alle ε-Reduktionen bearbeitet sind, wird die Menge R_ε gelöscht (Zeile 22). Danach wird durch das Kopieren aller Elemente von $U_{i,j+1}$ nach A erreicht, dass die Folgeaktionen für die neu erstellten Zustandsknoten bestimmt werden. Zum Abschluss wird der inkrementierte Index `j` zurückgeliefert (Zeile 24).

`shift`-Methode

Die `shift`-Methode in Listing 3.39 beschreibt wie Shift-Aktionen verarbeitet werden.

```
1  shift(i)=
2     U_{i+1,0} = ∅;
3     alreadySeenStates = ∅;
4     leaf = parseForest.createTreeNode(a_{i+1});
5     for(⟨v,s⟩: Q){
6        if(s ∉ alreadySeenStates){
7           w = graph.createStateVertex(s);
8           x = graph.createSymbolVertex(leaf);
9           graph.createEdge(x ← w);
10          for(⟨v′,s′⟩: Q){
11             if(s′ == s){
12                graph.createEdge(v′ ← x);
13             }
14          }
15          U_{i+1,0}.add(w);
16          alreadySeenStates.add(s);
17       }
18    }
```

Listing 3.39: Die `shift`-Methode. Quelle: [Tom86]

Zunächst wird in Zeile 1 die Menge $U_{i+1,0}$ initialisiert und in Zeile 4 ein Blatt für das folgende Eingabezeichen im Parse-Forest angelegt. Die Schleife in den Zeilen 5 bis 18 bearbeitet nacheinander alle Shift-Operationen.

Zunächst wird im GSS ein neuer Zustandsknoten w für den Folgezustand (Zeile 7) sowie ein Symbolknoten x mit einer Referenz auf das neu erzeugte Blatt angelegt (Zeile 8), bevor beide mit einer Kante verbunden werden (Zeile 9). Anschließend werden alle Zustandsknoten, die nach einer Shift-Aktion in den gleichen Folgezustand übergehen, mit w verbunden (Zeile 10 bis 14). Damit die auf diese Weise bearbeiteten Shift-Tupel nicht

erneut bearbeitet werden, gibt es die Menge `alreadySeenStates` und die Bedingung in Zeile 6. Der neu erzeugte Shift-Vertex wird in $U_{i+1,0}$ eingefügt.

Aufwandsbewertung

Bei der von Tomita angegebenen Aufwandsbetrachtung hängt die Effizienz des Algorithmus von der Anzahl der Mehrfacheintragungen in einer Zelle der Aktionstabelle ab. Gibt es höchstens eine Aktion pro Zelle, so hat er genauso wie der LR Parse-Algorithmus linearen Aufwand, da es im GSS keine Verzweigungen gibt. Für Aktionstabellen mit nur wenigen Mehrfacheintragungen ist der Aufwand fast linear. Bei sehr vielen Mehrfacheintragungen ist der Aufwand meistens $\mathcal{O}(n^3)$. Er kann diesen aber auch übersteigen.

Grammatiken mit unbeschränkter Mehrdeutigkeit oder mit Zyklen können vom GLR Algorithmus nicht geparst werden. Als Beispiel für eine unendliche Mehrdeutigkeit gibt Tomita die Grammatik in Formel (3.59) zusammen mit dem Eingabewort *xxx* an.

$$\begin{aligned} S &:= S\ S; \\ S &:= \varepsilon; \\ S &:= \text{"}x\text{"}; \end{aligned} \tag{3.59}$$

4 Anforderungen

Um die Anforderungen an die „Extractor Description Language“ zu erheben, wurde eine erste Version in einem ersten Schritt vom Autor dieser Masterarbeit aufgelistet. Diese Liste wurde in einem anschließenden Gespräch mit den Stakeholdern Herrn Dr. Volker Riediger und Herrn Daniel Bildhauer ergänzt und gemäß ihren Wünschen überarbeitet. Als Ergebnis dieses Prozesses entstanden die in diesem Kapitel aufgelisteten Anforderungen.

Da die einzelnen Anforderungen für die in dieser Masterarbeit zu entwickelnde Sprache eine unterschiedliche Relevanz haben, sind sie in die folgenden drei Kategorien eingeordnet, die in eckigen Klammern angegeben ist:

1. Muss: Muss-Anforderungen müssen umgesetzt werden.
2. Soll: Anforderungen dieser Kategorie beschreiben nicht die Hauptaufgabe der „Extractor Description Language“, sollten aber dennoch realisiert werden.
3. Kann: Mit Kann gekennzeichnete Anforderungen müssen im Rahmen dieser Masterarbeit nicht erfüllt werden. Eine Realisierung wäre jedoch wünschenswert.

Realisierte Anforderungen sind mit einem * versehen. Um eine bessere Übersicht über die Anforderungen zu erhalten, wurden sie in funktionale (Abschnitt 4.1) und nichtfunktionale Anforderungen (Abschnitt 4.2) unterteilt.

4.1 Funktionale Anforderungen

Die in diesem Abschnitt aufgeführten Anforderungen beschreiben Aspekte der „Extractor Description Language“, die die Definition der Grammatik (Abschnitt „Grammatikdefinition“ 4.1.1), die Angabe von semantischen Aktionen (Abschnitt „Semantische Aktionen“ 4.1.2), das Parsing (Abschnitt „Parsing“ 4.1.3) oder das Debugging (Abschnitt „Debugging“ 4.1.4) betreffen.

4.1.1 Grammatikdefinition

1.1.1 [Muss] * Die Regeln der zu definierenden Grammatik müssen in einer EBNF-ähnlichen Syntax notierbar sein.
1.1.2 [Muss] * Das Startsymbol bzw. die Startsymbole des Parse-Vorgangs müssen vom Nutzer definiert werden.
1.1.3 [Muss] * Zur Auflösung von Mehrdeutigkeit müssen Prioritäten für einzelne Regeln und Alternativen definierbar sein.
1.1.4 [Muss] * Als ein weiteres Mittel zur Auflösung von Mehrdeutigkeit muss die Assoziativität einzelner Regeln und Alternativen definierbar sein.
1.1.5 [Muss] * Regeln, die lexikalische Variablen bzw. Tokens definieren, müssen explizit als solche ausgewiesen werden.
1.1.6 [Muss] * Die zu ignorierenden Zeichen müssen im Rumpf von Regeln, die keine lexikalische Variablen bzw. Tokens definieren, nicht explizit vom Nutzer ergänzt werden.
1.1.7 [Muss] * Die beim Parsen zu ignorierenden Zeichen wie Leerzeichen müssen vom Nutzer definierbar sein.

Modularisierbarkeit

1.1.8 [Soll] * Die Grammatik soll modularisierbar sein.
    1.1.9 [Muss] * Die zu importierenden Module müssen angegeben werden.

Inselgrammatik

1.1.10 [Soll] * Die Definition von Inselgrammatiken soll ermöglicht werden.
    1.1.11 [Muss] * Der Anfang und das Ende der zu parsenden Daten müssen bei Inselgrammatiken angegeben werden.

4.1.2 Semantische Aktionen

In der „Extractor Description Language“ werden drei Arten von semantischen Aktionen unterschieden:

1. die Abbildung von Grammatiksymbolen auf Elemente eines TGraph-Schemas,
2. die Nutzung von Symboltabellen und
3. die Angabe von nutzerspezifischen semantischen Aktionen, die aus beliebigem Java-Code bestehen.

Abbildung auf ein TGraph-Schema

1.2.1 [Muss] * Das zu verwendende TGraph-Schema muss vom Nutzer angegeben werden.

1.2.2 [Soll] * Als Default-Mapping soll eine syntaktische Variable auf eine gleichnamige Knotenklasse abgebildet werden.

1.2.3 [Soll] Der Nutzer muss ein Variablen-spezifisches Mapping angeben können, welches syntaktische Variablen auf einen Knotentyp des angegebenen Schemas abbildet. Dadurch wird das Default-Mapping für die betroffene syntaktische Variable außer Kraft gesetzt.

1.2.4 [Soll] * Durch ein Regel-spezifisches Mapping sollen einzelne Regeln und Alternativen auf einen bestimmten Knoten- bzw. Kantentyp des angegebenen Schemas abgebildet werden. Dadurch wird das Variablen-spezifische und das Default-Mapping für die betroffene Regel bzw. Alternative außer Kraft gesetzt.

1.2.5 [Muss] * Das Setzen von Attributwerten eines Knotens bzw. einer Kante muss möglich sein.

1.2.6 [Soll] * Es soll dem Nutzer ermöglicht werden, die Positionen der aus einer syntaktischen Variablen hergeleiteten Eingabezeichen im Graphen zu speichern.

   1.2.7 [Muss] * Die Position der aus einer syntaktischen Variablen hergeleiteten Eingabezeichen muss aus der Zeile und Spalte des ersten Eingabezeichens und der Anzahl der hergeleiteten Eingabezeichen bestehen.

   1.2.8 [Muss] * Der Nutzer muss angeben können, in welchen Graphelement-Attributen die Positionsangabe gespeichert wird.

1.2.9 [Muss] * Verschiedene Knoten müssen zu einem einzigen Knoten verschmelzbar sein.

1.2.10 [Soll] Es soll definierbar sein, durch welchen Kantentyp alle Knoten, die durch Elemente des Regelrumpfs entstehen, mit dem Knoten des Regelkopfs verbunden werden sollen.

1.2.11 [Muss] * Von der zuvor genannten Regel abweichend muss es definierbar sein, durch welchen Kantentyp ein Knoten, der durch ein bestimmtes Element des Regelrumpfs entsteht, mit dem Knoten des Regelkopfs verbunden werden soll.

Symboltabellen

1.2.12 [Muss] * Es muss möglich sein, einen Symboltabellen-Stack für verschiedene Bezeichnerarten wie beispielsweise Klassen, Methoden und Variablen zu erstellen.

1.2.13 [Soll] * Der Gültigkeitsbereich einer Symboltabelle auf dem Stack, der in der zuvor aufgeführten Anforderung beschrieben wurde, soll definierbar sein.
1.2.14 [Soll] * Dem Nutzer soll es möglich sein, eine Symboltabelle im Graphen zu persistieren.
  1.2.15 [Muss] * Der Nutzer muss einen Knoten markieren können, der den Gültigkeitsbereich einer Symboltabelle darstellt.
  1.2.16 [Muss] * Der Nutzer muss für jeden Bezeichner-Knotentyp einen Kantentyp angeben können, durch den die Bezeichner-Knoten mit ihrem jeweiligen Gültigkeitsbereich-Knoten verbunden werden.

Nutzerspezifische semantische Aktionen

1.2.17 [Muss] * Es muss möglich sein, am Ende jedes Regelrumpfs eine beliebige vom Nutzer frei definierbare semantische Aktion anzugeben.
1.2.18 [Muss] * Vom Nutzer frei definierbare semantische Aktionen müssen aus syntaktisch korrektem Java-Code bestehen.
1.2.19 [Muss] * Der Nutzer muss auf die durch das Parsen des Regelrumpfs entstandenen Graphelemente und die erkannten Lexeme zugreifen können.
1.2.20 [Muss] * Es müssen globale Variablen definierbar sein, die von jeder frei definierbaren semantischen Aktion genutzt werden können.

4.1.3 Parsing

1.3.1 [Muss] * Wird beim Parsen eine Mehrdeutigkeit entdeckt, muss eine Exception geworfen werden.

4.1.4 Debugging

1.4.1 [Kann] * Falls eine Exception durch eine vom Nutzer definierten semantischen Aktion erzeugt wird, kann ihm mitgeteilt werden, welche semantische Aktion die Exception verursachte.
1.4.2 [Muss] * Sollte ein Eingabezeichen durch die definierte Grammatik nicht erkennbar sein, wird dies dem Nutzer mitgeteilt.
1.4.3 [Soll] * Im Falle von Mehrdeutigkeit sollen die Regeln, die diese verursachen, ausgegeben werden.

1.4.4 [Muss] * Wird vom Nutzer ein Graphelement-Typ oder Attribut angegeben, der bzw. das nicht im verwendeten TGraph-Schema existiert, muss eine Exception geworfen werden.

4.2 Nicht-funktionale Anforderungen

2.1 [Muss] * Es müssen Parsergeneratoren verwendet werden, die nach dem GLR- oder GLL-Algorithmus arbeiten.

2.2 [Muss] * Es muss ein externer Parsergenerator verwendet werden.

2.3 [Kann] * Der Parsergenerator sollte eine in Unicode kodierte Eingabe parsen können.

2.4 [Muss] * Der Parsergenerator muss einen in Java generierten Parser erzeugen können, bzw. einen in Java implementierten Interpreter verwenden.

2.5 [Muss] * Der Parsergenerator muss auf Windows, Linux und Mac OS lauffähig sein.

2.6 [Muss] * Bei der Programmierung und Dokumentation gelten die üblichen Regeln der Softwaretechnik, die hier nicht weiter erläutert werden.

5 Parsergeneratoren

Bei der Implementierung der in dieser Masterarbeit zu entwickelnden „Extractor Description Language" soll ein geeigneter Parsergenerator verwendet werden, um einen Extractor zu erzeugen. Aus diesem Grund wird in Abschnitt 5.1 zunächst eine Übersicht über mögliche Parsergeneratoren gegeben und einer ausgewählt, der dann in Abschnitt 5.2 näher betrachtet wird.

5.1 Übersicht über Parsergeneratoren

Bei der Suche nach Parsergeneratoren im Internet ist eine häufig genannte Empfehlung *ANTLR*[1]. Es ist in Java geschrieben und erhält eine EBNF-Grammatik als Eingabe, die um semantische Aktionen erweitert werden kann. Der resultierende Parser kann wahlweise in Java, C#, C++ oder Python generiert werden und arbeitet nach dem LL(k)-Algorithmus, der durch rekursiven Abstieg realisiert ist. Optional kann Backtracking aktiviert werden, wodurch die Mächtigkeit eines LL(*)-Parsers erreicht wird [ANT12, PF11]. Da für diese Masterarbeit LL(*) nicht ausreicht, wird ANTLR nicht weiter betrachtet.

Eine weitere Gruppe von Parsergeneratoren sind Generatoren für die sogenannten *„Packrat Parser"*, was so viel bedeutet wie sammelwütiger Parser. Ihr Input besteht aus einer erweiterten EBNF-Grammatik. Eine der Erweiterungen besteht darin, dass *Regeln und Alternativen angeordnet* sind. Das heißt, dass die Regel, die an oberster Stelle steht, bzw. die Alternative, die am weitesten links steht, zuerst ausgeführt wird. Sollte die Ausführung fehlschlagen, wird ein Backtracking durchgeführt. Des Weiteren werden Wiederholungen nach dem Prinzip des *„longest match"* interpretiert. Zusätzlich werden noch zwei neue Operatoren eingefügt: &*Expression* sowie !*Expression*. Mithilfe dieser Operatoren wird ein *Lookahead* realisiert, wodurch überprüft werden kann, ob die folgenden Eingabezeichen einen bestimmten Aufbau haben (&*Expression*) oder nicht (!*Expression*), ohne den Zeiger auf den nächsten zu erkennenden Eingabebuchstaben zu verändern [For02].

[1] `http://www.antlr.org/`

Durch diese Erweiterungen wird erreicht, dass sich Mehrdeutigkeiten in bestimmten Grammatiken auflösen lassen. Somit können „Packrat Parser“ nach einem *dem rekursiven Abstieg* ähnlichen Verfahren arbeiten, wodurch ein *linearer Zeitaufwand* erreicht wird. Allerdings benötigen sie einen *höheren Speicheraufwand* als die herkömmlichen LR- oder LL-Parser [For02]. Da trotz der erweiterten EBNF ihre Mächtigkeit geringer ist als die von GLL- oder GLR-Parsern werden auch Parsergeneratoren für „Packrat Parser“ nicht näher betrachtet.

Bei der Suche nach *GLR-Parsergeneratoren* lässt sich feststellen, dass es eine Vielzahl von Implementierungen gibt. So sind schon alleine auf der Internetseite [GLR12] bereits mehr als 10 verschiedene Parsergeneratoren aufgelistet, bei denen die Arbeitsweise der erzeugten Parser auf dem GLR-Algorithmus beruht. Um die Auswahl einzuschränken, werden nur die bekannteren GLR-Parsergeneratoren sowie diejenigen, deren Parser bzw. Interpreter in Java implementiert sind, kurz vorgestellt. Die Tabelle 5.1 gibt eine kurze Übersicht.

	Plattform	**Parser implementiert in**	**Metasprache**
The Meta-Environment	Linux	C	ASF+SDF
Stratego/XT (Spoofax)	alle	C/Java	ASF+SDF
Bison	alle	C/Java	BNF
Scannerless Boolean Parser	JVM	Java	EBNF
UltraGram	Windows	Java	BNF

Tabelle 5.1: Die GLR-Parsergeneratoren.

„The Meta-Environment“ . Als Eingabe erhält dieses Programm eine SDF-Definition, die um die „term rewriting“-Sprache ASF erweitert sein kann, wodurch die Notation von Transformationen ermöglicht wird [Met12a]. Aus dieser Eingabe wird zunächst eine Parsetabelle erzeugt, die zusammen mit den zu parsenden Daten dem im Programm enthaltenen GLR-Parser übergeben werden. Daraus wird dann ein Parse-Forest erzeugt [Met12c]. Die Programmkomponenten, die für die Erstellung der Parsetabelle und für das Parsen zuständig sind, sind in C implementiert und nur unter Linux lauffähig. Da jedoch die *Weiterentwicklung* des Programms im Jahre 2010 aufgrund des Nachfolgers Rascal (siehe unten) *eingestellt* wurde, wird „The Meta-Environment“ in dieser Masterarbeit nicht näher betrachtet [Met12b].

„Stratego/XT“ . Die Hauptaufgabe dieses Programms ist die Tree2Tree-Transformation von Eingabedaten. Hierzu muss die Syntax der Eingabe durch eine SDF-Definition beschrieben sein. Die Transformationen werden genauso wie beim zuvor beschriebenen „The Meta-Environment“ mithilfe von ASF notiert. Aus diesen Daten wird zunächst eine Parsetabelle generiert, bevor ein Parser mit ihr die Eingabedaten in

einen Parse-Forest umwandeln kann. Damit „Stratego/XT" besser nutzbar ist, wurde im Dezember 2011 das Eclipse-Plugin *Spoofax* veröffentlicht. In diesem ist der in C implementierte Parsetable-Generator für Linux, Windows und Mac OS sowie der in Java implementierte Parser enthalten [Str12b].

„Bison" ist ein Parsergenerator, der ursprünglich aus einer BNF-Syntaxbeschreibung nur einen in C implementierten LALR-Parser generieren konnte. In der aktuellen Version 2.5 besteht nun auch die Möglichkeit GLR-Parser zu erzeugen. Des Weiteren kam die Option hinzu, einen in Java implementierten Parser generieren zu lassen. Da sich letztere Option aber bisher nur *in einem experimentellen Status* befindet, wird auch „Bison" in dieser Masterarbeit nicht näher betrachtet [Bis12].

„Scannerless Boolean Parser" ist ein scannerlosen GLR-Parser. Er ist vollständig in Java geschrieben und generiert aus einer erweiterten EBNF-Definition eine Parsetable, die mithilfe eines in Java geschriebenen Interpreters eine Eingabe parsen kann [Meg06]. Da sich das Repository dieses Parsergenerators in einem *ungepflegten Zustand* befindet, wird auch dieses Programm nicht verwendet.

„UltraGram" . Hierbei handelt es sich um einen Parsergenerator, der aus einer erweiterten EBNF-Grammatik einen in Java implementierten GLR-Parser erzeugt. Da dieses Programm *nur unter Windows lauffähig* ist, wird es im Folgenden nicht weiter betrachtet [ult12].

Im Gegensatz zu den GLR-Parsergeneratoren gibt es nur wenige *Parsergeneratoren, die nach dem GLL-Algorithmus arbeiten*. Sie sind in der Tabelle 5.2 aufgelistet.

	Plattform	**Parser implementiert in**	**Metasprache**
Rascal	JVM	Java	Rascal
gll-combinators	JVM	Scala	Map-Funktionen

Tabelle 5.2: Die GLL-Parsergeneratoren.

„Rascal" ist der Nachfolger von „The Meta-Environment" und vollständig in Java implementiert. Als Eingabe dient eine Rascal-Beschreibung, welche eine Erweiterung von SDF und der „term rewriting"-Sprache ASF darstellt. Die Syntax-Beschreibung kann von einem Interpreter verarbeitet oder in einen generierten Parser überführt werden. Sowohl Interpreter als auch Parser sind in Java implementiert [Ras12a]. Ein formales Release ist bisher nicht veröffentlicht wurden, jedoch gibt es bereits eine herunterladbare Alpha-Version [Ras12b]. Da an dieser Version noch mit größeren Änderungen zu rechnen ist, wird dieses Programm nicht genauer untersucht.

„gll-combinators" ist eine Implementation des GLL-Algorithmus in Scala. Die Grammatik wird erstellt, indem jede BNF-Regel als Funktion definiert wird, die jede

Alternative im Regelrumpf auf ein beliebiges definierbares Ergebnis abbildet. Um eine Eingabe zu parsen, wird sie der entsprechenden Regel-Funktion der Grammatik übergeben und ein Ergebnis-Stream wird zurückgeliefert. Im Falle von mehrdeutigen Herleitungen wird zunächst ein Ergebnis zurückgeliefert. Die weiteren Ergebnisse werden dem „lazy evaluation"-Prinzip folgend nur bei Bedarf erzeugt [Spi12]. Aufgrund der *Implementation in Scala*, wird auch dieses Programm nicht weiter betrachtet.

Bei der Suche nach Parsergeneratoren, die in der Lage sind alle kontextfreien Sprachen zu parsen, konnte nur „Stratego/XT" als geeigneter Kandidat identifiziert werden und wird im folgenden Abschnitt näher untersucht.

5.2 Stratego/XT

Wie bereits im vorangegangenen Abschnitt erwähnt, handelt es sich bei *„Stratego/XT"* um ein Transformationsprogramm, das aus einer Eingabe mithilfe eines GLR-Parsers einen *Parse-Forest* erzeugt. Im Anschluss wird er mit den vom Benutzer definierten Transformationen umgebaut. Die für diese Masterarbeit relevante Anwendung von „Stratego/XT" besteht in der Erzeugung eines Parse-Forests aus einer Eingabe, wie es im nächsten Abschnitt beschrieben wird [str12a].

5.2.1 Allgemeine Funktionsweise

Um mithilfe von „Stratego/XT" Eingabedaten parsen zu können, muss ihre *Grammatik in der Metasprache SDF* beschrieben werden. Zur Erhöhung der Verständlichkeit werden für alle angegebenen Beispiele in den folgenden Abschnitten immer dieselbe Grammatik verwendet. Wie sie aussieht, ist in Abschnitt 5.2.2 beschrieben.

Wie Abbildung 5.1 zu sehen, muss die SDF-Beschreibung, die sich aus mehreren Modulen zusammensetzen kann, in eine *Parse-Tabelle* überführt werden. Hierzu wird das Programm sdf2Table[2] verwendet, welches zunächst die SDF-Regeln in die BNF überführt, damit der für das LR(k) Parsing verwendete Algorithmus zur Erzeugung einer Parse-Tabelle genutzt werden kann. Die erzeugte Tabelle wird zusammen mit den nummerierten BNF-Regeln in einer Datei gespeichert.

[2]Das in C geschrieben sdf2Table exisitiert für Windows, Linux und Mac OS, um Plattformunabhängigkeit zu erreichen. Zu finden ist es unter: `https://svn.strategoxt.org/repos/StrategoXT/spoofax-imp/trunk/org.strategoxt.imp.nativebundle/native`.

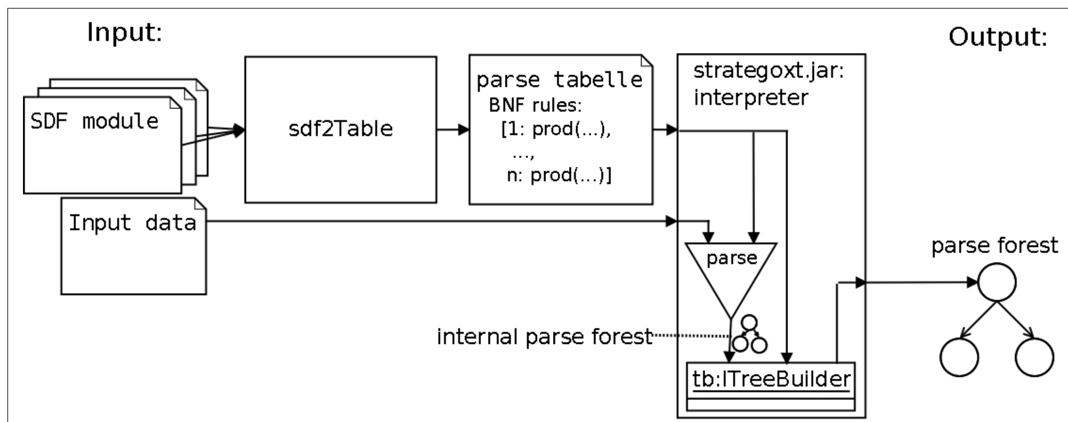


Abbildung 5.1: Das Parsen mithilfe von „Stratego/XT“.

Die soeben erzeugte Parse-Tabelle wird zusammen mit den Eingabedaten dem in der strategoxt.jar[3] enthaltenen *Interpreter* übergeben. In einem ersten Schritt parst dieser die Eingabedaten mithilfe der Parse-Tabelle nach dem GLR-Algorithmus und *erzeugt einen internen Parse-Forest*, der nur die verwendeten Regelnummern enthält. Daher bekommt der *ITreeBuilder* neben dem internen Parse-Forest auch die Parse-Tabelle mit den Regeln übergeben. Aus diesen Daten erzeugt er den *gewünschten Parse-Forest*, der als Ergebnis des Parse-Vorgangs dem Nutzer vom Interpreter zurückgeliefert wird.

Beim Start des Parse-Vorgangs besteht für den Nutzer die Möglichkeit dem Interpreter eine Instanz der drei mitgelieferten oder einer selbst erstellten *Implementation des Interfaces `ITreeBuilder`* mitzugeben. Wie die von den drei bereits existierenden `ITreeBuilder`n erzeugten Parse-Forests aussehen, wird in Abschnitt 5.2.5 beschrieben. Sollte kein Parse-Forest für diese Masterarbeit brauchbar sein, könnte eine eigene Implementation erstellt werden. Dies hätte den Vorteil, dass direkt beim Traversieren des internen Parse-Forests der gewünschte TGraph erzeugt werden kann. Um dies zu ermöglichen, enthält der Abschnitt 5.2.5 ebenfalls eine Beschreibung des `ITreeBuilder`-Interfaces und eine Erläuterung, wann die einzelnen Methoden vom Interpreter aufgerufen werden.

Da der `ITreeBuilder` den internen Parse-Forest übergeben bekommt, ist es bei einer eigenen Implementation notwendig, seinen Aufbau zu verstehen. Er wird in Abschnitt 5.2.4 beschrieben. Der *interne Parse-Forest* enthält nur die Nummern der *verwendeten Regeln*, weshalb der `ITreeBuilder` vom Interpreter über alle Regeln und ihre Nummern informiert wird. Da bei der Überführung in die BNF neue Regeln entstanden sind,

[3]Die in dieser Masterarbeit genutzte Version kann unter `http://hydra.nixos.org/build/2988379/download/2/strategoxt.jar` gefunden werden. Die aktuelle Version ist unter `http://hydra.nixos.org/job/strategoxt-java/strc-java-trunk/build` zu finden.

wird in Abschnitt 5.2.3 erklärt, wie diese SDF2BNF-Transformation funktioniert und wie die Regeln intern repräsentiert werden.

5.2.2 Eingabegrammatik

Wie bereits zuvor erwähnt, benötigt „Stratego/XT“ eine zur Definition in Abschnitt 3.2.3 konforme *SDF-Beschreibung als Eingabe*. Diese Form der Grammatik ermöglicht die Modularität, hat jedoch den Nachteil, dass die beschriebene Sprache nur die 256 Zeichen eines 8-Bit Encodings enthalten darf. Um dennoch die Verwendung von Unicode zu ermöglichen, wurde für „Stratego/XT“ ein Workaround angegeben[4], bei dem die nicht in ASCII enthaltenen Unicode-Zeichen auf die Zeichen mit den Dezimalwerten 253 bis 255 abgebildet werden. Unicode-Buchstaben werden auf 255, Unicode-Ziffern auf 254 und sonstige Unicode-Zeichen auf 253 abgebildet. Um beispielsweise Zahlen aus beliebigen Ziffern parsen zu können, kann folgende Regel genutzt werden:
`[0-9\254]* -> Number`

Des Weiteren muss beim Erstellen der Grammatik darauf geachtet werden, dass *keine unbegrenzte Mehrdeutigkeit* vorkommt, da sonst der zugrundeliegende GLR-Algorithmus nicht terminiert. Die in [Tom86] angegebene zyklische Grammatik, bei der der GLR-Algorithmus ebenfalls nicht terminieren sollte, konnte jedoch von „Stratego/XT“ geparst werden.

```
1  module Expression
2  imports Whitespace
3  imports Identifier
4  imports Number
5  exports
6    sorts Expression
7    context-free start-symbols Expression
8    context-free syntax
9      Identifier -> Expression {cons("Var")}
10     Number -> Expression {cons("Number")}
11     Expression "+" Expression -> Expression {cons("Plus")}
12     Expression "-" Expression -> Expression {cons("Minus")}
13     Expression "*" Expression -> Expression {cons("Mult")}
14     Expression "/" Expression -> Expression {cons("Div")}
15     "(" Expression ")" -> Expression {bracket}
```

Listing 5.1: Die SDF-Definition von Expression.

[4] `http://yellowgrass.org/issue/Spoofax/337`

Um den Aufbau der von „Stratego/XT“ erzeugten Parse-Forests zu veranschaulichen, wird das *Expression-Modul* in Listing 5.1 verwendet[5]. `Identifier` beschreibt einen Variablennamen, wie er in Java zulässig ist. `Number` repräsentiert eine Ganzzahl, die optional mit einem Vorzeichen versehen sein kann. Durch Zeile 7 wird erreicht, dass die syntaktische Variable `Expression` als Startpunkt der Herleitung jeder Eingabe genutzt wird. In den Zeilen 9 bis 15 werden neben Identifiern und Zahlen auch die Addition, Subtraktion, Multiplikation, Division sowie Klammerung von `Expression`s als eine `Expression` definiert. Indem auf die Angabe von Priorisierung und Assoziativität verzichtet wurde, konnte eine mehrdeutige Grammatik erstellt werden.

5.2.3 Erzeugung der Parse-Tabelle

Mithilfe des Tools *sdf2Table* können die im vorangegangen Abschnitt beschriebenen SDF-Module in eine Parse-Tabelle überführt werden. Mit der Angabe von `-m <Startmodul>` wird das Modul definiert, das die Startregel enthält, mit der die Eingabe geparst werden soll. Mit den Parametern `-c -p <searchSpace>` wird festgelegt, dass nach benötigten Modulen im `searchSpace` gesucht wird. Durch `-o <parseTable>.tbl` wird der Dateiname der zu erstellenden Parse-Tabelle bestimmt.

Jede Instanz des in Abschnitt 5.2.5 beschriebenen ITreeBuilder-Interfaces bekommt den internen Parse-Forest übergeben. Da dieser nur die Nummern der angewendeten Regel enthält, werden der Instanz ebenfalls die Regeln und ihre jeweilige Nummer mitgeteilt. Bei einer Implementation des Interfaces fällt auf, dass es mehr Regeln gibt, als in der Eingabegrammatik definiert wurden. Um dies zu verstehen, wird in Abschnitt 5.2.3.1 die von sdf2Table ausgeführte Transformation der Grammatikregeln in die BNF beschrieben. Wie die Regeln danach intern dargestellt werden, ist in Abschnitt 5.2.3.2 erläutert.

5.2.3.1 Regeltransformation in die BNF

Sdf2Table verwendet zur Erzeugung der Parse-Tabelle den gleichen Algorithmus wie LR(k)-Parser. Daher müssen die *SDF-Regeln in die BNF überführt* werden. Die Idee bei dieser Transformation ist die, dass alle SDF-Terme, die nicht in der BNF zulässig sind, als Variablen angesehen werden. Der Erhalt ihrer semantischen Bedeutung wird dadurch gewährleistet, dass für die neu erzeugten Variablen Regeln mit identischer Semantik generiert werden. So werden beispielsweise für die Option `A?` die Regeln `A -> A?` und `ε -> A?` erzeugt. Darüber hinaus werden Klammern um einen einzigen Term entfernt,

[5] Die Definition der importierten Module `Whitespace`, `Identifier` und `Number` sind in Abschnitt „Beispiel“ auf Seite 19 angegeben.

da sie in SDF keine Semantik haben. Wie die Transformation im Detail vonstatten geht, ist in [BKV07] definiert.

Bevor die Transformation im Detail erklärt wird, ist es notwendig zu wissen, dass sich eine SDF-Regeln aus einem Rumpf, einem Kopf und der Attributmenge zusammensetzt. Der Rumpf besteht aus beliebig vielen maximalen Termen. Der Kopf aus einem einzigen. Ein *maximaler Term* ist dabei ein Term, der in keinem anderen Term mehr enthalten ist. So stellt `"a"?` beispielsweise einen maximalen Term dar. `"a"` ist in diesem Beispiel auch ein Term jedoch kein maximaler, da es in dem Term `"a"?` enthalten ist.

In einem ersten Schritt werden für alle lexikalischen Startsymbole Regeln der Form

```
Startsymbol -> <START>
```

eingefügt. Für kontextfreie Startsymbole haben diese Regeln die Form

```
LAYOUT? Startsymbol LAYOUT? -> <START>.
```

Dadurch wird erreicht, dass es für den Parser ein einziges Startsymbol gibt, obwohl in der Grammatik mehrere definiert sein können. In der zuvor beschriebenen Beispielgrammatik käme auf diesem Weg die Regel

```
LAYOUT? Expression LAYOUT? -> <START>
```

hinzu.

Im Anschluss werden zwischen allen maximalen Termen eines kontextfreien Regelrumpfs `LAYOUT?` ergänzt. So wird beispielsweise aus der in Zeile 15 des Listing 5.1 angegebenen Regel

```
"("LAYOUT? Expression LAYOUT? ")"-> Expression {bracket}.
```

Für die in SDF zulässigen Terme in Regelrümpfen werden folgende Regeln solange erzeugt, bis ein Fixpunkt für die Regelmenge erreicht ist:

Literal. Das Literal `"a"` führt zur Generierung der Regel

```
[i] -> "a",
```

wobei `i` den Dezimalwert des Zeichens `a` darstellt.

Option. Für die Option `T?` werden die Regeln

```
ε -> T?
```

und

```
T -> T?
```

erzeugt.

Sequence. Die Sequence $(T_1 \ \ldots \ T_n)$ führt zur Erzeugung der Regel

$$T_1 \ \ldots \ T_n \ \text{->} \ (T_1 \ \ldots \ T_n).$$

Im Falle einer leeren Sequenz `()`, wird diese durch `empty` ersetzt und die folgende Regel wird erzeugt:

```
ε -> empty
```

Repetition. Für beide Repetitionsarten `T*` und `T+` werden die gleichen Regeln

```
ε -> T*
T* T* -> T* {left}
T+ -> T*
T -> T+
T* T+ -> T+
T+ T* -> T+
T+ T+ -> T+ {left}
```

generiert.

List. Die List `{T ","}*` oder `{T ","}+` führt zur Erzeugung von

```
ε -> {T ","}*
{T ","}* ","{T ","}* -> {T ","}* {left}
{T ","}+ -> {T ","}*
T -> {T ","}+
{T ","}* ","{T ","}+ -> T+
{T ","}+ ","{T ","}* -> T+
{T ","}+ ","{T ","}+ -> T+ {left}
```

Dabei kann an Stelle von `","` jeder beliebige andere Term verwendet werden.

Alternative. Für die Alternative $T_1|T_2$ werden die beiden Regeln

T_1 -> $T_1|T_2$

T_2 -> $T_1|T_2$

erzeugt.

LabeledTerm. Bei Termen, die mit einem Label versehen sind, wird das Label entfernt, da es in SDF keine Semantik besitzen.

Tuple. Tuple der Form <T_1, ..., T_n> werden in die Regel

"<"T_1 ","... ","T_n ">"-> <T_1, ..., T_n>

überführt.

FunctionTerm. Der FunctionTerm (T_1 ... T_n => T_m) -> T wird in die Regel

(T_1 ... T_n => T_m) "("T_1 ... T_n ")"-> T_m

transformiert.

Schließlich wird jeder maximale Term, welcher kein <START>, Literal oder Character-Class ist, mit *cf*(...) bzw. *lex*(...) umschlossen, je nachdem ob es sich um eine kontextfreie oder lexikalische Regel handelt. Für alle lexikalischen Terme *T* wird die Regel

lex(*T*) -> *cf*(*T*)

erzeugt, um beispielsweise die Verwendung einer lexikalischen Variablen in einer kontextfreien Regel zu erkennen.

5.2.3.2 Interne Regelrepräsentation

Bei der *Parse-Tabellen-Generierung* werden die Regeln in eine für den Interpreter verständliche Form überführt. Dabei wird für jede Regel

T_1 ... T_n -> T {A_1, ..., A_m}

ein Prädikat

$prod([sConv(T_1), ..., sConv(T_n)], sConv(T), attrs([aConv(A_1), ..., aConv(A_n)]))$

erzeugt. T_i, T sind maximale Terme und A_j Attribute. Zur besseren Verständlichkeit wurden $cf(...)$ bzw. $lex(...)$, die die maximalen Terme umschließen, weggelassen. Würden sie mit angegeben, sähe $sConv(T_1)$ beispielsweise so aus: $cf(sConv(T_1))$. Gibt es keine Attribute, so wird an Stelle von $attrs([aConv(A_1), ..., aConv(A_n)])$ der Wert *no-attrs* eingetragen.

$sConv(T)$ ist eine Funktion mit deren Hilfe die Terme in Prädikate überführt werden. Ihre Definition ist:

- $lit(T)$, falls T ein Literal "a" ist.
- $cilit(T)$, falls T ein Literal 'a' ist.
- $sort("T")$, falls T eine Variable ist.
- $char\text{-}class([i_1, ..., i_n])$, falls $T = [a_1...a_n]$ und i_k der Dezimalwert des ASCII-Zeichens a_k ist.
- $char\text{-}class([range(i, j)])$, falls $T = [a - b]$, i der Dezimalwert von a und j der Dezimalwert von b ist.
- $opt(sConv(S))$, falls $T = S?$ ist.
- $seq([sConv(S_1), ..., sConv(S_n)])$, falls $T = (S_1, ..., S_n)$ ist.
- $iter(sConv(S))$, falls $T = S+$ ist.
- $iter\text{-}star(sConv(S))$, falls $T = S*$ ist.
- $iter\text{-}sep(sConv(S_1), sConv(S_2))$, falls $T = \{S_1\ S_2\}+$ ist.
- $iter\text{-}star\text{-}sep(sConv(S_1), sConv(S_2))$, falls $T = \{S_1\ S_2\}*$ ist.
- $alt(sConv(S_1), sConv(S_2))$, falls $T = S_1 \mid S_2$ ist.
- $tuple(sConv(S_1), [sConv(S_2), ..., sConv(S_n)])$, falls $T =< S_1, S_2, ...S_n >$ ist.
- $func([sConv(S_1), ..., sConv(S_n)], sConv(S))$, falls $T = (S_1 \ldots S_n\text{=>}S)$ ist.

Die Transformation der Attribute geschieht mittels der Funktion $aConv(A)$. Ihre Definition ist:

- $assoc(a)$, falls A die Assoziativität a ausdrückt. a kann dabei die Werte `left`, `right`, `assoc` oder `non-assoc` haben.
- A, falls A eines der Attribute `avoid`, `prefer`, `reject` oder `bracket` ist.
- $term(A)$, sonst.

Wie die Prädikatrepräsentation der Regel in Zeile 10 des Listings 5.1 aussieht, ist in Formel (5.1) gezeigt.

$$prod([cf(sort(\text{"Number"}))], cf(sort(\text{"Expression"}))attrs([cons(\text{"Number"})])) \quad (5.1)$$

Wird eine solche transformierte Regel im Interpreter geladen, so erstellt er aus ihr einen Baum. Dabei wird aus jedem Prädikat ein Knoten, der seine Attribute als Kinder hat. Sollte ein Attribut aus einer Liste bestehen, so wird ein Listen-Knoten erzeugt, der seine Elemente als Kinder bekommt.

5.2.4 Interner Parse-Forest

Der in der strategoxt.jar enthaltene Interpreter erstellt mit der zuvor generierten Parse-Tabelle aus der Eingabe zunächst einen *internen Parse-Forest*, wie er durch den bereits in Abschnitt 3.4.2 vorgestellten GLR-Algorithmus erzeugt wird. Abbildung 5.2(a) zeigt, wie der Parse-Forest für die Eingabe

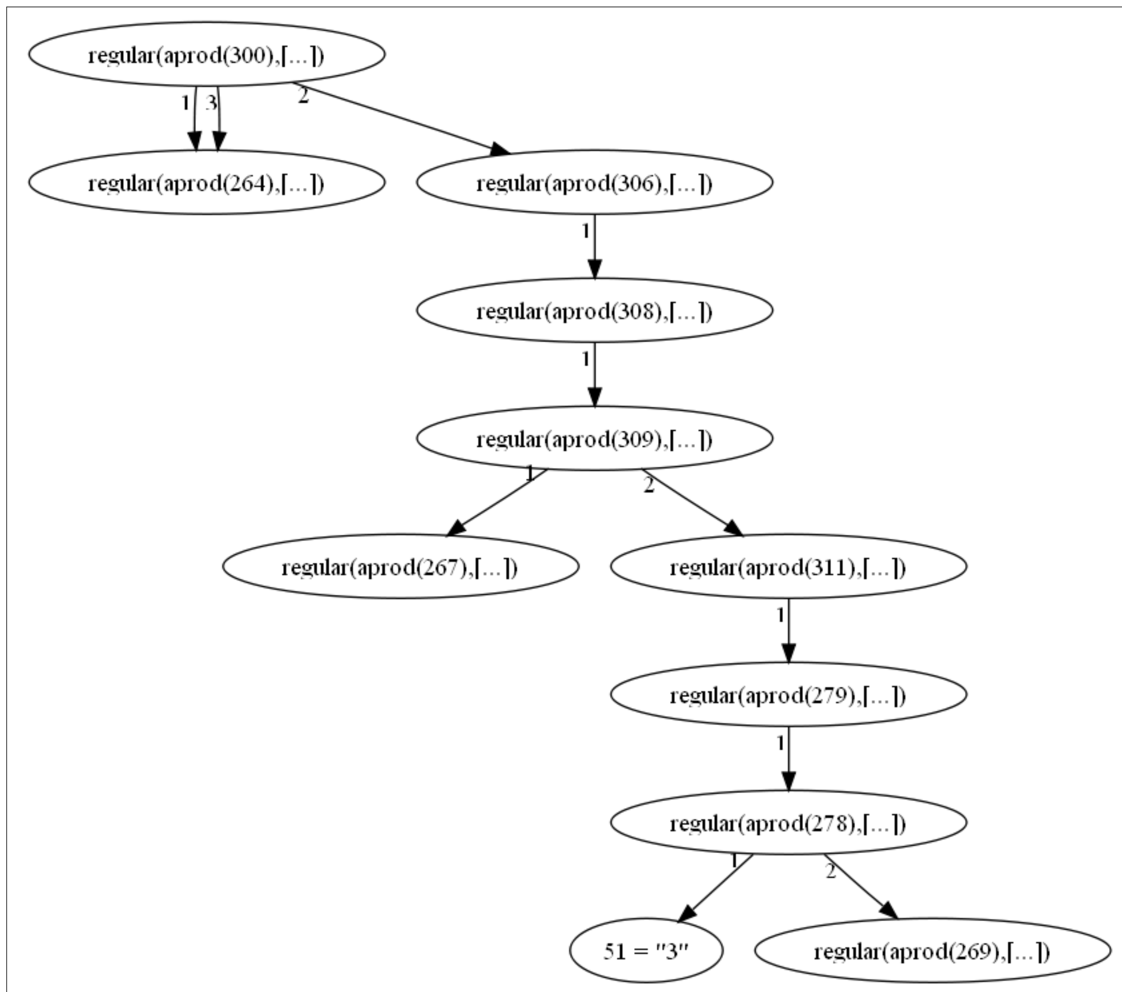
```
3
```

bei der in Abschnitt 5.2.2 erläuterten Beispielgrammatik aussieht.

In der Abbildung 5.2(a) sind zwei verschiedene Knotentypen gezeigt. Der *erste Typ* ist nur mit einem einzigen Knoten vertreten. Er ist mit `51 = "3"` beschriftet und repräsentiert ein erkanntes Eingabezeichen. Im internen Parse-Forest enthält dieser Knoten nur den Dezimalwert 51. Um eine bessere Verständlichkeit der Abbildung zu erreichen, wurde das entsprechende ASCII-Zeichen in der Abbildung mit angegeben.

Der *andere vertretene Knotentyp* ist beispielsweise mit `regular(aprod(300),[...])` beschriftet. `aprod(300)` steht für die angewendete Regel. Die beim dargestellten Parsevorgang benötigten Regeln mit ihren jeweiligen Nummern sind in der Tabelle 5.2(b) angegeben. `[...]` steht für die Liste der Kinder dieses Knotens. Für jeden maximalen Term im Regelrumpf exisitiert ein Kind im Parse-Forest. Die Reihenfolge der maximalen Terme entspricht der Anordnung der Kinder. Um diese Reihenfolge in der Abbildung zu verdeutlichen, sind an den Kantenanfängen die Inzidenznummern angegeben. Der *Prädikatname* `regular` bedeutet, dass es sich bei diesem Knoten um eine reguläre Regelanwendung handelt. *Darüber hinaus gibt es noch die Prädikatnamen*:

avoid(Rule,Children). Dieses Prädikat signalisiert, dass eine Grammatikregel, die bei der Definition mit dem Attribut `avoid` versehen wurde, angewendet wurde. Dies geschieht nur, wenn es ohne diese Regelanwendung keine alternative Herleitung für das Eingabewort mehr gäbe.



(a) Der interne Parse-Forest für die Eingabe 3.

Nummer	Regel
264	ε -> cf(LAYOUT?)
267	ε -> lex([+-]?)
269	ε -> lex([0-9]*)
278	[1-9] lex([0-9]*) -> lex(([1-9][0-9]*))
279	lex(([1-9][0-9]*)) -> lex([0]?|([1-9][0-9]*))
300	cf(LAYOUT?) cf(Expression) cf(LAYOUT?) -> <START>
306	cf(Number) -> cf(Expression) {cons("Number")}
308	lex(Number) -> cf(Number)
309	lex([+-]?) lex(UnsignedNumber) -> lex(Number)
311	lex([0]?|([1-9][0-9]*)) -> lex(UnsignedNumber)

(b) Die verwendeten Regeln zum Parsen der Eingabe 3.

Abbildung 5.2: Der interne Parse-Forest und die verwendeten Regeln für die Eingabe 3.

prefer(Rule,Children). `prefer` deutet an, dass eine mit `prefer` attributierte Regel genutzt wurde.

reject(Rule,Children). `reject` signalisiert, dass der Teilbaum, der einen Elternknoten mit diesem Prädikat besitzt, verworfen werden muss.

amb(Children). Durch einen mit diesem Prädikat versehenen Knoten, der für keine Regelanwendung steht, wird die Mehrdeutigkeit realisiert, die bei der Beschreibung des GLR-Algorithmus durch Unterknoten beschrieben wurde. Die Kinder dieses Knotens sind die Wurzelknoten aller möglichen Herleitungen.

cycle(Target). Ein Knoten mit diesem Prädikat repräsentiert Zyklen im Parse-Forest. Damit Traversierungsroutinen nicht in eine Endlosschleife geraten, hat dieser Knoten keine Kinder. Stattdessen wird die Rückwärtskante als ein von den Kindreferenzen unterscheidbarer Zeiger auf `Target` ausgedrückt.

Der Wurzelknoten des in Abbildung 5.2(a) gezeigten Parse-Forests hat als erstes und drittes Kind den Knoten (regular(aprod(264),[...]). Dies bedeutet, dass für das erste und dritte maximale Symbol des vom Wurzelknoten verwendeten Regelrumpfs dieselbe Herleitung existiert. Daher zeigen die erste und dritte Kante auf denselben Teilbaum.

5.2.5 `ITreeBuilder`

Der interne Parse-Forest wird zum Abschluss in einen *Parse-Forest* transformiert, der das *Ergebnis des Parse-Vorgangs* darstellt. Wie diese Transformation vonstatten geht, regelt das dem Interpreter mitgegebene `ITreeBuilder`-Objekt. Die Interaktion des Interpreters mit der `ITreeBuilder`-Instanz wird in Abschnitt 5.2.5.1 beschrieben. Wie die Methoden des `ITreeBuilder`-Interfaces definiert sind, wird in Abschnitt 5.2.5.2 näher betrachtet.

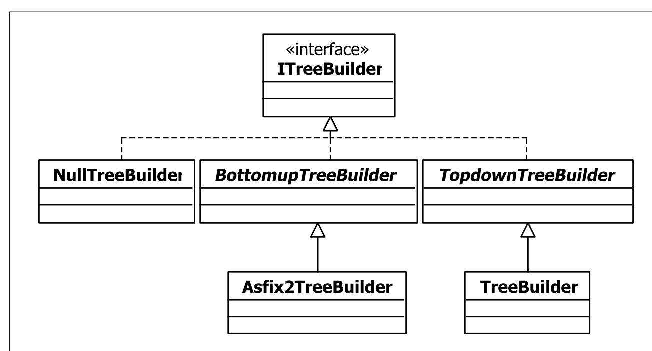


Abbildung 5.3: Das ITreeBuilder-Interface und seine Implementationen.

Abbildung 5.3 zeigt alle in „Stratego/XT“ enthaltenen Implementierungen des Interfaces `ITreeBuilder`. Der `NullTreeBuilder` erzeugt keinen Baum. Er wird eingesetzt,

wenn der Interpreter die Aufgabe eines Recognizers erfüllen soll. Der durch erzeugte Parse-Forest wird in Abschnitt 5.2.5.3 näher betrachtet. Mithilfe von `TreeBuilder` wird ein zusammengefallener Parse-Forest erzeugt. Wie dieser aussieht, wird in Abschnitt 5.2.5.4 beschrieben.

5.2.5.1 Interaktion des Interpreters mit dem `ITreeBuilder`

Die *Interaktion des Interpreters mit einem `ITreeBuilder`-Objekt* während dem Parse-Vorgang wird durch das Sequenzdiagramm in Abbildung 5.4 veranschaulicht.

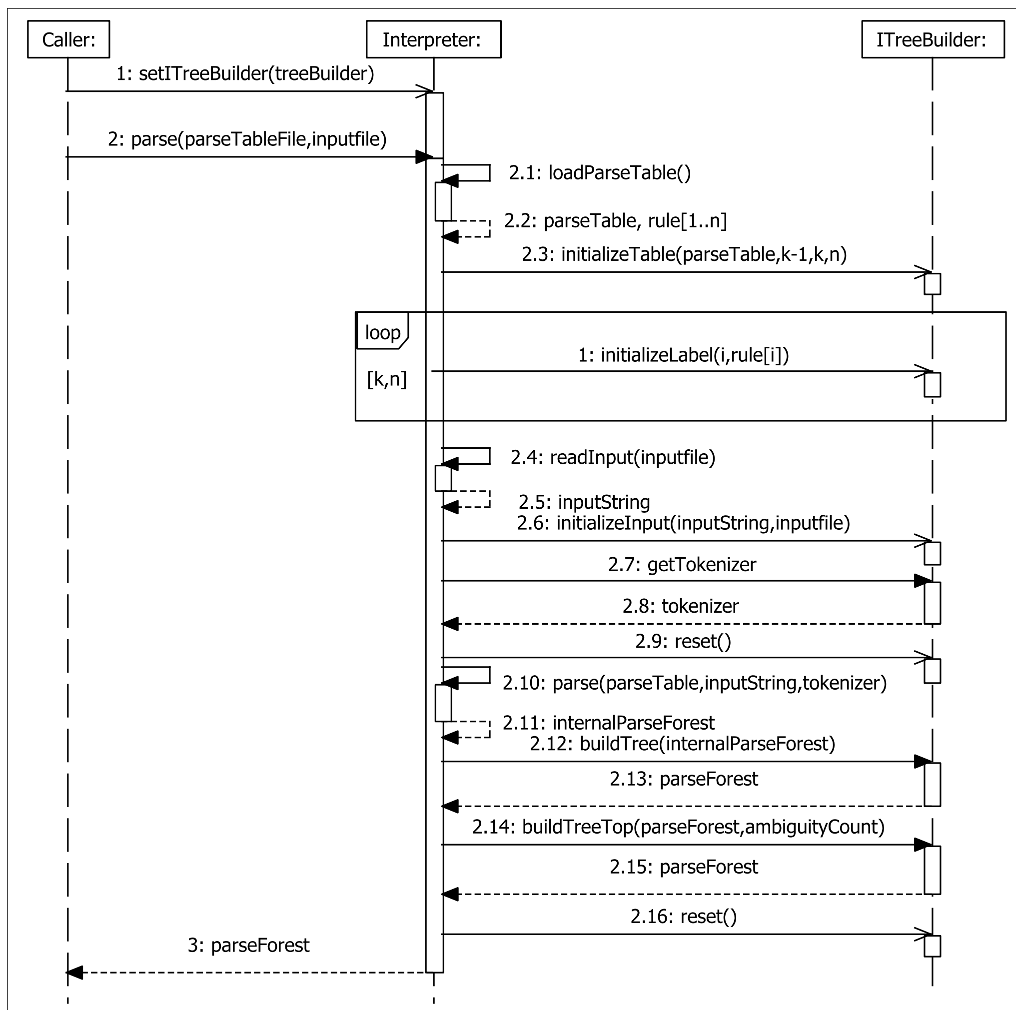


Abbildung 5.4: Die Verwendung des ITreeBuilders im Parse-Prozess.

In einem ersten Schritt teilt der Caller dem Interpreter mit, welcher `ITreeBuilder` verwendet werden soll. Danach ruft er die `parse()`-Methode auf und informiert den Inter-

preter dabei, wo die Parse-Tabelle zu finden ist (`parseTableFile`), mit deren Hilfe die angegebene Eingabedatei (`inputFile`) geparst werden soll.

Als Erstes lädt der Interpreter, die in `parseTableFile` gespeicherten Parse-Tabelle und Regeln. Durch den Aufruf von `initializeTable()` teilt er dem `ITreeBuilder` die soeben geladene Parse-Tabelle mit. Die letzten drei aktuellen Parameter geben die Anzahl der erkennbaren Zeichen (`k-1`) sowie die Nummer der ersten (`k`) und der letzten Regel (`n`) an. Da der im späteren Verlauf erzeugte interne Parse-Forest nur die Nummern der angewendeten Regeln enthält, wird dem `ITreeBuilder` nun mitgeteilt, hinter welcher Nummer sich welche Regel verbirgt. Hierzu wird für jede Regel nacheinander die Methode `initializeLabel` aufgerufen.

Nachdem dies geschehen ist, liest der Interpreter die Eingabedatei. Durch den Aufruf von `initializeInput` werden dem ITreeBuilder der Inhalt der Datei (`inputString`) und der Name der Eingabedatei (`inputfile`) mitgeteilt.

Im Anschluss holt sich der Interpreter mittels der `getTokenizer`-Methode den vom `ITreeBuilder` vorgeschlagenen Tokenizer. Falls `null` zurückgeliefert wird, verwendet der Interpreter seinen Default-Tokenizer. Bevor er den Parse-Vorgang mit dem entsprechenden Tokenizer startet, erfolgt der Aufruf von `reset()`. Als Ergebnis des Parsens hat der Interpreter den internen Parse-Forest (`internalParseForest`) erzeugt, welchen er dem `ITreeBuilder` durch die Methode `buildTree()` übergibt. Der resultierende `parseForest` wird an den Interpreter zurückgeliefert. Im Anschluss liefert er dem `ITreeBuilder` durch den Aufruf der Methode `buildTreeTop()` den soeben erhaltenen `parseForest` zusammen mit der Anzahl der im internen Parse-Forest enthaltenen Bäume (`ambiguityCount`). Das Ergebnis des soeben getätigten Methodenaufrufs liefert der Interpreter nach einem finalen `reset` dem Caller zurück.

5.2.5.2 Methoden des `ITreeBuilder`-Interfaces

`ITreeBuilder` ist ein Interface mit den folgenden Methoden, die in der UML-Notation angegeben werden:

initializeTable(table: ParseTable, productionCount: int, labelStart: int, labelEnd: int): void

Diese Methode bekommt mit dem formalen Parameter `table` die Parse-Tabelle des Interpreters übergeben. Die Bedeutung von `productionCount` ist weder dokumentiert noch wird es in einer Implementation verwendet. Da er für verschiedene Grammatiken konstant den Wert 256 hat, ließe sich vermuten, dass sie die Anzahl der ASCII-Zeichen darstellt. `labelStart` definiert die Nummer der ersten Regel

und hat immer den Wert 257. labelEnd gibt die Nummer der letzten Regeln wieder.

initializeLable(labelNumber: int, parseTreeProduction: IStrategoAppl): void
Diese Methode wird für jede vorhandene Regel nacheinander aufgerufen. Der formale Parameter parseTreeProduction steht für eine Baum-Repräsentation des in Abschnitt 5.2.3.2 eingeführten prod-Prädikates, welches für die Regel mit der in labelNumber angegebenen Nummer steht.

initializeInput(input: String, filename: String): void
Durch diese Methode bekommt der ITreeBuilder mit input die zu parsende Eingabe und mit filename den Dateinamen mitgeteilt, in dem sich die Eingabe befindet.

getTokenizer(): ITokenizer
Mithilfe dieser Methode kann dem Interpreter ein ITokenizer mitgegeben werden, der als Scanner für die Eingabe dient. Falls der default-Tokenizer verwendet werden soll, muss null zurückgeliefert werden.

reset(): void
Bevor der Parsevorgang gestartet wird, ruft der Interpreter diese Methode auf.

buildTree(node: AbstractParseNode): Object
Nachdem der Parsevorgang beendet ist, bekommt diese Methode durch den formalen Parameter node den internen Parse-Forest übergeben. Dieser wird bei durch diese Methode in den gewünschten Ausgabe-Parse-Forest überführt und zurückgeliefert. Es ist allerdings nicht vorgeschrieben, welches Object zurückgeliefert wird, da der Interpreter dieses Object nur weiterreicht ohne es zu verändern.

buildTreeTop(subtree: Object, ambiguityCount: int): Object
Durch diese Methode soll die Möglichkeit gegeben werden, eine neue Wurzel für den Parse-Forest zu erstellen. buildTreeTop() bekommt als subtree den von der Methode buildTree() zurückgelieferten Parse-Forest unverändert übergeben. Die Anzahl der im Parse-Forest enthaltenen Bäume wird durch den formalen Parameter ambiguityCount angegeben. Das Object, das diese Methode zurückliefert, stellt das Ergebnis des Parse-Vorganges des Interpreters dar.

5.2.5.3 Ausführlicher Parse-Forest

Der Asfix2TreeBuilder erzeugt einen *ausführlichen Parse-Forest*, der ähnlich dem internen Parse-Forest ist. Ein Unterschied besteht darin, dass die Prädikatenbezeichner wie beispielsweise regular nun einheitlich appl heißen. Darüber hinaus wird das Prädikat aprod(<Regelnummer>) durch den Baum der prod-Repräsentation ersetzt, welche

für die Regel mit der Nummer `<Regelnummer>` steht. Ein weiterer Unterschied besteht darin, dass die direkten Kinder eines Knotens im internen Parse-Forest durch einen Listenknoten ersetzt werden, der die Referenzen auf die Kinder hat. Es wird ebenfalls ein neuer Wurzelknoten erzeugt. Als Kinder erhält er die alte Wurzel sowie ein weiterer Knoten, in dem die Anzahl der Mehrdeutigkeiten angegeben sein soll. Allerdings ist dieser Wert immer 0 (siehe die Anmerkung zur `buildTreeTop`-Methode im vorangegangenen Abschnitt). Die Knoten des internen Parse-Forests, die Zyklen oder Mehrdeutigkeit anzeigen, bleiben erhalten.

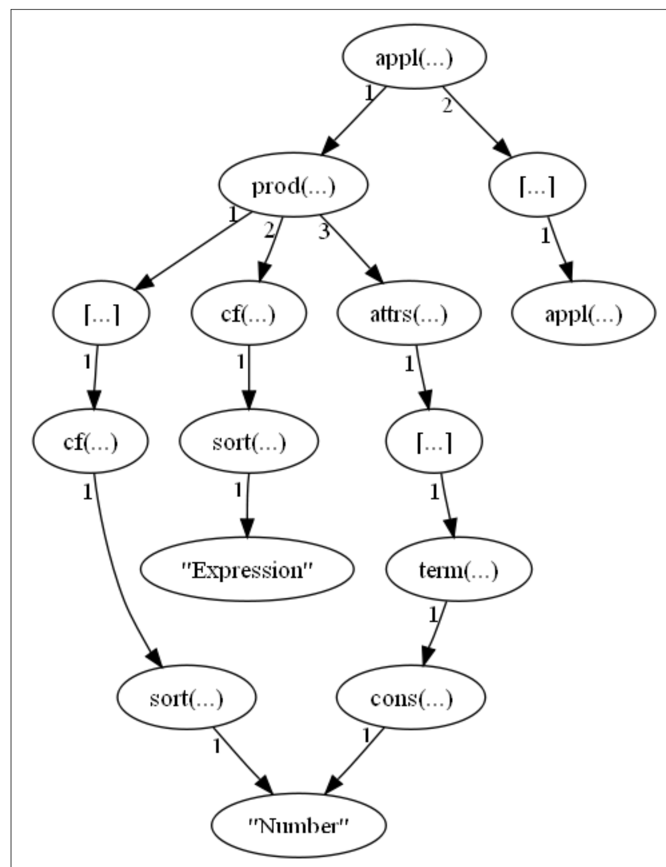


Abbildung 5.5: Ausschnitt aus dem Parse-Forest für die Eingabe 3.

Da der so entstandene Parse-Forest selbst für kleine Eingaben sehr groß wird, ist in Abbildung 5.5 nur die Repräsentation des Knotens `regular(aprod(306),[...])` gezeigt. Wie anhand des Knotens `"Number"` in der Abbildung zu erkennen ist, werden Teilbäume, die mehrfach benötigt werden, wiederverwendet.

Der ausführliche Parse-Forest der beim Parsen der Eingabe 3 entsteht, ist in der Prädikaten-Repräsentation in den Gleichungen (5.2) bis (5.12) angegeben. Um die Verständlichkeit zu erhöhen wurde an Stelle der prod-Prädikate die Regel angegeben.

(5.2) *parsetree*((5.3), 0)

(5.3) *appl*(cf(LAYOUT?) cf(Expression) cf(LAYOUT?) -> <START>, [(5.4), (5.5), (5.4)])

(5.4) *appl*(ε -> cf(LAYOUT?), [])

(5.5) *appl*(cf(Number) -> cf(Expression) {cons(Number)}, [(5.6)])

(5.6) *appl*(lex(Number) -> cf(Number), [(5.7)])

(5.7) *appl*(lex([+-]? lex(UnsignedNumber) -> lex(Number), [(5.8), (5.9)])

(5.8) *appl*(ε -> lex([+-]?), [])

(5.9) *appl*(lex([0] | ([1-9] [0-9]*)) -> lex(UnsignedNumber), [(5.10)])

(5.10) *appl*(lex([1-9] [0-9]*) -> lex([0] | ([1-9] [0-9]*)), [(5.11)])

(5.11) *appl*([1-9] lex([0-9]*) -> lex([1-9] [0-9]*), [3, (5.12)])

(5.12) *appl*(ε -> lex([0-9]*), [])

5.2.5.4 Zusammengefallener Parse-Forest

Beim *zusammengefallenen Parse-Forest* wird für jeden Knoten v des internen Parse-Forests, der für die *Anwendung einer `cons("V")`-attributierten Regel* steht, ein *Knoten mit der Beschriftung V erzeugt*. Allerdings werden nur die cons-Attribute von kontextfreien Regeln berücksichtigt. Sollte sich unterhalb von v ein weiterer Knoten w befinden, der eine `cons("W")`-attributierte Regel besitzt, so wird im zusammengefallenen Parse-Forest W zum Kind von V. Ließen sich aus v die Zeichen $\mathtt{a}_1$ bis $\mathtt{a}_n$ herleiten, ohne dass dabei eine Regel mit einem cons-Attribut angewendet werden muss, so werden diese Zeichen zu "$a_1 \ldots a_n$"zusammengefasst und als Blatt unter V gehängt. Es werden jedoch nur die Zeichen $\mathtt{a}_i$ berücksichtigt, die in einem „`lexical syntax`"-Block definiert wurden.

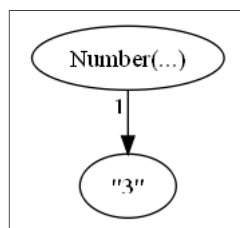


Abbildung 5.6: Der ParseTree für die Eingabe 3.

Abbildung 5.6 zeigt den zusammengefallenen Baum, der beim Parsen der Eingabe 3 entsteht. Wie zu erkennen ist, werden keine Anwendungen von lexikalischen Regeln oder Regeln ohne cons-Attribut dargestellt.

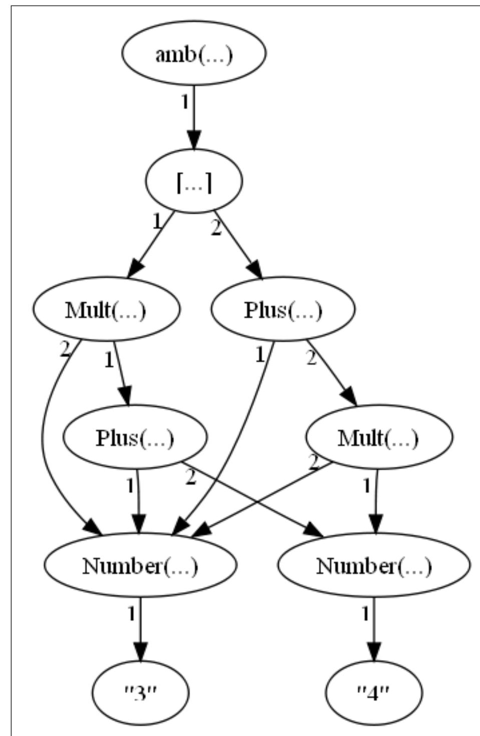


Abbildung 5.7: Der Parse-Forest für die Eingabe 3+4*3.

Der zusammengefallene Parse-Forest, der bei der Eingabe 3+4*3 entsteht. ist in Abbildung 5.7 zu sehen. An diesem Parse-Forest ist die mehrfache Verwendung des Teilbaumes *Number*("3") erkennbar. Darüber hinaus ist anhand des Wurzelknotens der Umgang mit Mehrdeutigkeit gezeigt.

6 Extractor Description Language (EDL)

Die „Extractor Description Language“ (EDL) stellt eine *Erweiterung der SDF-Grammatik um semantische Aktionen* dar. Bevor in Abschnitt 6.2 erläutert wird, welche Aktionen möglich sind, wird die EDL im folgenden Abschnitt anhand eines Beispiels veranschaulicht.

6.1 Beispiel

Um ein realistisches Beispiel einer „Extractor Description Language“-Anwendung anzugeben, wird in Abschnitt 6.1.2 eine EDL-Grammatik gezeigt, mit deren Hilfe eine TGraph-Repräsentation von einigen Java-Statements erstellt werden kann. Da die Betrachtung aller in der „Java Language Spezification“ [GJSB05] definierten Statements an dieser Stelle zu umfangreich wäre, wurden for-Schleifen, try-catch-Blöcke und switch-Anweisungen in diesem Beispiel ignoriert. Schließlich folgt in Abschnitt 6.1.3 eine Beschreibung, wie eine exemplarischer Java-Block anhand der EDL-Grammatik in einen Graph überführt wird.

Um den TGraphen, der mithilfe der EDL-Grammatik erzeugt werden soll, besser verstehen zu können, wird im folgenden Abschnitt zunächst das verwendete Schema vorgestellt.

6.1.1 TGraph-Schema

Das in diesem Abschnitt vorgestellte Schema entstammt dem grabaja-Tool[1] und stellt ein Metamodell von Java-Klassen dar. In Abschnitt „Regeln zur Namensvergabe“ wird zunächst beschrieben, nach welchen Regeln die Namen für die Knoten- und Kantenklassen vergeben wurden. Der zur Repräsentation der im Rahmen dieses Beispiels betrachteten Statements notwendige Schema-Teil wird in Abschnitt „Repräsentation von Statements“ vorgestellt.

[1] Mithilfe des Tools „graph based java“ (grabaja) ist es möglich, eine Java-Klasse in einen TGraphen und auch wieder zurück zu überführen. Das Tool ist unter der URL `https://github.com/jgralab/grabaja` zu finden. Das dazugehörige Schema befindet sich unter `https://github.com/jgralab/grabaja/blob/02d4eac29de4a1dc0d0c1f1c72a30d75315d6153/java5.tg`.

Regeln zur Namensvergabe

Knotenklassen werden nach dem repräsentierten Java-Konstrukt benannt. So werden beispielsweise If-Statements durch eine Knotenklasse vom Typ `If` repräsentiert. Um auszudrücken, dass es sich bei einem bestimmten Konstrukt um beispielsweise ein Statement handelt, wird eine abstrakte Knotenklasse `Statement` erzeugt, von der alle Repräsentanten von Statement-Konstrukten abgeleitet sind. So ist beispielsweise `If` ein Subtyp von `Statement`.

Die Namen der Kantenklassen beginnen immer mit einem Verb und sollen die Kantenrichtung erkennen lassen. Des Weiteren sollen der Bezeichner die Semantik der Kante widerspiegeln. Kanten mit ähnlicher Semantik bekommen einen gemeinsamen abstrakten Supertyp. So hat die abstrakte Kantenklasse `IsConditionOf` die Semantik, dass ein `Expression`-Knoten die Bedingung einer `Statement`-Instanz darstellen kann. Da sowohl If- als auch While-Statements Bedingungen haben, werden zwei Subtypen von `IsConditionOf` benötigt. Um die Eindeutigkeit ihrer Bezeichner zu bewahren, wird ihr Name um den Typ des Omega-Knotentyps erweitert. Somit ergeben sich zwei Klassen, die `IsConditionOfIf` und `IsConditionOfWhile` heißen.

Repräsentation von Statements

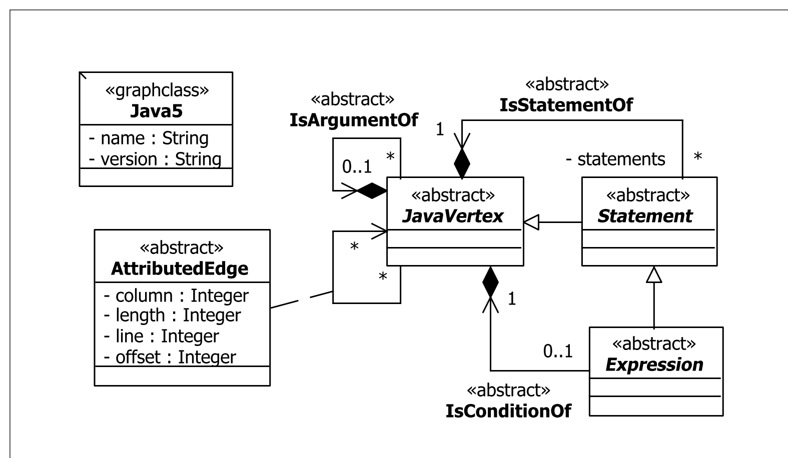


Abbildung 6.1: Die Graphklasse und die Top-Level-Knoten und -Kanten.

Abbildung 6.1 zeigt die Graphklasse `Java5`. Sie enthält den Namen und die Version des aktuellen Graphen. Die abstrakte Knotenklasse `JavaVertex` ist die Superklasse aller Knotentypen und die Kantenklasse `AttributedEdge` die Superklasse aller Kantentypen. Letztere besitzt Attribute, mit deren Hilfe die durch den Alpha-Knoten einer Kante

repräsentierten Zeichen in der geparsten .java-Datei identifiziert werden können. Alle weiteren in dieser Abbildung gezeigten Graphelement-Klassen werden entweder weiter unten erklärt oder stellen Generalisierungen von benötigten Kantenklassen dar.

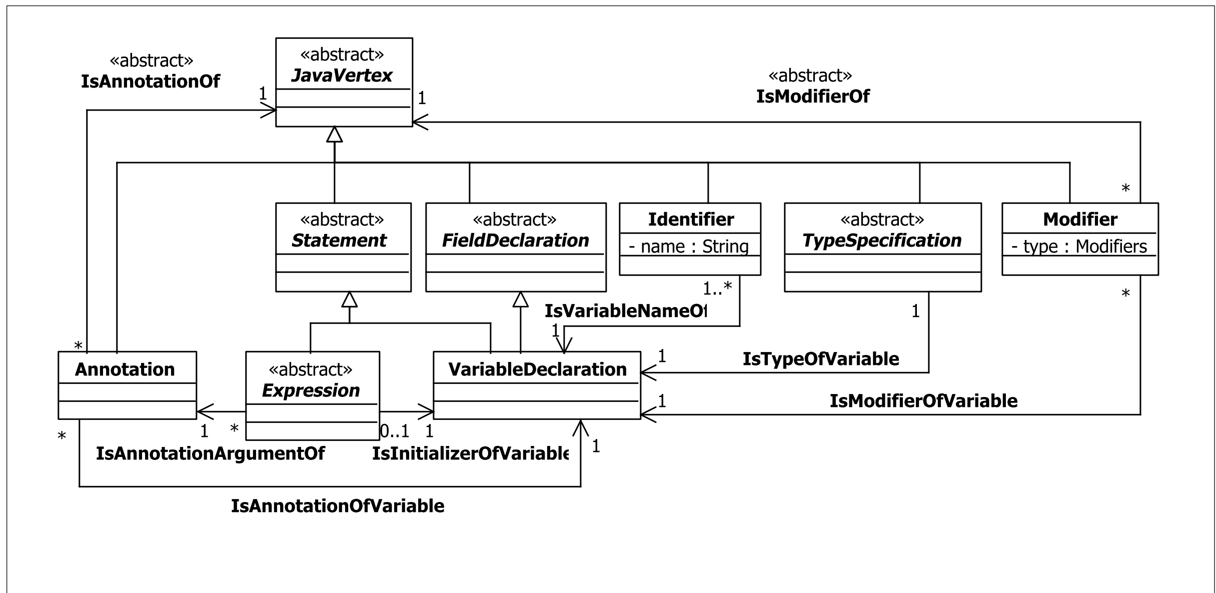


Abbildung 6.2: Die Repräsentation einer Variablen-Deklaration.

Variablen-Deklarationen der Form,

```
@SuppressWarnings("unused")
final String aVariable = "value";
```

die in einem Statement-Block vorkommen, werden durch `VariableDeclaration`-Knoten repräsentiert. Sollten mehrere Variablen definiert werden, so wird für jede Variable eine neue `VariableDeclaration`-Instanz erzeugt. Die Existenz einer Annotation wird durch eine `Annotation`-Instanz dargestellt, die mittels einer Kante vom Typ `IsAnnotationOfVariable` verbunden ist. Das Argument `"unused"` wird durch einen `Expression`-Knoten repräsentiert und mit der zuvor erwähnten `Annotation` durch eine Kante vom Typ `IsAnnotationArgumentOf` verbunden.

Der Modifier `final` wird durch einen `Modifier`-Knoten und der Variablentyp `String` durch eine `TypeSpecification`-Instanz repräsentiert. Der Name der Variablen wird im `name`-Attribut einer `Identifier`-Instanz gespeichert. Die Initialisierung `"value"` wird durch einen `Expression`-Knoten repräsentiert. Die vier soeben beschriebenen Knoten sind mit einer entsprechenden Kante des in der Abbildung gezeigten Typs mit dem `VariableDeclaration`-Objekt verbunden.

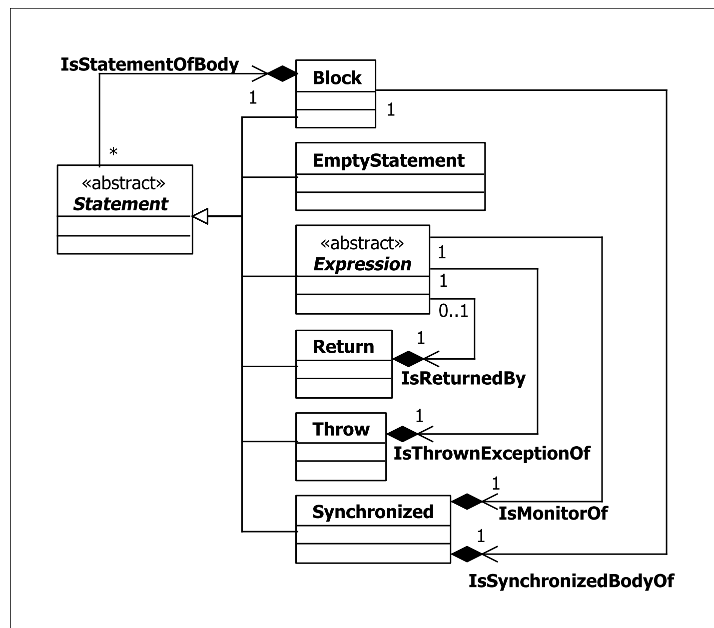


Abbildung 6.3: Die Repräsentation einiger weiterer Statements.

Abbildung 6.3 zeigt die Repräsentation von Statement-Blocks durch `Block`-Knoten. Die in diesem Block enthaltenen Statements sind durch `IsStatementOfBody`-Kanten mit den `Block`-Instanzen verbunden. Leere Statements werden durch `EmptyStatement`-Knoten und Statements, die nur aus einem Java-Ausdruck bestehen, durch `Expression`-Instanzen repräsentiert.

`return;` wird durch `Return`-Knoten dargestellt. Sollte ein Rückgabewert definiert sein, so wird sein Graph-Repräsentant mit einer `IsReturnedBy`-Kante angeschlossen. Throw-Statements werden in Instanzen der Knotenklasse `Throw` und des inzidenten Kantentyps `IsThrownExceptionOf` überführt.

Die Existenz von synchronisierten Statement-Blocks wird durch `Synchronized`-Knoten im Graphen dargestellt. Das jeweilige definierte Monitor-Objekt ist über eine Kante vom Typ `IsMonitorOf` erreichbar. Der enthaltene Statement-Block ist durch eine `IsSynchronizedBodyOf`-Instanz mit dem entsprechenden `Synchronized`-Objekt verbunden.

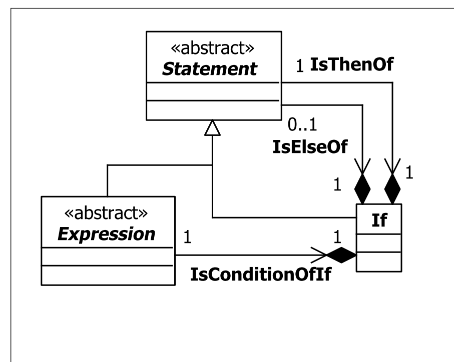


Abbildung 6.4: Die Repräsentation einer if-Verzweigung.

Die in Abbildung 6.4 dargestellte `If`-Knotenklasse repräsentiert If-Verzweigungen im Java-Code. Die Bedingung ist durch eine `IsConditionOfIf`-Kante und das Then-Statement durch eine Kante vom Typ `IsThenOf` erreichbar. Sollte ein Else-Statement existieren, so wird es mittels `IsElseOf`-Instanzen mit dem `If`-Knoten verbunden.

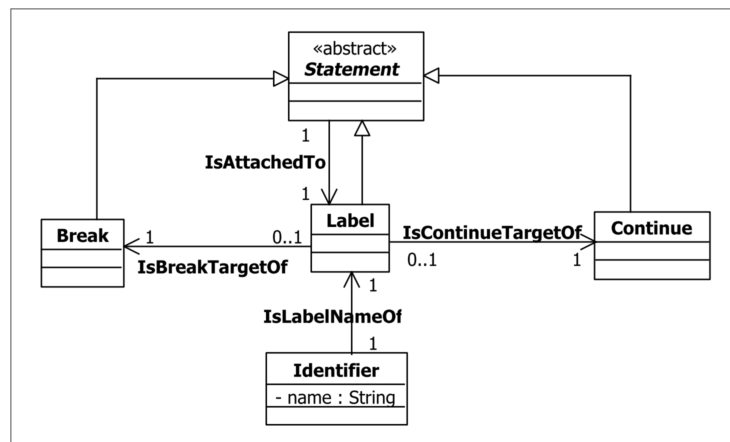


Abbildung 6.5: Die Repräsentation von Labels und ihrer Verwendung.

Wurde ein Label definiert, so wird dieses durch einen `Label`-Knoten repräsentiert. Das mit einem Label versehene Statement kann über eine Kante vom Typ `IsAttachedTo` erreicht werden, wie in Abbildung 6.5 zu sehen. Der Name des Labels ist im `name`-Attribut des zur `Label`-Instanz adjazenten `Identifier`-Objekts enthalten. Wird dieses Label in einem break- oder continue-Statement referenziert, so sind ihre Repräsentanten durch entsprechende Kanten mit dem `Label`-Knoten verbunden.

Abbildung 6.6 zeigt die Repräsentation von while- und do-while-Schleifen. Erstere werden durch `While`-Knoten dargestellt. Ihre Schleifenbedingungen sind über Kanten vom Typ `IsConditionOfWhile` und ihre Schleifenrümpfe über `IsLoopBodyOfWhile`-Kanten erreichbar. Da sich die Namen der Graphelement-Klassen, die eine do-while-Schleife

darstellen, nur durch das Infix `Do` unterscheiden, wird ihre Repräsentation hier nicht näher erläutert.

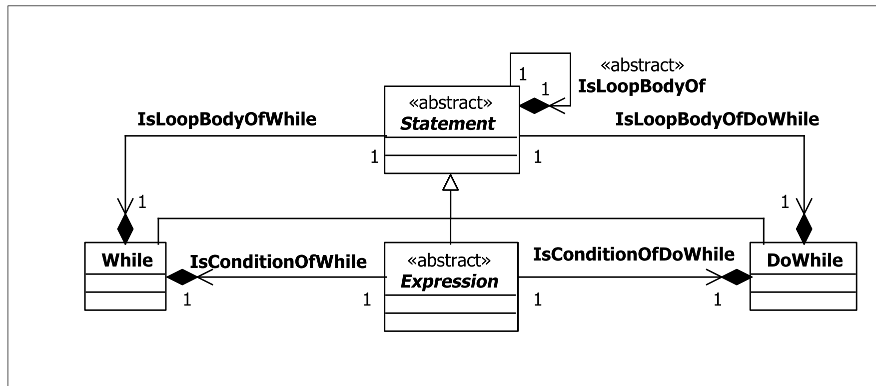


Abbildung 6.6: Die Repräsentation von while- und do-while-Schleifen.

6.1.2 EDL-Grammatik

Die dieser EDL-Grammatik zugrundeliegenden SDF-Module sind *im „Stratego/XT"-Repository*[2] zu finden. Ähnlich wie beim Schema im vorangegangenen Abschnitt, werden nur die zur Erkennung der betrachteten Java-Statements notwendigen Module aufgeführt. Welche Bestandteile des TGraph-Schemas innerhalb von EDL nutzbar sind, wird im Abschnitt 6.2 beschrieben. Um die Verständlichkeit zu erhöhen, wurden sie auf die für dieses Beispiel relevanten Definitionen reduziert und die EDL-Bestandteile hervorgehoben.

```
1  module java-15/Main
2  imports ...
3  schema de.uni_koblenz.jgralab.grabaja.java5schema.Java5Schema
4  default values
5    AttributedEdge.offset = offset(alpha);
6    AttributedEdge.line = line(alpha);
7    AttributedEdge.column = column(alpha);
8    AttributedEdge.length = length(alpha);
9  symbol tables
10   VariablesTable<Identifier>
11   LabelsTable<Label>
12 exports
13   context-free start-symbols CompilationUnit
```

Listing 6.1: Das EDL-Modul `Main`.

[2]Die SDF-Module, die den Java-Quellcode beschreiben, sind unter der URL `https://svn.strategoxt.org/repos/StrategoXT/java-front/trunk/syntax/src/languages/java-15` zu finden.

Listing 6.1 zeigt das *Startmodul* dieser EDL-Grammatik, von dem aus jeder Parse-Vorgang gestartet wird. Sein Name ist prinzipiell frei wählbar, wurde jedoch `Main` genannt, damit es vom Menschen besser als Startmodul identifiziert werden kann. Es importiert alle Module, die zum Erkennen von Java-Code notwendig sind. Da die Importe für dieses Beispiel keine weitere Bedeutung haben und nur die Lesbarkeit des Moduls verringern würden, wurden sie in Zeile 2 durch `...` ersetzt.

In Zeile 3 wird das im vorangegangenen Abschnitt vorgestellte Java-Schema als das in dieser Grammatik verwendete TGraph-Schema definiert. Da alle Kantenklassen von `AttributedEdge` abgeleitet sind, hat jede Kante die Attribute `offset`, `line`, `column` und `length`, mit deren Hilfe die durch den Alpha-Knoten repräsentierten Zeichen der Eingabedatei identifiziert werden können. Damit das *Setzen der entsprechenden Default-Werte der Attribute* nicht für jede erzeugte Kante erneut definiert werden muss, wird dies in dem **`default values`**-Block in den Zeilen 4 bis 8 vorgenommen. So wird beispielsweise in Zeile 5 definiert, dass für jede erzeugte Kante des Typs `AttributedEdge` oder eines Subtyps das `offset`-Attribut den `offset`-Wert des Alpha-Knotens erhalten soll. Auch wenn es in diesem Beispiel nicht vorkommt, so können in einem **`default values`**-Block beliebige Attribute mit Default-Werten versehen werden, wie es in Abschnitt 6.2.3.2 näher beschrieben wird.

Zum Parsen von Java-Statements werden für die Variablen und die Labels zwei verschiedene Symboltabellen benötigt. Daher werden im **`symbol tables`**-Block in den Zeilen 9 bis 11 zwei Symboltabellen deklariert. *In EDL besteht eine Symboltabelle aus einem Stack von Maps*, die einen eindeutig identifizierenden Wert auf einen Knoten abbildet. Jede Map repräsentiert dabei einen Namensraum, wie beispielsweise in einem Statement-Block im Rumpf einer While-Schleife. Die auf dem Stack darunter befindliche Map repräsentiert den Namensraum des Statement-Blocks, in dem sich die While-Schleife befindet.

In Zeile 10 wird eine Symboltabelle mit dem Namen *`VariablesTable`* deklariert. Durch diese Deklaration wird automatisch eine Symboltabelle mit leerem Stack angelegt. Um durch spätere semantische Aktionen eine neue Map auf den Stack zu legen oder ein Element in die Symboltabelle einzufügen, kann sie durch ihren Namen angesprochen werden. Innerhalb der spitzen Klammern in Zeile 10 wird definiert, dass in dieser Symboltabelle nur Knoten vom Typ `Identifier` eingefügt werden dürfen. In Zeile 11 wird eine weitere Symboltabelle *`LabelsTable`* angelegt, in der `Label`-Knoten eingefügt werden können.

Zeile 13 legt `CompilationUnit` als das Start-Symbol zum Parsen von vollständigen .java-Dateien fest. Im Folgenden wird jedoch nur die EDL-Grammatik für Statement-Blöcke angegeben.

```
1  module java-15/statements/Blocks
2  imports ...
3  global actions
4    pattern _(*) -> BlockStmt
5      #$=$0;#
6  exports
7    sorts BlockStmt Block
8    context-free syntax
9      @Symboltable{VariablesTable,LabelsTable}
10     rule "{" (BlockStmt #IsStatementOfBody($0,$);#)* "}" -> Block
11
12     rule LocalVarDeclStmt -> BlockStmt
13     rule Stmt             -> BlockStmt
14     ...
```

Listing 6.2: Das EDL-Modul `Blocks`.

Das in Listing 6.2 dargestellte Modul definiert einen *Statement-Block* in Java. Wie anhand der SDF-Regel in Zeile 10 zu erkennen, besteht er aus einer Folge von einzelnen Statements (`BlockStmt`), die von geschweiften Klammern umschlossen sind.

Jeder Block stellt in Java einen *eigenen Namensraum* für Variablen und Labels dar. Dieser Namensraum entspricht dem Teilbaum des Parse-Trees, der die Anwendung einer Block-Regel als Wurzel hat. Daher muss schon bei der Preorder-Traversierung des Parse-Trees eine neue Map erzeugt und auf den Symboltabellen-Stack gelegt werden. Wird bei der Postorder-Traversierung der Knoten, der für die Anwendung der Regel steht, wieder verlassen, so muss die Map vom Stack entfernt werden. Um dies zu erreichen, wird die Block-Regel mit einer `Symboltable`-Annotation versehen (Zeile 9), durch die die Symboltabellen `VariablesTable` und `LabelsTable` mit der Regel in Zeile 10 assoziiert werden.

Da der Regelkopf in Zeile 10 aus der syntaktischen Variablen `Block` besteht und ihr Name identisch mit dem Knotentyp `Block` ist, greift hier das *Default-Mapping*. Dadurch wird als erste semantische Aktion für den Regelkopf ein `Block`-Knoten erzeugt. Durch den Aufruf von `IsStatementOfBody($0,$)` wird sobald ein `BlockStmt` erkannt wurde, der hieraus synthetisierte Knoten (`$0`) durch eine `IsStatementOfBody`-Kante mit dem zuvor erzeugten `Block`-Knoten verbunden. Letzterer ist als Knoten des Regelkopfs durch `$` erreichbar.

In Zeile 12 wird eine lokale Variablen-Deklaration (Listing 6.4) und in Zeile 13 ein Statement (Listings 6.5 bis 6.8) als `Block-Stmt` definiert. Da in beiden Regeln der durch die

jeweilige Variable im Rumpf synthetisierte Wert als Ergebnis der Regel zurückgegeben werden soll, genügt es die identische semantische Aktion `$=$0;` nur einmal zu definieren. Hierzu wird sie in den Zeilen 3 bis 5 im **`global actions`**-Block aufgeführt, in dem semantische Aktionen für mehrere Regeln desselben Moduls definiert werden können. Um festzulegen, für welche Regeln diese Aktion auszuführen ist, werden durch das Pattern `_(*) -> BlockStmt` in Zeile 4 alle Regeln identifiziert, die einen beliebigen Rumpf haben und deren Kopf aus der syntaktischen Variablen `BlockStmt` besteht. Die Wildcard `_` steht für einen beliebigen Term und `_(*)` für eine beliebige Anzahl von ihnen.

```
1  module java-15/classes/FieldDeclarations
2  imports ...
3  global actions
4    pattern VarDeclId _(0..2) -> VarDecl
5      # $=VariableDeclaration();
6        IsVariableNameOf($0,$);
7      #
8  exports
9    sorts VarDecl VarDeclId VarInit ...
10   context-free syntax
11     rule VarDeclId             -> VarDecl
12     rule VarDeclId "=" VarInit -> VarDecl
13       #IsInitializerOfVariable($2,$);#
14
15     rule Id       -> VarDeclId
16       # $=Identifier();
17         $.name=$0;
18         VariablesTable.declare($0,$);
19       #
20     ...
21     rule Expr      -> VarInit
22       #$=$0;#
23     ...
```

Listing 6.3: Das EDL-Modul `FieldDeclarations`.

Um die semantischen Aktionen des EDL-Moduls, welches die *lokale Variablen-Deklaration* beschreibt (Listing 6.4), besser verstehen zu können, wird in Listing 6.3 zunächst definiert, wie die *Deklaration des Variablennamens* (Zeile 11) *und die optionale Initialisierung* (Zeile 12) in Java aufgebaut sind.

In Zeile 15 wird der Variablenname als `Id` definiert, was für einen gültigen Java-Bezeichner steht. Als Ergebnis dieser Regel wird ein `Identifier`-Knoten erzeugt (Zeile 16) und sein `name`-Attribut auf das durch die lexikalische Variable `Id` erkannte Lexem

(`$0`) gesetzt (Zeile 17). Da es sich bei dem soeben genannten Knoten um einen Repräsentanten einer Variablen handelt, muss er in die Symboltabelle `VariablesTable` eingefügt werden. Dies geschieht durch den Aufruf der `declare`-Methode in Zeile 18 (siehe Abschnitt 6.2.4). Durch den aktuellen Parameter `$0` wird ausgedrückt, dass das durch `$0` referenzierte Lexem den `Identifier`-Knoten eindeutig identifiziert.

In Zeile 21 wird der Wert, mit dem eine Variable initialisiert wird, als Java-Ausdruck definiert. Der durch die kontextfreie Variable `Expr` synthetisierte Knoten wird in Zeile 22 als Ergebnis der Regel festgelegt.

Für die beiden `VarDecl`-Regeln (Zeile 11 und 12) muss ein `VariableDeclaration`-Knoten synthetisiert werden (Zeile 5) und mittels einer `IsVariableNameOf`-Kante mit dem durch `VarDeclId` synthetisierten `Identifier`-Knoten verbunden werden (Zeile 6). Um diese semantischen Aktionen nicht mehrfach definieren zu müssen, wurden sie in dem **`global actions`**-Block in den Zeilen 4 bis 7 aufgeführt. Das Pattern in Zeile 4 beschreibt Regeln, deren Rumpf aus `VarDeclId` gefolgt von 0 bis 2 beliebigen Termen (`_(0..2)`) und deren Kopf aus `VarDecl` besteht.

```
1  module java-15/statements/LocalVariableDeclarations
2  imports ...
3  exports
4    sorts LocalVarDeclStmt LocalVarDecl
5    context-free syntax
6      rule LocalVarDecl ";" -> LocalVarDeclStmt {prefer}
7        #$=$0;#
8
9      rule
10     # $anno=list();
11       $mod=list();
12     #
13      (
14        (Anno #$anno.add($0);#)
15        | (VarMod #$mod.add($0);#)
16      )*
17      Type
18      {VarDecl "," #lift($0);#}+
19     # $=$2;
20       IsAnnotationOfVariable($anno,$);
21       IsModifierOfVariable($mod,$);
22       IsTypeOfVariable($1,$);
23     #
24     -> LocalVarDecl {prefer}
```

Listing 6.4: Das EDL-Modul `LocalVariableDeclarations`.

Listing 6.4 definiert, wie in Java *lokale Variablen deklariert* werden. Bei der Ausführung der semantischen Aktionen im Rumpf der in den Zeilen 9 bis 24 definierten Regel, werden in Zeile 10 und 11 zunächst zwei leere Listen angelegt und den beiden temporären Variablen `$anno` und `$mod` zugewiesen. Für jede durch `Anno` erkannte Annotation wird der synthetisierte `Annotation`-Knoten an die erste Liste gehängt (Zeile 14). Die durch `VarMod` synthetisierten `Modifier`-Knoten werden in die zweite Liste eingefügt (Zeile 15).

In Zeile 18 wird definiert, dass nur die durch `VarDecl` synthetisierten Knoten vom Typ `VariableDeclaration` in der durch `{VarDecl ","}+` erzeugten Liste *list* enthalten sind. Im Anschluss wird *list* als Ergebnis der Regel festgelegt (Zeile 19). Danach wird in Zeile 20 jeder `Annotation`-Knoten (`$anno`) durch eine `IsAnnotationOfVariable`-Kante und in Zeile 21 jeder `Modifier`-Knoten (`$mod`) durch jeweils eine Kante vom Typ `IsModifierOfVariable` mit jedem Knoten des Regelkopfs verbunden. Zum Schluss wird der erkannte Typ der Variablendeklaration (`$1`) per `IsTypeOfVariable`-Kante an die `VariableDeclaration`-Instanzen gehängt (Zeile 22).

Für die Regel in Zeile 6 wird die soeben synthetisierte Liste von `VariableDeclaration`-Knoten dem Regelkopf zugewiesen (Zeile 7). Diese Liste wird in Listing 6.2 als Ergebnis der `BlockStmt`-Regel zurückgegeben (Zeile 13), und in Zeile 10 wird jeder enthaltene Knoten durch eine `IsStatementOfBody`-Kante mit dem `Block`-Knoten verbunden.

```
1  module java-15/statements/Statements
2  imports ...
3  global actions
4    pattern "if" "(" Expr ")" Stmt _(0..2) -> Stmt
5      # $=If();
6        IsConditionOfIf($2,$);
7        IsThenOf($4,$);
8      #
9  exports
10   sorts Stmt  ...
11   context-free syntax
12    rule "if" "(" Expr ")" Stmt                 -> Stmt {prefer}
13    rule "if" "(" Expr ")" Stmt   "else" Stmt -> Stmt
14      #IsElseOf($6,$);#
```

Listing 6.5: Das EDL-Modul `Statements` (Teil 1).

Das *EDL-Modul* `Statements` ist aufgrund der Übersichtlichkeit auf vier Listings aufgeteilt. Listing 6.5 zeigt den ersten Teil, indem die If-Statements definiert werden. Da beide If-Regeln (Zeile 12 und 13) mit den identischen Termen `"if""("Expr ")"Stmt` begin-

nen, müssen dieselben semantischen Aktionen für beide Regeln definiert werden. Um Redundanz zu vermeiden, wird in den Zeilen 3 bis 8 ein **global actions**-Block definiert, mit dessen Hilfe semantische Aktionen nur einmal definiert werden brauchen und dann für alle Regeln ausgeführt werden, auf die ein bestimmtes Pattern zutrifft. Hierzu wird in Zeile 4 zunächst das besagte Pattern angegeben, durch das die beiden If-Regeln identifiziert werden können. _ ist dabei eine Wildcard, die für einen beliebigen Term steht und `_(0..2)` repräsentiert 0 bis 2 beliebige Terme. Für jede Regel, auf die dieses Pattern zutrifft, wird zunächst ein neuer `If`-Knoten dem Regelkopf zugewiesen (Zeile 5). Der durch `Expr` synthetisierte Knoten wird mittels einer `IsConditionOf`-Kante und der von `Stmt` zurückgegebene Knoten mittels einer `IsThenOf`-Kante mit dem Regelkopf verbunden.

Die Regel in Zeile 13 definiert einen zusätzlichen Else-Teil. Nachdem die Pattern-spezifischen semantischen Aktionen ausgeführt wurden, wird in Zeile 14 der den Else-Teil repräsentierende Knoten mittels einer `IsElseOf`-Kante mit dem `If`-Knoten des Regelkopfs verbunden.

```
15  rule Block      -> Stmt #$=$0;#
16  rule ";"        -> Stmt #$=EmptyStatement();#
17  rule Expr ";"   -> Stmt #$=$0;#
18
19  rule "return" #$=Return();#
20  (Expr #IsReturnedBy($0,$);#)?
21  ";" -> Stmt
22
23  rule "throw" Expr ";" -> Stmt
24    # $=Throw();
25      IsThrownExceptionOf($1,$);
26    #
27
28  rule "synchronized" "(" Expr ")" Block -> Stmt
29    # $=Synchronized();
30      IsMonitorOf($2,$);
31      IsSynchronizedBodyOf($4,$);
32    #
```

Listing 6.6: Das EDL-Modul `Statements` (Teil 2).

Der zweite Teil des `Statements`-Moduls in Listing 6.6 definiert einen Statement-Block als Statement (Zeile 15). Der durch `Block` (siehe Listing 6.2) synthetisierte Knoten wird dem Regelkopf zugewiesen. Für das leere Statement wird ein `EmptyStatement`-Knoten

erzeugt (Zeile 16) und für ein Expression-Statement wird der synthetisierte `Expression`-Knoten zurückgeliefert (Zeile 17).

Für Return-Statements wird zunächst ein `Return`-Knoten erzeugt, an den der optionale Ausdruck `Expr` durch eine `IsReturnedBy`-Kante angebunden wird, sofern dieser existiert (Zeile 19 bis 21). Bei Throw- (Zeile 23 bis 26) und Synchronized-Statements (Zeile 28 bis 32) werden entsprechende Knoten v für den Regelkopf erzeugt und die im Regelrumpf synthetisierten Elemente werden durch entsprechende Kanten mit v verbunden.

```
33  rule Id ":" Stmt -> Stmt
34    # $=Label();
35      LabelsTable.declare($0,$);
36      $ident=Identifier();
37      $ident.name= $0;
38      IsLabelNameOf($ident,$);
39      IsAttachedTo($2,$);
40    #
41
42  rule
43  "break" #$=Break();#
44  (Id #IsBreakTargetOf(LabelsTable.use($0),$);#)?
45  ";" -> Stmt
46
47  rule
48  "continue" #$=Continue();#
49  (Id #IsContinueTargetOf(LabelsTable.use($0),$);#)?
50  ";" -> Stmt
```

Listing 6.7: Das EDL-Modul `Statements` (Teil 3).

Der in Listing 6.7 dargestellte dritte Teil des `Statements`-Moduls beschreibt die Deklaration und Verwendung von Labels in Java. Ersteres wird durch die Regel in Zeile 33 definiert. Hierfür wird zunächst in Zeile 34 ein `Label`-Knoten erzeugt. Dieser wird in die Symboltabelle `LabelsTable` durch Aufruf der Methode `declare()` eingefügt (Zeile 35). Seine eindeutige Identifizierung geschieht über das durch die lexikalische Variable `Id` erkannte Lexem (`$0`). Im Anschluss wird in Zeile 36 der temporären Variable `$ident` ein neuer `Identifier`-Knoten zugewiesen wird. Sein `name`-Attribut wird in Zeile 37 entsprechend gesetzt. Über eine `IsLabelNameOf`-Kante wird er mit dem `Label`-Knoten verbunden (Zeile 38), dessen zugehöriges Statement mittels einer `IsAttachedTo`-Kante angeschlossen wird (Zeile 39).

Für ein erkanntes Break-Statement wird in Zeile 43 ein `Break`-Knoten erzeugt. Als nächstes muss der durch die Variable `Id` identifizierte Label-Knoten bestimmt werden. Hierzu wird in Zeile 44 durch den Aufruf der `use`-Methode in der Symboltabelle `LabelsTable` nach einem `Label`-Knoten gesucht, der über den Wert von `Id` identifiziert wird. Dieser wird durch eine Kante vom Typ `IsBreakTargetOf` mit dem Regelkopf verbunden. Die semantischen Aktionen für Continue-Statements sind bis auf die Bezeichner der erzeugten Graphelemente identisch.

```
51  rule "while" "(" Expr ")" Stmt -> Stmt
52    # $=While();
53      IsConditionOfWhile($2,$);
54      IsLoopBodyOfWhile($4,$);
55    #
56
57  rule "do" Stmt "while" "(" Expr ")" ";" -> Stmt
58    # $=DoWhile();
59      IsConditionOfDoWhile($4,$);
60      IsLoopBodyOfDoWhile($1,$);
61    #
62  ...
```

Listing 6.8: Das EDL-Modul `Statements` (Teil 4).

Listing 6.8 enthält den vierten und letzten Teil des `Statements`-Moduls. In Zeile 51 wird ein While-Statement definiert. Zunächst wird für den Regelkopf ein `While`-Knoten erzeugt (Zeile 52), mit dem der die Abbruch-Bedingung repräsentierende `Expression`-Knoten durch eine `IsConditionOfWhile`-Kante (Zeile 53) und der den Schleifenrumpf repräsentierende `Statement`-Knoten durch eine `IsLoopBodyOfWhile`-Kante verbunden werden (Zeile 54). Für das in Zeile 57 dargestellte Do-While-Statement wird analog verfahren.

6.1.3 Erzeugung des Graphen

Mithilfe der im vorangegangenen Abschnitt gezeigten EDL-Grammatik soll nun der in Listing 6.9 gezeigte Java-Block geparst werden.

```
1  {
2     @SuppressWarnings("unused")
3     final int i;
4  }
```

Listing 6.9: Der zu parsende Java-Block.

Zunächst wird eine neue Instanz der `Java5`-Graphklasse erzeugt, sowie zwei neue Symboltabellen angelegt, wie sie im `Main`-Modul (Zeile 9 bis 11 des Listings 6.1) definiert wurden. Da es im exemplarischen Java-Block keine Label gibt, wird im Folgenden nur die Symboltabelle mit Namen `VariablesTable` aufgeführt.

`VariablesTable` = [{}]
`Graph` = ∅

Die erste bearbeitete Regel ist die `Block`-Regel in Zeile 10 des Listings 6.2. Zunächst wird aufgrund der `Symboltable`-Annotation eine neue Map in die `VariablesTable` eingefügt. Im Anschluss wird wegen des Default-Mappings ein `Block`-Knoten im Graph erzeugt.

`VariablesTable` = [{}, {}]
`Graph` =

v1:Block

Die nächsten zur Herleitung der Eingabe angewendeten Regeln sind die `BlockStmt`- (Zeile 12 in Listing 6.2) und die `LocalVarDeclStmt`-Regeln (Zeile 6 in Listing 6.4). Für sie werden zunächst keine semantischen Aktionen ausgeführt.

Im Zuge der Abarbeitung der `LocalVarDecl`-Regel (Zeilen 9 bis 24 in Listing 6.4) werden zunächst zwei leere Listen erzeugt und den beiden temporären Variablen `$anno` und `$mod` zugewiesen.

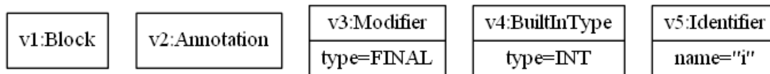
Da sich in der Eingabe eine `SupressWarnings`-Annotation befindet, wird von der nicht aufgeführten `Anno`-Regel ein `Annotation`-Knoten im Graph synthetisiert und in die `$anno`-Liste eingefügt. Analoges geschieht für den `final`-Modifier. Die Eingabe `int` wird durch die `Type`-Regel erkannt, die einen `BuildInType`-Knoten synthetisiert.

`VariablesTable` = [{}, {}]
`Graph` =

v1:Block | v2:Annotation | v3:Modifier (type=FINAL) | v4:BuiltInType (type=INT)

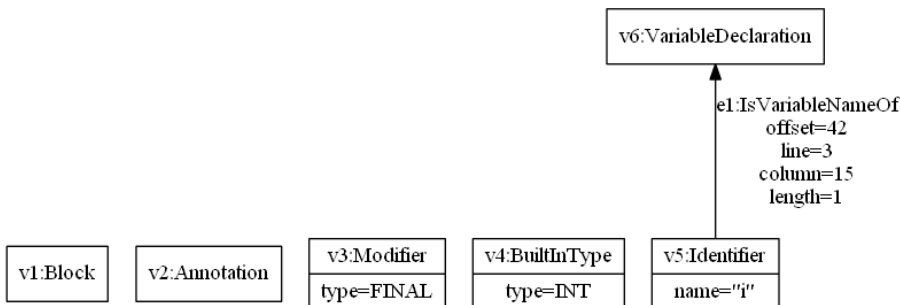
Als nächstes wird wird die Variablen-Deklarationen `i` erkannt. Hierzu wird zunächst die `VarDecl`-Regel in Zeile 11 des Listings 6.3 und im Anschluss die `VarDeclId`-Regel (Zeilen 15 bis 19) angewendet. Bei der Anwendung der letzteren, wird ein neuer Knoten $v4$ vom Typ `Identifier` erzeugt. Bevor er in die Symboltabelle `VariablesTable` eingefügt wird, erhält sein `name`-Attribut das durch `Id` erkannte Lexem `i`. Schließlich wird $v5$ zurückgegeben.

`VariablesTable` = [{}, {`"i"->`$v5$}]
`Graph` =



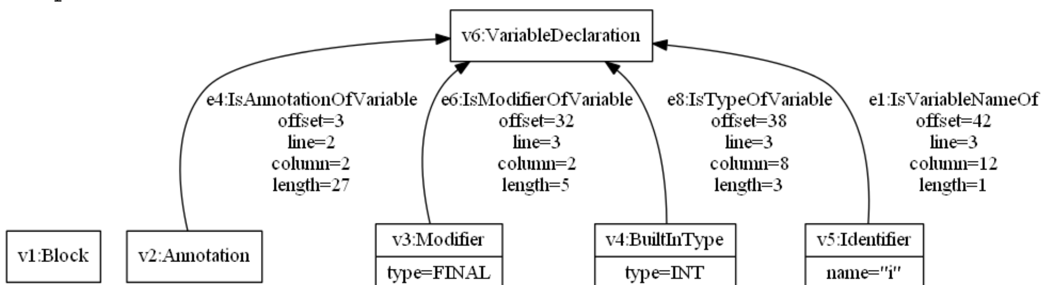
Bevor die `VarDecl`-Regel verlassen wird, wird ein `VariableDeclaration`-Knoten $v6$ erzeugt, der durch eine `IsVariableNameOf`-Kante mit $v5$ verbunden wird. Durch die **`default values`**-Deklaration im Startmodul (Zeilen 4 bis 8 in Listing 6.1) werden die Attribute `offset`, `line`, `column` und `length` automatisch mit den entsprechenden Werten versehen.

`VariablesTable` = [{}, {`"i"->`$v5$}]
`Graph` =



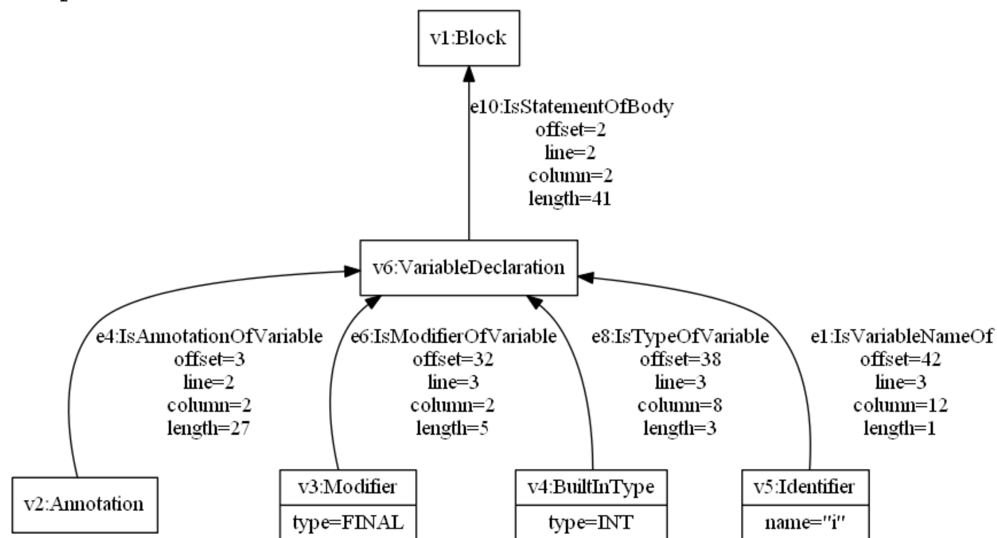
Nachdem nun alle Variablen-Deklarationen erkannt wurden, wird vor dem Verlassen der `LocalVarDecl`-Regel der `Annotation`-Knoten $v2$ durch `IsAnnotationOfVariable`-Kanten, der `Modifier`-Knoten $v3$ durch `IsModifierOfVariable`-Kanten und der Knoten $v4$ vom Typ `BuiltInType` durch `IsTypeOfVariable`-Kanten mit dem Knoten $v6$ verbunden.

`VariablesTable` = [{}, {`"i"->`$v5$}]
`Graph` =



Durch die `Block`-Regel in Zeile 10 des Listings 6.2 wird der synthetisierten Knoten vom Typ `VariableDeclaration` mittels einer `IsStatementOfBody`-Kanten mit dem Knoten $v1$ vom Typ `Block` verbunden. Zum Abschluss wird die oberste Map von der Symboltabelle `VariablesTable` entfernt.

`VariablesTable` = [{}]
`Graph` =



6.2 Semantische Aktionen

Der in den Listings des vorangegangenen Beispiels unmarkierte *SDF-Anteil der EDL-Grammatik* wird genutzt, um mithilfe von „Stratego/XT" einen Parse-Forest aufzubauen. Dieser Parse-Forest wird per Tiefensuche traversiert und dabei die semantischen Aktionen ausgeführt. Durch diese Art der Ausführung muss für jede Regel definiert werden, welchen Rückgabewert sie erzeugt. Darüber hinaus wird ein Zugriff auf die erzeugten Werte im Regelrumpf benötigt. Dieser Zugriff und die Definition des Rückgabewerts wird in Abschnitt 6.2.1 näher erläutert.

Da es in SDF-Grammatiken üblich ist, an Stelle von Alternativen im Rumpf mehrere Regeln anzugeben, kann es zu redundanten semantischen Aktionen kommen. Um dies zu vermeiden, enthält die EDL ein Generalisierungskonzept, mit dem für eine Menge von Regeln eines definierten Aufbaus dieselben Aktionen ausgeführt werden können (siehe Abschnitt 6.2.2).

Die in einer EDL-Grammatik zulässigen semantischen Aktionen lassen sich in drei Kategorien einteilen:

1. *Schema-spezifische semantische Aktionen* (siehe Abschnitt 6.2.3)
2. die *Nutzung von Symboltabellen* (siehe Abschnitt 6.2.4)
3. *Nutzer-spezifische semantische Aktionen* (siehe Abschnitt 6.2.5)

6.2.1 Zugriff auf Elemente einer Regel

Um den Zugriff auf Elemente einer Regel anschaulicher erklären zu können, wird die folgende Regel aus Listing 6.4 exemplarisch betrachtet:

```
(Anno | VarMod)* Type {VarDecl ","}+ -> LocalVarDecl
```

Der Regelkopf kann durch `$` in jeder semantischen Aktion referenziert werden, egal wo sie definiert ist. Ihm kann durch `$ =` *X* ein expliziter Wert zugewiesen werden, wobei *X* ein EDL-Ausdruck ist, die ein Objekt erzeugt. Bei dieser Zuweisung wird ein zuvor durch `$` referenziertes Graphelement mit dem neu zugewiesenen Element gemerged (siehe Abschnitt 6.2.3.1). Erhält `$` weder implizit (durch ein Default-Mapping siehe Abschnitt 6.2.2) noch explizit einen Wert, so hat es den Wert `null`. Der Wert von `$` entspricht dem *Rückgabewert* der aktuellen Regel.

Der Zugriff auf die maximalen Terme[3] im Regelrumpf geschieht durch `$i`, wobei i angibt, um den wievielten maximalen Term es sich handelt. Der erste Term wird durch `$0` identifiziert. Auf die Terme innerhalb eines maximalen Terms kann auf der Ebene des Regelrumpfs nicht zugegriffen werden. Jeder zusammengesetzte Term erzeugt eine neue Sichtbarkeitsebene. Im Falle einer Alternative wie bei

```
"a"| "b"
```

werden zwei Ebenen erzeugt, nämlich `"a"` und `"b"`. Jede Sichtbarkeitsebene setzt sich wieder aus maximalen Termen zusammen, auf die nur in genau dieser Ebene per `$i` zugegriffen werden kann. Die Tabelle 6.1 zeigt die einzelnen Ebenen der weiter oben eingeführten Beispielregel und für welche Terme die `$i` stehen.

Sichtbarkeitsebene	Zugriff auf einzelne Elemente

`(Anno | VarMod)* Type {VarDecl ","}+`	`$0 = (Anno | VarMod)*`
	`$1 = Type`
	`$2 = {VarDecl ","}+`

`(Anno | VarMod)`	`$0 = Anno | VarMod`
`Anno`	`$0 = Anno`
`VarMod`	`$0 = VarMod`
`VarDecl ","`	`$0 = VarDecl` `$1 = ","`

Tabelle 6.1: Die Sichtbarkeitsebenen einer Beispielregel

[3]Ein maximaler Term ist ein Element einer Konkatenation. Semantische Aktionen im Regelrumpf werden nicht als maximale Terme bezeichnet.

Sollte ein maximaler Term ein Tuple der Form `<` T_0 `, ..., ` T_n `>`[4] enthalten, so hat `$i` die folgenden Werte:

```
$0 = "<"
$1 = T0
$2 = ","
...
$2n+1 = ">"
```

Im Falle eines Terms vom Typ FunctionTerm `(`T_0 `...` T_n `=> T)` hat `$i` die Werte:

```
$0 = T0
...
$n = Tn
```

Welchen Wert `$i` hat, hängt davon ab, welchen Term er referenziert. Handelt es sich bei dem i-ten Term T um

- ein *Literal* oder eine *CharacterClass*, so handelt es sich bei `$i` um das erkannte Lexem.
- eine syntaktische Variable, die für die Anwendung einer *kontextfreien Regel* steht, so referenziert `$i` das synthetisierte Graphelement. Sollte für die angewendete Regel kein Rückgabewert definiert sein, hat `$i` den Wert `null`.
- eine syntaktische Variable, die die Anwendung einer *lexikalischen Regel* repräsentiert, besteht `$i` default-mäßig aus dem erkannten Lexem. Sollte in der EDL-Grammatik definiert sein, dass für diese lexikalische Variable ein Graphelement erzeugt wird, so referenziert `$i` dieses.
- eine *Option* $S?$, so hat `$i` den durch S synthetisierten Wert, falls S bei der aktuellen Regelanwendung benötigt wurde. Ansonsten hat `$i` den Wert `null`.
- eine *Sequence* (S), so hat `$i` den durch S synthetisierten Wert.
- eine *Repetition* der Form $S+$ oder $S*$, so besteht `$i` aus einer Liste, die alle durch S synthetisierten Werte enthält.
- eine *List* der Form $\{S_1\ S_2\}+$ oder $\{S_1\ S_2\}*$, so besteht `$i` aus einer Liste, die alle durch S_1 und S_2 synthetisierten Werte enthält.
- eine *Alternative* $S_1 \mid S_2$, so hat `$i` den durch S_1 synthetisierten Wert, falls bei der Regelanwendung S_1 hergeleitet wurde. Ansonsten hat es den durch S_2 synthetisierten Wert.
- ein *Tuple* oder einen *FunctionTerm*, so hat `$i` den Wert `null`.

S, S_1 und S_2 stehen in der obigen Aufzählung für maximale Terme. Bestehen sie nur aus einem einzigen der zuvor aufgelisteten Terme, so stellt sein synthetisierter Wert das

[4] `...`, `n`, `n+i` und `2n+i` stellen Metasymbole dar.

Ergebnis von S, S_1 und S_2 dar. So synthetisiert `BlockStmt*` in Zeile 10 des Listings 6.2 eine Liste von `Statement`-Knoten.

Besteht ein maximaler Term jedoch aus einer Konkatenation von mehreren Termen, so ist der durch diese Ebene synthetisierte Wert im default-Fall `null`. Um einen anderen Wert zurückzugeben, kann die `lift(`*X*`);`-Operation verwendet werden. *X* stellt dabei einen EDL-Ausdruck dar, der ein Objekt zurückliefert. So kann beispielsweise durch

```
(Anno | VarMod)* Type {VarDecl "," #lift($0)#}+ -> LocalVarDecl
```

erreicht werden, dass die durch `$2` referenzierte List im Regelrumpf als Wert eine Liste von Variablen-Deklarationen hat. Werden innerhalb eines maximalen Terms mehrere `lift()`-Operationen definiert, so wird nur der zuletzt definierte Wert zurückgegeben.

Vergabe von Alias-Bezeichnern

Um anstatt von `$` oder `$i` sprechendere Namen zu verwenden, können die in SDF Semantik-freien Labels verwendet werden, um Alias-Bezeichner zu vergeben. So kann beispielsweise durch

```
(Anno | VarMod)* type:Type {VarDecl ","}+ -> head:LocalVarDecl
```

der durch den Term `Type` synthetisierte Wert neben `$1` auch durch `$type` referenziert werden. Ein solcher Alias-Bezeichner ist nur auf der Ebene sichtbar, auf der er definiert wurde. Der Regelkopf kann in diesem Beispiel auch als `$head` angesprochen werden und ist genauso wie `$` in jedem Term sichtbar.

Temporäre Variablen

Soll ein erzeugtes Objekt gespeichert werden, ohne `$` oder ein `$i` zu überschreiben, so kann es einer temporären Variable der Form `$varName` zugewiesen werden, wobei `varName` für einen Bezeichner steht. Durch eine initiale Zuweisung wird die Variable deklariert und kann in allen Termen genutzt werden. Sollte eine temporäre Variable genauso wie ein Alias-Bezeichner heißen, so ist in dem maximalen Term, indem das Alias definiert wurde, die temporäre Variable nicht sichtbar.

Zugriff auf Whitespace

In SDF ist es üblich, Kommentare als Teil der zu ignorierenden Zeichen (`LAYOUT`) zu definieren, da bei der BNF-Transformation (siehe Abschnitt 5.2.3.1) in kontextfreien Regeln `LAYOUT?` automatisch zwischen allen Termen ergänzt wird. So wird beispielsweise aus

```
LocalVarDecl ";" -> LocalVarDeclStmt
```

die BNF-Regel

```
cf(LocalVarDecl) cf(LAYOUT?) ";" -> cf(LocalVarDeclStmt)
```

erzeugt. Wäre `LocalVarDeclStmt` ein kontextfreies Startsymbol, so würde darüber hinaus noch die Regel

```
cf(LAYOUT?) cf(LocalVarDeclStmt) cf(LAYOUT?) -> <START>
```

generiert werden. Auf diese Weise ist eine *explizite Definition der möglichen Vorkommen von Kommentaren unnötig* und die Grammatik wird verständlicher. Um beide Vorteile für EDL zu erhalten, ist dieses Konzept der Kommentar-Behandlung übernommen und so umgesetzt worden, dass auf die Ergebnisse aller angewendeten `LAYOUT`-Regeln zugegriffen werden kann.

Beispiel zum Zugriff auf Whitespace

Um den Zugriff auf die Whitespace-Informationen zu verdeutlichen, wird die initiale Java-Grammatik um die Definition von einzeilige Java-Kommentaren erweitert.

```
1  module java-15/lex/Comments
2  imports ...
3  exports
4    sorts Comment EOLCommentChars ...
5    lexical syntax
6      rule Comment -> LAYOUT
7        #$ = $0;#
8
9      rule "//" EOLCommentChars LineTerminator -> Comment
10       #$ = lexem($1);#
11
12     rule ~[\n\r]* -> EOLCommentChars
13     ...
14
15   lexical restrictions
16     EOLCommentChars -/- ~[\n\r]
17     ...
```

Listing 6.10: Das EDL-Modul `Comments`.

Listing 6.10 zeigt das EDL-Modul `Comments`, welches die in Java gültigen Kommentare definiert. Aus Gründen der Übersichtlichkeit wurde das Modul auf die Regeln für einzeilige Kommentare beschränkt. Ihr syntaktischer Aufbau wird durch die Regel in Zeile 9 beschrieben. Dabei steht `EOLCommentChars` für eine beliebig lange Sequenz aus Zeichen, die von `'\n'` und `'\r'` verschieden sind (Zeile 12). Durch die Restriktion in

Zeile 16 wird das Prinzip des „longest match“ angewendet, indem ausgedrückt wird, dass das hinter `EOLCommentChars` folgende Zeichen ein Zeilenumbruch sein muss. `LineTerminator` in Zeile 9 steht für einen Zeilenumbruch. Durch die semantische Aktion in Zeile 10 wird das durch `EOLCommentChars` erkannte Lexem als Ergebnis der `Comment`-Regel festgelegt.

In Zeile 6 wird `Comment` als `LAYOUT` definiert, wodurch erreicht wird, dass Kommentare durch die in kontextfreien Regeln automatisch ergänzten `LAYOUT?` erkannt werden. Als Ergebnis dieser Regelanwendung wird in Zeile 7 das durch `Comment` zurückgelieferte Lexem festgelegt. Im Rahmen dieses Beispiels wird davon ausgegangen, dass für alle weiteren `LAYOUT`-Regeln, die keine Kommentare definieren, `null` zurückgegeben wird.

```
4  ...
5  context-free syntax
6    rule LocalVarDecl ";" -> LocalVarDeclStmt {prefer}
7      #
8        ...
9        {
10         System.out.println(#getWhitespaceBefore($2)#);
11       }
12     #
13 ...
```

Listing 6.11: Das EDL-Modul `LocalVariableDeclarations`
(erweiterter Auszug aus Listing 6.4).

Listing 6.11 zeigt einen Auszug aus dem Modul `LocalVariableDeclarations`[5], welches um die nutzerspezifische semantische Aktion in den Zeilen 9 bis 11 erweitert wurde.

Die kontextfreie Regel in Zeile 6 definiert ein Statement, welches aus der Java-Deklaration von lokalen Variablen (`LocalVarDecl`) besteht und durch ein Semikolon abgeschlossen wird. Durch den Aufruf von `getWhitespaceBefore($2)` in Zeile 10 wird eine Liste l erzeugt, die alle von `null` verschiedenen Ergebnisse der zwischen `LocalVarDecl` und `";"` angewendeten `LAYOUT`-Regeln enthält. Da in diesem Beispiel die `Comment`-Regeln den erkannten Kommentar als String und alle anderen `LAYOUT`-Regeln `null` zurückliefern, enthält l nur die Kommentarinhalte. Schließlich wird l auf der Kommandozeile ausgegeben.

Würde der in Listing 6.12 gezeigte Java-Code mit der zuvor beschriebenen Grammatik geparst werden, so würde auf der Kommandozeile die folgende Ausgabe erscheinen:

```
[ stores the name of the current file]
```

[5] Das vollständige Modul ist in Listing 6.4 zu sehen.

```
1 String fileName // stores the name of the current file
2     ;
```

Listing 6.12: Java-Code mit Kommentar.

Methoden zum Zugriff auf Whitespace

Um innerhalb einer semantischen Aktion Zugriff auf die durch `LAYOUT`-Regeln erzeugten Werte zu erhalten, gibt es in EDL die Methode `getWhitespaceBefore()`. Wie bereits im vorangegangenen Beispiel erklärt, liefert

`getWhitespaceBefore($2)`

eine Liste aller von `null` verschiedenen Ergebnisse von `LAYOUT`-Regeln, die direkt vor dem durch `$2` referenzierten maximalen Term angewendet wurden.

Sollte die zu parsende Eingabe ein Präfix oder Suffix aus `LAYOUT` haben, so kann hinter der Definition eines kontextfreien Startsymbols eine semantische Aktion definiert werden, die als finale Aktion eines Parse-Vorgangs ausgeführt wird. Innerhalb dieser Aktion sind die initialen `LAYOUT` über `getPrefixWhitespace()` und die abschließenden `LAYOUT` über `getSuffixWhitespace()` zugreifbar.

6.2.2 Generalisierungskonzept

Die allgemeinste Form der semantischen Aktion stellt das *Default-Mapping* dar. Hierbei wird für jeden Kopf einer kontextfreien Regel, der nur aus einer syntaktischen Variablen besteht, eine Instanz einer gleichnamigen nicht-abstrakten Knotenklasse erzeugt. Ein Beispiel stellt die `Block`-Regel in Zeile 10 des Listings 6.2 dar, in der ein Knoten eines gleichnamigen Typs für den Regelkopf erstellt wird.

Um für eine spezifischere Menge von kontextfreien Regeln semantische Aktionen zu definieren, gibt es die `globale Definition`, wie sie in Abschnitt 6.2.2.1 näher erläutert wird. Schließlich besteht noch die Möglichkeit, eine spezifische kontextfreie Regel mit semantischen Aktionen zu versehen (siehe Abschnitt 6.2.2.2). Hierbei können die Aktionen innerhalb des Regelrumpfs oder hinter der Regel angegeben werden.

Die Reihenfolge, in der die semantischen Aktionen ausgeführt werden, ist:

1. Default-Mapping
2. globale semantische Aktionen vor Regelausführung
   a) Sollten mehrere Pattern auf eine Regel zutreffen, so werden die jeweiligen Aktionen in der Reihenfolge ausgeführt, wie sie in der EDL-Grammatik definiert sind.
3. semantische Aktionen im Regelrumpf
   a) Aktionen auf einer Sichtbarkeitsebene werden während der Traversierung des internen Parse-Forests entsprechend ihres Definitionsorts im Regelrumpf von links nach rechts ausgeführt.
   b) Gibt es bei 3a eine tiefere Sichtbarkeitsebene, so werden zunächst die Aktionen dieser Ebene ausgeführt.
4. globale semantische Aktionen nach Regelausführung
   a) Sollten mehrere Pattern auf eine Regel zutreffen, so werden die jeweiligen Aktionen in der Reihenfolge ausgeführt, wie sie in der EDL-Grammatik definiert sind.
5. Regel-spezifische semantische Aktionen hinter der Regel

Nachdem alle Anwendungen der in der Grammatik definierte Regeln bearbeitet und die jeweiligen semantischen Aktionen ausgeführt wurden, werden die Aktionen hinter dem Start-Symbol behandelt.

6.2.2.1 Globale semantische Aktionen

Globale semantische Aktionen werden für jede Regel ausgeführt, auf die ein definierbares Regel-Pattern passt und die sich im selben Modul wie das Pattern befinden. Ein solches Pattern besteht aus den gleichen Elementen wie eine Regeldefinition in SDF ohne die Angabe von Attributen. Allerdings gibt es den zusätzlichen Wildcard-Term `"_"` (Unterstrich), der für einen beliebigen Term stehen kann. Darüber hinaus kann das wiederholte Auftreten eines Terms durch eine wie auch in UML-Klassendiagrammen verwendete Multiplizität (z.B. `(0..*)` oder `(5)`) beschrieben werden. Um beispielsweise semantische Aktionen für die beiden If-Regeln in Listing 6.5 zu definieren, müsste das folgende Pattern verwendet werden:

```
"if""("Expr ")"Stm _(0..2) -> Stm
```

Der syntaktische Aufbau für Pattern-spezifische semantische Aktionen sieht wie folgt aus:

```
"@Before"? "pattern"Pattern "#"(SemAct)+ "#"
```

Mittels der Annotation `@Before` kann definiert werden, dass die semantischen Aktionen vor den Regel-spezifischen Aktionen ausgeführt wird. `Pattern` steht für das zuvor beschriebene Regel-Pattern und `SemAct` für die in den Abschnitten 6.2.3 bis 6.2.5 beschriebenen semantischen Aktionen. Innerhalb eines EDL-Moduls werden solche Definitionen in einem Block definiert, der mit den Schlüsselwörtern **`global actions`** eingeleitet wird. Er muss vor den exports- und hiddens-Blöcken stehen. Ein solche Definition von semantischen Aktionen ist in den Zeilen 3 bis 8 des Listings 6.5 zu sehen. Wird in einer semantischen Aktion hinter einem Pattern `$i` verwendet, so wird der `i`-te maximale Term im Rumpf der Regel referenziert, die mit dem Pattern matcht.

6.2.2.2 Regel-spezifische semantische Aktionen

Regel-spezifische semantische Aktionen sind an eine einzige Regel gebunden und müssen zwischen zwei `#` stehen. Sie können hinter einer Regel (Zeile 24 und 26 des Listings 6.6) oder an jeder beliebigen Stelle im Regelrumpf vorkommen (Zeile 19 und 20 des Listings 6.6). Kommen sie in einer Option oder Alternative vor, werden sie nur ausgeführt, wenn die entsprechende Option oder Alternative zur Herleitung der Eingabe aufgerufen wird. Die innerhalb einer Repetition oder List definierten semantischen Aktionen, werden für jeden Aufruf einer Regel bei der Herleitung ausgeführt.

Um unterscheiden zu können, ob eine semantische Aktion zu Beginn eines Regelrumpfs gehört oder die letzten Aktion ist, die hinter der vorangegangenen Regel steht, muss jede neue Regel mit dem Schlüsselwort `rule` eingeleitet werden.

6.2.3 Schema-spezifische semantische Aktionen

Schema-spezifische semantische Aktionen führen zur Erzeugung von Graphelementen (Abschnitt 6.2.3.1) oder zum Setzen von Attributwerten eines Graphelements (Abschnitt 6.2.3.2). Jede dieser semantischen Aktionen muss mit einem **`";"`** abgeschlossen werden.

Um solche Aktionen ausführen zu können, muss zunächst ein TGraph-Schema in der EDL-Grammatik angegeben werden. Hierzu muss im Startmodul die Zeile

```
schema SchemaName
```

angegeben werden. `SchemaName` ist dabei der Name des Schemas. Befinden sich noch weitere Schema-Deklarationen in anderen Modulen, so werden diese ignoriert.

6.2.3.1 Erzeugen von Graphelementen

Um Graphelemente zu erzeugen, werden semantische Aktionen genutzt, die in ihrer Form den Konstruktoraufrufen in Java ähneln. So wird ein `If`-Knoten beispielsweise durch

```
If();
```

erzeugt. Sollte die Knotenklasse durch den einfachen Namen nicht eindeutig identifizierbar sein, so muss der qualifizierte Name verwendet werden. Der erzeugte Knoten kann durch

```
$ = If();
```

dem Regelkopf zugewiesen werden. Anstelle von `$` kann der Knoten auch einem beliebigen Element des Regelrumpfs (`$i`) zugewiesen werden.

Die Erzeugung von Kanten geschieht ähnlich:

```
IsThenOf($5,$);
```

In diesem Fall wird vom fünften Element des Regelrumpfs eine `IsThenOf`-Kante zum Regelkopf erzeugt. Sollte der einfache Name nicht ausreichen, die Kantenklasse eindeutig zu identifizieren, muss der qualifizierte Name verwendet werden. Der erste aktuelle Parameter ist der Startknoten und der zweite der Endknoten der Kante. Die erzeugte Kante wird zurückgegeben, damit sie dem Regelkopf `$` oder einem Element des Regelrumpfs `$i` zugewiesen werden kann. Sollte eine der beiden Knotenreferenzen `null` sein oder auf ein Objekt zeigen, zu dem keine Kante dieses Typs erzeugt werden kann, wird eine Exception geworfen.

In manchen Fällen kann eine Hyperkante[6] erwünscht sein. Ein Beispiel wäre die Regel

```
"{"BlockStmt* "}"-> Block
```

in Zeile 10 des Listings 6.2, in dem alle durch `BlockStmt*` erzeugten `Statement`-Knoten mit dem `Block`-Knoten verbunden werden sollen. Da Hyperkanten in TGraphen nicht realisiert sind, werden die `Statement`-Instanzen durch je eine binäre Kante mit dem `Block`-Knoten verbunden. Um in einem solchen Fall das Erzeugen von mehreren Kanten zu vereinfachen, kann bei der Kantenerzeugung anstatt eines einzelnen Knotens direkt eine Liste von Knoten übergeben werden, wie beispielsweise:

```
AnEdgeClass(αList, ωList);
```

Mit dieser Aktion wird von jedem Knoten in der `αList` zu jedem Knoten der `ωList` eine `AnEdgeClass`-Kante erzeugt. In dem Fall liefert diese Aktion keine Kante zurück. Sollte sich in der `αList` oder `ωList` ein Objekt befinden, welches kein gültiger α- oder ω-Knoten für Kanten vom Typ `AnEdgeClass` ist, so wird eine Exception geworfen. Eine Anwendung dieses Konstrukts ist in Zeile 11 des Listings 6.2 zu sehen. `$1` steht dabei für

[6] Eine Hyperkante ist eine Kante mit beliebig vielen Anfangs- und Endknoten.

die Liste der durch `BlockStmt*` erzeugten `Statement`-Knoten. Solle es eine leere Liste sein, so wird keine Kante erzeugt.

Soll ein Element als Ergebnis einer Regelanwendung zurückgeliefert werden, so muss es dem Kopf der entsprechenden Regel zugewiesen werden. Dies kann durch die folgende Anweisung geschehen:

```
$ = $3;
```

Sollte `$` zuvor eine Kante referenziert haben, so wird diese gelöscht. Stand `$` für einen Knoten, so wird dieser mit `$3` verschmolzen. Im Detail bedeutet dies, dass die zu `$` inzidenten Kanten auf `$3` umgelegt werden. Kanten, bei denen dies aufgrund des Schemas nicht möglich ist, werden zusammen mit dem ehemaligen `$`-Knoten gelöscht.

6.2.3.2 Setzen von Attributwerten

Das Setzen von Attributwerten eines Graphelements kann durch Zuweisungen der folgenden Form geschehen:

`$i.attr1 =` X_1`;` bzw.

`$.attr1 =` X_1`;`

Die Attribute der Graph-Instanz können durch

`graph().attr1 =` X_1`;`

belegt werden. `graph` ist eine Funktion, die das Graph-Objekt zurückliefert. Um die Erzeugung der Graph-Instanz muss sich der Nutzer nicht kümmern, da dies automatisch geschieht.

In den zuvor gegebenen Beispielen steht X_i für eine der folgenden EDL-Ausdrücke:

null-Konstante, der Form `null`.

Integer-Konstante, dessen Syntax der einer Integer-Zahl in Java entspricht.

Long-Konstante, dessen Syntax der einer Long-Zahl in Java entspricht.

Double-Konstante, dessen Syntax der einer Double-Zahl in Java entspricht.

String-Konstante, dessen Syntax der eines String-Literals in Java entspricht wie beispielsweise `"A String\n"`.

Enumeration-Konstante der Form `AnEnumerationDomain.ENUM_CONSTANT`.

Listen-Konstrukt der Form `list(`X_1`,` X_2`, ...,` X_n`)`.[7]

Set-Konstrukt der Form `set(`X_1`,` X_2`, ...,` X_n`)`.

[7] Sollte es notwendig sein, auf ein Element einer Liste zuzugreifen, so kann dies genauso wie bei einem Array durch Angabe von

```
"["Index "]"
```

geschehen. Darüber hinaus kann auf eine Liste jede im Java-Interface `List` definierte Methode aufgerufen werden.

Map-Konstrukt der Form `map(`X_{key1}`,`X_{value1}`, ..., `X_{keyn}`,`X_{valuen}`)`.

Record-Konstrukt der Form `ARecordDomain(`X`)`. Dabei stellt X eine Map dar, die jedem Attribut des Records einen Wert zuweist.

`$i`, wobei `$i` auf eine lexikalische Variable, ein Literal oder eine CharacterClass im Regelrumpf verweisen muss. Steht `$i` für einen zusammengesetzten Term, so muss vom Nutzer sichergestellt werden, dass `$i` einem gültigen X_i-Wert entspricht.

Attributwert eines anderen Graphelements der Form `$.attr` bzw. `$i.attr`. Dabei können Integer-, Long-, Double- und String-Werte ineinander überführt werden. In allen anderen Fällen müssen die Attribut-Domänen identisch sein.

Nutzerspezifische semantische Aktion, die einen Wert erzeugt, die der Attribut-Domäne entspricht.

Parsing-Information (siehe weiter unten).

Zugriff auf Parsing-Informationen

Da EDL für das Parsen von einer oder mehreren Dateien gedacht ist, kann die Positionsangabe der erkannten Zeichen hilfreich sein. Daher sind die in Tabelle 6.2 gezeigten Positionsangaben für jeden Term in einer Regel zugreifbar.

Funktion	Wertebereich	Beschreibung
`file`	String	Name der geparsten Datei
`offset`	int ≥ 0	Index des ersten erkannten Zeichens
`line`	int ≥ 1	Zeile des ersten erkannten Zeichens
`column`	int ≥ 0	Spalte des ersten erkannten Zeichens
`length`	int ≥ 0	Anzahl der erkannten Zeichen
`lexem`	String	erkannte Lexem

Tabelle 6.2: Die in EDL definierten Funktionen zum Zugriff auf Positionsangaben und Lexeme.

Die Funktion `file` liefert für jeden möglichen Term immer den Namen der aktuell geparsten Datei. Daher benötigt sie keine Parameter und kann durch `file()` aufgerufen werden. Alle anderen Funktionen benötigen einen Term als Parameter, für den sie die entsprechende Positionsangabe zurückliefern sollen. Die Terme werden durch die bereits zuvor erläuterten `$i` und `$` identifiziert. Ein möglicher Aufruf wäre `line($2)`.

Default-Werte von Attributen

Wie in dem in Abschnitt 6.1.1 vorgestellten Java5-Schema kann es vorkommen, dass alle Knoten- oder Kantenklassen einen gemeinsamen Obertyp haben, in dem die Attribute für die Positionsangaben definiert sind. Da es für den Anwender umständlich wäre, für jede erzeugte Kante die entsprechenden Attribute zu setzen, kann diese Aktion im Startmodul als Default-Aktion beim Erzeugen eines Graphelements von einem bestimmten Typ geschehen. Ein solcher Deklarationsblock kann im Startmodul unter der schema-Deklaration durch die Schlüsselwörter **default values** erstellt werden. Existieren solche Blöcke in anderen Modulen derselben Grammatik, werden diese ignoriert und eine Warnung ausgeben.

```
4  default values
5    AttributedEdge.offset = offset(alpha);
6    AttributedEdge.line = line(alpha);
7    AttributedEdge.column = column(alpha);
8    AttributedEdge.length = length(alpha);
```

Listing 6.13: Der **default values**-Block aus Listing 6.1.

Listing 6.13 zeigt ein Beispiel für einen **default values**-Block. Innerhalb eines solchen Blocks können für die Attribute von Graphelementen Default-Werte definiert werden. So bedeutet die Angabe in Zeile 5: Falls in einer beliebigen semantischen Aktion eine Kante vom Typ `AttributedEdge` oder einer ihrer Subtypen erzeugt wird, so erhält das `offset`-Attribut automatisch den Wert `offset(alpha)`. Das reservierte Schlüsselwort `alpha` referenziert dabei den jeweiligen Alpha-Knoten. Alternativ gibt es auch noch das Schlüsselwort `omega` mit analoger Bedeutung. Sollten für den Alpha- oder Omega-Knoten keine Positionsangaben vorliegen, so wird -1 genommen.

Im Falle von

```
AttributedEdge.offset = 1;
AttributedEdge.offset = 2;
```

hat das `offset`-Attribut für Kanten vom Typ `AttributedEdge` den Wert 2, weil die zweite Zeile die letzte notierte Zuweisung ist.

6.2.4 Symboltabellen

In EDL werden Symboltabellen durch einen *Stack von Maps* dargestellt, die einen frei definierbaren String auf einen Knoten abbilden. Jede Map repräsentiert einen *eigenen Gültigkeitsbereich* für die in ihr enthaltenen String. Die oberste Map auf dem Stack repräsentiert

den aktuellen Gültigkeitsbereich. So könnte sie beispielsweise die Variablen enthalten, die in einer bestimmten Java-Methode definiert sind. Die auf dem Stack darunterliegende Map würde demnach die Variablen enthalten, die in der Klasse deklariert wurden, in der die Methode definiert ist.

```
1 symbol tables
2   VarAndMethodTable<Identifier -->IsDefinedIn,
3                       MethodDeclaration -->IsDeclaredIn>:Block
4   LabelsTable<Label -->IsDefinedIn>:Block
```

Listing 6.14: Die Deklaration von Symboltabellen.

Um Symboltabellen in der EDL-Grammatik nutzen zu können, müssen sie zunächst *deklariert* werden. Hierzu muss ein ***symbol tables**-Konstrukt* erstellt werden. Wird es im Startmodul angegeben, so sind diese Tabellen in allen Modulen sichtbar. Ansonsten sind sie nur im definierten Modul gültig. Ein Beispiel eines solchen Konstrukts ist in Listing 6.14 zu sehen. In Zeile 2 und 3 wird eine Symboltabelle mit Namen `VarAndMethodTable` deklariert, in die `Identifier` und `MethodDeclaration`-Knoten eingefügt werden können. Um die Symboltabelle im Graphen persistieren zu können, wird durch die Angabe von `:Block` festgelegt, dass jede Map auf dem Stack durch einen Knoten v_{Block} vom Typ `Block` im Graphen repräsentiert wird. Um das Enthaltensein eines `Identifier`-Knotens $v_{Identifier}$ in einer Map auszudrücken, wird durch `-->IsDefinedIn` definiert, dass jeder $v_{Identifier}$ durch eine Kante vom Typ `IsDefinedIn`[8] mit dem entsprechenden v_{Block} verbunden werden soll. Analog hierzu werden für `MethodDeclaration`-Knoten `IsDeclaredIn`[9]-Kanten erzeugt.

Ein **symbol tables**-Block kann mehrere Symboltabellen-Definitionen enthalten. Die folgenden SDF-Regeln beschreiben den Aufbau einer einzelnen Definition:

```
STableName "<"STableElems ">"(":"PersistentVertexClass)?
                              -> STableDef
VertexClassName -> PersistentVertexClass
{STableElem ","}+ -> STableElems
VertexClassName PersistentEdgeClass? -> STableElem
("-->"|"<--") EdgeClassName -> PersistentEdgeClass
```

[8] Der Kantentyp `IsDefinedIn` ist nicht im zu Beginn dieses Kapitels vorgestellten Java-Schemas enthalten, sondern wurde hier neu eingeführt, um ein Beispiel für die Pesistenz von Symboltabellen im Graph geben zu können.

[9] `IsDeclaredIn` ist ebenfalls ein Kantentyp, der nicht im Java-Schema enthalten ist und nur für dieses Beispiel erzeugt wurde.

Bei `STableName` handelt es sich um einen gültigen SDF-Bezeichner, über den die Tabelle in den semantischen Aktionen aufrufbar ist. Bei der Benennung der definierten Symboltabellen muss der Nutzer sicherstellen, dass es zu keinen Namensüberschneidungen kommt. Mittels der `STableElem`-Deklarationen können die Knotenklassen bestimmt werden, deren Instanzen in die Symboltabelle eingefügt werden können.

Soll eine Symboltabelle, wie durch die Anforderung 1.2.14 gewünscht, im Graphen persistiert werden, so müssen `PersistentVertexClass` und für jedes `STableElem` ein `PersistentEdgeClass` definiert sein. Ersteres steht für den Typ eines Knotens, der den Namensraum dieser Symboltabelle repräsentiert. So stellt im Beispiel (Listing 6.14) ein `Block`-Knoten den Namensraum für alle Variablen dar, die innerhalb des jeweiligen Statement-Blocks definiert werden. `PersistentEdgeClass` definiert den Typ der Kanten, die ein Element der Symboltabelle und den Namensraum-Repräsentanten miteinander verbindet. Durch `-->` bzw. `<--` wird ausgedrückt, ob das Symboltabellen-Element der Start- oder Endknoten der Kante ist.

Erzeugung und Sichtbarkeitsbereich von Symboltabellen

Für jede Symboltabellen-Definition wird ein *Symboltabellen-Stack* erzeugt. Durch eine den Annotationen in Java ähnelnde semantische Aktion der Form

```
"@Symboltable{"{SymTableName ","}+ "}"-> SymTableAnnotation
```

kann eine Regel mit Symboltabellen-Stacks assoziiert werden. `SymTableName` steht dabei für den Namen einer Symboltabelle, wie sie innerhalb eines **`symbol tables`**-Blocks definiert wurde.

```
9  @Symboltable{VariablesTable,LabelsTable}
10 rule "{" (BlockStmt #IsStatementOfBody($0,$);#)* "}" -> Block
```

Listing 6.15: Assoziation von Symboltabellen mit einer Regel (Auszug aus Listing 6.2).

Listing 6.15 zeigt, wie die Block-Regel in Zeile 10 mit den beiden angegebenen Symboltabellen `VariablesTable` und `LabelsTable` assoziiert wird. Dies bedeutet, dass sobald bei der Traversierung des internen Parse-Forests ein Knoten v erreicht wird, der für die Anwendung dieser Block-Regel steht, für jede der beiden angegebenen Symboltabellen eine neue Map erzeugt und auf den jeweiligen Stack gelegt wird. Die so erzeugte Map bleibt solange auf dem Stack, bis v nach der Traversierung seiner Kinder wieder verlassen wird.

Da es notwendig sein kann, nach dem Parse-Vorgang Zugriff auf die Symboltabellen zu haben, werden die Symboltabellen durch `public` Felder im generierten `GraphBuilder` referenziert.

Nutzung von Symboltabellen

Um auf Symboltabellen zuzugreifen, wird die folgende Notation verwendet, die an den Aufruf einer Methode in Java erinnert:

```
SymTableName.MethodName(Params);
```

Ähnlich wie bei PDL [Dah95] steht `MethodName` dabei für eine der folgenden Methoden mit den zugehörigen Parametern `Params`:

declare(Key, Vertex).
Beim Aufruf dieser Methode wird der Knoten `Vertex` zusammen mit dem eindeutig identifizierenden Wert `Key` in die Top-Map des jeweiligen Stacks eingefügt. Bei `Key` kann es sich um ein beliebiges Objekt handeln. Befindet sich in der Top-Map bereits ein temporärer Knoten v, der durch `Key` identifiziert werden kann, so wird v mit `Vertex` verschmolzen (siehe Beschreibung der Methode `use` weiter unten). Sollte v nicht temporär sein, so wird eine Exception geworfen. Die Methode liefert den Knoten `Vertex` zurück.

use(Key).
Diese Methode sucht zunächst in der obersten Map m des Stacks nach einem Knoten, der durch `Key` eindeutig identifiziert werden kann. Konnte in m kein entsprechender Knoten gefunden werden, so wird in der auf dem Stack darunterliegenden Map gesucht. Der erste gefundene Knoten wird zurückgeliefert. Falls kein entsprechendes Element gefunden wurde, so wird ein temporärer Knoten erzeugt, in die Map eingefügt und zurückgegeben.

useOrDeclare(Key, Vertex).
`useOrDeclare` arbeitet zunächst wie `use`. Sollte ein Knoten gefunden werden, so wird `Vertex` gelöscht. Ansonsten arbeitet es wie `declare`. Diese Methode kann hilfreich sein, falls Sprachen wie MatLab geparst werden sollen, in denen eine Variable durch eine initiale Zuweisung automatisch deklariert werden.

getTemporaryVertices().
`getTemporaryVertices` liefert eine Liste aller temporären Knoten zurück, die sich in der gesamten Symboltabelle befinden.

Wie durch die Anforderung 1.2.14 gewünscht, soll es möglich sein eine Symboltabelle im Graphen zu persistieren. Dies bedeutet, dass für jede erzeugte Map ein Knoten bestimmt werden soll, der den für diese Map gültigen Namensraum repräsentiert. Falls die

Persistenz einer Symboltabelle gewünscht wird, muss bei der Symboltabellen-Definition ein `PersistentVertexClass` angegeben sein, wie es weiter oben beschrieben wurde. Durch

```
SymTableName.nameSpace = v;
```

kann der Knoten v vom Typ `PersistentVertexClass` als Namensraum-Repräsentant für die oberste Map auf dem Symboltabellen-Stack `SymTableName` gesetzt werden. Alle in dieser Map bereits enthaltenen Knoten werden durch jeweils eine Instanz der für ihren Typ angegebenen `PersistentEdgeClass`-Kantenklasse mit v verbunden. Werden nach dem Setzen von `nameSpace` weitere Knoten in die Map eingefügt, so werden gleichzeitig die entsprechenden Persistenz-Kanten erzeugt. Für jede erzeugte Map darf das Attribut `nameSpace` höchstens einmal gesetzt werden.

6.2.5 Nutzerspezifische semantische Aktionen

Bei den nutzerspezifischen semantischen Aktionen werden lokale und globale Definitionen unterschieden. Erstere können im Regelrumpf und hinter der Regel und in einem **`global actions`**-Block vorkommen. Sie bestehen aus korrektem Java-Code, der mit geschweiften Klammern umschlossen ist. Kommt eine lokale nutzerspezifische semantische Aktion auf der rechten Seite einer Zuweisung oder als aktueller Parameter vor, so muss durch ein return-Statement innerhalb des `{}`-Blocks der zurückzugebende Wert festgelegt sein. Um die in den vorangegangenen Abschnitten beschriebenen semantischen Aktionen der EDL innerhalb eines solchen Blocks nutzen zu können, müssen sie zwischen zwei `#` stehen.

Um den Java-Code der lokalen nutzerspezifischen semantischen Aktionen zu vereinfachen, kann im Startmodul mit den Schlüsselwörtern **`user code`** ein Java-Block eingeleitet werden, in dem Methoden und Felder definiert werden können, die in den nutzerspezifischen semantischen Aktionen verwendet werden können. Die Verwendung von EDL-spezifischen semantischen Aktionen innerhalb eines solchen Blocks muss ebenfalls durch umschließende `#` signalisiert werden. Ein **`user code`**-Block ist für alle Module sichtbar, egal wo er definiert wurde. Bei der Vergabe von Methoden- und Feldnamen müssen Namenskollisionen vermieden werden.

Da es in nutzerspezifischen semantischen Aktionen möglich ist, auch andere Java-Klassen wie beispielsweise `ArrayList` zu verwenden, müssen die benötigten import-Statements vom Nutzer explizit angegeben werden. Hierzu muss ein durch die Schlüsselwörter **`import declarations`** eingeleiteter Block alle benötigten Java-Imports enthalten, wie sie auch in .java-Dateien vorkämen, jedoch ohne das Java-Schlüsselwort `import`. Die import-Statements sind für alle Module sichtbar, egal wo sie definiert wurden.

6.3 Inselgrammatik

Um eine EDL-Beschreibung als Inselgrammatik zu verwenden, muss im Startmodul genau einmal definiert sein, wo der zu parsende Anteil beginnt und wo er endet. Sollten weitere Definitionen dieser Art existieren, so werden diese ignoriert und eine Warnung wird ausgegeben.

```
1 island start
2   exclusive <script[^>]*type="text/javascript"[^>]*>
3 island end
4   exclusive </script>
```

Listing 6.16: Die Definition einer Inselgrammatik zum Parsen des Javascript-Anteils einer HTML-Seite.

Listing 6.16 zeigt ein Beispiel in dem der notwendige EDL-Code steht, um nur den Javascript-Anteil einer HTML-Seite zu parsen. In Zeile 1 und 2 wird der Anfang der zu parsenden Insel einer HTML-Seite definiert. Dieser besteht aus einem `script`-Tag, dessen `type`-Attribut den in diesem Tag enthaltenen Code als Javasript ausweist. Dieses Tag wird in Zeile 2 durch einen regulären Ausdruck beschrieben, dessen erkanntes Lexem nicht zum parsenden Code gehört, was durch das Schlüsselwort **`exclusive`** signalisiert wird. In den Zeilen 3 und 4 wird das schließende `script`-Tag als das Ende der Insel markiert.

Die durch SDF-Regeln beschriebene Syntax dieser Definitionen hat die folgende Form:

```
"island""start"IslandBorderDef -> IslandStart
"island""end"IslandBorderDef -> IslandEnd
("inclusive"|"exclusive") RegExp -> IslandBorderDef
```

`RegExp` steht für einen regulären Ausdruck, der auch in Java gültig wäre. Die durch ihn beschriebene Sprache darf das leere Wort nicht enthalten. Das Schlüsselwort `inclusive` signalisiert, dass das durch `RexExp` erkannte Wort Teil des zu parsenden Anteils der Eingabe ist.

Bei der Verwendung von eingebetteten Flag-Ausdrücken wie `(?i)`, wodurch die Groß- und Kleinschreibung ignoriert wird, muss beachtet werden, dass sie für alle definierten Inselgrammatik-Ausdrücke gelten, unabhängig von ihrem Definitionsort.

7 Konzeptioneller Systementwurf

Die Aufgabe des im Rahmen dieser Masterarbeit entstandenen Systems besteht in der Überführung einer *Eingabedatei mithilfe einer EDL-Grammatik und eines TGraph-Schemas in einen Graphen*.

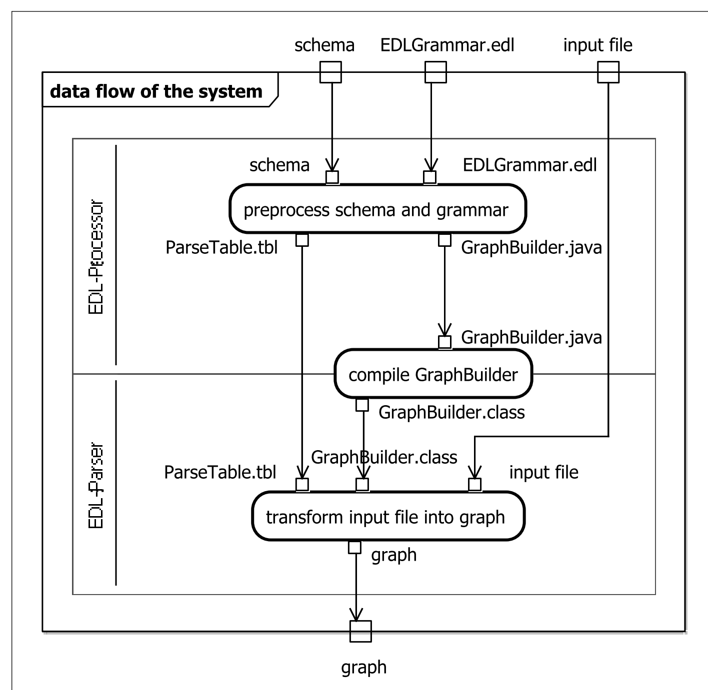


Abbildung 7.1: Der Datenfluss im vollständigen System.

Die Arbeitsweise des Systems ist, wie in Abbildung 7.1 dargestellt, in zwei Schritte unterteilt:

1. Zur *Vorverarbeitung* durch den `EDL-Processor` muss die verwendete EDL-Grammatik sowie das benötigte Schema festgelegt werden. Im Rahmen der Vorverarbeitung, wie sie im Abschnitt 7.1 näher erläutert wird, werden eine Parse-Tabelle und eine `GraphBuilder`-Klasse erzeugt. Dieser Schritt muss nur bei einer Änderung der EDL-Grammatik oder des Schemas wiederholt werden.
2. Bevor die `EDL-Parser`-Komponente mit dem Schritt des *Parsings* beginnen kann, muss zunächst die `GraphBuilder`-Klasse kompiliert werden und die zu parsende

Eingabedatei festgelegt werden. Letztere wird mithilfe des `GraphBuilder`s und der Parse-Tabelle in den gewünschten Graphen überführt und zurückgeliefert (siehe Abschnitt 7.2). Das Parsing muss für jede Eingabedatei erneut ausgeführt werden.

7.1 Vorverarbeitung

Abbildung 7.2 zeigt den *Ablauf der Vorverarbeitung*, bei der aus dem übergebenen Schema und der EDL-Grammatik eine Parse-Tabelle und die `GraphBuilder`-Klasse erzeugt werden.

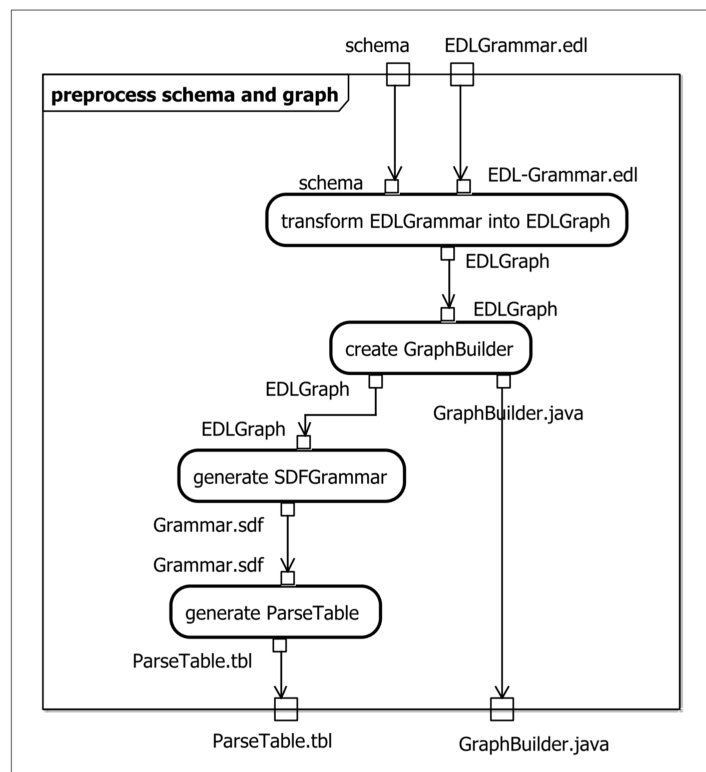


Abbildung 7.2: Die Vorverarbeitung der EDL-Grammatik.

In einem ersten Schritt wird die *EDL-Grammatik in einen abstrakten Syntaxgraphen namens* `EDLGraph` *überführt*. Dabei wird das Default-Mapping von syntaktischen Variablen auf gleichnamige Knoten-Klassen vorgenommen und überprüft, ob die schema-spezifischen semantischen Aktionen valide sind. So wird beispielsweise getestet, ob die gesetzten Attribute eines Knotens oder einer Kante existieren.

Im Anschluss wird der soeben erzeugte `EDLGraph` genutzt, um eine *`GraphBuilder`-Klasse zu generieren*, die für die Ausführung der in der EDL-Grammatik definierten semantischen Aktionen zuständig ist. Darüber hinaus wird der `EDLGraph` ebenfalls benötigt, um eine SDF-Grammatik zu erzeugen, aus der schließlich die Parse-Tabelle generiert wird.

Komponenten der Vorverarbeitung

Für die *Vorverarbeitung ist der `EDL-Processor` zuständig*, wie er in Abbildung 7.3 zu sehen ist. Vom Nutzer kann diese Komponente durch das `EDLProcessor`-Interface angesprochen werden, um die benötigte EDL-Grammatik sowie das Schema anzugeben und die Vorverarbeitung zu starten. Darüber hinaus hat diese Komponente die Aufgabe, die in ihr enthaltenen Subsysteme so zu koordinieren, dass die im vorangegangenen Abschnitt erläuterten Vorverarbeitungsschritte ausgeführt werden.

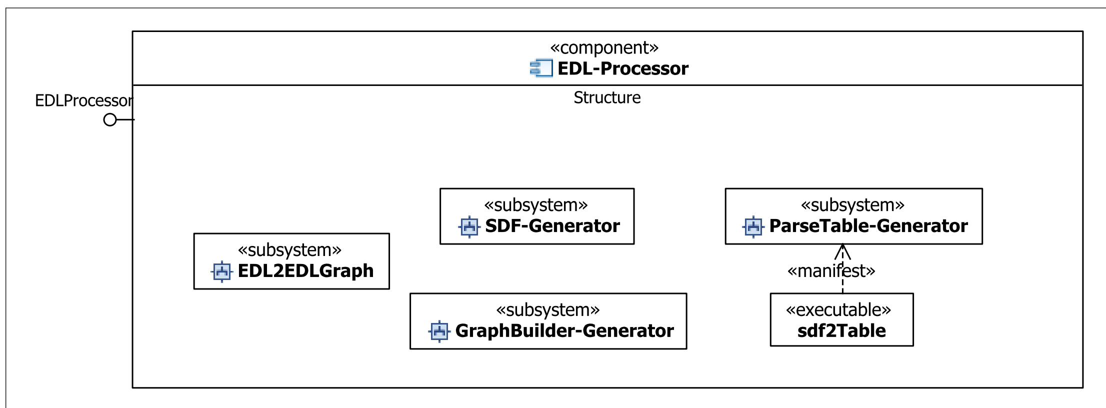


Abbildung 7.3: Die für die Vorverarbeitung zuständigen Komponenten.

Mithilfe von `EDL2EDLGraph` wird die EDL-Grammatik in den abstrakte Syntaxgraphen (`EDLGraph`) überführt, aus dem der `GraphBuilder-Generator` die `GraphBuilder`-Klasse erzeugt. Der `SDF-Generator` nutzt den `EDLGraph`, um eine gültige SDF-Grammatik zu generieren. Diese wird dem in „Stratego/XT“ enthaltenen Tool `sdf2Table` übergeben, welches die Parse-Tabelle erzeugt.

7.2 Erzeugung des Graphen aus der Eingabedatei

In Abbildung 7.4 wird gezeigt, wie die *Eingabedatei* mithilfe der in der Vorverarbeitung erzeugten Parse-Tabelle und des `GraphBuilders` *in den gewünschten Graphen überführt* wird.

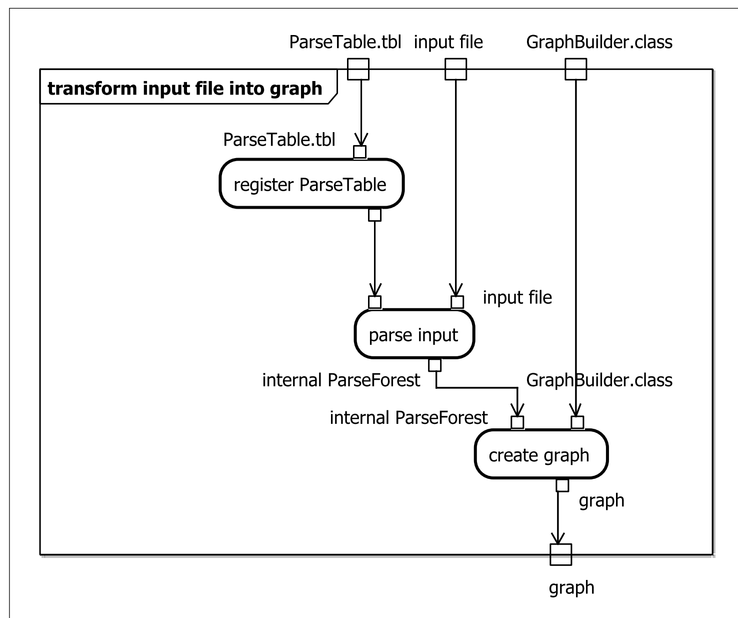


Abbildung 7.4: Die Erzeugung eines Graphen aus der Eingabedatei.

In einem ersten Schritt wird die Parse-Tabelle bei dem in der im „Stratego/XT" enthaltenen Interpreter registriert. Dieser parst im Anschluss die Eingabedatei und erzeugt einen internen Parse-Forest, aus dem schließlich mithilfe des `GraphBuilders` der gewünschte Graph aufgebaut wird.

Komponenten der Grapherzeugung

Die *für die Grapherzeugung notwendigen Komponenten* sind in Abbildung 7.5 dargestellt. Dabei stellt `EDL-Parser` die zentrale Komponente dar, über die der Nutzer den Vorgang starten kann.

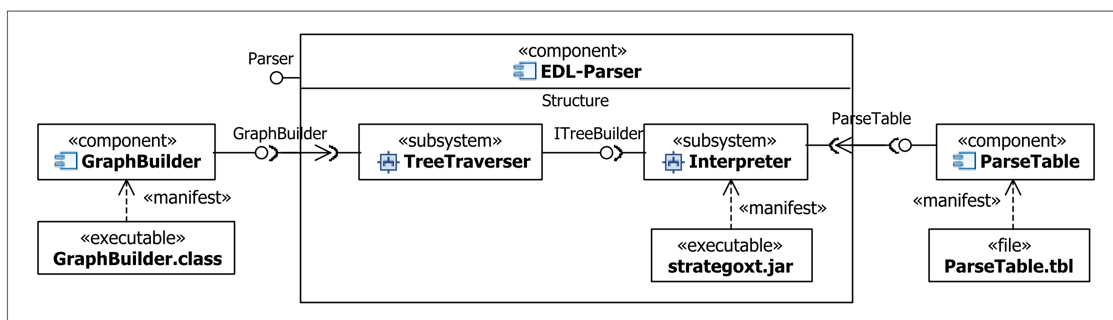


Abbildung 7.5: Die für das Parsing der Eingabedatei zuständigen Komponenten.

Innerhalb der Komponente `EDL-Parser` wird der in der `strategoxt.jar` enthaltene `Interpreter` genutzt, um mithilfe der `ParseTable`-Komponente die Eingabedatei zu parsen. Manifestiert wird `ParseTable` durch die bei der Vorverarbeitung in einer .tab-Datei persistierten Parse-Tabelle. Wie bereits in Abschnitt 5.2 beschrieben, nutzt der `Interpreter` einen `ITreeBuilder`, um den internen Parse-Forest in die gewünschte Ausgabe zu überführen. Der `EDL-Parser` verfügt über eine `ITreeBuilder`-Komponente, die `TreeTraverser` heißt. Diese traversiert den internen Parse-Forest und führt dabei mithilfe einer `GraphBuilder`-Komponente die semantischen Aktionen aus, wodurch der gewünschte Graph aufgebaut wird. Der `GraphBuilder` wird durch eine Java-Klasse manifestiert, die aus der im Vorverarbeitungsschritt erzeugten Java-Datei `GraphBuilder.java` kompiliert wurde.

8 Parser

In diesem Kapitel wird die *Implementation der `EDL-Parser`-Komponente* behandelt, die eine oder mehrere Eingabedateien unter Zuhilfenahme einer Parse-Tabelle und eines `GraphBuilder`s in einen Graphen überführt. Vor der Nutzung der `Parser`-Komponente muss eine in EDL geschriebene Grammatik mittels der *`EDL-Processor`-Komponente* vorverarbeitet werden (siehe Kapitel 7 und 9). Dabei wird eine Parse-Tabelle und ein `GraphBuilder` generiert. Letzterer führt die in der Grammatik definierten semantischen Aktionen aus.

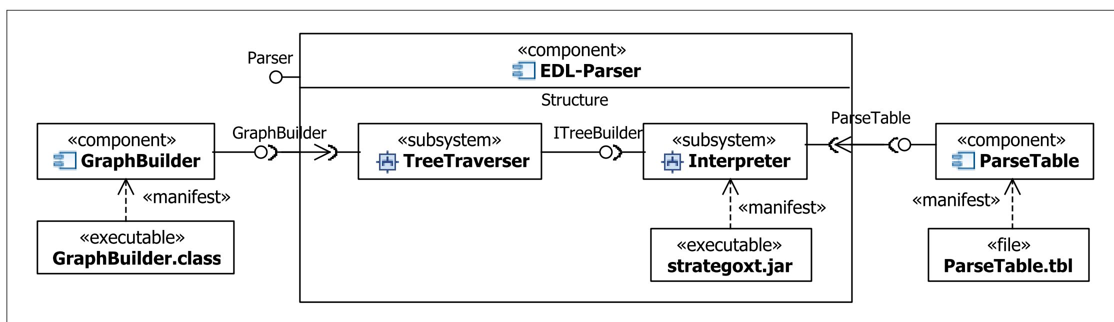


Abbildung 8.1: Die `Parser`-Komponente.

Abbildung 8.1 zeigt die `EDL-Parser`-Komponente, die einen `Interpreter` zur Überführung der Eingabe in einen internen Parse-Forest nutzt. In der `strategoxt.jar` ist eine Implementation des `Interpreter`s enthalten, die eine im Vorverarbeitungsschritt generierte Parse-Tabelle benötigt. Darüber hinaus nutzt der `Interpreter` die Komponente `TreeTraverser`, um den internen Parse-Forest zu traversieren. Dabei verwendet der `TreeTraverser` einen im Vorverarbeitungsschritt generierten `GraphBuilder`, um den Ausgabegraphen aufzubauen.

Der Parse-Vorgang kann durch den *Aufruf des generierten `GraphBuilder`s* gestartet werden:

```
java MyGraphBuilder [-e <Encoding>] [-d] [-v] [-debug]
      -o <File> -i <File>
```

Die folgenden Kommandozeilenparameter werden benötigt:

Parameter	Beschreibung
-i <File> --input <File>	Definiert die zu parsende Eingabedatei, die in einen Graphen überführt wird.
-o <File> --output <File>	Mithilfe dieses Parameters wird der Name der Datei angegeben, in die der erzeugte Graph gespeichert wird.

Darüber hinaus gibt es noch die folgenden optionalen Parameter:

Parameter	Beschreibung
-e <Encoding> --encoding <Encoding>	Mithilfe dieses optionalen Parameters kann das Encoding der Eingabedateien festgelegt werden. Ansonsten wird das default-Encoding der JVM genommen.
--debug	Bei Angabe dieses optionalen Parameters wird der Debug-Modus aktiviert. Dies bedeutet, dass der interne Parse-Forest zum Debuggen ausgegeben wird, egal ob der Parse-Forest eindeutig oder mehrdeutig ist. Sollte bei der Ausführung der semantischen Aktionen eine Exception auftreten, so wird er ebenfalls ausgegeben und der Knoten markiert, der den Fehler verursacht hat.
-d <Format> --dot <Format>	Mit diesem optionalen Parameter wird der Dot-Modus aktiviert. Der Parser erzeugt eine grafische Ausgabe des internen Parse-Forests im gewünschten Format, anstatt der default-Konsolenausgabe.
-vb --verbose	Bei Angabe dieses optionalen Parameters wird der Verbose-Modus aktiviert, um bei der Ausgabe alle Regel-Anwendungen im internen Parse-Forest auszugeben. Ansonsten werden nur die Anwendungen der in der Grammatik definierten Regeln ausgegeben.

Im folgenden Abschnitt wird zunächst ein Beispiel angegeben, das einen Überblick über die Arbeitsweise der `Parser`-Komponente geben soll. Im Anschluss werden die Repräsentation einer Regel (Abschnitt 8.2), die Realisierung eines Stacks für die Regelanwendungen im internen ParseForest (Abschnitt 8.3) und die Implementation der Komponen-

te `TreeTraverser` vorgestellt, die den internen Parse-Forest traversiert und dabei die Ausführung der Aktionen anstößt (Abschnitt 8.4). Die verschiedenen Debug-Ausgaben werden im Abschnitt 8.5 erläutert, bevor im folgenden Abschnitt 8.6 die Implementation des Symboltabellen-Stacks erklärt wird. Schließlich wird im Abschnitt 8.5 die Realisierung der Inselgrammatiken vorgestellt.

8.1 Beispiel

Um die Realisierungsbeschreibungen in den folgenden Abschnitten besser verstehen zu können, wird hier anhand einer *Beispielgrammatik zur Erkennung einer Zahl mit optionalem Währungszeichen* die Arbeitsweise der `Parser`-Komponente verdeutlicht.

```
1  module Main
2  ...
3  exports
4     sorts Start Value Currency
5     context-free start-symbols Start
6     lexical syntax
7        [0]          -> Value {definedAs("...")}
8        [1-9] [0-9]* -> Value {definedAs("...")}
9
10       [\$\253] -> Currency {definedAs("...")}
11    lexical restrictions
12       Value -/- [0-9]
13    context-free syntax
14       Value Currency? -> Start {definedAs("...")}
```

Listing 8.1: Die Beispielgrammatik.

Listing 8.1 zeigt die aus *reinem SDF* bestehende Beispielgrammatik, wie sie im Vorverarbeitungsschritt erzeugt würde. Um die Übersichtlichkeit zu erhöhen, wurde in Zeile 2 der Import des Moduls weggelassen, welches die zu ignorierenden Zeichen (`LAYOUT`) definiert. Für jede definierte Regel ist ein *`definedAs`-Attribut* gesetzt, um bei der späteren BNF-Transformation die in der Grammatik definierten Regeln eindeutig identifizieren zu können. In den Klammern dieses Attributs steht die Regel, wie sie in der ursprünglichen in EDL erstellten Grammatik aufgeführt war. Da dies die Lesbarkeit dieses Listings reduzieren würde, ist die Regel zwischen den Klammern durch `"..."` ersetzt worden. Der Sinn des `definedAs`-Attributs wird weiter unten bei der Erläuterung des Listings 8.2 anschaulicher beschrieben.

In den Zeilen 7 und 8 wird eine natürliche Zahl inklusive der 0 als `Value` definiert. Durch die Restriktion in Zeile 12, in der hinter `Value` keine Ziffer mehr kommen darf, wird für `Value` das Prinzip des „longest match" angewendet. In Zeile 10 werden die zulässigen Währungssymbole (`Currency`) beschrieben. Dies sind `$` und ein beliebiges Unicode-Zeichen, das weder ein Buchstabe noch eine Zahl ist (in SDF repräsentiert durch: `\253`). Durch die Regel in Zeile 14 wird definiert, dass ein Wert von einem optionalen Währungszeichen gefolgt werden kann. Schließlich wird in Zeile 5 `Start` als das Startsymbol festgelegt, von dem aus jede Eingabe geparst werden soll.

```
257: cf(Currency) -> cf(Currency?)
258:  -> cf(Currency?)
259: cf(LAYOUT) -> cf(LAYOUT?)
260:  -> cf(LAYOUT?)
261: lex([0-9]+) -> lex([0-9]*)
262:  -> lex([0-9]*)
263: lex([0-9]*) -> cf([0-9]*)
264: lex([0-9]*) lex([0-9]*) -> lex([0-9]*) {left}
265: lex([0-9]*) lex([0-9]+) -> lex([0-9]+)
266: lex([0-9]+) lex([0-9]*) -> lex([0-9]+)
267: lex([0-9]+) lex([0-9]+) -> lex([0-9]+) {left}
268: lex([0-9]+) -> cf([0-9]+)
269: [0-9] -> lex([0-9]+)
...
279: cf(Value) cf(LAYOUT?) cf(Currency?)
                        -> cf(Start) {definedAs("...")}
280: lex(Currency) -> cf(Currency)
281: [\$\253] -> lex(Currency) {definedAs("...")}
282: [1-9] lex([0-9]*) -> lex(Value) {definedAs("...")}
283: lex(Value) -> cf(Value)
284: [0] -> lex(Value) {definedAs("...")}
285: cf(LAYOUT?) cf(Start) cf(LAYOUT?) -> <START>
...
```

Listing 8.2: Die aus der Beispielgrammatik generierten BNF-Regeln.

Bei der Generierung der Parse-Tabelle werden die Regeln der erzeugten Grammatik in BNF umgewandelt, wie in Abschnitt 5.2.3.1 beschrieben. Listing 8.2 zeigt die aus der Beispielgrammatik *generierten BNF-Regeln*, wie sie der `TreeTraverser` vom `Interpreter` übergeben bekommt[1]. Jede Regel wird mit einer eindeutige Nummer ≥ 257 versehen, über die sie im internen Parse-Forest referenziert wird. Im gezeigten Listing sind die Re-

[1] Der `TreeTraverser` erhält die Regeln als ein Baum aus Objekt-Knoten. Zum besseren Verständnis wird in Listing 8.2 die textuelle Repräsentation angegeben.

geln 270 bis 278 sowie 286 bis 291 weggelassen, da sie der Definition von `LAYOUT` dienen und für das Verständnis dieses Beispiels keine Bedeutung haben.

Regel 285 ist die Start-Regel, von der aus jeder Parse-Vorgang gestartet wird. Sie definiert das in der Grammatik angegebene Start-Symbol, das vom optionalen `LAYOUT` umschlossen ist. Diese Option wird durch die beiden Regeln 259 und 260 ausgedrückt.

Die erste hier vorgestellte in der Grammatik definierte Regel ist 284, da sie mit dem Attribut `definedAs` versehen ist. Im Listing wurde sie aus diesem Grund hervorgehoben. Da der Kopf aus einer lexikalischen Variablen besteht, wird durch Regel 283 die Abbildung auf eine gleichbenannte kontextfreie Variable definiert.

Auch Regel 282 wurde in der Grammatik definiert. Im Rumpf enthält sie die lexikalische Variable `lex([0-9]*)`, deren Semantik durch die Regeln 261 bis 269 ausgedrückt wird. Bei der BNF-Generierung werden CharacterClasses optimiert, wie beispielsweise aus `~[\15]` die CharacterClass `[\0-\14\16-\255]` wird. Dabei geht die *ursprüngliche Darstellung* unwiderruflich verloren, weshalb die Syntax der in der Grammatik definierten Regel zur Unterstützung der Nutzer beim Debugging *in den Klammern des `definedAs`-Attributs* gespeichert wird. In diesem Beispiel war allerdings keine Optimierung nötig.

Durch die Regel 281 wird die Überführung der lexikalischen Variable `lex(Currency)` in die gleichbenannte kontextfreie Variable `cf(Currency)` (Regel 280) bedingt.

Da es sich bei Regel 279 um eine kontextfreie Regel handelt, wird zwischen konkatenierten Termen im Regelrumpf die Option `cf(LAYOUT?)` ergänzt. Die Semantik dieser Option wird durch die Regeln 257 und 258 ausgedrückt.

Mithilfe der Parse-Tabelle, die aus den BNF-Regeln erzeugt wurde, wird nun *die Eingabe* *"50 €"* geparst. Abbildung 8.2 zeigt den resultierenden internen Parse-Forest. Bevor dieser dem `TreeTraverser` übergeben wird, teilt der `Interpreter` ihm noch den Namen der geparsten Datei, ihren Inhalt, sowie alle Regeln mit. Letztere kapselt der `TreeTraverser` in Instanzen der Wrapper-Klasse `Rule`, wie sie in Abschnitt 8.2 vorgestellt werden.

Im *internen Parse-Forest* repräsentieren die inneren Knoten und Blätter, die für ε-Produktionen stehen, je eine angewendete Regel, die über ihre Nummer eindeutig identifiziert wird. Die zur Herleitung benötigten in der Grammatik definierten Regeln sind in der Abbildung markiert. Blätter, die für die Erkennung eines Lexems stehen, enthalten den Dezimalwert des erkannten Zeichens. Sollte dieser Wert für ein darstellbares Zeichen stehen, wird dieses hinter einem Gleichheitszeichen angegeben.

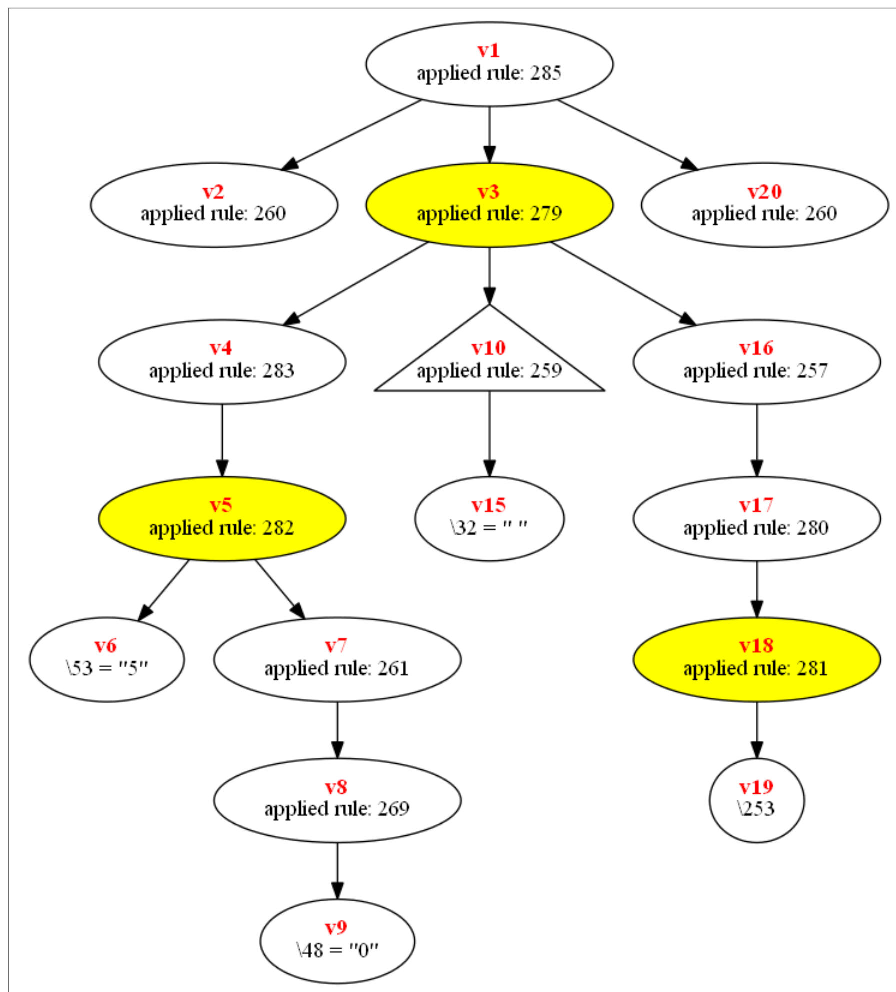


Abbildung 8.2: Der erzeugte interne Parse-Forest für die Eingabe `"50 €"`.

Darüber hinaus wurde jeder Knoten mit einem *Bezeichner vi* versehen, bei der *i* die Reihenfolge angibt, in der der interne Parse-Forest vom `TreeTraverser` per „depth first"-Strategie traversiert wird. Der Knoten $v10$ ist dabei der Wurzelknoten der linearen Herleitung des Lexems im Blatt $v15$. Die weggelassenen Knoten $v11$ bis $v14$ stehen für die Anwendungen von `LAYOUT`-Regeln und würden bei der folgenden Beschreibung der Traversierung keine neuen Erkenntnisse bringen.

Traversierung des internen Parse-Forests

Der interne Parse-Forest aus Abbildung 8.2 wird von der `TreeTraverser`-Komponente per „depth first"-Strategie traversiert. Dabei wird neben dem Call-Stack ein weiterer Stack `stack` benötigt, der den Pfad zur Wurzel speichert (siehe Abschnitt 8.3). Da die

Eingabe nur aus einer einzigen Zeile besteht, werden für jedes Element des Stacks in diesem Beispiel nur der `offset`- und `length`-Wert des erkannten Lexems, sowie das Ergebnis der Regelanwendung mitgeführt. Die Referenz auf das `Rule`-Objekt, welches die jeweils angewendete Regel repräsentiert, wird in diesem Beispiel durch die Nummer der Regel ersetzt. Jedes Element des Stacks besitzt auch eine Referenz `ruleApply` auf das Elternelement, welches für die letzte Anwendung einer in der Grammatik definierten Regel steht. Darüber hinaus ist dem Stack der Name der Eingabedatei und ihr Inhalt als Character-Array bekannt, werden jedoch nicht aufgeführt, da sie in diesem Beispiel unveränderlich sind.

Während der Taversierung veranlasst der `TreeTraverser` den `GraphBuilder` die in der Grammatik definierten semantischen Aktionen auszuführen. Da allerdings in diesem Beispiel keine semantischen Aktionen definiert sind, verändert sich dadurch nichts.

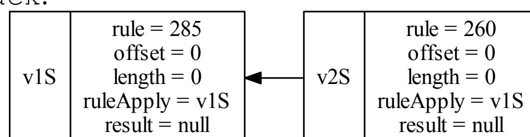
Betrete Knoten $v1$

Der `TreeTraverser` erzeugt ein neues Element $v1S$ auf dem Stack. Dabei hat das Attribut `length` den Wert `0`, um zu signalisieren, dass durch diese Regelanwendung zum aktuellen Traversierungszeitpunkt noch kein Zeichen der Eingabe erkannt ist. Da dies die Anwendung einer Start-Regel ist, referenziert `ruleApply` sich selbst.
Regel 285: cf(LAYOUT?) cf(Start) cf(LAYOUT?) -> <START>
stack:

v1S	rule = 285 offset = 0 length = 0 ruleApply = v1S result = null

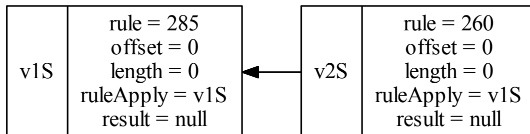
Betrete Knoten $v2$

Beim Betreten des Knotens $v2$ wird ein neues Element auf den `stack` gelegt und `ruleApply` auf $v1S$ gesetzt.
Regel 260: ε `-> cf(LAYOUT?)`
stack:



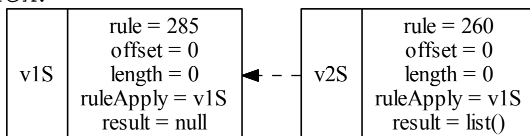
Verlasse Knoten $v2$

Beim Verlassen des Knotens $v2$ wird das `length`-Attribut auf den Wert 0 gesetzt und `result` behält den Wert `null`, da dies die Anwendung einer ε-Regel ist.
Regel 260: `ε -> cf(LAYOUT?)`
`stack`:



Rückkehr zu Knoten $v1$

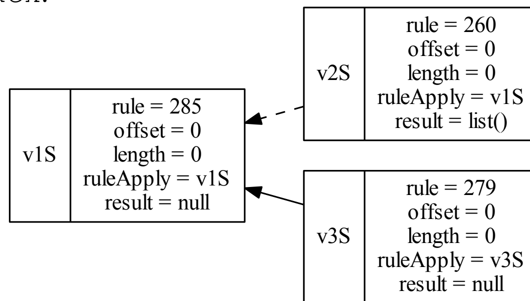
Bei der Rückkehr zu Knoten $v1$ wird das `length`-Attribut von $v1S$ auf den Wert von $v2S$ gesetzt. Da es sich bei dem soeben verlassenen Knoten um die Anwendung einer `LAYOUT`-Regel handelt, wird als default-Wert eine leere Liste erzeugt und `result` zugewiesen. Der Knoten $v2S$ kann noch nicht vom `stack` gelöscht werden, da er noch über semantische Aktionen zugreifbare Werte enthalten kann. Um zu verdeutlichen, dass sich $v2S$ nicht mehr auf dem eigentlichen Stack befindet, wird die Kante gestrichelt dargestellt.
Regel 285: `cf(LAYOUT?) cf(Start) cf(LAYOUT?) -> <START>`
`stack`:



Betrete Knoten $v3$

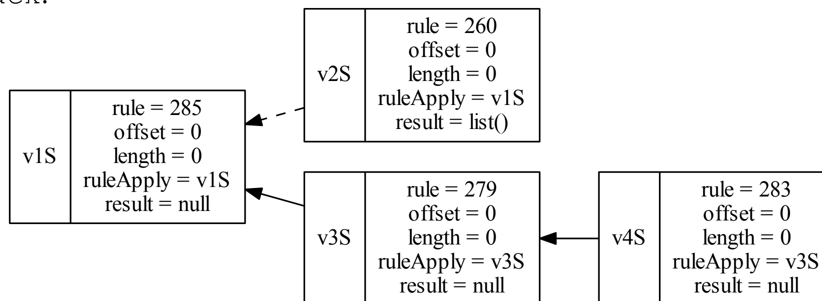
Beim Betreten des Knotens $v3$ wird ein neues Element $v3S$ auf den Stack gelegt. Da es sich bei diesem Knoten um die Anwendung einer in der Grammatik definierten Regel handelt, wird `ruleApply` auf $v3S$ gesetzt. Sollte in dieser Aktion `$` ein Wert w zugewiesen bekommen, so wird `result` auf diesen Wert w gesetzt.

Regel 279: `cf(Value) cf(LAYOUT?) cf(Currency?) -> cf(Start)`
`stack`:



Betrete Knoten $v4$

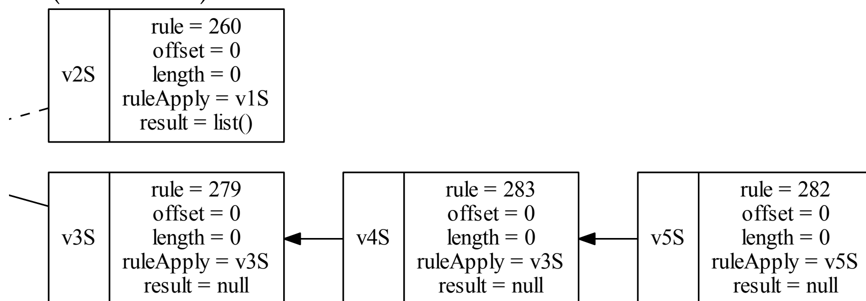
Beim Betreten des Knotens $v4$ wird ein neues Element $v4S$ auf den Stack gelegt. Da diese Regel nicht in der Grammatik definiert wurde, wird `ruleApply` auf `v3S` gesetzt.
Regel 283: `lex(Value) -> cf(Value)`
`stack`:



Betrete Knoten $v5$

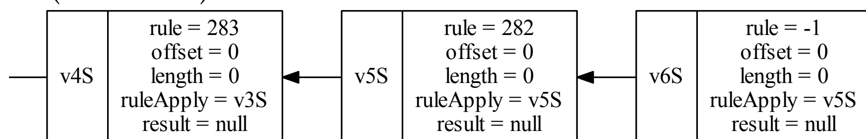
Beim Betreten des Knotens $v5$ wird ein neues Element $v5S$ auf den Stack gelegt. Da es sich bei diesem Knoten um die Anwendung einer in der Grammatik definierten Regel handelt, wird `ruleApply` auf $v5S$ gesetzt.

Regel 282: `[1-9] lex([0-9]*) -> lex(Value)`
`stack` (Ausschnitt):



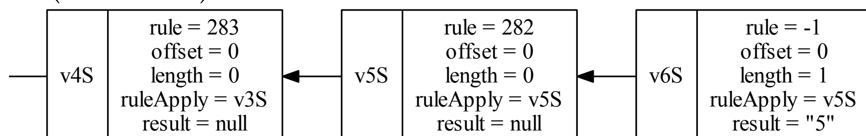
Betrete Knoten $v6$

Beim Betreten des Knotens $v6$ wird ein neues Element $v6S$ auf den Stack gelegt. Bei diesem Knoten handelt es sich um keine Regelanwendung, sondern um die Erkennung des Lexems `"5"`. Aus diesem Grund wird `rule` auf -1, `length` auf 1, `result` auf `"5"` und `ruleApply` auf $v5S$ gesetzt. Für ein erkanntes Zeichen wird keine semantische Aktion ausgeführt.
`stack` (Ausschnitt):



Verlasse Knoten $v6$

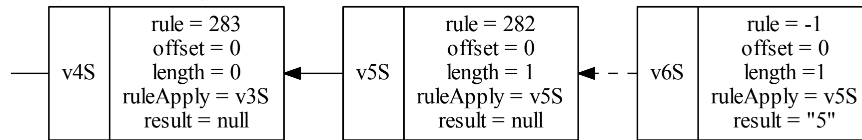
Beim Verlassen des Knotens $v6$ wird `length` auf 1 und `result` auf `"5"` gesetzt.
`stack` (Ausschnitt):



Rückkehr zu Knoten $v5$

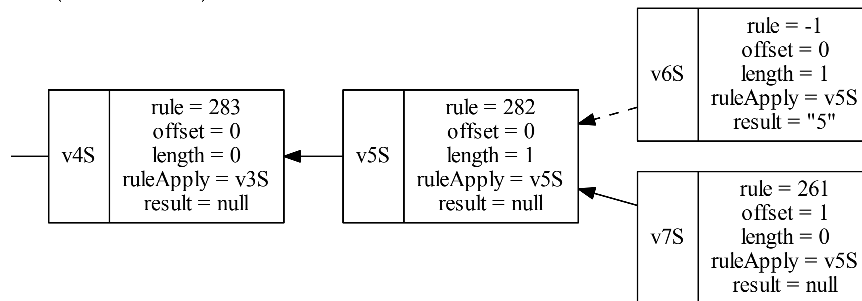
Bei der Rückkehr zum Knoten $v5$ wird `length` auf 1 gesetzt. Sollte in semantischen Aktionen durch `$0` auf den Wert des ersten Terms zugegriffen werden, so kann sein Wert über das Attribut `result` des Stack-Elements $v6S$ erreicht werden.

Regel 282: `[1-9] lex([0-9]*) -> lex(Value)`
`stack` (Ausschnitt):



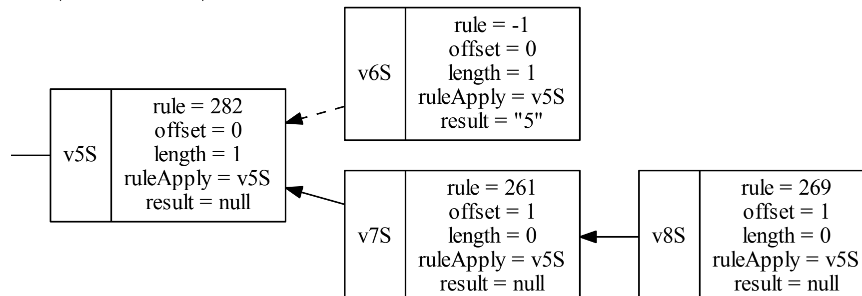
Betrete Knoten $v7$

Beim Betreten des Knotens $v7$ wird ein neues Element $v7S$ auf den Stack gelegt. `offset` wird auf $v5s$.`offset` $+ v5S$.`length` $= 1$ und `ruleApply` auf $v5S$ gesetzt.
Regel 261: `lex([0-9]+) -> lex([0-9]*)`
`stack` (Ausschnitt):



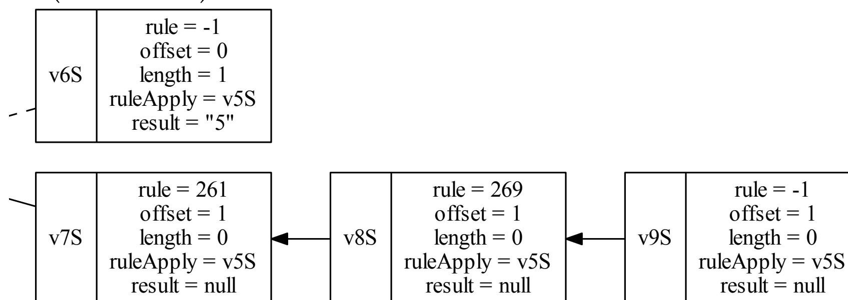
Betrete Knoten $v8$

Beim Betreten des Knotens $v8$ wird ein neues Element $v8S$ auf den Stack gelegt. `offset` wird auf $v7s$.`offset` $+ v7S$.`length` $= 1$ und `ruleApply` auf $v5S$ gesetzt.
Regel 269: `[0-9] -> lex([0-9]+)`
`stack` (Ausschnitt):



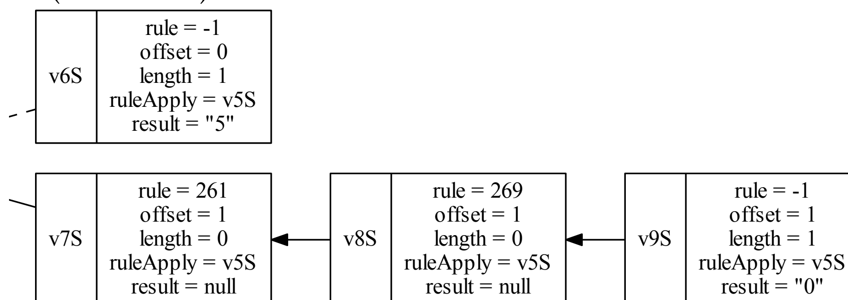
Betrete Knoten $v9$

Beim Betreten des Knotens $v9$ wird ein neues Element $v9S$ auf den Stack gelegt. `offset` wird auf $v8s$.`offset`$+v8S$.`length` $= 1$ und `ruleApply` auf $v5S$ gesetzt. Da durch dieses Blatt das Lexem `"0"` erkannt wird, erhält `rule` den Wert -1.
`stack` (Ausschnitt):



Verlasse Knoten $v9$

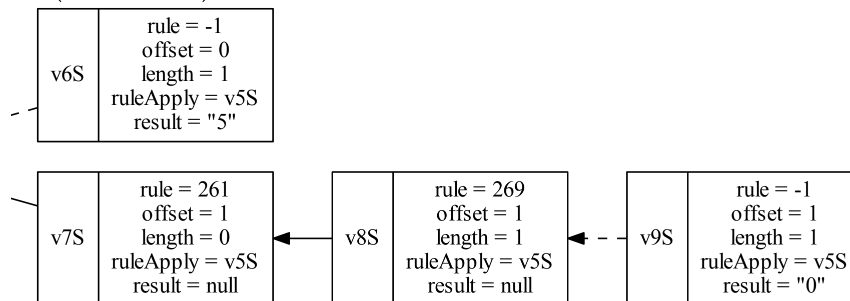
Beim Verlassen des Knotens $v9$ wird `length` auf 1 und `result` auf `"0"` gesetzt.
`stack` (Ausschnitt):



Rückkehr zu Knoten $v8$

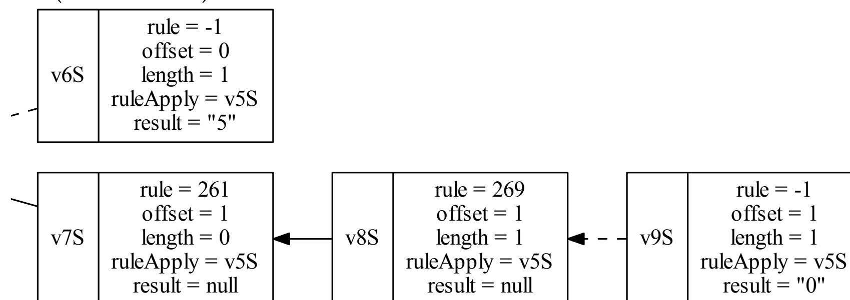
Bei $v8$ handelt es sich um die Anwendung einer aus `[0-9]*` generierten BNF-Regel, bei der ein Element von `[0-9]` erkannt wird. Daher wird bei der Rückkehr zu Knoten $v8$ `length` auf den Wert $v8S$.`length` $+$ $v9S$.`length` gesetzt und im Anschluss die semantische Aktion ausgeführt, die zwischen `[0-9]` und `*` definiert ist. Sollte in dieser Aktion auf `$0` zugegriffen werden, so entspricht dies aufgrund der Sichtbarkeitsebenen in EDL dem `result`-Attribut des Kindelements $v9S$. Sollte auf `$` zugegriffen werden, so ist sein Wert durch $v8s$`.ruleApply.result` zu erreichen.

Regel 269: `[0-9] -> lex([0-9]+)`
`stack` (Ausschnitt):



Verlasse Knoten $v8$

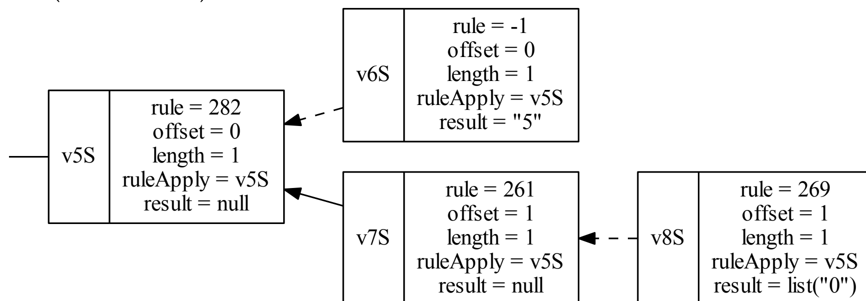
Regel 269: `[0-9] -> lex([0-9]+)`
`stack` (Ausschnitt):



Rückkehr zu Knoten $v7$

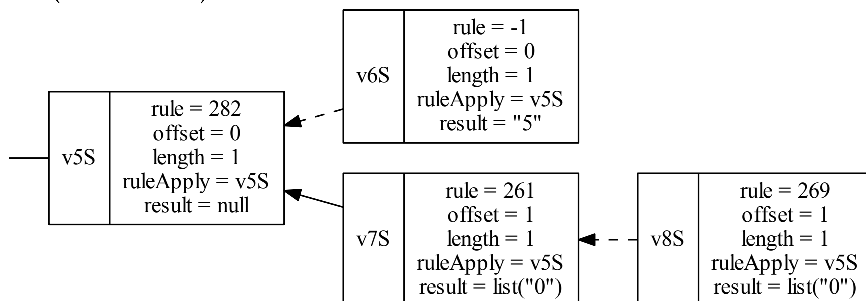
Bei der Rückkehr zu Knoten $v7$ wird ein automatischer Wert generiert, da $v8S$`.result` bisher nicht gesetzt ist und die Repitition nur aus dem einzigen maximalen Term `[0-9]` besteht. Daher wird $v8S$`.result` auf eine neue Liste mit den Ergebnissen aller Kindknoten gesetzt (`list(`$v9S$`.result)`). Des Weiteren kann das Stack-Element $v9S$ gelöscht werden, da es nun nicht mehr erreicht werden kann.
Bei $v7$ handelt es sich um die Anwendung einer aus `[0-9]*` generierten BNF-Regel, aus der direkt kein Zeichen der Eingabe erkannt werden kann. Daher wird bei der Rückkehr zu Knoten $v7$ das Attribut `length` auf den Wert $v7S$`.length` + $v8S$`.length` gesetzt.

Regel 261: `lex([0-9]+) -> lex([0-9]*)`
`stack` (Ausschnitt):



Verlasse Knoten $v7$

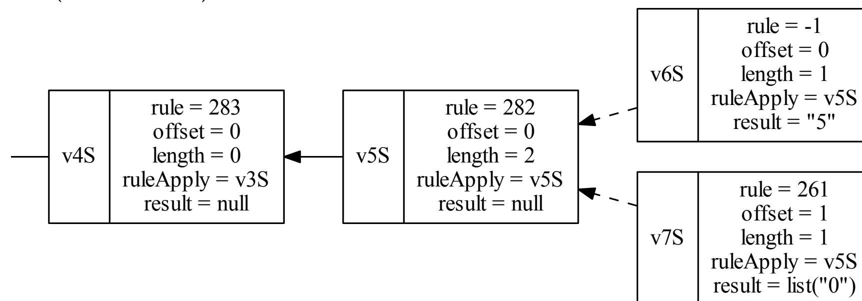
Beim Verlassen des Knotens $v7$ wird ein automatischer Wert generiert. Daher wird `result` auf die Liste des ersten Kinds $v8S$`.result` gesetzt.
Regel 261: `lex([0-9]+) -> lex([0-9]*)`
`stack` (Ausschnitt):



Rückkehr zu Knoten $v5$

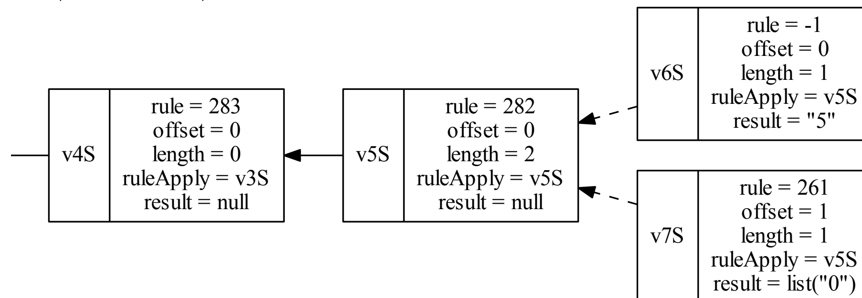
Bei $v5$ handelt es sich um eine in der Grammatik definierten Regel. Daher wird bei der Rückkehr zu Knoten $v5$ das Attribut `length` auf den Wert $v5S$`.length` + $v7S$`.length` gesetzt. Des Weiteren kann das Stack-Element $v8S$ gelöscht werden, da es nun nicht mehr erreicht werden kann.

Regel 282: `[1-9] lex([0-9]*) -> lex(Value)`
`stack` (Ausschnitt):



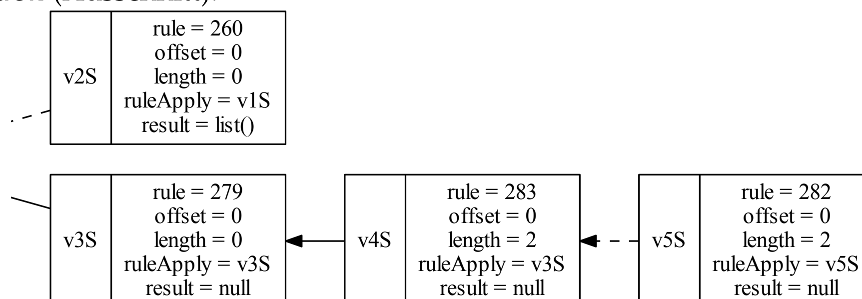
Verlasse Knoten $v5$

Regel 282: `[1-9] lex([0-9]*) -> lex(Value)`
`stack` (Ausschnitt):



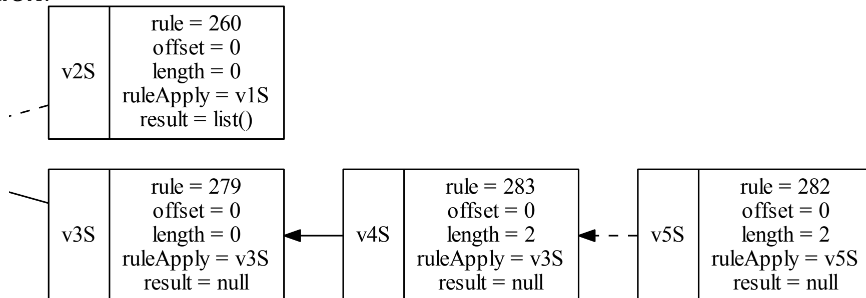
Rückkehr zu Knoten $v4$

Bei der Rückkehr zu Knoten $v4$ können die Stack-Elemente $v6S$ und $v7S$ gelöscht werden, da sie nun nicht mehr erreicht werden können. Des Weiteren wird das Attribut `length` auf den durch die Formel $v4S$.`length` + $v5S$.`length` berechneten Wert gesetzt.
Regel 283: `lex(Value) -> cf(Value)`
`stack` (Ausschnitt):



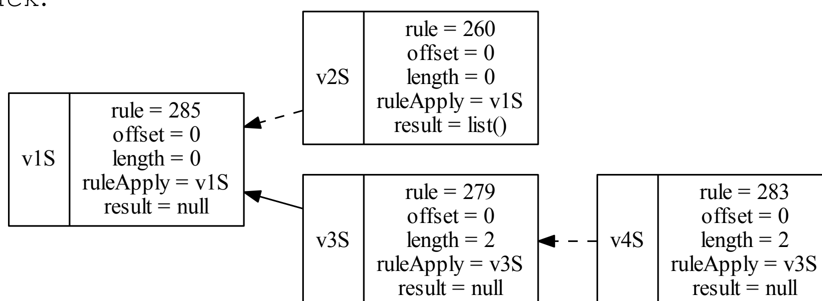
Verlasse Knoten $v4$

Beim Verlassen des Knotens $v4$ wird `result` auf $v5S$`.result` gesetzt, da die angewendete Regel 283 nur `lex(Value)` auf `cf(Value)` abbildet.
Regel 283: `lex(Value) -> cf(Value)`
`stack`:



Rückkehr zu Knoten $v3$

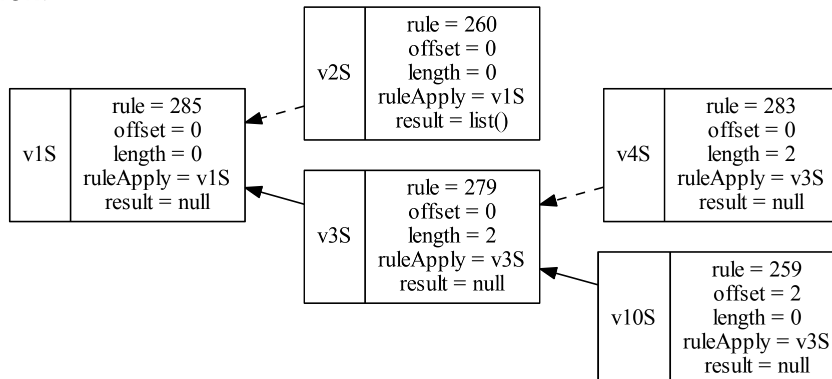
Bei der Rückkehr zu Knoten $v3$ kann das Stack-Element $v5S$ gelöscht und das Attribut `length` auf den durch die Formel $v3S$`.length` + $v4S$`.length` berechneten Wert gesetzt werden.
Regel 279: `cf(Value) cf(LAYOUT?) cf(Currency?) -> cf(Start)`
`stack`:



Betrete Knoten $v10$

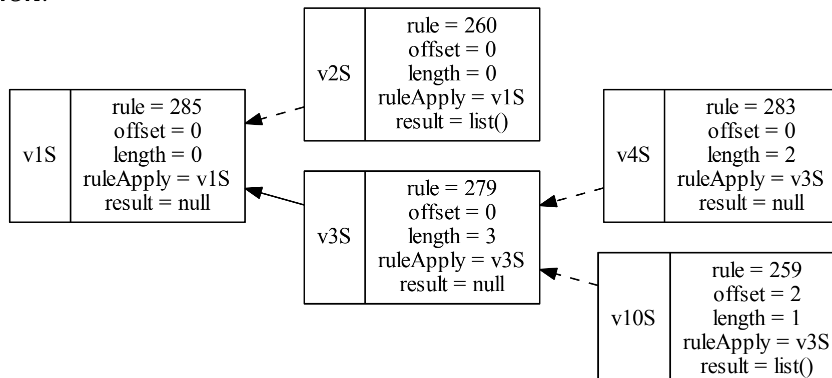
Beim Betreten des Knotens $v10$ wird ein neues Element $v10S$ auf den Stack gelegt. `offset` wird auf $v3s$`.offset` + $v3S$`.length` $= 2$ und `ruleApply` auf $v3S$ gesetzt. Um dieses Beispiel abzukürzen, wird als nächster Schritt die Rückkehr zu $v3$ behandelt.

Regel 259: `cf(LAYOUT) -> cf(LAYOUT?)`
`stack`:



Rückkehr zu Knoten $v3$

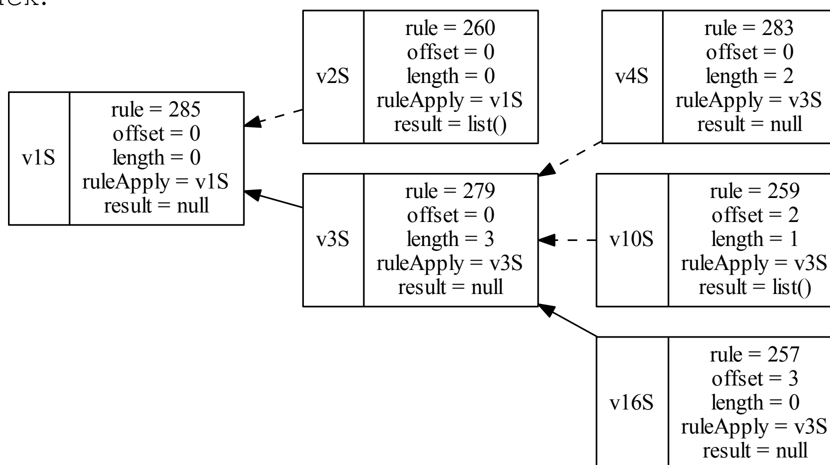
Bei der Rückkehr zu Knoten $v3$ wird das Attribut `length` auf den durch die Formel $v3S.\texttt{length} + v10S.\texttt{length}$ berechneten Wert 3 gesetzt.
Regel 279: `cf(Value) cf(LAYOUT?) cf(Currency?) -> cf(Start)`
`stack`:



Betrete Knoten $v16$

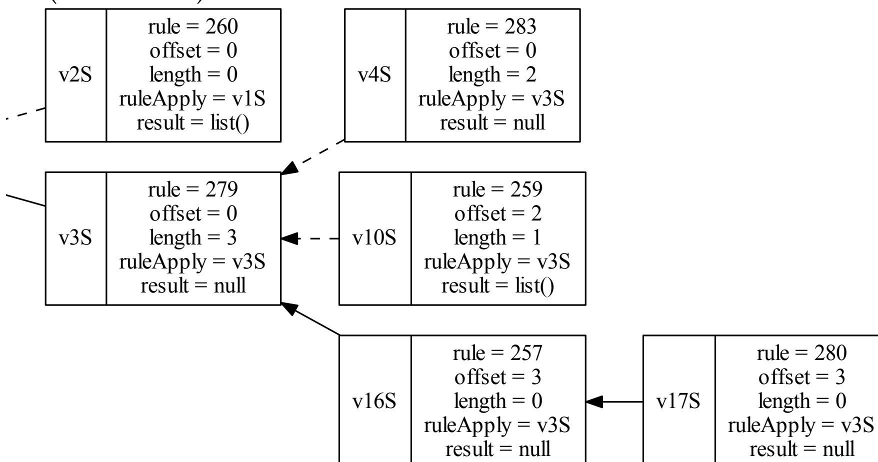
Beim Betreten des Knotens $v16$ wird ein neues Element $v16S$ auf den Stack gelegt. `offset` wird auf $v3s.\texttt{offset} + v3S.\texttt{length} = 3$ und `ruleApply` auf $v3S$ gesetzt.

Regel 257: `cf(Currency) -> cf(Currency?)`
`stack`:



Betrete Knoten $v17$

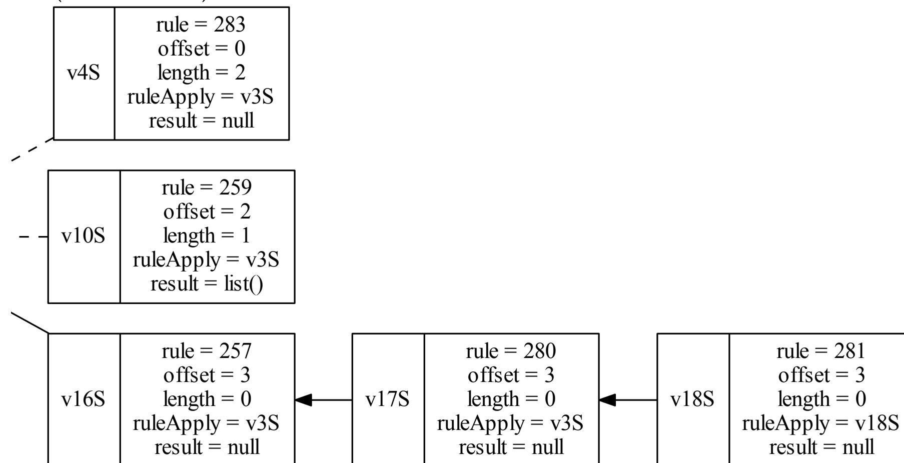
Beim Betreten des Knotens $v17$ wird ein neues Element $v17S$ auf den Stack gelegt. `offset` wird auf $v16s$.`offset` + $v16S$.`length` und `ruleApply` auf $v3S$ gesetzt.
Regel 280: `lex(Currency) -> cf(Currency)`
`stack` (Ausschnitt):



Betrete Knoten $v18$

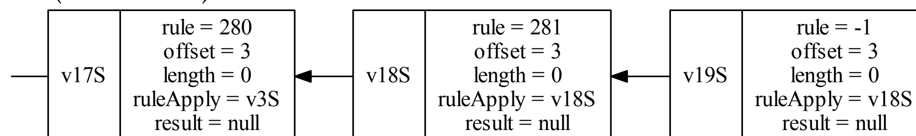
Beim Betreten des Knotens $v18$ wird ein neues Element $v18S$ auf den Stack gelegt. `offset` wird auf $v17s$.`offset` + $v17S$.`length` und `ruleApply` auf $v18S$ gesetzt, da es sich bei dieser Regel um eine in der Grammatik definierte Regel handelt.

Regel 281: `[$\253] -> lex(Currency)`
`stack` (Ausschnitt):



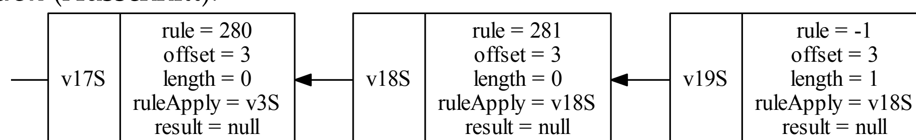
Betrete Knoten $v19$

Beim Betreten des Knotens $v19$ wird ein neues Element $v19S$ auf den Stack gelegt. `offset` wird auf $v18s$.`offset`+$v18S$.`length` und `ruleApply` auf $v18S$ gesetzt. Mit diesem Blatt wurde das Lexem `"€"` erkannt und `rule` wird auf -1 gesetzt.
`stack` (Ausschnitt):



Verlasse Knoten $v19$

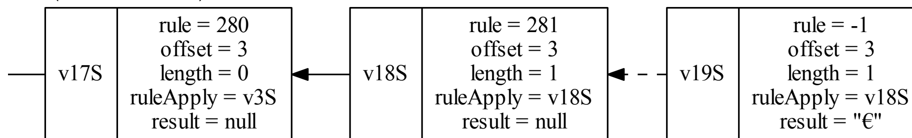
Beim Verlassen des Knotens $v19$ wird `length` auf 1 gesetzt.
`stack` (Ausschnitt):



Rückkehr zu Knoten $v18$

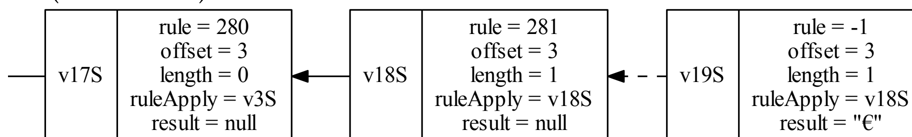
Bei der Rückkehr zu Knoten $v18$ wird das Attribut `length` auf den durch die Formel $v18S$.`length` + $v19S$.`length` berechneten Wert gesetzt.
Da es sich bei dem durch $v19$ erkannten Lexem um ein Unicode-Zeichen handelt, ist im

internen Parse-Forest der Wert `\253` angegeben. Daher muss bei dem im `stack` gespeicherten Inhalt der geparsten Datei nachgesehen werden, welches Zeichen sich an der Position $v19S$`.offset` befindet und dieser Wert $v19S$`.result` zugewiesen werden.
Regel 281: `[$\253] -> lex(Currency)`
`stack` (Ausschnitt):



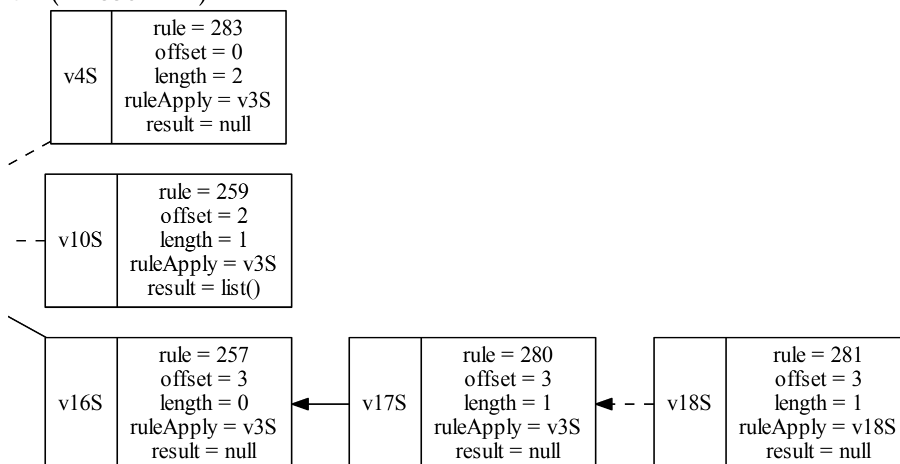
Verlasse Knoten $v18$

Regel 281: `[$\253] -> lex(Currency)`
`stack` (Ausschnitt):



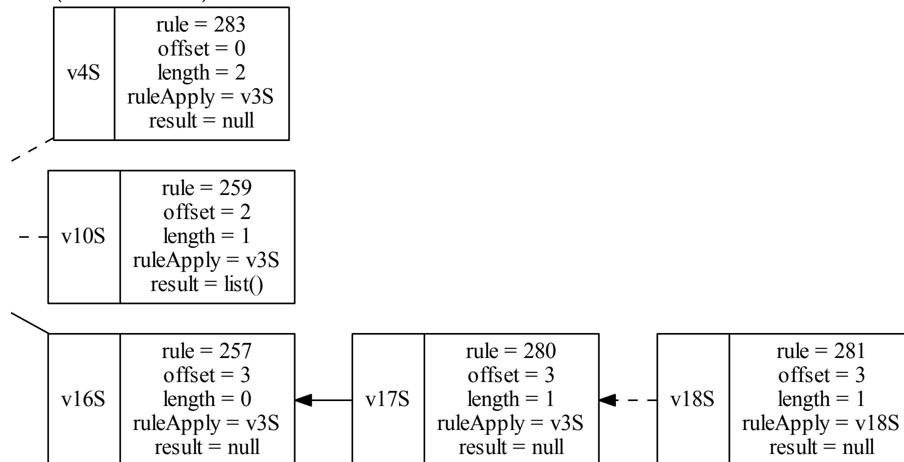
Rückkehr zu Knoten $v17$

Bei der Rückkehr zu Knoten $v17$ wird das Attribut `length` auf den durch die Formel $v17S$`.length` + $v18S$`.length` berechneten Wert gesetzt. Da die angewendete Regel 280 nur zur Überführung von `lex(Currency)` nach `cf(Currency)` dient, wird `result` auf $v18S$`.result` gesetzt. Darüber hinaus kann $v19S$ nun gelöscht werden.
Regel 280: `lex(Currency) -> cf(Currency)`
`stack` (Ausschnitt):



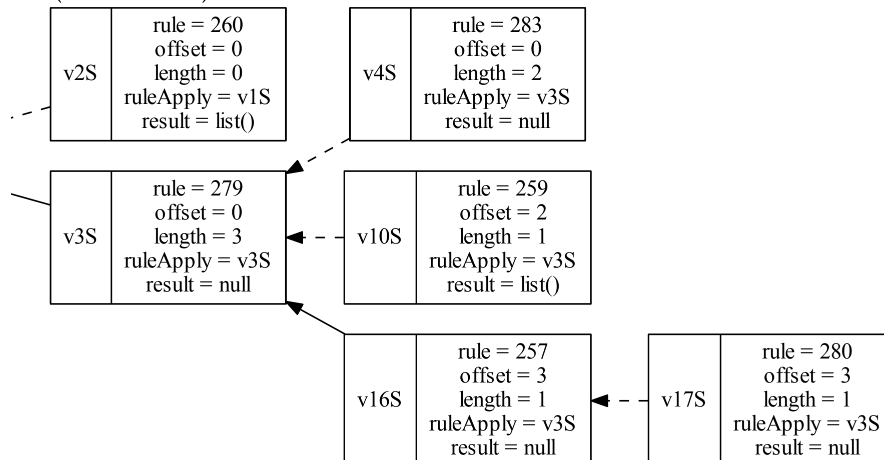
Verlasse Knoten $v17$

Regel 280: `lex(Currency) -> cf(Currency)`
`stack` (Ausschnitt):



Rückkehr zu Knoten $v16$

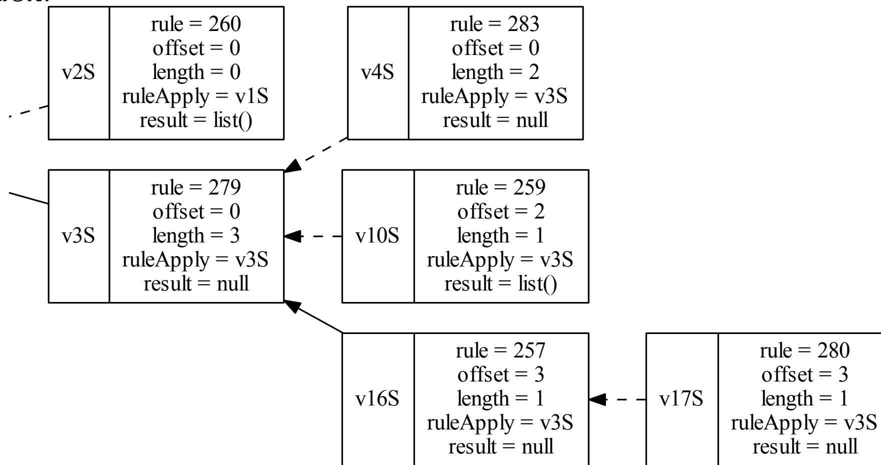
Bei der Rückkehr zu Knoten $v16$ kann das Stack-Element $v18S$ gelöscht und das Attribut `length` auf den durch die Formel $v16S$.`length`+$v17S$.`length` berechneten Wert gesetzt werden.
Regel 257: `cf(Currency) -> cf(Currency?)`
`stack` (Ausschnitt):



Verlasse Knoten $v16$

Regel 257: `cf(Currency) -> cf(Currency?)`

`stack:`

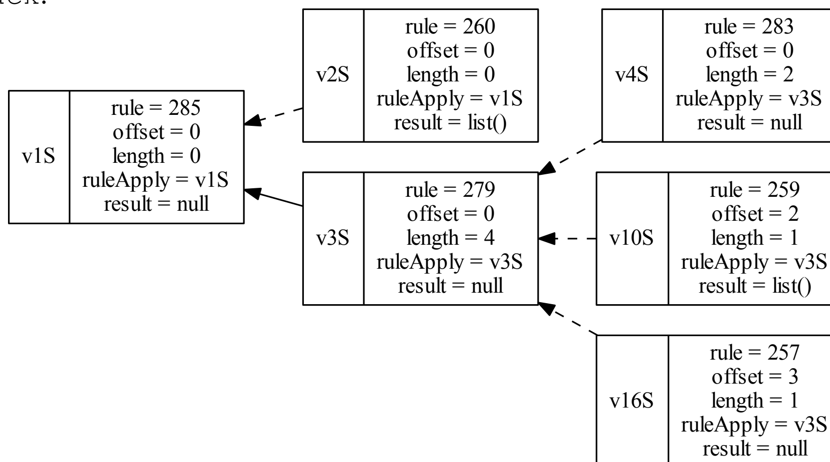


Rückkehr zu Knoten $v3$

Bei der Rückkehr zu Knoten $v3$ wird das Attribut `length` auf den durch die Formel $v3S$`.length` $+$ $v16S$`.length` berechneten Wert gesetzt und $v17S$ gelöscht.

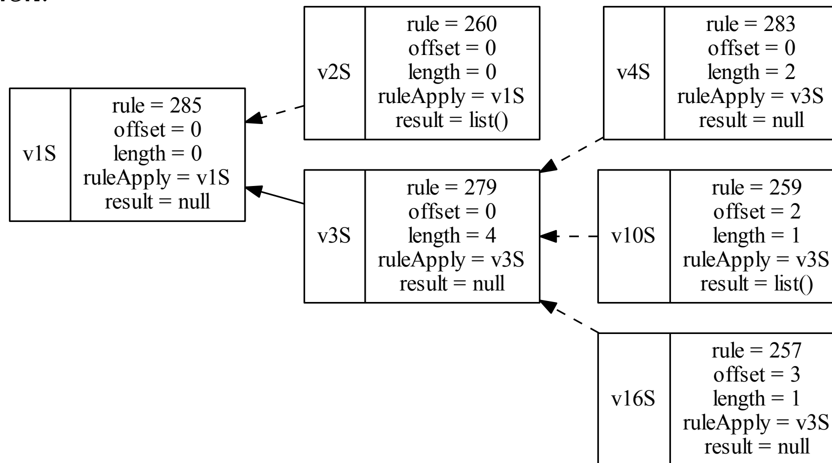
Regel 279: `cf(Value) cf(LAYOUT?) cf(Currency?) -> cf(Start)`

`stack:`



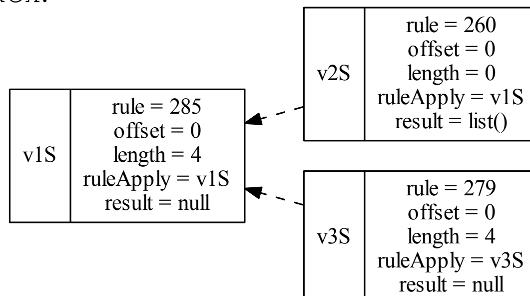
Verlasse Knoten $v3$

Regel 279: `cf(Value) cf(LAYOUT?) cf(Currency?) -> cf(Start)`
`stack`:



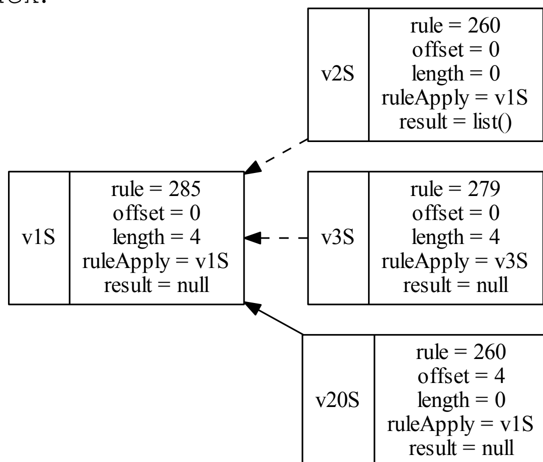
Rückkehr zu Knoten $v1$

Bei der Rückkehr zu Knoten $v1$ wird das Attribut `length` auf den durch die Formel $v1S$.`length` + $v3S$.`length` berechneten Wert gesetzt. Die Stack-Elemente $v4S$, $v10S$ und $v16S$ können gelöscht werden.
Regel 285: `cf(LAYOUT?) cf(Start) cf(LAYOUT?) -> <START>`
`stack`:



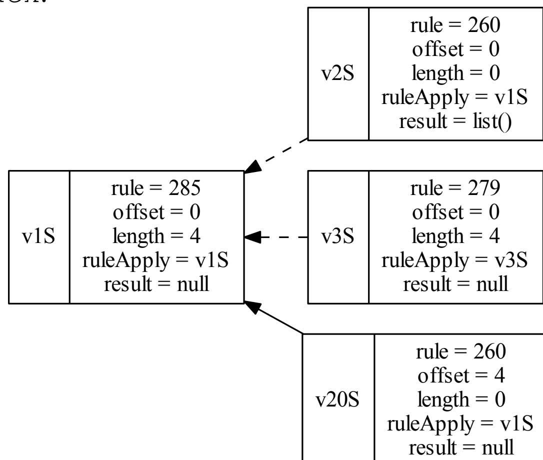
Betrete Knoten $v20$

Beim Betreten des Knotens $v20$ wird ein neues Element auf den `stack` gelegt und `offset` auf den Wert $v1S$`.length+`$v1S$`.offset` gesetzt.
Regel 260: ε `-> cf(LAYOUT?)`
`stack`:



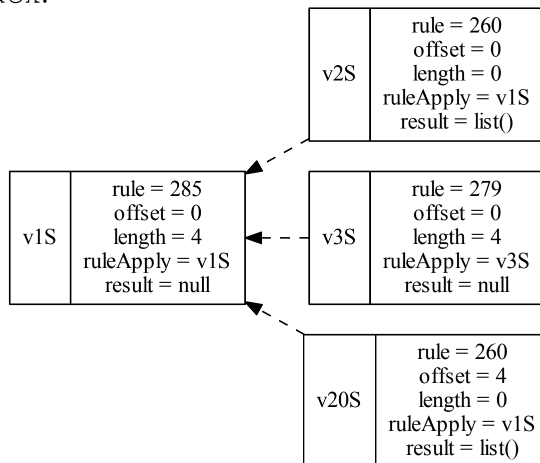
Verlasse Knoten $v20$

Beim Verlassen des Knotens $v20$ wird das `length`-Attribut auf den Wert 0 gesetzt.
Regel 260: ε `-> cf(LAYOUT?)`
`stack`:



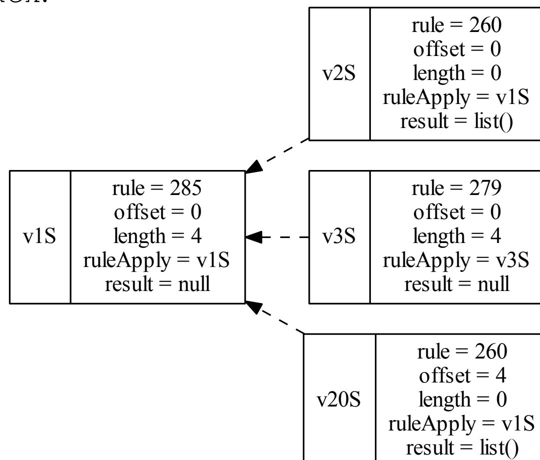
Rückkehr zu Knoten $v1$

Bei der Rückkehr zu Knoten $v1$ wird das Attribut `length` auf den durch die Formel $v1S$.`length` + $v20S$.`length` berechneten Wert gesetzt und $v20S$.`result` bekommt eine leere Liste, da dies die Anwendung einer `LAYOUT`-Regel ist.
Regel 285: `cf(LAYOUT?) cf(Start) cf(LAYOUT?) -> <START>`
`stack`:



Verlasse Knoten $v1$

Regel 285: `cf(LAYOUT?) cf(Start) cf(LAYOUT?) -> <START>`
`stack`:



8.2 Repräsentation einer Regel

Die `TreeTraverser`-Komponente erhält vom `Interpreter` alle in der Grammatik definierten sowie bei der BNF-Transformation erzeugten Regeln. Jede *Regel* wird dabei durch eine *Baumstruktur* repräsentiert. Da der `EDL-Parser` bestimmte Informationen aus diesem Baum extrahieren muss, wurde eine Klasse implementiert, die die Baumstruktur wrapped und eine Schnittstelle zur Extraktion der benötigten Informationen bietet. Einmal extrahierte Daten werden zur späteren Wiederverwendung gespeichert.

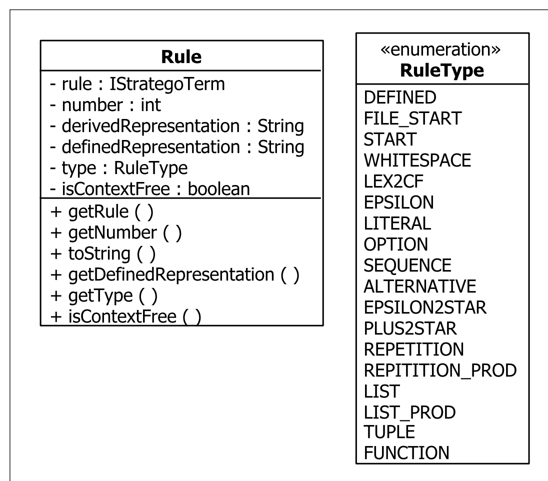


Abbildung 8.3: Die Repräsentation einer Regel.

Abbildung 8.3 zeigt die Wrapper-Klasse `Rule`. Um eine `Rule`-Instanz zu erzeugen, muss ein *Konstruktor* mit der Baumrepräsentation einer Regel und ihrer eindeutig identifizierende Nummer aufgerufen werden. Erstere wird im Feld `rule` gespeichert und ist über den Getter

`getRule(): IStrategoTerm`

abrufbar. Letztere wird im Feld `number` vermerkt und auf sie kann über den Getter

`getNumber(): int`

zugegriffen werden.

Zum Debuggen wird eine String-Repräsentation der Regel benötigt. Beim ersten Aufruf der Methode

`toString(): String`

wird die gewünschte String-Repräsentation des Regel-Baums erzeugt, für spätere Aufrufe im Feld `derivedRepresentation` gespeichert und schließlich zurückgegeben. Für in der Grammatik definierte Regeln gibt es noch eine alternative String-Darstellung in der Form, wie die Regel vom Nutzer geschrieben wurde. Damit diese Darstellung bei der BNF-Transformation nicht verloren geht, wurde sie in den generierten SDF-Modulen

durch das Attribut `definedAs` gespeichert und ist im Regel-Baum zu finden. Durch einen Aufruf von

```
getDefinedRepresentation(): String
```

kann diese Darstellung aus dem Baum extrahiert werden. In dem String-wertigen Feld `definedRepresentation` wird die extrahierte Darstellung für zukünftige Aufrufe dieser Methode gespeichert.

Die Methode

```
isContextFree(): boolean
```

überprüft, ob die Regel `rule` kontextfrei ist. Dies ist der Fall, wenn der Regelkopf bzw. im Falle des Typs FUNCTION das erste Element im Regelrumpf mit `cf` beginnt. Dieser Rückgabewert ist im Feld `isContextFree` gespeichert, welches beim Konstruktoraufruf initialisiert wird.

`RuleType` einer Regel

Wie anhand der exemplarischen Arbeitsweise des `EDL-Parsers` in Abschnitt 8.1 zu erkennen, sind die Aktionen abhängig vom Typ einer Regel, der durch die Methode

```
getType(): RuleType
```

abgefragt werden kann. Nach ihrem erstmaligem Aufruf wird der Typ im Feld `type` gespeichert. Welche Regel-Typen es gibt, wird in der Enumeration `RuleType` definiert. In der Grammatik definierte Regeln haben den Typ:

DEFINED. Regeln dieses Typs wurden in der Grammatik definiert. Sie sind über die Existenz des Attributs `definedAs` identifizierbar.

Die folgenden Typen beschreiben Regeln, die bei der BNF-Transformation erzeugt wurden:

FILE_START. Regeln dieses Typs haben einen Kopf, der ausschließlich aus `<Start>` besteht. Sie repräsentieren Regeln, die zur Erkennung von ganzen Eingabedateien dienen.

START. Dieser Typ identifiziert Regeln, die den Start der Herleitung der Eingabe repräsentieren. Für jedes in der Grammatik definierte Start-Symbol *S* wird eine Regel erzeugt, die *S* auf `<START>` abbildet. Ihr Kopf besteht daher nur aus `<START>`.

WHITESPACE. Regeln vom Typ WHITESPACE dienen der Herleitung der in kontextfreien Regeln automatisch ergänzten `cf(LAYOUT?)`. In jeder Grammatik gibt es genau drei Regeln dieses Typs:

```
cf(LAYOUT) -> cf(LAYOUT?)
ε -> cf(LAYOUT?)
cf(LAYOUT) cf(LAYOUT) -> cf(LAYOUT) {left}
```

LEX2CF. Damit eine Variable V, die durch eine lexikalische Regel definiert wurde, auch in kontextfreien Regeln verwendet werden kann, wird die folgende Regel vom Typ LEX2CF erzeugt:

`lex(`V`) -> cf(`V`)`

EPSILON. Dieser Typ beschreibt ε-Regeln, die aus einer leeren Sequence `()` oder Option T`?` erzeugt wurden:

ε `-> cf(`T`?)`

ε `-> lex(`T`?)`

ε `-> cf(())`

ε `-> lex(())`

LITERAL. Für mit " oder ' umschlossene Literale L werden Regeln erzeugt, deren Kopf aus `"`L`"` bzw. `'`L`'` besteht. Diese Regeln haben den Typ `LITERAL`.

OPTION. Regeln vom Typ OPTION realisieren die Erkennung von Term T der Option T`?`, sofern sie nicht vom Typ WHITESPACE sind. Diese Regeln haben eine der folgenden Formen:

`cf(`T`) -> cf(`T`?)`

`lex(`T`) -> lex(`T`?)`

SEQUENCE. Regeln dieses Typs, die die Semantik der Sequence `(`$T_1 \ldots T_n$`)` beschreiben, haben die Form:

T_1 `cf(LAYOUT?) ... cf(LAYOUT?)` T_n `-> cf((`$T_1 \ldots T_n$`))`

$T_1 \ldots T_n$ `-> lex((`$T_1 \ldots T_n$`))`

ALTERNATIVE. Regeln vom Typ ALTERNATIVE realisieren die Semantik der Alternative T_1 `| ... |` T_n. Sie haben eine der folgenden Formen:

T_1 `-> cf(`T_1 `| ... |` T_n`)`

`...`

T_n `-> cf(`T_1 `| ... |` T_n`)`

T_1 `-> lex(`T_1 `| ... |` T_n`)`

`...`

T_n `-> lex(`T_1 `| ... |` T_n`)`

EPSILON2STAR. Dies ist der Typ von Regeln, die zur Erkennung des leeren Worts aus $T\star$ und $\{T_1\ \ T_2\}\star$ dienen. Sie haben die folgende Form:

ε `->` `cf(`$T\star$`)`

ε `->` `cf({`$T_1\ \ T_2$`}*)`

ε `->` `lex(`$T\star$`)`

ε `->` `lex({`$T_1\ \ T_2$`}*)`

PLUS2STAR. PLUS2STAR ist der Typ von Regeln, die $T+$ und $\{T_1\ \ T_2\}+$ in $T\star$ und $\{T_1\ \ T_2\}\star$ überführen. Sie haben die folgende Form:

`cf(`$T+$`) -> cf(`$T\star$`)`

`cf({`$T_1\ \ T_2$`}+) -> cf({`$T_1\ \ T_2$`}*)`

`lex(`$T+$`) -> lex(`$T\star$`)`

`lex({`$T_1\ \ T_2$`}+) -> lex({`$T_1\ \ T_2$`}*)`

REPETITION. Dieser Typ identifiziert Regeln, die zur Realisierung der Semantik von $T\star$ bzw. $T+$ erzeugt werden und zur Herleitung von mindestens zwei T dienen. Ihre Form baut sich wie folgt auf:

`cf(`$T\star$`) cf(LAYOUT?) cf(`$T\star$`) -> cf(`$T\star$`) {left}`

`cf(`$T\star$`) cf(LAYOUT?) cf(`$T+$`) -> cf(`$T+$`)`

`cf(`$T+$`) cf(LAYOUT?) cf(`$T\star$`) -> cf(`$T+$`)`

`cf(`$T+$`) cf(LAYOUT?) cf(`$T+$`) -> cf(`$T+$`) {left}`

`lex(`$T\star$`) lex(`$T\star$`) -> lex(`$T\star$`) {left}`

`lex(`$T\star$`) lex(`$T+$`) -> lex(`$T+$`)`

`lex(`$T+$`) lex(`$T\star$`) -> lex(`$T+$`)`

`lex(`$T+$`) lex(`$T+$`) -> lex(`$T+$`) {left}`

REPETITION_PROD. REPETITION_PROD ist der Typ von Regeln, die im Rahmen der BNF-Transformation von $T\star$ und $T+$ erzeugt werden und zur Erkennung eines T führen. Ihr Aufbau entspricht einer der folgenden Regeln:

T `-> cf(`$T+$`)`

T `-> lex(`$T+$`)`

LIST. Dieser Typ identifiziert Regeln, die die Semantik von {T_1 T_2}* bzw. {T_1 T_2}+ realisieren, zur Herleitung von mindestens zwei T_1 und der Erkennung eines T_2 dienen. Ihre Form baut sich wie folgt auf:

```
cf({T1 T2}*) cf(LAYOUT?) T2 cf(LAYOUT?) cf({T1 T2}*)
        -> cf({T1 T2}*) {left}
cf({T1 T2}*) cf(LAYOUT?) T2 cf(LAYOUT?) cf({T1 T2}+)
        -> cf({T1 T2}+)
cf({T1 T2}+) cf(LAYOUT?) T2 cf(LAYOUT?) cf({T1 T2}*)
        -> cf({T1 T2}+)
cf({T1 T2}+) cf(LAYOUT?) T2 cf(LAYOUT?) cf({T1 T2}+)
        -> cf({T1 T2}+) {left}
lex({T1 T2}*) T2 lex({T1 T2}*) -> lex({T1 T2}*) {left}
lex({T1 T2}*) T2 lex({T1 T2}+) -> lex({T1 T2}+)
lex({T1 T2}+) T2 lex({T1 T2}*) -> lex({T1 T2}+)
lex({T1 T2}+) T2 lex({T1 T2}+) -> lex({T1 T2}+) {left}
```

LIST_PROD. LIST_PROD ist der Typ von Regeln, die im Rahmen der BNF-Transformation von {T_1 T_2}* und {T_1 T_2}+ erzeugt werden und zur Erkennung eines T_1 führen. Ihr Aufbau entspricht einer der folgenden Regeln:

```
T1 -> cf({T1 T2}+)
T1 -> lex({T1 T2}+)
```

TUPLE. Regeln dieses Typs entstehen bei der Transformation von $<T_1, \ldots, T_n>$. Sie haben die folgende Form:

```
"<" cf(LAYOUT?) T1 ","cf(LAYOUT?) ... cf(LAYOUT?) ","
        cf(LAYOUT?) Tn cf(LAYOUT?) ">" -> cf(<T1,...,Tn>)
"<" T1 "," ... "," Tn ">" -> lex(<T1,...,Tn>)
```

FUNCTION. Regeln vom Typ FUNCTION werden bei der BNF-Transformation aus dem Term ($T_1 \ldots T_n$ => T) erzeugt. Sie haben die Form:

```
cf((T1...Tn => T)) cf(LAYOUT?) "(" cf(LAYOUT?)
      T1 cf(LAYOUT?) ... cf(LAYOUT?) Tn cf(LAYOUT?) ")" -> T
lex((T1...Tn => T)) "(" T1...Tn ")" -> T
```

8.3 Realisierung des Stacks

Wie anhand des Beispiels in Abschnitt 8.1 zu erkennen, wird zur Ausführung der semantischen Aktionen ein Stack benötigt, der den Pfad zur Wurzel des Parse-Forests speichert.

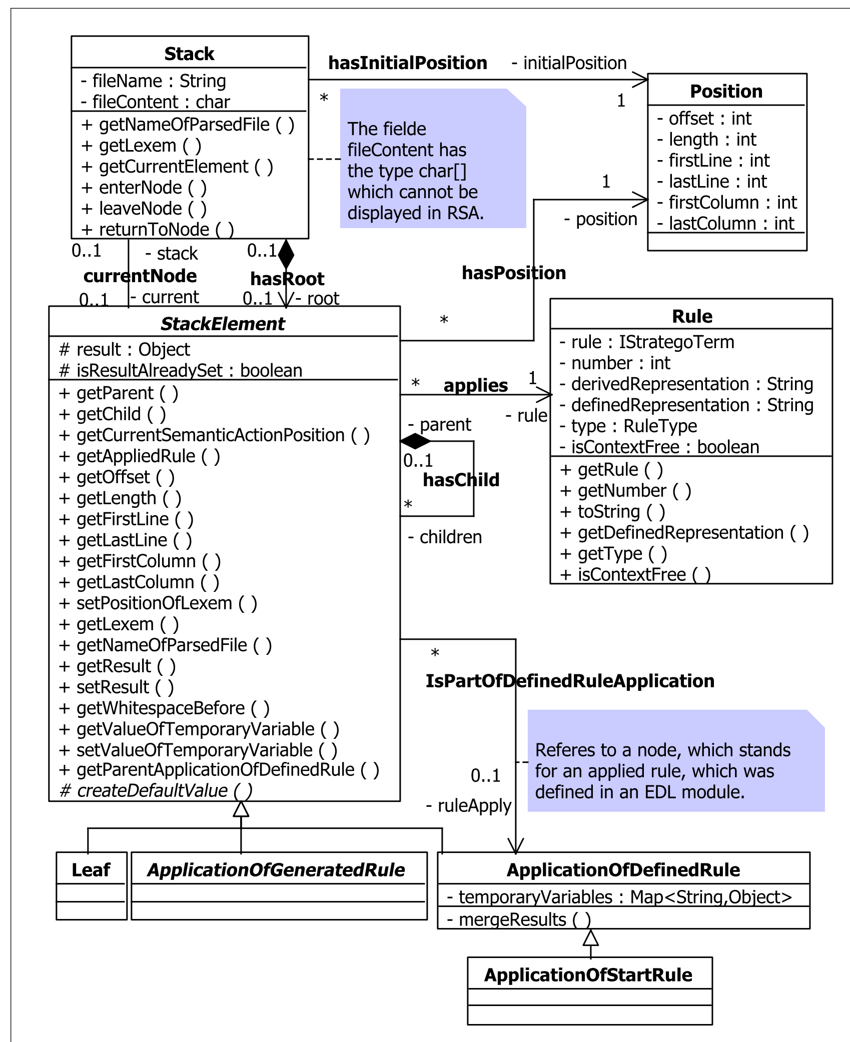


Abbildung 8.4: Der Stack.

Abbildung 8.4 zeigt den Aufbau des Stacks, der durch die Klasse `Stack` realisiert wird. Bei der Erzeugung eines neuen `Stack`-Elements wird der Name der geparsten Datei `fileName` als String, der zu parsende Inhalt `fileContent` als char-Array und seine Position `initialPosition` in der Datei übergeben. Erstere ist über die Methode

```
getNameOfParsedFile(): String
```

abrufbar. Über die Methode

```
getLexem(offset: int, length: int): String
```

kann das durch eine Regelanwendung erkannte Lexem zurückgegeben werden. Durch `offset` und `length` wird der benötigte Teil von `fileContent` eindeutig identifiziert.

Elemente des Stacks sind Instanzen vom abstrakten Typ `StackElement`. Das Bottom-Element des Stacks kann über `root` und das aktuelle Element über `current` identifiziert

werden. Letzteres kann über die Methode

`getCurrentElement(): StackElement`

abgefragt werden.

`StackElement` hat drei direkte Subklassen. Sollte die angewendete Regel in der Grammatik definiert worden sein, so wird die Klasse `ApplicationOfDefinedRule` instanziiert. Ein Objekt ihrer Subklasse `ApplicationOfStartRule` wird erzeugt, wenn eine Regel vom Typ `START` angewendet wird. In allen anderen Fällen von Regelanwendungen ist das aktuelle `StackElement` vom abstrakten Typ `ApplicationOfGeneratedRule`, der für jeden von `DEFINED` und `START` verschiedenen `RuleType` *t* eine Subklasse mit dem Namen `ApplicationOf`*t*`Rule` besitzt. Sie sind aus Gründen der Übersichtlichkeit nicht in der Abbildung dargestellt. Die Blätter des internen Parse-Forests, die ein Lexem erkennen, werden durch Instanzen der Klasse `Leaf` dargestellt.

Jede `StackElement`-Instanz verfügt über eine Referenz auf das Eltern-Element im Stack `parent` und seine Kinder `children`. Alle Kinder eines Elements müssen aufgehoben werden, da sie noch über für semantische Aktionen benötigte Informationen verfügen. Um das Eltern-Element zu erhalten kann die Methode

`getParent(): StackElement`

verwendet werden. Um Zugriff auf ein einzelnes Kind zu erhalten, wird die Methode

`getChild(index: int): StackElement`

benötigt. Welches der Kind-Elemente gewünscht wird, kann über `index` spezifiziert werden. Über die Methode

`gerCurrentSemanticActionPosition(): int`

kann die Anzahl der Kinder erfragt werden.

Da jedes `StackElement` bis auf `Leaf` die Anwendung einer Regel repräsentiert, gibt es eine Referenz auf die angewendete Regel `rule`, auf die über die Methode

`getAppliedRule(): Rule`

zugegriffen werden kann. Die einzelnen Felder und Methoden der Klasse `Rule` wurden im vorangegangenen Abschnitt vorgestellt.

In EDL ist es möglich auf die Positionsangaben des erkannten Lexems eines jeden Terms zuzugreifen. Daher hat jedes `StackElement` eine Referenz (`position`) auf eine Instanz der Klasse `Position`, welche die Felder `offset`, `length`, `firstLine`, `lastLine`, `firstColumn` und `lastColumn` sowie entsprechende Getter und Setter besitzt. Die Methoden sind aus Platzgründen nicht in der Abbildung dargestellt. Um den Zugriff auf diese Informationen zu vereinfachen, besitzt jedes `StackElement` die gleichen Getter wie `Position`. Alle sechs Positionsangaben können über die Methode

`setPositionOfLexem(offset: int, length: int,`

```
        firstLine: int, lastLine: int,
        firstColumn: int, lastColumn: int): void
```

gleichzeitig gesetzt werden. Um von einem `StackElement` auf den Namen der zur Zeit geparsten Datei zugreifen zu können, wird die Methode

```
getNameOfParsedFile(): String
```

benötigt. Diese greift über die Referenz `stack` auf die gleichnamige Methode des jeweiligen `Stack`-Objekts zu. Über die Methode

```
getLexem(): String
```

kann das bisher erkannte Lexem des aktuellen `StackElement`s abgerufen werden.

Jede angewendete Regel produziert ein Ergebnis, das im Feld `result` gespeichert wird, auf das über die Methode

```
getResult(): Object
```

zugegriffen werden kann. In EDL kann der Wert einer Regelanwendung durch Zuweisung an `$` oder durch die `lift`-Methode gesetzt werden. Beides wird durch einen Aufruf der Methode

```
setResult(newResult: Object): void
```

realisiert. Sollte das alte und das neue Ergebnis vom Typ `Vertex` sein, so werden beiden Knoten verschmolzen, was durch die private Methode

```
mergeResults(oldResult: Vertex, newResult: Vertex): void
```

in der Klasse `ApplicationOfDefinedRule` realisiert ist. Für maximale Terme können in EDL Default-Werte generiert werden, sofern kein Ergebnis explizit angegeben wurde. Da auch `null` vom Nutzer als Ergebnis festgelegt werden kann, wird das Feld mit Namen `isResultAlreadySet` benötigt, welches beim ersten Aufruf von `setResult` auf `true` gesetzt wird[2].

Realisierung der Sichtbarkeit von Variablen

Die in EDL definierten *Sichtbarkeitsebenen* entsprechen je einem zusammengesetzten Term. Da im Rahmen der BNF-Transformation für zusammengesetzte Terme eigene Regeln generiert werden, kann jede als eigene Sichtbarkeitsebene angesehen werden.

Im Rahmen des Desugaring-Prozesses bei der Vorverarbeitung (siehe Abschnitt 9.3.1) werden für kontextfreie Regeln die Indizes `i` in `$i` so angepasst, dass die automatisch ergänzten `cf(LAYOUT?)` ebenfalls durch `$i` referenziert werden können. Dadurch wird erreicht, dass die in EDL definierte Methode `getPrefixWhitespace()` sowie ihr Gegenstück `getSuffixWhitespace()` nicht durch eigene Java-Methoden realisiert wer-

[2] `setResult()` wird nur von in EDL definierten semantischen Aktionen aufgerufen. Beim Erzeugen von Default-Werten bleibt `isResultAlreadySet` bei `false`.

den müssen. Die Anpassung der Indizes `$i` hat den weiteren Vorteil, das `$i` durch den Aufruf `getChild(i)` realisiert werden kann. `getWhitespaceBefore()` ist durch die Methode

```
getWhitespaceBefore(indexOfTerm: int): List<Object>
```

realisiert, da so auch auf die Whitespace vor dem ersten maximalen Term zugegriffen werden kann.

`$` und temporäre Variablen sind innerhalb der gesamten in der Grammatik definierten Regel sichtbar. Das Feld `result` in der Klasse `ApplicationOfDefinedRule` entspricht `$`. Für temporäre Variablen ist diese Klasse um die Map `temporaryVariables` erweitert worden, die jedem Variablennamen einen Wert zuordnet. Auf diesen kann durch die Methode

```
getValueOfTemporaryVariable(name: String): Object
```

zugegriffen werden. Um einen neuen Wert zu setzten, wird die Methode

```
setValueOfTemporaryVariable(name: String, value: Object): void
```

verwendet.

Um innerhalb eines maximalen Terms schnellen Zugriff auf diese Variablen zu gewährleisten, hat jedes `StackElement` eine Referenz auf das Element, das für die Anwendung einer in der Grammatik definierten Regel steht (`ruleApply`). Über die Methode

```
getParentApplicationOfDefinedRule(): ApplicationOfDefinedRule
```

kann auf das referenzierte Objekt zugegriffen werden.

Aktualisierung des Stacks

Wie in Abschnitt 8.4 näher beschrieben, traversiert der `TreeTraverser` den internen Parse-Forest nach der „depth first“-Strategie. Wird dabei ein Knoten betreten, verlassen oder zu einem Elternknoten zurückgekehrt, ruft der `TreeTraverser` eine der folgenden Methoden des `Stack`s auf, um diesen zu aktualisieren:

`enterNode(rule: Rule): void`

Diese Methode wird aufgerufen, wenn der `TreeTraverser` einen neuen Knoten des internen Parse-Forests betritt. Sollte dieser Knoten für eine Regelanwendung stehen, so wird die entsprechende Regel als aktueller Parameter übergeben. Ansonsten hat `rule` den Wert `null`.

Vorgehen:

1. Erzeuge eine neue `StackElement`-Instanz e, deren genauer Typ sich nach `rule` richtet und lege sie auf den Stack.
2. Sollte e das Bottom-Element sein, so initialisiere die Position mit den Angaben in `initialPosition`. Ansonsten hat e das Elternelement p und die Positionsangabe wird wie folgt berechnet:

   e`.offset = `p`.offset + `p`.length`

   e`.length = 0`

   e`.firstLine = `p`.lastLine`

   e`.lastLine = `p`.lastLine`

   e`.firstColumn = `p`.lastColumn`

   e`.lastColumn = `p`.lastColumn`

`leaveNode(): void`

Diese Methode wird aufgerufen, wenn der `TreeTraverser` einen Knoten des internen Parse-Forests verlässt.

Vorgehen:

Falls e ein Blatt des internen Parse-Forests ist, so wird höchstens ein Eingabezeichen erkannt. Ist das erkannte Eingabezeichen `'\n'`, dann werden folgende Positionsangaben angepasst:

e`.length = 1`

e`.lastLine = `e`.firstLine +1`

e`.lastColumn = 0`

Wurde ein anderes Zeichen erkannt, werden `lastLine` und `lastColumn` wie folgt verändert:

e`.length = 1`

e`.lastLine = `e`.firstLine`

e`.lastColumn = `e`.firstColumn +1`

Ansonsten handelt es sich bei dem Blatt um die Anwendung einer ε-Regel. Dabei werden folgende Positionsangaben verändert:

e`.lastLine = `e`.firstLine`

e`.lastColumn = `e`.firstColumn`

`returnToNode(): void`

Diese Methode wird ausgeführt, wenn der `TreeTraverser` einen Knoten verlassen hat und zu dem Elternknoten zurückkehrt.

Vorgehen:

1. Wurde das Ergebnis des aktuellen `StackElements` e noch nicht gesetzt, so wird der Default-Wert berechnet (siehe den folgenden Abschnitt).
2. Die Kinder von e werden gelöscht, da sie nun keine relevanten Informationen mehr besitzen.
3. Die Positionsangabe des Elternelements p, zu dem zurückgekehrt wird, werden aktualisiert:
   p.length = p.length + e.length
   p.lastLine = e.lastLine
   p.lastColumn = e.lastColumn
4. Der Zeiger auf das aktuelle Element `current` wird von e auf sein Elternelement p gesetzt.

Erzeugung der Default-Werte

Beim Verlassen eines Knotens des internen Parse-Forests wird die `leaveNode()`-Methode des `Stack`s aufgerufen. Sollte durch eine semantischen Aktion noch kein Ergebnis gesetzt worden sein, so wird ein Default-Wert als `result` berechnet. Wie diese Berechnung aussieht, ist abhängig vom Typ der angewendeten Regel und wird durch eine Implementation der abstrakten Methode

```
createDefaultValue(): void
```

in der jeweiligen Subklasse von `StackElement` realisiert.

Die berechneten Default-Werte sind die folgenden:

DEFINED. Regeln dieses Typs wurden in der Grammatik definiert. Ihr Default-Wert ist immer `null`.

FILE_START. Regeln dieses Typs dienen der Erkennung von ganzen Eingabedateien. Falls in ihrem Rumpf genau eine von WHITESPACE verschiedene Regel r Anwendung findet, so wird der durch r bestimmte Wert als `result` gesetzt.

START. Dieser Typ identifiziert Regeln, die den Start der Herleitung der Eingabe repräsentieren. Ihr Default-Wert berechnet sich genauso wie bei FILE_START.

WHITESPACE. Regeln vom Typ WHITESPACE dienen der Herleitung der in kontextfreien Regeln automatisch ergänzten `cf(LAYOUT?)`. Für Regeln dieses Typs wird eine Liste als `result` erzeugt. Der Inhalt dieser Liste ist bei:

1. ε `-> cf(LAYOUT?)`
   eine leere Liste.
2. `cf(LAYOUT) -> cf(LAYOUT?)`
   - eine leere Liste, falls das Ergebnis des Kinds `cf(LAYOUT) null` ist,
   - das Ergebnis der dritten Regel, falls diese zur Herleitung von `cf(LAYOUT)` genutzt wurde,
   - und ansonsten eine Liste, die das Ergebnis des Kinds `cf(LAYOUT)` enthält.
3. `cf(LAYOUT) cf(LAYOUT) -> cf(LAYOUT) {left}`
   eine Liste, die die Ergebnisse beider Kinder enthält, sofern diese von `null` verschieden sind. Sollte das erste Kind eine erneute Anwendung dieser Regel sein, so wird dessen Ergebnisliste um das von `null` verschiedene Ergebnis des zweiten Kinds erweitert.

LEX2CF. Regeln dieses Typs haben immer genau ein Kind, dessen Ergebnis als `result` gespeichert wird.

EPSILON. Dieser Typ beschreibt ε-Regeln und hat als Default-Wert einen leeren String.

LITERAL. Für mit " oder ' umschlossene Literale L wird das erkannte Lexem als `result` gesetzt.

OPTION. Regeln vom Typ OPTION haben nur genau ein Kind, dessen Ergebnis als Default-Wert gesetzt wird.

SEQUENCE. Regeln dieses Typs beschreiben die Semantik der Sequence $(T_1 \ldots T_n)$. Sollte sie nur aus einem Element bestehen, so wird sein Ergebnis als `result` gespeichert. Ansonsten ist der Wert `null`.

ALTERNATIVE. Regeln vom Typ ALTERNATIVE haben nur ein Kind, dessen Ergebnis der Default-Wert der Regel darstellt.

EPSILON2STAR. Dies ist der Typ von Regeln, die zur Erkennung des leeren Worts aus $T\star$ und $\{T_1 \;\; T_2\}\star$ dienen. Als Default-Wert erzeugen sie eine leere Liste.

PLUS2STAR. PLUS2STAR ist der Typ von Regeln, die T+ und $\{T_1 \;\; T_2\}+$ in $T\star$ und $\{T_1 \;\; T_2\}\star$ überführen. Das Ergebnis des Kinds wird als `result` übernommen.

REPETITION. Dieser Typ identifiziert Regeln zur Realisierung der Semantik von $T\star$ und $T+$, die zur Herleitung von mindestens zwei T dienen. Ihr Default-Wert ist die Ergebnisliste des ersten Kinds, an die die Ergebnisliste des letzten Kinds gehängt wird.

REPETITION_PROD. REPETITION_PROD ist der Typ von Regeln, die im Rahmen der BNF-Transformation von T`*` und T+ erzeugt werden und zur Erkennung eines T führen. Als Ergebnis erzeugen sie eine Liste, die das Ergebnis des einzigen Kinds enthält.

LIST. Dieser Typ identifiziert Regeln zur Realisierung der Semantik von `{`T_1 T_2`}*` und `{`T_1 T_2`}+`, die zur Herleitung von mindestens zwei T_1 und der Erkennung eines T_2 dienen. Ihr Default-Wert ist die Ergebnisliste des ersten Kinds, an die das Ergebnis des mittleren Kindes (T_2) und die Ergebnisliste des letzten Kinds gehängt wird. Sollte das Ergebnis dieses `StackElement`s oder einer seiner Kinder durch eine semantische Aktion bereits gesetzt worden sein, enthält die Ergebnisliste nur die gesetzten Werte.

LIST_PROD. LIST_PROD ist der Typ von Regeln, die im Rahmen der BNF-Transformation von `{`T_1 T_2`}*` und `{`T_1 T_2`}+` erzeugt werden und zur Erkennung eines T_1 führen. Als Ergebnis erzeugen sie eine Liste, die das Ergebnis des einzigen Kinds enthält.

TUPLE. Regeln dieses Typs entstehen bei der Transformation von <T_1, ... , T_n>. Ihr Default-Wert ist `null`.

FUNCTION. Regeln vom Typ FUNCTION werden bei der BNF-Transformation aus dem Term `(`$T_1 \ldots T_n$ `=>` T`)` erzeugt. Ihr Default-Wert ist `null`.

Bewahrung der Werte in Lists

In EDL gibt es die bereits in SDF vorhandenen *List-Konstrukte, die um semantische Aktionen erweitert werden können*, wie beispielsweise in:

`{#`S_1`#` T_1 `#`S_2`#` T_2 `#`S_3`#}*`

Dieses Beispiel beschreibt eine Folge von T_1-Elementen, die durch den Separator T_2 voneinander getrennt sind. Vor jedem erkannten T_1 wird die semantische Aktion S_1 ausgeführt und dahinter S_2. Nur nach der Erkennung eines T_2 wird die Aktion S_3 ausgeführt.

Zur *Erkennung eines einzelnen T_1-Elements* wird der in Abbildung 8.5 gezeigte interne Parse-Forest benötigt. Zum besseren Verständnis wird direkt die angewendete Regel angegeben an Stelle ihrer eindeutig identifizierenden Nummer. Damit der Forest nicht durch automatisch ergänzte `LAYOUT?` zu groß wird, wird angenommen, dass es sich bei der oben angegebenen List um eine lexikalische Definition handelt. Wird dieser Parse-Forest durch den `TreeTraverser` traversiert, so wird beim Betreten des Knotens $v2$ die semantische Aktion S_1 ausgeführt und sobald er wieder verlassen wird S_2. Da kein T_2 erkannt wurde, wird S_3 nicht ausgeführt.

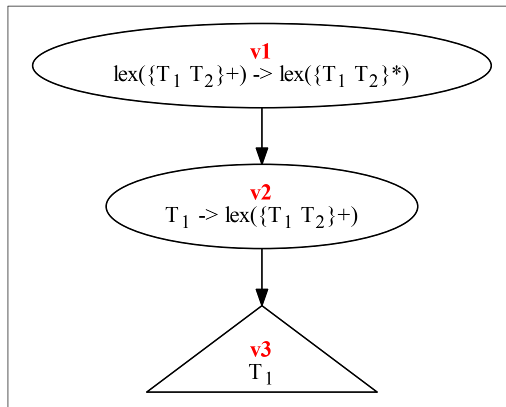


Abbildung 8.5: Der interne Parse-Forest zur Erkennung eines T_1-Elements.

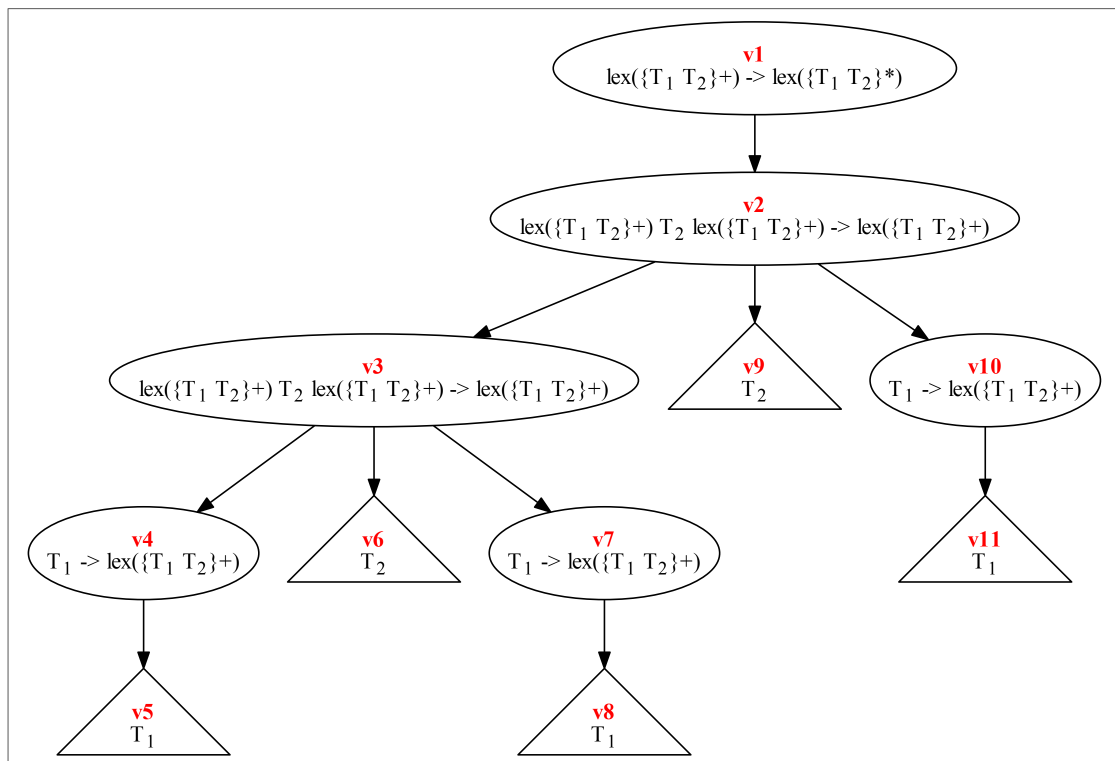


Abbildung 8.6: Der interne Parse-Forest zur Erkennung von $T_1T_2T_1T_2T_1$.

Abbildung 8.6 zeigt den internen Parse-Forest zur *Erkennung der Sequenz* $T_1T_2T_1T_2T_1$. Die Reihenfolge der ausgeführten semantischen Aktionen ist in diesem Fall:

1. Beim Betreten von $v4$ wird S_1 ausgeführt.
2. Beim Verlassen von $v4$ wird S_2 ausgeführt.
3. Bei der Rückkehr zu $v3$ nach Bearbeitung von $v6$ (T_2) wird S_3 ausgeführt.
4. Beim Betreten von $v7$ wird S_1 ausgeführt.
5. Beim Verlassen von $v7$ wird S_2 ausgeführt.
6. Bei der Rückkehr zu $v2$ nach Bearbeitung von $v9$ (T_2) wird S_3 ausgeführt.
7. Beim Betreten von $v10$ wird S_1 ausgeführt.
8. Beim Verlassen von $v10$ wird S_2 ausgeführt.

Wie anhand dieses Beispiels zu erkennen ist, wird die semantische Aktion S_3 nur für jedes erkannte T_2 ausgeführt. Würde innerhalb von S_3 das Element T_1 einen neuen Wert zugewiesen bekommen, so würden sich nur die für $v4$ und $v7$ berechneten Ergebnisse verändern und $v10$ behielte den ursprünglichen Wert. Aus diesem Grund ist eine Zuweisung an T_1 innerhalb von S_3 in EDL verboten.

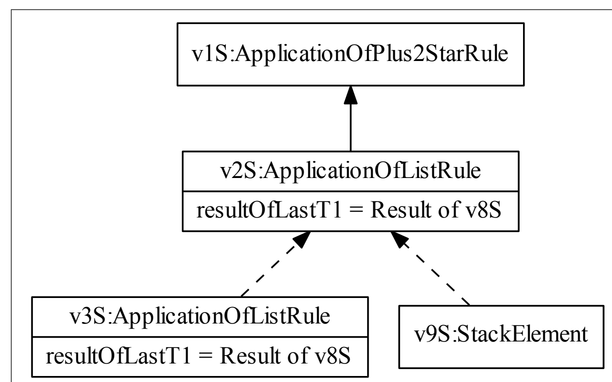


Abbildung 8.7: Der `Stack` in Schritt 6.

Ein lesender Zugriff auf T_1 ist allerdings in S_3 gestattet. Nach der Rückkehr zu Knoten v_9 (Schritt 6) wäre der Wert vom T_1-Element der des Knotens $v8$. Der verwendete `Stack` sieht allerdings wie in Abbildung 8.7 gezeigt aus und der Repräsentant $v8S$ vom Knoten $v8$ ist bereits entfernt worden. Aus diesem Grund gibt es in `StackElements` vom Typ `ApplicationOfListRule` ein Feld mit Namen `resultOfLastT1` in dem das Ergebnis des zuletzt erkannten T_1 gespeichert wird.

8.4 TreeTraverser

Der Interpreter von `Stratego/XT` wird genutzt, um die Eingabe mithilfe einer Parse-Tabelle zu parsen und einen *internen Parse-Forest* aufzubauen. Letzterer wird mithilfe eines `ITreeBuilders` verarbeitet (siehe Abschnitt 5.2.5). Im Rahmen dieser Masterarbeit wird der `TreeTraverser` als Implementation dieses Interfaces genutzt, um den internen Parse-Forest zu traversieren und den `GraphBuilder` zu veranlassen, die gewünschten semantischen Aktionen auszuführen.

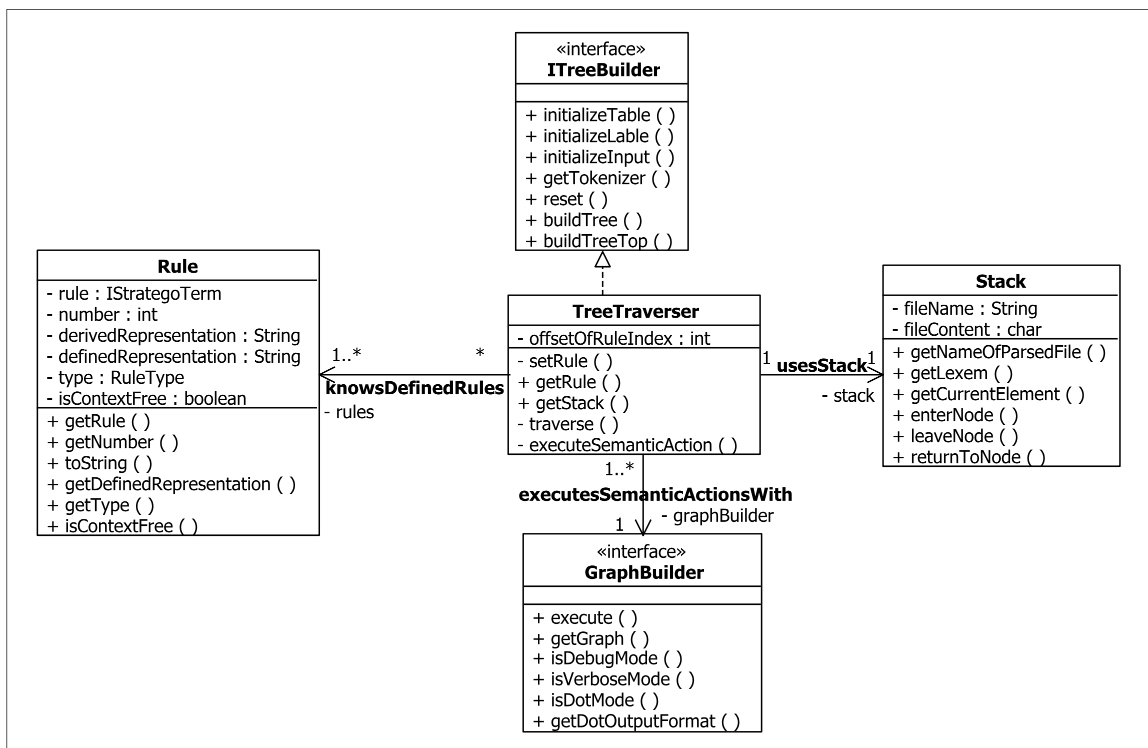


Abbildung 8.8: Die Realisierung der `TreeTraverser`-Komponente.

Abbildung 8.8 zeigt den `TreeTraverser`, dessen Instanz beim Interpreter vor dem Parse-Vorgang registriert werden muss. Beim Aufruf des entsprechenden Konstruktors muss eine Referenz auf den aktuellen `GraphBuilder` mitgegeben werden.

Implementation des `ITreeBuilder`-Interfaces

Die Methoden des `ITreeBuilder`-Interfaces sind wie folgt implementiert:

initializeTable(table: ParseTable, productionCount: int, labelStart: int, labelEnd: int): void
Der Parameter `labelStart` ist die niedrigste eindeutig identifizierende Nummer

einer Regel und wird im Feld `offsetOfRuleIndex` gespeichert. Alle Nummern bis einschließlich `labelEnd` identifizieren weitere Regeln. Daher kann mit diesen beiden Information ein neues `Rule`-Array `rules` erzeugt werden, da seine Länge durch

```
labelEnd - labelStart + 1
```

berechnet werden kann. Damit für den Zugriff auf dieses Array die eindeutig identifizierende Nummer einer Regel verwendet werden kann, ohne den Wert des Felds `offsetOfRuleIndex` berücksichtigen zu müssen, gibt es die folgenden Methoden:

```
getRule(index: int): Rule
setRule(index: int, rule: Rule): void
```

initializeLabel(labelNumber: int, parseTreeProduction: IStrategoAppl): void

Für jede vorhandene Regel wird diese Methode nacheinander aufgerufen. Dabei wird ein neues `Rule`-Objekt erzeugt, das die Regel `parseTreeProduction` wrapt. Dieses Objekt wird in das Array `rules` eingefügt.

initializeInput(input: String, filename: String): void

Beim Aufruf dieser Methode wird ein neuer `Stack` angelegt und im Feld `stack` gespeichert. Der entsprechende Konstruktor bekommt den Namen der geparsten Datei `filename` als String und den Inhalt der Datei `input` als char-Array übergeben. Auf `stack` kann durch den entsprechenden Getter zugegriffen werden.

getTokenizer(): ITokenizer

Da der Default-`ITokenizer` des Interpreters verwendet werden soll, liefert diese Methode `null` zurück.

reset(): void

Diese Methode wird mit einem leeren Rumpf realisiert, da sie nicht benötigt wird.

buildTree(node: AbstractParseNode): Object

Der formale Parameter `node` referenziert die Wurzel des internen Parse-Forests, der durch einen Aufruf der Methode `traverse()` traversiert wird (siehe den folgenden Abschnitt) und dabei die entsprechenden semantischen Aktionen ausführt. Im Anschluss wird der `node` zurückgegeben.

buildTreeTop(subtree: Object, ambiguityCount: int): Object

Beim Aufruf dieser Methode ist `subtree` die Wurzel des internen Parse-Forest, wie er von der Methode `buildTree()` zurückgegeben wurde. Sollte der Debug-Modus gewählt sein, was über die Methode `isDebugMode()` des `GraphBuilder`s bestimmt werden kann, so wird `subtree` ausgegeben (siehe Abschnitt 8.5). Schließlich wird der im `GraphBuilder` erzeugte Graph zurückgegeben.

Traversierung des internen Parse-Forests

Sobald der Parse-Vorgang des Interpreters abgeschlossen ist, *erhält der* `TreeTraverser` *den internen Parse-Forest*. Bei der Traversierung nach der „depth first"-Strategie durch die `traverse()`-Methode wird der `Stack` aktualisiert und der `GraphBuilder` veranlasst, die entsprechenden semantischen Aktionen auszuführen. Sollte ein Knoten gefunden werden, der für eine *Mehrdeutigkeit* steht, wird die Traversierung durch das Werfen einer *Exception abgebrochen*. Dabei wird der interne Parse-Forest ausgegeben, damit der Nutzer in der Lage ist, die Mehrdeutigkeit zu verstehen und in der Grammatik zu beheben.

```
1  private void traverse(AbstractParseNode currentNode, AbstractParseNode root) {
2     // check if currentNode is an ambiguity node
3     ...
4     // enter a new node
5     stack.enterNode(getRule(currentNode.getLabel()));
6     for (AbstractParseNode child : currentNode.getChildren()) {
7        executeSemanticAction(root, currentNode);
8        traverse(child, root);
9        // return to parent node
10       stack.returnToNode();
11    }
12    // leave node
13    executeSemanticAction(proot, currentNode);
14    stack.leaveNode();
15 }
```

Listing 8.3: Die Methode traverse().

Listing 8.3 zeigt die `traverse()`-Methode. Als Parameter erhält sie den aktuellen Knoten des internen Parse-Forests (`currentNode`) sowie seine Wurzel (`root`).

In Zeile 5 wird der *aktuelle Knoten* durch Aufruf von `enterNode()` auf den `Stack` gelegt. Als aktuellen Parameter erhält diese Methode das `Rule`-Objekt, das die angewendete Regel wrapped. Sollte `currentNode` ein Blatt sein, das für die Erkennung eines Eingabezeichens steht, so liefert `getRule()` `null` zurück.

Mithilfe der Schleife in den Zeilen 6 bis 11 wird *über alle Kinder* iteriert. Dabei gibt es für jeden maximalen Term im Rumpf der angewendeten Regel ein Kind. In Zeile 7 wird durch den Aufruf der Methode

```
executeSemanticAction(root: AbstractParseNode,
                      currentNode: AbstractParseNode): void
```

der `GraphBuilder` veranlasst, die semantischen Aktionen vor dem aktuellen maximalen Term auszuführen. Sollte dabei eine Exception auftreten und der Debug-Modus akti-

viert sein, so wird der *durch `root` referenzierte interne Parse-Forest* ausgegeben und dabei der aktuelle Knoten `currentNode` als Verursacher der Exception markiert.

Durch den rekursiven Aufruf der `traverse`-Methode in Zeile 8 wird der Teilbaum mit dem aktuellen Kind als Wurzel traversiert. Sobald dieser Teilbaum vollständig behandelt wurde, wird in Zeile 10 die `returnToNode()`-Methode des `Stack`s aufgerufen.

Nachdem alle Kinder behandelt sind, wird zunächst die abschließende semantische Aktion einer Regelanwendung ausgeführt, bevor die `leaveNode()`-Methode aufgerufen wird (Zeile 14). Diese Reihenfolge ist notwendig, da ansonsten durch `leaveNode()` ein Default-Wert erzeugt würde, bevor die letzte semantische Aktion ausgeführt wäre.

8.5 Debug-Ausgaben

Um den Nutzer beim *Debuggen* zu unterstützen, wird im Falle einer Mehrdeutigkeit der *interne Parse-Forest ausgegeben*. Falls der Debug-Modus aktiviert wurde, wird der Forest nach seiner Traversierung ausgegeben. Sollte bei der Ausführung der semantischen Aktionen eine Exception auftreten, so wird der Forest ausgegeben und die Regelanwendung markiert, die die Exception verursacht hat.

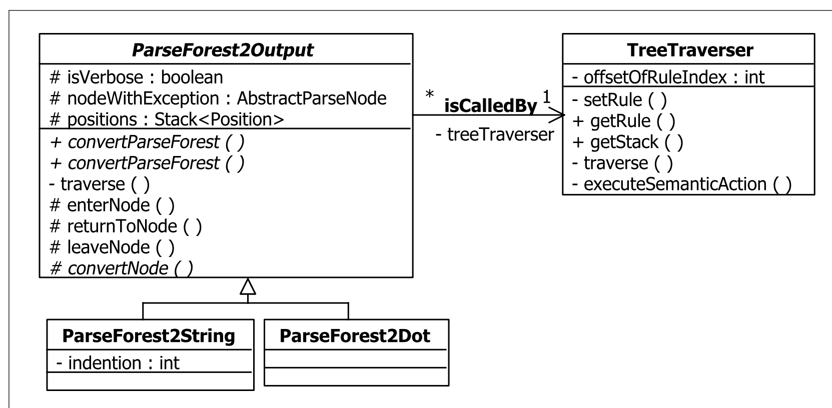


Abbildung 8.9: Die Realisierung des `Debugging`s.

Abbildung 8.9 zeigt die abstrakte Klasse *`ParseForest2Output`*, die die *Superklasse aller Ausgabeformate* ist. Der Konstruktor benötigt den aktuellen `TreeTraverser`, um an die im internen Parse-Forest durch Nummern identifizierte Regeln zu gelangen und um die erkannten Unicode-Zeichen richtig darstellen zu können. Über die beiden Methoden

```
convertParseForest(root: AbstractParseNode,
                   isVerbose: boolean): String
```

und

```
convertParseForest(root: AbstractParseNode,
                isVerbose: boolean,
                nodeWithException: AbstractParseNode): String
```

wird die Konvertierung des internen Parse-Forests `root` angestoßen. Durch `isVerbose`, was im gleichnamigen Feld gespeichert wird, kann bestimmt werden, ob die Anwendung jeder Regel oder nur die Anwendung der in der Grammatik definierten Regeln dargestellt werden sollen. Die zweite Methode wird aufgerufen, falls im Debug-Modus bei einer semantischen Aktion eine Exception aufgetreten ist. Der verantwortliche Knoten wird über den Parameter `nodeWithException` übergeben und im gleichnamigen Feld gespeichert. Als Ergebnis wird ein String produziert, der den auf der Konsole auszugebenden Text repräsentiert.

Die soeben vorgestellten Methoden rufen

```
traverse(appendable: Appendable, node: AbstractParseNode): void
```

auf, die den internen *Parse-Forest per „depth first"-Strategie* durchlaufen. Dabei werden die Methoden

```
enterNode(node: AbstractParseNode): void
returnToNode(node: AbstractParseNode): void
leaveNode(node: AbstractParseNode): void
```

aufgerufen, um den `java.util.Stack` `positions` zu füllen und die Positionsangaben der Knoten zu aktualisieren. Ihre Funktionsweise ist analog zu den im Abschnitt 8.3 beschriebenen gleichnamigen Methoden.

Die Repräsentation eines einzelnen Knotens wird schließlich durch die Methode

```
convertNode(appendable: Appendable,
                node: AbstractParseNode): void
```

erzeugt.

In der `EDL-Parser`-Komponente sind zwei Ausgabeformate vordefiniert: die *Ausgabe auf der Konsole* und die Erzeugung einer *grafischen Ausgabe mittels dot*. Beide Varianten werden in den folgenden Abschnitten vorgestellt.

Konsolenausgabe

Bei dieser Form der Ausgabe, die durch die Klasse *`ParseForest2String`* realisiert ist, erzeugen die `convertParseForest()`-Methoden eine String-Repräsentation des internen Parse-Forests, die in der aufrufenden Methode auf der Konsole ausgegeben werden kann.

```
cf(LAYOUT?) cf(Start) cf(LAYOUT?) -> <START> (line 1,...
    -> cf(LAYOUT?) (line 1, column 0, length 0)
    cf(Value) cf(LAYOUT?) cf(Currency?) -> cf(Start){definedAs...
        lex(Value) -> cf(Value) (line 1, column 0, length 2)
            [1-9] lex([0-9]*) -> lex(Value){definedAs("...")}...
                "5" (line 1, column 0, length 1)
                lex([0-9]+) -> lex([0-9]*) (line 1, column 1,...
                    [0-9] -> lex([0-9]+) (line 1, column 1,...
                        "0" (line 1, column 1, length 1)
        cf(LAYOUT) -> cf(LAYOUT?) (line 1, column 2, length 1)
            ...
        cf(Currency) -> cf(Currency?) (line 1, column 3,...
            lex(Currency) -> cf(Currency) (line 1, column 3,...
                [\$\253] -> lex(Currency){definedAs("...")}...
                    "€" (line 1, column 3, length 1)
    -> cf(LAYOUT?) (line 1, column 4, length 0)
```

Listing 8.4: Die String-Repräsentation des internen Parse-Forests aus Abbildung 8.2 im *Verbose-Modus*.

Listing 8.4 zeigt die String-Repräsentation des internen Parse-Forests aus Abbildung 8.2, der die Herleitung der Eingabe `"50 €"` darstellt. Da in diesem Listing der *Verbose-Modus* angenommen wird, werden alle Regelanwendungen aufgeführt. Aus Gründen der Übersichtlichkeit sind die Whitespace-Herleitung in Zeile 11 und aus Platzgründen manche Zeilen im Listing durch `...` abgekürzt worden.

Hinter jedem Knoten ist ein Tripel angegeben, in dem die Zeile des ersten erkannten Zeichens z, die Spalte vor z und die Anzahl der erkannten Zeichen angegeben wird. Alle Kinder eines Knotens werden eingerückt dargestellt. Die Länge der Einrückung wird über das Feld `indention` bestimmt, welches bei Aufruf der abgeleiteten Methoden `enterNode()` und `leaveNode()` aktualisiert wird. Sollte ein Knoten eine Mehrdeutigkeit repräsentieren oder eine Exception verursacht haben, so wird er mittels >>>> markiert.

```
Value Currency? -> Start (line 1, column 0, length 4)
    [1-9] [0-9]* -> Value (line 1, column 0, length 2)
        "5" (line 1, column 0, length 1)
        "0" (line 1, column 1, length 1)
    [\$\253] -> Currency (line 1, column 3, length 1)
        "€" (line 1, column 3, length 1)
```

Listing 8.5: Die String-Repräsentation des internen Parse-Forests aus Abbildung 8.2 im *Default-Modus*.

Listing 8.5 zeigt den gleichen internen Parse-Forest, wobei dieses Mal nur die in der Grammatik definierten Regeln ausgegeben werden. Sollten `LAYOUT`-Regeln in der Grammatik definiert sein, so werden diese ebenfalls nicht ausgegeben.

Grafische Ausgabe

Sollte der Dot-Modus gewählt sein, so muss die Klasse *`ParseForest2Dot`* genutzt werden. Um eine Instanz dieses Typs zu erzeugen, benötigt der Konstruktor zusätzlich zum Inhalt der geparsten Eingabe *das gewünschte Ausgabeformat*.

In dieser Klasse wandeln die beiden `convertParseForest()`-Methoden den internen Parse-Forest in eine .dot-Datei um, aus der im Anschluss mittels dot eine Grafik erzeugt wird. Es wird zunächst versucht, beide Dateien im aktuellen Ordner zu erzeugen. Sollte dies fehlschlagen, so werden sie im temporären Verzeichnis des Betriebssystems gespeichert. Der Pfad zur erzeugten Grafik wird als Ergebnis des Methodenaufrufs zurückgegeben.

Abbildung 8.10 zeigt den internen Parse-Forest aus Abbildung 8.2, der die Herleitung der Eingabe `"50 €"` darstellt. Da er im *Verbose-Modus* erstellt wurde, sind alle Regelanwendungen dargestellt. Die gestrichelte Kante zu `...` bedeutet, dass die Herleitung des Whitespace-Zeichens in der Abbildung abgekürzt ist, da sie sonst zu groß wäre.

Für jede angewendete Regel ist die Zeile des ersten erkannten Eingabezeichens z, die Spalte vor z und die Anzahl der durch die Regelanwendung erkannten Zeichen angegeben. Sollte ein Knoten eine Mehrdeutigkeit repräsentierten oder eine Exception bei der Ausführung der entsprechenden semantischen Aktionen verursachen, so wird er rot dargestellt. Blätter, die für ein erkanntes Lexem stehen, werden gelb dargestellt. Anwendungen von kontextfreien Regeln werden hellblau und von lexikalischen Regeln werden hellgrün dargestellt.

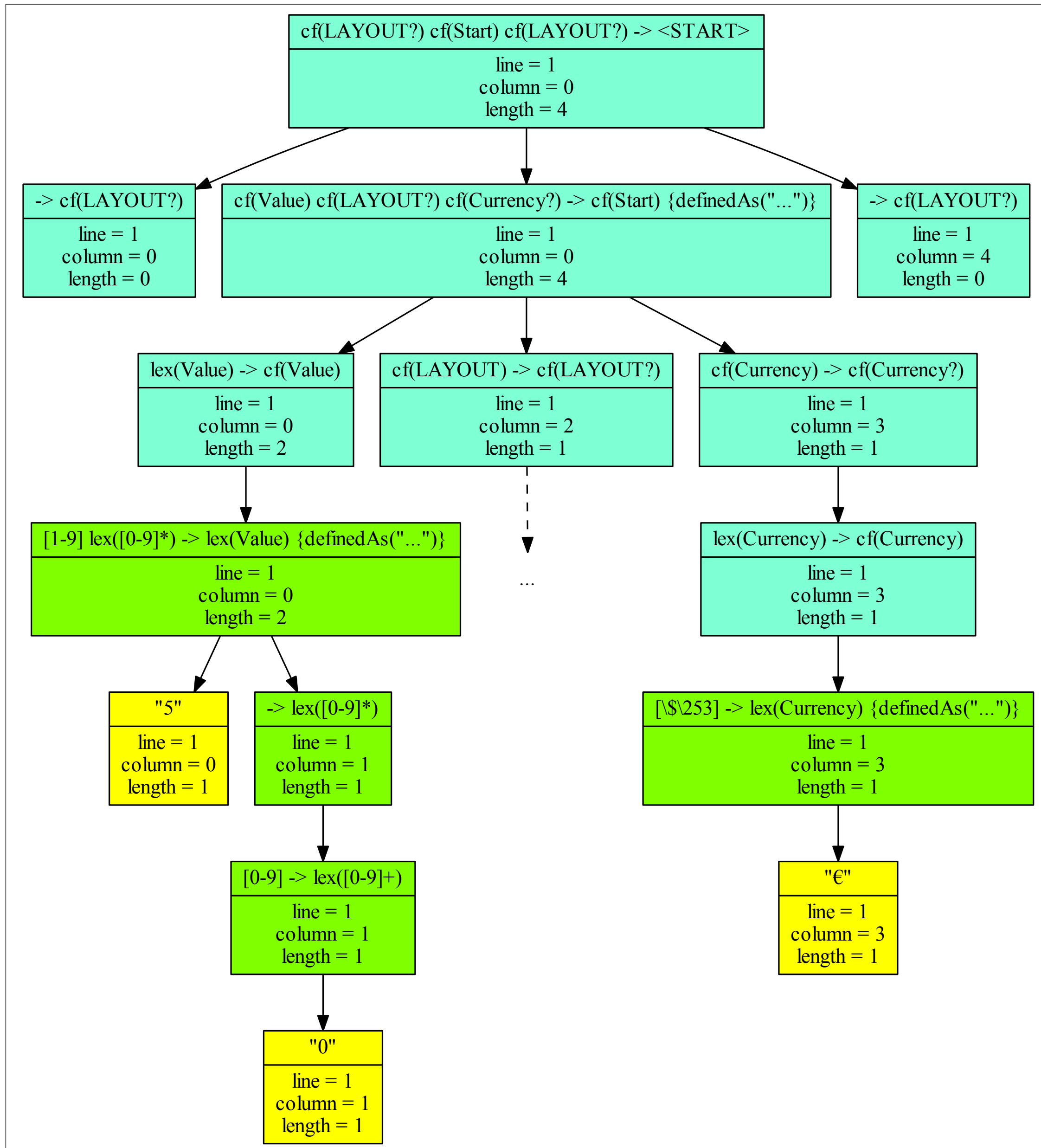


Abbildung 8.10: Die grafische Repräsentation des internen Parse-Forests aus Abbildung 8.2 im *Verbose-Modus*.

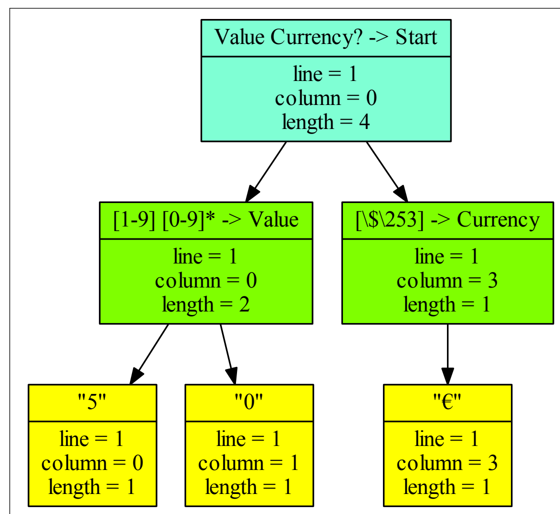


Abbildung 8.11: Die grafische Repräsentation des internen Parse-Forests aus Abbildung 8.2 im *Default-Modus*.

Abbildung 8.11 zeigt die grafische Repräsentation des internen Parse-Forests im Default-Modus. Hier sind nur die Anwendungen der in der Grammatik definierten Regeln gezeigt. Sollten `LAYOUT`-Regeln in der Grammatik definiert sein, so werden diese ebenfalls nicht ausgegeben.

8.6 Realisierung des Symboltabellen-Stacks

In EDL ist es möglich *Symboltabellen* zu verwenden, die aus einem *Stack von Maps* bestehen (siehe Abschnitt 6.2.4).

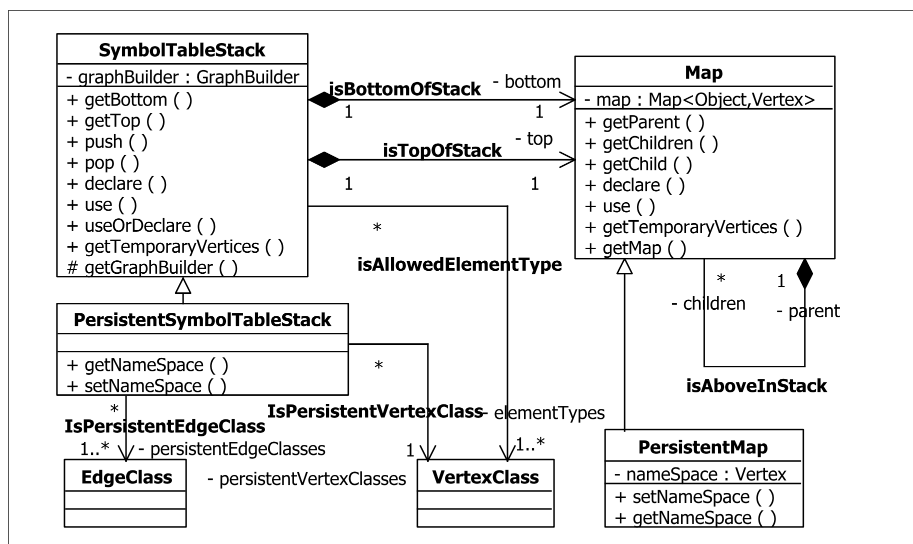


Abbildung 8.12: Der Symboltabellen-Stack.

Abbildung 8.12 zeigt die Realisierung der Symboltabellen durch `SymbolTableStack`. Wird ihr Konstruktor aufgerufen, wird automatisch eine neue `Map` als Bottom-Element erzeugt und durch `bottom` referenziert. Dies hat den Vorteil, dass vordefinierte Bezeichner direkt festgelegt werden können, ohne zunächst weitere Methoden aufrufen zu müssen.

Mithilfe der Methode

push(): void

wird eine neue `Map` erzeugt und auf den Symboltabellen-Stack gelegt. Durch einen Aufruf von

pop(): void

kann die oberste `Map` wieder entfernt werden. Da die gesamte Symboltabelle am Ende des Parse-Vorgangs zugreifbar sein soll, *darf keine* `Map` *endgültig gelöscht werden*. Um dies zu realisieren, bilden alle `Map`s eine *Baumstruktur*, in der das Eltern-Element durch `parent` und die Kinder über `children` referenziert werden. Innerhalb dieses Baums kann das aktuell oberste Stack-Element über die der Klasse `SymbolTableStack` bekannte Referenz *top* identifiziert werden. Durch Rückverfolgen der `parent`-Referenzen kann jedes Element im Stack bis `bottom` identifiziert werden. Das Entfernen eines Elements vom Stack mittels `pop()` kann durch das Versetzen der `top`-Referenz erreicht werden. Durch die Methoden

```
getBottom(): Map
getTop(): Map
getParent(): Map
getChildren(): List<Map>
getChild(index: int): Map
```

können die entsprechenden `Map`s zugegriffen werden.

In der Symboltabellen-Definition in EDL müssen die Knotenklassen der einzufügenden Elemente definiert werden. Diese Knotenklassen werden in `SymbolTableStack` durch `elementTypes` referenziert. Im Falle von persistenten Symboltabellen wird eine Instanz der Klasse *PersistentSymbolTableStack* erzeugt, die ein Subtyp von `SymbolTableStack` ist. Ihre Elemente sind analog dazu *PersistentMaps*.

Bei persistenten Symboltabellen muss für jede erzeugte `PersistentMap` ein Knotentyp definiert sein, durch dessen Instanzen sie im Graphen persistiert werden kann. Dieser Typ ist durch *persistentVertexClass* referenziert. Sollte in einer in EDL erstellten Grammatik dem `namespace`-Attribut ein Wert zugewiesen werden, wird durch Aufruf der Methode

`setNameSpace(v: Vertex): void`

das gleichnamige Attribut der `PersistentMap` gesetzt. Dadurch wird dieser `Map` ein

persistenter Knoten im Graphen zugeordnet. Darüber hinaus muss für jeden Elementtyp der persistenten Symboltabelle eine Kantenklasse definiert sein, durch die die eingefügten Knoten mit dem `nameSpace`-Knoten einer `PersistentMap` verbunden werden. Diese Kantentypen werden durch die Referenz *`persistentEdgeClasses`* gespeichert. Dabei gilt, dass Elemente des *i*-ten Knotentyps aus `elementTypes` durch Instanzen des *i*-ten Kantentyps aus `persistentEdgeClasses` persistiert werden.

Die in EDL zulässigen *Symboltabellen-Operationen* werden durch die Methoden

```
declare(key:Object, value: Vertex): Vertex
use(key:Object): Vertex
useOrDeclare(key:Object, value: Vertex): Vertex
getTemporaryVertices(): List<Vertex>
```

mit der in Abschnitt 6.2.4 beschriebenen Semantik realisiert. Da der Nutzer die Möglichkeit hat, nach dem Parse-Vorgang auf die Symboltabellen zuzugreifen, kann er über die Methode

```
getMap(): Map<Object,Vertex>
```

einer `Map` die enthalte `java.util.Map` bekommen.

Zur Erzeugung von temporären Knoten sowie den persistierenden Kanten wird eine Referenz auf den aktuellen Graphen benötigt. Der sich im durch das Feld `graphBuilder` referenzierten `GraphBuilder` befindet und über den entsprechenden Getter zugegriffen werden kann.

8.7 Inselgrammatiken

Zur Realisierung der *Inselgrammatiken*, wie sie in Abschnitt 6.3 definiert sind, wird die Methode

```
determineIslands(input: String, inclusiveStartPattern: String,
            exclusiveStartPattern: String,
            inclusiveEndPattern: String,
            exclusiveEndPattern: String): List<Position>
```

des `GraphBuilder`s aufgerufen. Diese verodert zunächst alle vier Eingabepattern und wendet diesen neuen regulären Ausdruck auf die zu parsende Eingabe `input` an. Jeder gematchte Teil-String wird zunächst daraufhin überprüft, ob er ein Anfang oder Ende einer Insel repräsentiert und ob er inklusiv oder exklusiv ist. Mithilfe dieser Ergebnisse werden *`Position`-Objekte* erzeugt, die die zu parsenden Inseln definieren.

Sollte als inklusives Start-Pattern `s` und als exklusives Ende-Pattern `e` definiert sein, dann werden aus der Eingabe

`ccscscseccccesccc`

die zu parsenden Inseln `scscs` und `sccc` extrahiert. Im Allgemeinen gilt, dass ein Inselstart durch den ersten Match eines Start-Patterns am Eingabe-Anfang oder hinter dem letzten Inselende begonnen wird. Die dadurch begonnene Insel endet entweder mit dem Ende der Eingabe oder mit dem ersten folgenden Match eines Ende-Patterns. Im Falle eines inklusiven Patterns zählt der jeweilige Insel-Anfang oder -Ende mit zu der Insel und ansonsten nicht.

Allgemein gilt beim Matchen, dass ein Eingabezeichen durch das Pattern erkannt wird, dessen erkannter Teilstring am frühesten in der Eingabe begonnen hat. Somit wird mit dem inklusiven Start-Pattern `se` und dem inklusiven Ende-Pattern `es` in der Eingabe

`eses`

keine Insel erkannt, da nur zwei mal das Ende-Pattern erkannt wird. Sollten zwei Pattern einen identischen Teilstring erkennen können, so erfolgt die Zuordnung nach folgender Hierarchie:

1. inklusives Start-Pattern
2. exklusives Start-Pattern
3. inklusives Ende-Pattern
4. exklusives Ende-Pattern

Wenn das Pattern `e` jedem der vier Typen zugeordnet ist, wird die obige Eingabe als eine einzige Insel (`eses`) erkannt werden, da alle `e` mit einem inklusive Start-Pattern matchen.

8.8 Ausführung der semantischen Aktionen

Bei der Traversierung des internen Parse-Forests veranlasst der `TreeTraverser` den bei ihm registrierten `GraphBuilder` die aktuellen semantischen Aktionen auszuführen (siehe Abschnitt 8.4). In diesem Abschnitt wird anhand eines EDL-Moduls veranschaulicht, *wie der `GraphBuilder` die auszuführenden semantischen Aktionen eindeutig identifizieren kann*.

Beispiel

```
module Number
...
exports
   sorts Cipher Number
   lexical start-symbols Number
   lexical syntax
      rule [0-9] -> Cipher

      rule #$ = 0;#
           Cipher
             #$ = {return ((Integer)#$#) * 10 +
                     Integer.parseInt((String)#lexem($0)#);
                  };
             #
           + -> Number

      rule #$ = 0;#
           Cipher
             #$ = {return ((Double)#$#) * 10 +
                     Double.parseDouble((String)#lexem($0)#);
                  };
             #
           +
           [\,]
           Cipher+
           #$ = {String value = "0." + #lexem($2)#;
                 return ((Double)#$#)+ Double.parseDouble(value);
                };
           #
            -> Number
```

Listing 8.6: Das Beispielmodul `Number`.

Listing 8.6 zeigt das *Modul* `Number`, welches die Ziffernfolgen der ganzen und Fließkommazahlen erkennt und daraus ihren Wert berechnet. In Zeile 7 wird definiert, was eine Ziffer ist. Die Regel in den Zeilen 9 bis 15 behandelt *ganze Zahlen*. Zunächst wird der bisherige Wert in Zeile 9 auf 0 gesetzt. Für jede erkannte Ziffer wird der bisherige Wert mit 10 multipliziert (Zeile 11) und der Wert der neu erkannten Ziffer aufaddiert.

Die Regel zur Erkennung von *Fließkommazahlen* ohne Exponent (Zeile 17 bis 30) arbeitet für die Vorkommastellen analog. Der einzige Unterschied besteht darin, dass jetzt

Double-Werte erzeugt werden. Nachdem alle Nachkommastellen erkannt wurden, wird in Zeile 26 das erkannte Lexem mit `"0."` konkateniert und der entsprechende Double-Wert auf das bisherige Ergebnis addiert (Zeile 27).

```
...
259: lex(Cipher+) -> lex(Cipher*)
260:  -> lex(Cipher*)
261: lex(Cipher*) -> cf(Cipher*)
262: lex(Cipher*) lex(Cipher*) -> lex(Cipher*) {left}
263: lex(Cipher*) lex(Cipher+) -> lex(Cipher+)
264: lex(Cipher+) lex(Cipher*) -> lex(Cipher+)
265: lex(Cipher+) lex(Cipher+) -> lex(Cipher+) {left}
266: lex(Cipher+) -> cf(Cipher+)
267: lex(Cipher) -> lex(Cipher+)
268: lex(Cipher+) [\,] lex(Cipher+) -> lex(Number) {definedAs()}
269: lex(Cipher+) -> lex(Number) {definedAs()}
270: [0-9] -> lex(Cipher) {definedAs()}
271: lex(Number) -> <START>
...
```

Listing 8.7: Die aus dem Beispielmodul `Number` generierten BNF-Regeln.

Aus dem soeben vorgestellten EDL-Modul wird im Vorverarbeitungsschritt eine *Parse-Tabelle* und ein *`GraphBuilder`* generiert. Listing 8.7 zeigt die BNF-Regeln, die aus dem Modul `Number` erzeugt wurden.

Regel 271 wurde aus dem lexikalischen Start-Symbol erzeugt und die Regeln 268-270 sind die in der Grammatik definierten Regeln, was an dem `definedAs`-Attribut zu erkennen ist. Aus Gründen der Übersichtlichkeit wurde der Inhalt dieses Attributs nicht dargestellt. Um die Semantik des Terms `Cipher+` zu bewahren, wurden die Regeln 259 bis 267 erzeugt.

Arbeitsweise der `execute()`-Methode

Da für die Herleitung aller drei Vorkommen von `Cipher+` in beiden `Number`-Regeln *dieselben BNF-Regeln verwendet werden, jedoch unterschiedliche semantische Aktionen ausgeführt* werden sollen, muss bei der Semantik der `execute()`-Methode des `GraphBuilder`s berücksichtigt werden, wo im EDL-Modul die auszuführenden Aktionen definiert wurden.

Um dies zu gewährleisten, werden zunächst alle BNF-Regeln bestimmt, die in der Grammatik definiert wurden: d.h. Regeln vom Typ *DEFINED und START*. Im obigen Beispiel sind dies die Regeln 268 bis 271. Da allerdings nur die Regeln 268 bis 269 mit semantischen Aktionen versehen sind, sieht die generierte `execute()`-Methode wie in Listing 8.8 gezeigt aus.

```
1  @Override
2  public void execute(Stack stack) {
3     StackElement currentElement = stack.getCurrentElement();
4     assert ...;
5     StackElement parent = currentElement
6              .getParentApplicationOfDefinedRule();
7     switch (parent.getAppliedRule().getNumber()) {
8     // #############
9     // Module Number
10    // #############
11    case 269:
12       // lex: Cipher+ -> Number
13       execute_Rule269(currentElement, parent);
14       break;
15    case 268:
16       // lex: Cipher+ [\,] Cipher+ -> Number
17       execute_Rule268(currentElement, parent);
18       break;
19    }
20 }
```

Listing 8.8: Die generierte `execute()`-Methode.

In Zeile 3 wird aus dem aktuellen `Stack` das aktuelle Element extrahiert und in den Zeilen 5 bis 6 das `StackElement`, das die zuletzt angewendete in der Grammatik definierte Regel *r* repräsentiert. Anhand der Nummer von *r* wird in den Zeilen 7 bis 19 entschieden, welche Folge-Methode aufgerufen wird.

Um die Lesbarkeit des generierten Codes zu erhöhen, werden die `case`-Blöcke nach Modulen geordnet erzeugt. Die Regeln eines Moduls werden dabei durch einen Kommentar eingeleitet, wie in den Zeilen 8 bis 10 zu sehen. Jeder `case`-Fall enthält einen Kommentar, in dem die entsprechende Regel in der Form angegeben wird, wie sie vom Nutzer in der Grammatik erstellt worden ist. Wie in Zeile 12 zu sehen, enthält es darüber hinaus noch die Information, dass es sich um eine lexikalische Regel handelt.

```
/**
 * Rule 269: lex(Cipher+) -> lex(Number) {definedAs()}
 */
private void execute_Rule269(StackElement currentElement,
      StackElement parent) {
   int currentPos = parent.getCurrentSemanticActionPosition();
   if (currentElement == parent) {
      switch (currentPos) {
      case 0:
         execute_Rule269_Position0(currentElement);
         break;
      }
   } else {
      switch (currentPos - 1) {
      case 0:
         execute_Rule269_Term0(currentElement, parent);
         break;
      }
   }
}
```

Listing 8.9: Die generierte `execute_Rule269()`-Methode.

Die generierte `execute_Rule269()`-Methode ist in Listing 8.9 zu sehen. Im JavaDoc-Kommentar in Zeile 2 ist die BNF-Regel 269 zu sehen. In Zeile 6 wird die Position im Rumpf der Regel 269 bestimmt, an der zur Zeit die Abarbeitung steht.

Sollte das oberste Element auf dem `Stack` die Anwendung der Regel 269 repräsentieren (Zeile 7), so muss anhand der aktuellen Position bestimmt werden, welche Aktion ausgeführt werden soll (Zeile 8 bis 12). Da im `Number`-Modul auf der Ebene des Rumpfs dieser Regel nur eine initiale semantische Aktion definiert ist, wird nur im Falle der Position 0 die entsprechende Aktion durch die Methode `execute_Rule269_Position0()` ausgeführt (Zeile 9 bis 13).

Sollte das oberste Element auf dem `Stack` allerdings die Anwendung einer anderen BNF-Regel repräsentieren, so muss sie aus einem Term im Regelrumpf hergeleitet worden sein. Für die Regel 269

```
lex(Cipher+) -> lex(Number) {definedAs()}
```

gibt es im Rumpf nur den Term `lex(Cipher+)` für den weitere BNF-Regeln existieren, die nicht vom Typ DEFINED sind. Da dieser Term die Position 0 im Regelrumpf besitzt, wird in den Zeilen 15 bis 17, die Methode `execute_Rule269_Term0` aufgerufen.

```
1  /**
2   * Rule 269: ## lex(Cipher+) -> lex(Number) {definedAs()}
3   */
4  private void execute_Rule269_Position0(StackElement currentElement) {
5     ...
6  }
```

Listing 8.10: Die generierte `execute_Rule269_Position0()`-Methode.

Listing 8.10 zeigt die Methode `execute_Rule269_Position0()`. Im JavaDoc-Kommentar wird in Zeile 2 die aktuelle Regel gezeigt. Die aktuelle Position, an der die durch diese Methode ausgeführten semantischen Aktionen definiert wurden, ist mit `##` markiert. In Zeile 5 wurde der Code ausgeblendet, der `$ = 0;` ausführt.

```
1  /**
2   * Rule 269: <u>lex(Cipher+)</u> -> lex(Number) {definedAs()}
3   */
4  private void execute_Rule269_Term0(StackElement currentElement,
5        StackElement parent) {
6     StackElement nextParent =
7           skipElementsWithNoSemanticAction(parent);
8     switch (nextParent.getAppliedRule().getNumber()) {
9     case 267:
10       // lex(Cipher) -> lex(Cipher+)
11       execute_Rule269_Term0_Rule267(currentElement, nextParent);
12       break;
13    }
14 }
```

Listing 8.11: Die generierte `execute_Rule269_Term0()`-Methode.

Die `execute_Rule269_Term0()`-Methode ist in Listing 8.11 zu sehen. Der aktuelle Term ist im JavaDoc-Kommentar in Zeile 2 mit den HTML-Tags versehen, die es unterstrichen darstellen.

In den Zeilen 6 und 7 wird wird ein `StackElement` gesucht, das sich im `Stack` über `parent` befindet und für eine Regelanwendung steht, bei der semantische Aktionen möglich sind. Zur Herleitung des Terms `lex(Cipher+)` könnten die folgenden Regeln angewendet werden:

```
263: lex(Cipher*) lex(Cipher+) -> lex(Cipher+)
264: lex(Cipher+) lex(Cipher*) -> lex(Cipher+)
265: lex(Cipher+) lex(Cipher+) -> lex(Cipher+) {left}
267: lex(Cipher) -> lex(Cipher+)
```

Jedoch werden die Regeln 263 und 264 vom Interpreter nicht verwendet, sodass sie

im internen Parse-Forest nicht vorkommen können. Die Regel 265 wird genutzt, um eine Folge von `lex(Cipher+)` zu erzeugen und nur Regel 267 dient der Erkennung einer einzelnen `Cipher`. Daher wäre dies die einzige Regel, bei der semantische Aktionen möglich sind. Daher wird in den Zeilen 9 bis 12 des Listings 8.11 die Methode `execute_Rule269_Term0_Rule267()` aufgerufen. Der Kommentar in Zeile 10 hilft dem Code-Verständnis, da die entsprechende Regel gezeigt wird.

Im Allgemeinen sind bei generierten BNF-Regeln der folgenden Typen semantische Aktionen möglich:

- EPSILON. Jedoch nur bei den folgenden beiden Regeln:

    `ε -> cf(())`

    `ε -> lex(())`
- OPTION.
- SEQUENCE.
- ALTERNATIVE.
- REPETITION_PROD.
- LIST.
- LIST_PROD.
- TUPLE.
- FUNCTION.

```
/**
 * Rule 269: <u>lex(Cipher+)</u> -> lex(Number) {definedAs()}<br>
 * Rule 267: lex(Cipher) -> lex(Cipher+)
 */
private void execute_Rule269_Term0_Rule267(StackElement currentElement,
      StackElement parent) {
   int currentPos = parent.getCurrentSemanticActionPosition();
   if (currentElement == parent) {
      switch (currentPos) {
      case 1:
         execute_Rule269_Term0_Rule267_Position1(currentElement);
         break;
      }
   }
}
```

Listing 8.12: Die generierte `execute_Rule269_Term0_Rule267()`-Methode.

Listing 8.12 zeigt die `execute_Rule269_Term0_Rule267()`-Methode. Sollte es sich bei dem aktuellen `StackElement` um die Anwendung dieser Regel handeln, so wird durch das Switch-Statement in den Zeilen 9 bis 13 für die möglichen Positionen im Regelrumpf, an denen semantische Aktionen möglich sind, entsprechende Methodenaufrufe erzeugt.

Da im Number-Modul für diese Regel nur hinter Cipher eine Aktion definiert ist, wird nur die Position 1 betrachtet und in Zeile 11 die entsprechende Methode aufgerufen.

Da der Term lex(Cipher) nur durch Anwendung einer vom Nutzer erstellten Regel hergeleitet werden kann, wird für diesen Term kein weiterer Methodenaufruf erzeugt. Sollte die entsprechende Cipher-Regel semantische Aktionen besitzen, so werden diese direkt durch die initiale execute()-Methode ausgeführt.

```
/**
 * Rule 269: <u>lex(Cipher+)</u> -> lex(Number) {definedAs()}<br>
 * Rule 267: lex(Cipher) ## -> lex(Cipher+)
 */
private void execute_Rule269_Term0_Rule267_Position1(
      StackElement currentElement) {
   ...
}
```

Listing 8.13: Die generierte execute_Rule269_Term0_Rule267_Position1()-Methode.

Die in Listing 8.13 gezeigte Methode führt die semantischen Aktionen aus, die für jede erkannte Cipher ausgeführt werden sollen. In diesem Beispiel wurde in Zeile 7 der Code abgekürzt, der zur Multiplikation des bisherigen Werts mit 10 und dem anschließenden Addieren der aktuellen Ziffern führt.

Die semantischen Aktionen der anderen Number-Regel werden durch die folgenden drei Methoden ausgeführt:

excute_Rule268_Position0()

excute_Rule268_Term0_Rule267_Position1()

excute_Rule268_Position3()

9 EDL-Processor

Die `EDL-Processor`-Komponente erzeugt aus einer in EDL erstellten Grammatik eine Parse-Tabelle und einen `GraphBuilder`. Diese Artefakte werden von der im vorangegangenen Kapitel vorgestellten `Parser`-Komponente genutzt, um eine Eingabe in den gewünschten TGraphen zu überführen.

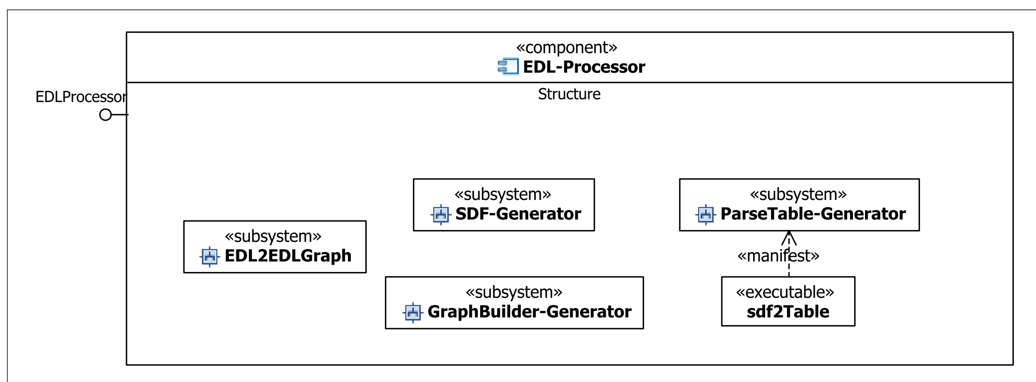


Abbildung 9.1: Die `EDL-Processor`-Komponente.

Wie in Abbildung 9.1 zu erkennen, besteht die `EDL-Processor`-Komponente aus vier Subsystemen:

1. Der *`EDL2EDLGraph`* überführt die in EDL erstellte Grammatik in einen abstrakten Syntaxgraphen (`EDLGraph`) (siehe Abschnitt 9.1).
2. Der *`SDF-Generator`* generiert aus dem `EDLGraph` valide SDF-Module (siehe Abschnitt 9.2).
3. Die erzeugten SDF-Module werden vom *`ParseTable-Generator`* in die Parse-Tabelle überführt. Als Realisierung dieser Komponente wird das in „Stratego/XT“ enthaltene Programm `sdf2Table` verwendet.
4. Der *`GraphBuilder-Generator`* erzeugt schließlich aus dem `EDLGraph` einen in Java implementierten `GraphBuilder`, der für die Ausführung der semantischen Aktionen zuständig ist, die in der in EDL beschriebenen Grammatik definiert sind (siehe Abschnitt 9.3).

Die Vorverarbeitung kann durch den folgenden *Aufruf der Klasse EDLPreprocessor* gestartet werden:

```
java EDLPreprocessor -i <SearchPath> -m <NameOfStartModule>
      -o <OutputPath> -p <PackagePrefix> [-s <Schema.tg>]
```

Die folgenden Kommandozeilenparameter werden benötigt:

-i <SearchPath> --input <SearchPath>	Definiert das Basisverzeichnis, in dem das Startmodul und alle von ihm importierten Module gesucht werden.
-m <NameOfStartModule> --module <NameOfStartModule>	Gibt den qualifizierten Namen des Startmoduls an. Es muss sich in der Datei <SearchPath><StartModule>.edl befinden.
-o <OutputPath> --output <OutputPath>	Definiert das Verzeichnis, in dem die generierten Dateien gespeichert werden.
-p <PackagePrefix> --packagePrefix <PackagePrefix>	Definiert das Paket-Präfix des generierten GraphBuilders.

Darüber hinaus gibt es noch den folgenden optionalen Parameter:

-s <Schema.tg> --schema <Schema.tg>	Mithilfe dieses optionalen Parameters kann das zu verwendende Schema definiert werden. Wird dieser Parameter weggelassen, so muss der JVM die in der Schema-Sektion des EDL-Moduls definierte Schema-Klasse bekannt sein.

9.1 EDL2EDLGraph

Die Aufgabe von *EDL2EDLGraph* besteht darin, eine in EDL erstellte Grammatik in einen abstrakten Syntaxgraphen (EDLGraph) zu überführen.

Abbildung 9.2 veranschaulicht die Arbeitsweise von EDL2EDLGraph. Als Eingabe benötigt er eine in EDL geschriebene Grammatik (Grammar.edl), die in einen EDLGraph überführt werden soll. Darüber hinaus verwendet EDL2EDLGraph eine vorgefertigte *Parse-Tabelle, die aus einer Grammatik für EDL generiert wurde* (EDLParseTable.tbl). Diese Grammatik wird in Abschnitt 9.1.1 näher betrachtet. Des Weiteren verwendet die Komponente EDL2EDLGraph einen EDLGraphBuilder.class, um den EDLGraphen aufzubauen. EDLGraphBuilder.class wurde zuvor aus einem *von Hand geschriebenen GraphBuilder* kompiliert.

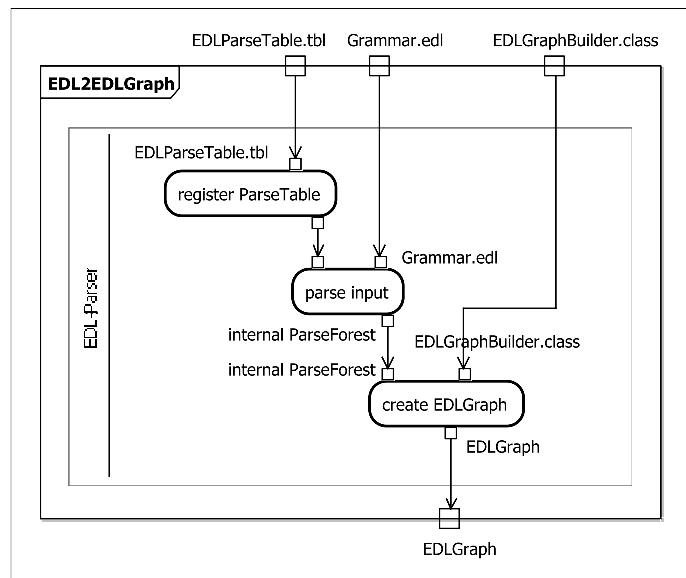


Abbildung 9.2: Die Arbeitsweise des `EDL2EDLGraph`s.

Um seine Aufgabe erfüllen zu können, *nutzt* `EDL2EDLGraph` *die* `EDL-Parser`*-Komponente* (siehe Kapitel 7 und 8). Diese registriert zunächst die `EDLParseTable.tbl` und parst im Anschluss die in EDL geschriebene Grammatik `Grammar.edl`. Der resultierende interne Parse-Forest wird mithilfe des `EDLGraphBuilder`s in den `EDLGraph`en überführt. Sein Schema wird in Abschnitt 9.1.2 erläutert. Schließlich folgt im Abschnitt 9.1.3 die Implementationsbeschreibung des `EDLGraphBuilder`s.

9.1.1 EDL-Grammatik

Da EDL auf SDF aufbaut, wurde zur Beschreibung der EDL-Syntax *eine SDF-Grammatik erweitert*. Diese zugrundeliegende SDF-Grammatik und alle von ihr referenzierten Module entstammen dem „Meta-Environment"-Repository[1] und dem „Stratego/XT"-Repository[2].

Diese Grammatik war allerdings unübersichtlich, weshalb sie *neu strukturiert* wurde. So ist beispielsweise die Definition der für Regeln zulässigen Attribute, die sich zuvor über die Module `Kernel`, `Modules`, `Priority`, `Restrictions` und `SDF2` erstreckte, in ein einziges Modul `Attribute` aggregiert. Darüber hinaus ist bei der Neustrukturierung auf die Erweiterbarkeit eines Moduls um EDL-spezifische Sektionen wie beispielsweise die Schema-Sektion geachtet worden (siehe Abschnitt 9.1.1.1).

[1] `http://svn.meta-environment.org/sdf-library/trunk/`

[2] `https://svn.strategoxt.org/repos/StrategoXT/java-front/trunk/syntax/src/languages/java-15`

Im Anschluss werden als besondere Aspekte der EDL-Grammatik die Zuordnung der semantischen Aktionen zu den einzelnen Termen (Abschnitt 9.1.1.2), die Erkennung der nutzerspezifischen semantischen Aktionen (Abschnitt 9.1.1.3) und die Restriktionen bezüglich der Bezeichnervergabe (Abschnitt 9.1.1.4) beschrieben.

9.1.1.1 Erweiterbarkeit

Um die Erweiterbarkeit der SDF-Grammatik zu verdeutlichen, wird in diesem Abschnitt erklärt, wie ein SDF-Modul um die Schema-Sektion der EDL erweitert wurde. Die Regeln, die EDL-spezifische Syntax beschreiben, sind in den einzelnen Listings markiert.

```
1  module grammar/Module
2  imports
3     grammar/sections/Main
4  ...
5  exports
6     context-free syntax
7        "module" ModuleName InitialSection* Section* -> Module
8
9        Keyword   -> ModuleName {reject}
10
11       "module" -> Keyword
12    ...
```

Listing 9.1: Das Modul `grammar/Module`.

Listing 9.1 zeigt das Modul `grammar/Module`, welches die Syntax eines Moduls definiert: nämlich der mit dem Schlüsselwort `module` eingeleitete Modulname (`ModuleName`), gefolgt von initialen Sektionen (`InitialSection`) sowie den folgenden Sektionen wie beispielsweise der **`exports`**-Sektion. Die Definition der `InitialSection` und `Section` geschieht nicht in diesem Modul sondern in den importierten Modulen des `sections`-Pakets (Zeile 3).

Durch die Regel in Zeile 9 wird erreicht, dass Schlüsselwörter (`Keyword`) nicht als Modulnamen verwendet werden dürfen. In Zeile 12 wird `module` als ein solches reserviertes Wort festgelegt. Um weitere Schlüsselwörter als Modulname zu verbieten, genügt es in den importierten Modulen nur noch ein Literal als `Keyword` zu definieren, wie in Listing 9.3 beschrieben.

```
module grammar/sections/Main
imports
   grammar/sections/ExportsSection
   ...
   grammar/sections/SchemaSection
```

Listing 9.2: Das Modul `grammar/sections/Main`.

Wie in Listing 9.2 zu sehen, dient das `grammar/sections/Main`-Modul dem Import aller definierten Sektionen. Um die SDF-Grammatik um die Schema-Sektion zu erweitern, wird in Zeile 5 das neu erstellte Modul `grammar/sections/SchemaSection` importiert. Eine Anpassung des `grammar/Module`-Moduls wird dadurch unnötig.

```
module grammar/sections/SchemaSection
...
exports
   context-free syntax
      SchemaSection -> InitialSection

      "schema" TypeName  -> SchemaSection

      "schema" -> Keyword
   ...
```

Listing 9.3: Das Modul `grammar/sections/SchemaSection`.

Schließlich zeigt Listing 9.3 das neu erstellte `grammar/sections/SchemaSection`-Modul. In Zeile 7 wird der Aufbau einer Schema-Sektion definiert. `TypeName` steht dabei für den Namen eines TGraph-Schemas. Durch die Regel in Zeile 5 wird die soeben definierte Sektion als initiale Sektion eines Moduls festgelegt. Um zu verhindern, dass ein Modul den Bezeichner `schema` erhält, wird es in Zeile 9 als reserviertes Schlüsselwort ausgewiesen.

9.1.1.2 Zuordnung von semantischen Aktionen zu Termen

In EDL können an jeder Stelle im Regelrumpf semantische Aktionen definiert werden. Da aus der SDF-Beschreibung in Abschnitt 3.2.3 die Zuordnung einer semantischen Aktion zu einem bestimmten Term nicht hervor geht, wird sie in den im Paket ***terms*** enthaltenen Modulen definiert.

```
module grammar/terms/Term
...
SemanticAction*  (Term SemanticAction*)+ -> Terms
SemanticAction*                          -> Terms

"(" SemanticAction* Term SemanticAction* ")"  -> Term {bracket}
"(" SemanticAction*  ")"                      -> Term
"(" SemanticAction* Term SemanticAction*
                  (Term SemanticAction*)+ ")" -> Term

Term SemanticAction* "?"                      -> Term
Term SemanticAction* "+"                      -> Term
Term SemanticAction* "*"                      -> Term

"{" (SemanticAction* Term SemanticAction*)
               (Term SemanticAction*) "}" "+" -> Term
"{" (SemanticAction* Term SemanticAction*)
               (Term SemanticAction*) "}" "*" -> Term

Term SemanticAction* "|" SemanticAction* Term -> Term {right}
...
```

Listing 9.4: Das Modul grammar/terms/Term.

Listing 9.4 zeigt das grammar/terms/Term-Modul, welches die Syntax der am häufigsten verwendeten zusammengesetzten Terme definiert. Bei einer Folge von Termen (Zeile 3 und 4) und einer Sequence (Zeile 6 bis 9) kann es initiale semantische Aktionen geben, die vor dem ersten enthaltenen Term stehen. Alle folgenden Aktionen sind dem vorausgegangenen Term zugeordnet.

Bei unären zusammengesetzten Termen wie die Option (Zeile 11) und der Repitition (Zeile 12 und 13) sind alle semantischen Aktionen zwischen dem Term und dem Operatorzeichen dem Term zugeordnet. Lists (Zeile 15 bis 18) bestehen aus zwei Termen. Mit dem ersten werden alle führenden und direkt folgenden Aktionen assoziiert. Dem zweiten Term sind nur die folgenden semantischen Aktionen zugeordnet.

Bei Alternatives, wie in Zeile 20 definiert, sind die Aktionen links vom Operator mit dem linken Term assoziiert und Aktionen rechts vom Operator mit dem rechten Term.

9.1.1.3 Erkennung der nutzerspezifischen semantischen Aktionen

In EDL ist es möglich, durch Angabe von mit geschweiften Klammern umschlossenen Java-Code nutzerspezifische semantische Aktionen zu definieren. Innerhalb des Java-Codes können die EDL-Aktionen durch das Umschließen mit `#` genutzt werden, wie anhand der **`user code`**-Sektion in Listing 9.5 exemplarisch gezeigt ist.

```
1  user code {
2     String filename;
3     {
4        filename = #file()#;
5     }
6  }
```

Listing 9.5: Der zu erkennende nutzerspezifische Java-Code.

In Zeile 2 wird eine Variable mit Namen `filename` angelegt und im Code-Block in den Zeilen 3 bis 5 mit dem Namen der aktuell geparsten Datei initialisiert. Um an den Dateinamen zu gelangen wird die EDL-Funktion `file` genutzt (Zeile 4).

Wie anhand dieses Beispiels zu sehen, ist es zulässig geschweifte Klammern innerhalb einer **`user code`**-Sektion zu verwenden. Da diese Klammern allerdings auch den gesamten nutzerspezifischen Code umschließen, ist es notwendig beim Parsen die Schachtelung der Klammern nachzuvollziehen.

```
1  module grammar/semantic-actions/UserCode
2  ...
3  lexical syntax
4     ~[\{\}\#]* -> JavaCode
5  lexical restrictions
6     ~[\{\}\#]* -/- ~[\{\}\#]
7  context-free syntax
8     "{" "}"                          -> UserCode {prefer}
9     "{" {JavaCode Content}+ "}" -> UserCode
10
11    UserCode                         -> Content
12    SemanticAction                   -> Content
13    ...
```

Listing 9.6: Das Modul `grammar/semantic-actions/UserCode`.

Listing 9.6 zeigt das Modul `grammar/semantic-actions/UserCode`, welches den nutzerspezifischen Code (`UserCode`) definiert. Zeile 4 beschreibt, dass Java-Code aus ei-

ner möglicherweise leeren Folge von beliebigen Zeichen außer `{`, `}` und `#` besteht. Die geschweiften Klammern dürfen nicht als Java-Code erkannt werden, da nur so ihre Schachtelung eindeutig erkannt werden kann. `#` ist ausgeschlossen, damit EDL-spezifische semantische Aktionen eindeutig identifiziert werden können. Durch Angabe der Restriktion in Zeile 6 wird für den Java-Code das Prinzip des „longest match" angewendet.

In Zeile 8 wird der leere `UserCode` definiert. Alternativ dazu kann `UserCode` auch aus einer von `Content` unterbrochenen Folge von Java-Code bestehen (Zeile 9). Dabei steht `Content` entweder für Java-Code, der in geschweiften Klammer steht, oder eine von `#` umschlossene EDL-Aktion (Zeile 11 und 12). Sollte eine nutzerspezifische semantische Aktion mit einem EDL-Statement beginnen oder enden, so wird das erste bzw. letzte `JavaCode` zum leeren Wort abgeleitet.

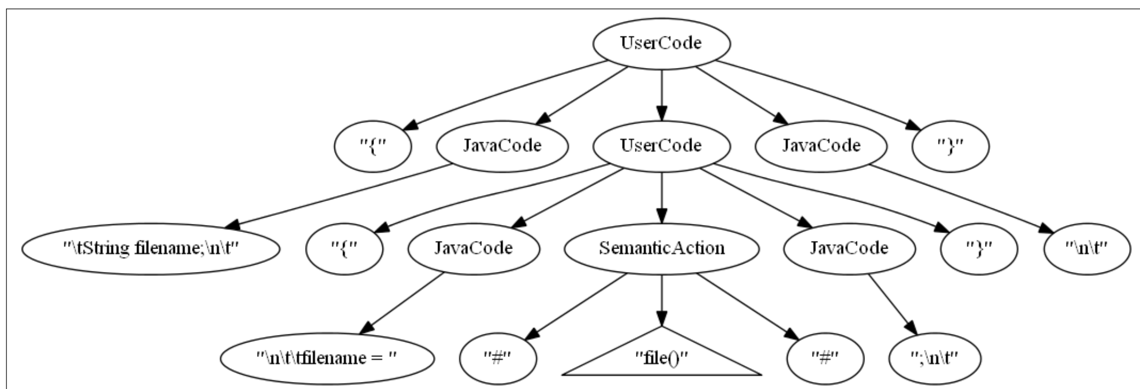


Abbildung 9.3: Die Herleitung des Beispiels in Listing 9.5.

Abbildung 9.3 zeigt die Herleitung des zu Beginn dieses Abschnitts angegebenen Beispiels in Listing 9.5 ohne die Schlüsselwörter **`user code`**. Die inneren Knoten stehen für die Anwendung einer Regel und die Blätter für die erkannten Lexeme. Der dreieckige Knoten steht für den Teilbaum, der zur Herleitung von `file()` dient.

9.1.1.4 Restriktionen bei der Bezeichnervergabe

Um Methodennamen, Graphelement-Klassen, Enumerations, ihre Konstanten und Attribute von Graphelementen schon beim Parsen eindeutig erkennen zu können, sind folgende Restriktionen bei der Bezeichnervergabe getroffen worden:

- *Graphelement-Klassen*, *Records* und *Enumerations* müssen mit einem großen Buchstaben, _ oder `$` beginnen, um sie von Methodennamen zu unterscheiden.
- *Methodennamen* müssen mit einem kleinen Buchstaben beginnen, damit sie von Konstruktor-Aufrufen der Graphelemente unterschieden werden können.

- *Attribute* von Graphelementen oder Records müssen mit einem Kleinbuchstaben, _ oder $ beginnen.
- *Enumeration-Konstanten* müssen mit einem Großbuchstaben beginnen, um sie von Graphelement-Attributen unterscheiden zu können.

9.1.2 EDL-Schema

Das neu aus der Grammatik hergeleitete *EDL-Schema* beschreibt die Menge aller möglichen abstrakten Syntaxgraphen von EDL. Die in diesem Abschnitt vorgestellten Schema-Elemente stellen nur Auszüge dar, mit denen die beschriebenen Aspekte verdeutlicht werden sollen. Das vollständige Schema ist im Anhang A abgebildet. Um zu verdeutlichen welche Knotenklassen EDL-spezifische Syntax repräsentieren, sind diese gelb eingefärbt. Die Navigationsrichtung der Kantenklassen sind im Rahmen des EDL-Schemas als Leserichtung zu verstehen.

Paketstruktur des Schemas

Ähnlich wie die im vorangegangenen Abschnitt vorgestellte Grammatik ist das EDL-Schema in Pakete unterteilt.

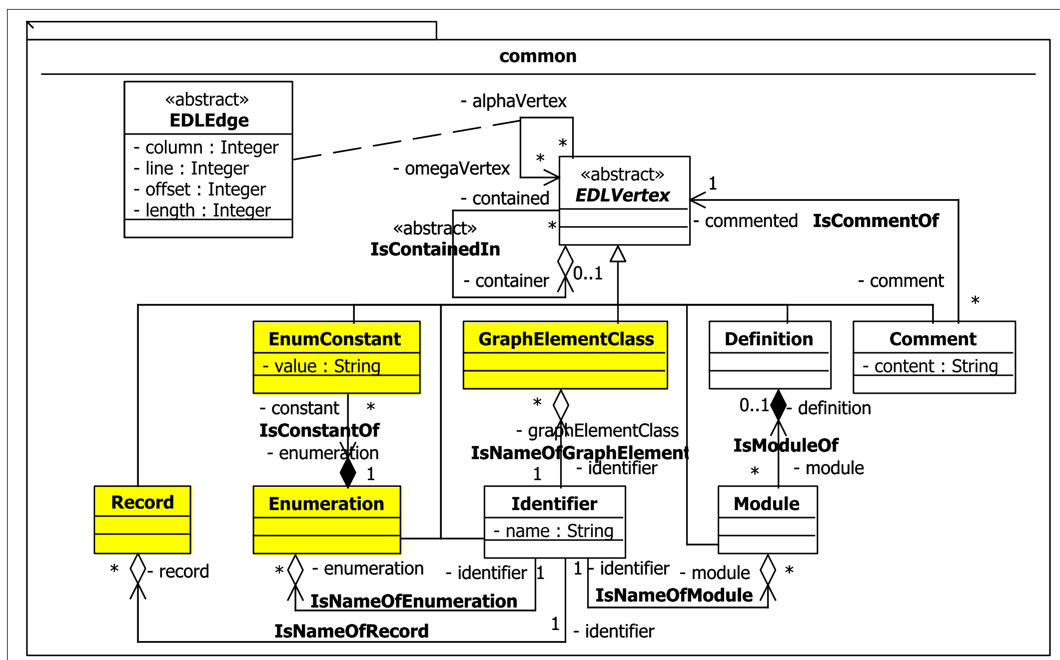


Abbildung 9.4: Der Inhalt des ***common***-Pakets (Ausschnitt).

Abbildung 9.4 zeigt den Inhalt des ***common**-Pakets*. Es enthält die abstrakte Knotenklasse *`EDLVertex`*, die die Superklasse aller Knotenklassen im EDL-Schema darstellt. Analog dazu ist *`EDLEdge`* die abstrakte Superklasse aller Kantenklassen. Sie enthält Attribute, die die genaue Position festlegen, an der sich das durch den Alphaknoten repräsentierte Lexem in der Eingabe befindet.

Da in EDL-Modulen *Kommentare* an beliebiger Stelle stehen können, muss jeder Knoten mit einem `Comment`-Knoten verbunden werden können. Dies wird durch die Kantenklasse `IsCommentOf` erreicht.

Module werden durch die Knotenklasse `Module` repräsentiert. Ihr Name wird durch eine adjazente `Identifier`-Instanz im Graphen repräsentiert. Ähnlich wie bei SDF können alle Module einer Grammatik in einer einzigen Definitionsdatei stehen. Daher kann jeder `Module`-Knoten optional einer `Definition`-Instanz zugeordnet sein.

Die Knotenklassen `GraphElementClass` sowie `Enumeration` und die zugehörige Klasse `EnumConstant` repräsentieren Elemente des im Startmodul von EDL importierten TGraph-Schemas. `Record` steht für Records des importierten Schemas.

Darüber hinaus befinden sich im ***common***-Paket noch *weitere abstrakte Kantenklassen*, die Supertypen von Klassen verschiedener anderer Pakete sind. Aus Gründen der Übersichtlichkeit werden sie hier nicht gezeigt. Die anderen Pakete sind:

section
grammar
term
semantic-action

Jedes dieser Pakete entspricht einem gleichnamigen Paket in der EDL-Grammatik und enthält alle Graphelement-Klassen, die die Syntax eines EDL-Moduls repräsentieren, welche durch die im jeweiligen Paket enthaltenen Module beschrieben wird.

Struktur des `EDLGraph`en

Das EDL-Schema ist so angelegt, dass jede Instanz *als Grundstruktur einen Syntaxbaum* hat, die dem Aufbau des repräsentierten EDL-Moduls entspricht. Die Typen der Baumkanten sind Kompositionskantenklassen.

Abbildung 9.5 verdeutlicht, dass die Kinder eines `Module`-Knotens in diesem Baum im Falle einer initialen Sektion `InitialSection`- und sonst `Section`-Instanzen sind. Erreichbar sind die Kinder durch Kanten der Kompositionsklasse `IsSectionOf`.

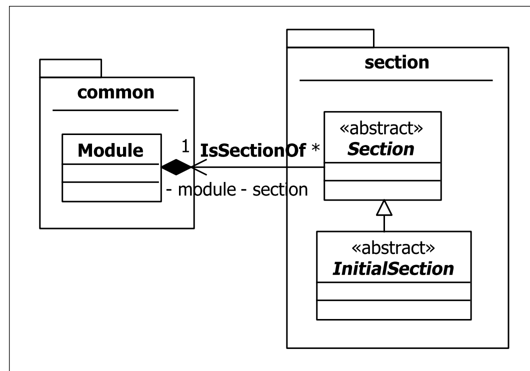


Abbildung 9.5: Beispiel zur Verdeutlichung der Baumstruktur.

Da die Repräsentation des SDF-Anteils im `EDLGraph`en nach seinem Aufbau unverändert bleibt, kann auf Sehnen innerhalb des SDF-Syntaxbaums verzichtet werden. Ein Beispiel einer solchen möglichen aber nicht realisierten Sehne ist im Listing 9.6 zu sehen: Die Restriktion in Zeile 6 wirkt sich auf den in Zeile 4 verwendeten Term `~[\{\}\#]*` aus. Die einzige Ausnahme bildet der Import eines Moduls, was durch eine Kante zum importieren Modul ausgedrückt wird. Diese Information ist für die Auswertung des EDL-Anteils notwendig, da z.B. Symboltabellen im Startmodul definiert, jedoch in allen importierten Modulen sichtbar sind.

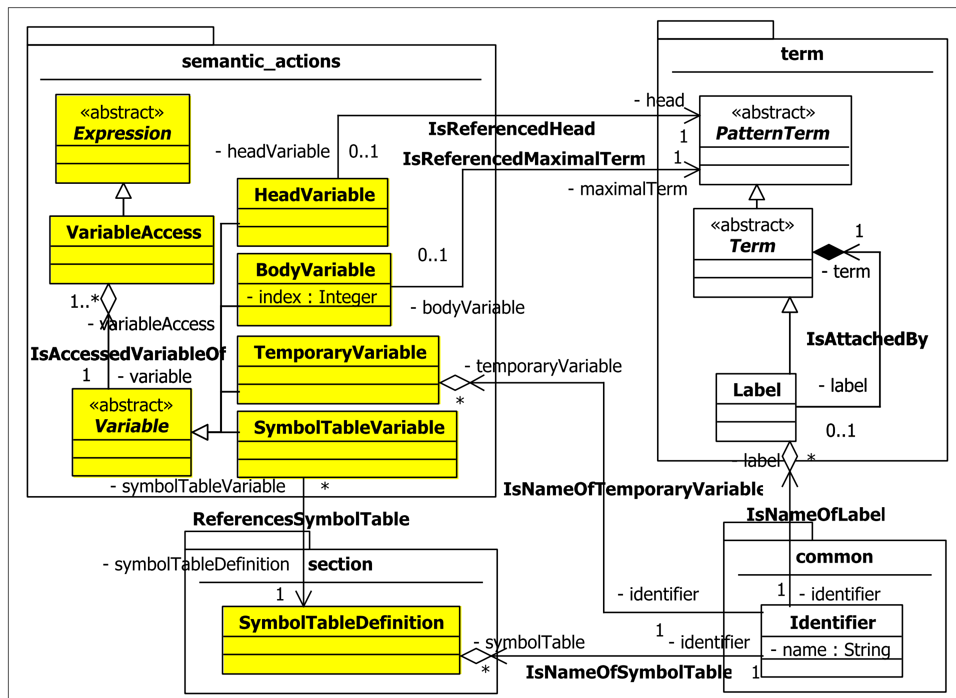


Abbildung 9.6: Die Repräsentation der Variablen.

Innerhalb des EDL-spezifischen Syntaxbaums existieren mehrere Sehnen, da im Vorfeld der Generierung des `GraphBuilder`s verschiedene Graph2Graph-Transformationen ausgeführt werden (siehe Abschnitt 9.3.1). So zeigt beispielsweise Abbildung 9.6 die Sehnen, die durch die Variablen erzeugt werden.

Variablen werden durch Instanzen der abstrakten Knotenklasse `Variable` repräsentiert. Damit für jede Verwendung dieser Variable derselbe Knoten verwendet wird, kann er über mehrere `IsAccessedVariableOf`-Kanten mit je einem `VariableAccess`-Knoten verbunden sein, wie in Abbildung 9.7 zu erkennen ist.

In EDL sind vier verschiedene Variablenarten bekannt nämlich Variablen,

1. die den Kopf einer Regel referenzieren (`HeadVariable`). Sie werden durch Kanten vom Typ `IsReferencedHead` mit dem `Term`-Knoten des jeweiligen Regelkopfs verbunden.
2. die für einen maximalen Term stehen (`BodyVariable`). Der referenzierte `Term` wird durch eine `IsReferencedMaximalTerm`-Kante angeschlossen. Darüber hinaus wird die Nummer des referenzierten Terms, wie sie in der Grammatik definiert wurde, zusätzlich im Attribut `index` persistiert, da im Falle von globalen semantischen Aktionen die Zuordnung zu einem speziellen Term noch nicht hergestellt werden kann. Variablen der Form `$labelname` werden ebenfalls durch Knoten dieses Typs repräsentiert, da das referenzierte `Label` ebenfalls einen neuen maximalen Term bildet.
3. die als temporäre Variablen definiert sind (`TemporaryVariable`).
4. die eine Symboltabelle referenzieren (`SymbolTableVariable`). Die Definition der referenzierte Symboltabelle (`SymbolTableDefinition`) ist über eine Kante vom Typ `ReferencesSymbolTable` erreichbar.

```
1 # $temp = "Content";
2   $temp = $temp;
3 #
```

Listing 9.7: Die zur Verdeutlichung der Variablenrepräsentation benutzte semantische Aktion.

Listing 9.7 zeigt eine semantische Aktion, in der in Zeile 1 die temporäre Variable `$temp` den String `"Content"` zugewiesen bekommt. In der nächsten Zeile wird die Variable sich selbst zugewiesen.

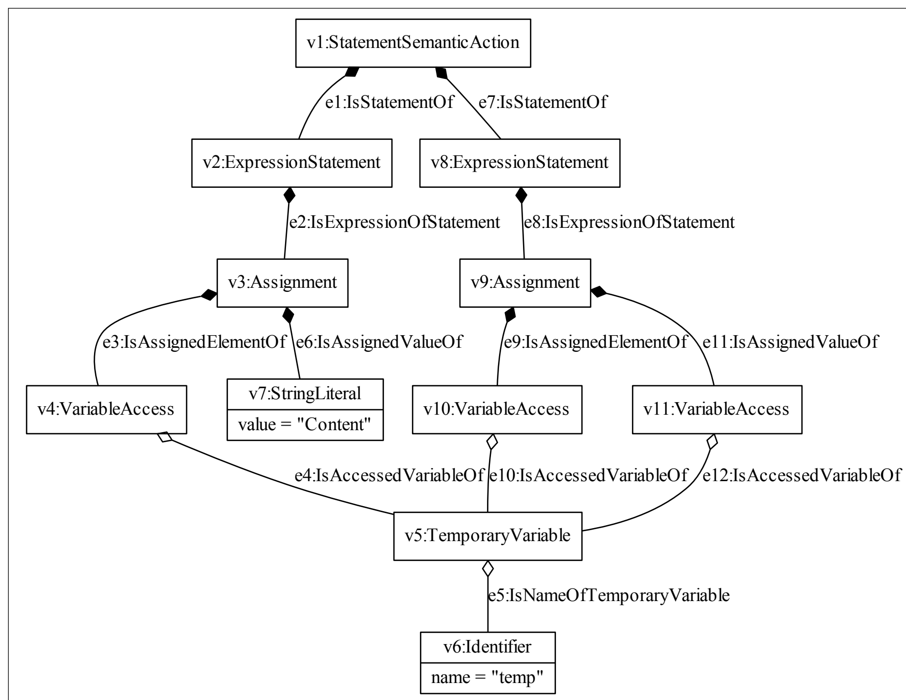


Abbildung 9.7: Der `EDLGraph` der semantischen Aktion aus Listing 9.5.

Abbildung 9.7 zeigt den `EDLGraph` der semantischen Aktion aus Listing 9.7. Aus Gründen der Übersichtlichkeit sind die Kantenattribute, die die Positionsangaben beinhalten, nicht aufgeführt. Die Kompositionskanten stellen den Syntaxbaum dar. Die Knoten $v4$, $v10$ und $v11$ vom Typ `VariableAccess` sind mit demselben `TemporaryVariable`-Knoten $v5$ verbunden, der nicht Bestandteil des Syntaxbaums ist. Dies hat den Vorteil, dass falls bei einer späteren Optimierung der für eine Selbstzuweisung stehende Knoten $v8$ gelöscht würde, der Knoten $v5$ erhalten bliebe, da er nur über eine Aggregationskante mit dem von $v8$ aufgespannten Syntaxbaum verbunden ist[3].

9.1.3 Implementation der `EDL2EDLGraph`-Komponente

Bei der Implementation der *`EDL2EDLGraph`-Komponente* wurde eine Implementation des `GraphBuilder`-Interfaces erstellt, damit der im vorangegangenen Kapitel 8 beschriebene EDL-Parser genutzt werden kann, um .edl-Dateien in einen `EDLGraph`en zu überführen.

[3]Wird in JGraLab ein Knoten gelöscht, so werden auch alle per Kompositionskanten angeschlossene Kindknoten gelöscht.

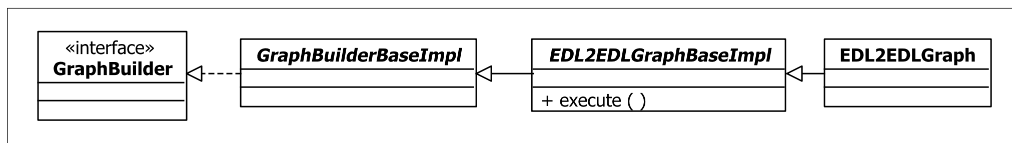


Abbildung 9.8: Die Realisierung der `EDL2EDLGraph`-Komponente.

Abbildung 9.8 zeigt die Vererbungshierarchie der `EDL2EDLGraph`-Klasse. Das *Interface `GraphBuilder`* beschreibt die Schnittstelle, die die Nutzersicht auf `GraphBuilder` beschreibt. Die abstrakte `GraphBuilderBaseImpl`-Klasse enthält die Basisimplementation sowie Methoden, die den generierten Code vereinfachen.

Aus der in SDF geschriebenen Grammatik, die die Syntax von EDL-Modulen beschreibt, wurde eine Parse-Tabelle generiert, die der EDL-Parser benötigt, um .edl-Dateien zu parsen und dabei den internen Parse-Forest aufzubauen. Der `TreeTraverser` traversiert diesen und ruft die `execute()`-Methode des registrierten `GraphBuilder`s auf. `execute()` delegiert den Aufruf zu der Methode, die die auszuführenden semantischen Aktionen enthält (siehe Abschnitt 8.8).

Da die Parse-Tabelle für EDL rund 950 Regeln enthält, wäre ein manuelles Erstellen der `execute()`-Methode zu aufwändig. Daher wurde ein Hilfsprogramm erstellt, das die *Klasse `EDL2EDLGraphBaseImpl` generiert*. Sie enthält leere Methoden für alle möglichen Positionen, an denen semantische Aktionen ausgeführt werden können. Die Klasse `EDL2EDLGraph` überschreibt schließlich die generierten Methoden, die die Positionen repräsentieren, an denen die Aktionen zum Aufbau des `EDLGraph`en ausgeführt werden. Um die Verständlichkeit zu erhöhen, wurde beim Überschreiben der automatisch generierte JavaDoc-Kommentar kopiert, da er veranschaulicht, wann die in dieser Methode enthaltenen Aktionen ausgeführt werden.

Die Trennung von generiertem und von Hand geschriebenem Code in zwei verschiedene Klassen erhöht die Wartbarkeit: Da bei einer Änderung der Grammatik eventuell neue Regeln erzeugt werden, wird eine erneute Generierung von `EDL2EDLGraphBaseImpl` nötig. Der von Hand geschriebene Code bliebe dabei erhalten.

Da sich die generierten Methodennamen wie beispielsweise bei der Veränderung von Regel-Nummern beim Erstellen einer neuen Regel verändern können, muss bei einer *erneuten Generierung von `EDL2EDLGraphBaseImpl` auch `EDL2EDLGraph` angepasst* werden. Um dies zu vereinfachen, wurde eine *Hilfsprogramm* geschrieben, das für alle Methoden in `EDL2EDLGraph`, die eine generierte Methode überschreiben, den Inhalt des JavaDoc-Kommentars mit dem der neu generierten Methode überprüft. Sollte er übereinstimmen, so braucht diese Methode nicht angepasst zu werden. Sollte es Abweichungen geben, oder eine Methode eventuell nicht mehr vorhanden sein, so bekommt der

Nutzer dies über die Konsole mitgeteilt. Die daraus resultierenden Anpassungen an `EDL2EDLGraph` müssen jedoch von Hand vorgenommen werden.

Eigenschaften eines `EDLGraph`en

Um die spätere `GraphBuilder`-Generierung zu vereinfachen, werden folgende *Eigenschaften des erzeugten `EDLGraph`en sichergestellt*:

- Schema-Elemente:
  - `Enumeration`-Knoten repräsentieren `EnumDomain`s des definierten Schemas.
  - `EnumConstant`-Knoten repräsentieren Enumeration-Konstanten, die in der diesem Knoten zugeordneten `EnumDomain` definiert sind.
  - `Record`-Knoten repräsentieren `RecordDomain`s des definierten Schemas.
  - `GraphElementClass`-Knoten repräsentieren Knoten- oder Kanten-Klassen des definierten Schemas.
  - Die zu `Enumeration`-, `Record`- und `GraphElementClass`-Knoten adjazenten `Identifier`-Knoten enthalten den jeweiligen qualifizierten Namen.
  - Verschiedene `Enumeration`-, `Record`- und `GraphElementClass`-Knoten stehen für verschiedene Elemente des definierten Schemas.
- Initiale Sektionen:
  - Der Graph enthält nur die Repräsentanten der „island start"- und „island end"-Sektionen des Start-Moduls.
  - Es gibt höchstens einen `UserCodeSection`-Knoten, der ein direktes Kind vom Knoten ist, der das Start-Modul repräsentiert. Der Inhalt aller „user code"-Sektionen von allen Modulen ist in ihm enthalten. Die Reihenfolge, in der sie innerhalb eines Moduls definiert sind, bleibt erhalten.
  - Jeder `Module`-Knoten hat höchstens ein Kind vom Typ `SymbolTables`. Sollte es in den geparsten .edl-Datei mehrere „symbol tables"-Sektionen geben, so werden die in ihnen enthaltenen Deklarationen als Kind des ersten Knotens vom Typ `SymbolTables` dargestellt. Die definierten persistierenden Kantenklassen können Instanzen der jeweiligen Knotenklassen miteinander verbinden. Der Name der definierten Symboltabellen ist innerhalb eines Moduls eindeutig.
  - Es gibt höchstens einen `ImportDeclarations`-Knoten, der ein direktes Kind vom Knoten ist, der das Start-Modul repräsentiert. Er enthält den Inhalt aller „import declarations"-Sektionen von allen Modulen.

  - Jeder `Module`-Knoten hat höchstens ein Kind vom Typ `GlobalAction`. Sollte es in den geparsten .edl-Datei mehrere „global actions“-Sektionen geben, so werden die in ihnen enthaltenen Pattern als Kind des ersten `GlobalAction`-Knotens dargestellt. Alle Pattern haben nur hinter sich eine semantische Aktion.
  - Im Graphen gibt es höchstens einen `DefaultValues`-Knoten, der die Definitionen aller im Start-Modul definierten „default values“-Sektionen darstellt. Die in dieser Sektion gesetzten Attribute sind der entsprechenden Graphelement-Klasse des definierten Schemas bekannt.
  - Der Graph enthält nur die Repräsentation der ersten im Start-Modul definierten „schema“-Sektion. Das importierte Schema muss dem `ClassLoader` der JVM bekannt sein.

- Regel-Attribute:
  - Es gibt keine Attribute mit Namen `definedAs`, da dieses Attribut von EDL benötigt wird, um beispielsweise die in EDL definierten Regeln in der Parse-Tabelle wiederzuerkennen.
- Terme:
  - Der Wert des `value`-Felds von `Literal` besitzt keine umschließenden " oder '.
  - `Label`-Knoten sind nicht Teil der Term-Bäume im Graphen.
  - `Wildcard`-Knoten dürfen nur als direkte Kinder von `Pattern`-Knoten vorkommen.
  - `Multiplicity`-Knoten dürfen nur zu `PatternTerm`-Knoten adjazent sein, die direkte Kinder von `Pattern`-Knoten sind.
  - Die minimale Multiplizität ist kleiner oder gleich der maximalen Multiplizität.
- Statements:
  - Leere Statements werden im Graphen nicht repräsentiert.
  - `ExpressionStatement`-Knoten haben einen Knoten vom Typ `MethodCall`, `ConstructorCall`, `Assignment` oder `UserCode` als direktes oder indirektes Kind.
- Ausdrücke:
  - Es gibt maximal zwei `BooleanLiteral`-Knoten im Graphen, die die Werte `true` und `false` repräsentieren.
  - Der Graph enthält jeweils maximal einen Knoten vom Typ `NullLiteral`, `AlphaConstant` und `OmegaConstant`.
  - Der Wert des `value`-Felds von `StringLiteral`-Knoten enthält keine umschließenden ".

– `ConstructorCall`-Knoten haben entweder einen adjazenten Knoten vom Typ `GraphElementClass` oder einen adjazenten `Record`-Knoten. Die Anzahl der Parameter entspricht der in der EDL-Beschreibung angegebenen Zahl.
– Für jede in der geparsten Grammatik vorkommende Variable gibt es entsprechend ihrer Sichtbarkeitsbereiche genau einen `Variable`-Knoten.
– Variablen, die einen Term im Regelrumpf referenzieren, kommen nicht in Pattern vor, die vor dem Regelrumpf ausgeführt werden sollen.
– Innerhalb einer Regel ist sichergestellt, dass einen Term im Regelrumpf referenzierende Variablen erst nach diesem Term verwendet werden[4].

- Nutzerspezifischer Code:
  – `JavCode`-Knoten repräsentieren nutzerspezifischen Code, der mindestens ein von Whitespace verschiedenes Zeichen enthält.
  – `UserCode`-Knoten haben mindestens ein Kind, d.h. sie repräsentieren keinen leeren nutzerspezifischen Code.
  – `UserCode`-Knoten, die an einer Position stehen, an der ein Rückgabe-Wert benötigt wird, enthalten ein `return`-Statement.

9.2 SDF-Generator

Der durch `EDL2EDLGraph` erzeugte `EDLGraph` wird der `SDF-Generator`-Komponente übergeben. Diese *erzeugt aus dem SDF-Anteil des Graphen valide SDF-Module*, die von dem in „Stratego/XT“ enthalten Programm `sdf2table` in eine ParseTabelle überführt werden.

Die `SDF-Generator`-Komponente ist durch die *Klasse `SDFGenerator`* realisiert. Sie traversiert den `EDLGraph`en und erzeugt eine textuelle Repräsentation für alle Graphelemente, die SDF-Code darstellen[5]. Damit der so erzeugte Text in eine einzige Datei geschrieben werden kann, wird eine SDF-Definitionsdatei an dem Ort erzeugt, der beim Aufruf des `EDLPreprocessor`s angegeben wurde. Diese Datei wird *mit dem Schlüsselwort `definition` eingeleitet und enthält die Konkatenation aller benötigten Module*. Der Name dieser Datei setzt sich aus dem Namen der Graphklasse des in den EDL-Modulen definierten Schemas gefolgt von `.def` zusammen. Sollte kein Schema importiert worden sein, so wird anstatt des Graphklassen-Bezeichners der Name des Start-Moduls genommen.

[4]Für Pattern kann diese Zusicherung nicht getroffen werden.
[5]Die Graphelement-Klassen, deren Instanzen SDF-Code repräsentieren, sind im Schema im Anhang A durch unmarkierte Knotenklassen und die diese Klassen verbindende Kantenklassen ausgedrückt.

Nachdem die Definitionsdatei generiert wurde, wird *mithilfe von* `sdf2table` *im gleichen Verzeichnis die Parse-Tabelle erzeugt*. Sie ist in einer Datei persistiert, die bis auf die Endung `.tbl` den gleichen Namen wie die Definitionsdatei trägt. Durch den Aufruf

```
sdf2Table -m <NameOfMainModule> -i <File>.def -o <File>.tbl
```

wird die Parse-Tabelle erstellt. Schließlich liefert die `SDF-Generator`-Komponente den Pfad zur Parse-Tabelle zurück, damit sie von der `GraphBuilder-Generator`-Komponente verwendet werden kann, um den `GraphBuilder` zu erzeugen (siehe folgenden Abschnitt).

9.3 GraphBuilder-Generator

Die `GraphBuilder-Generator`-Komponente *generiert* aus dem `EDLGraph`en und der erstellten Parse-Tabelle *einen* `GraphBuilder`, der die vom Nutzer definierten semantischen Aktionen ausführt.

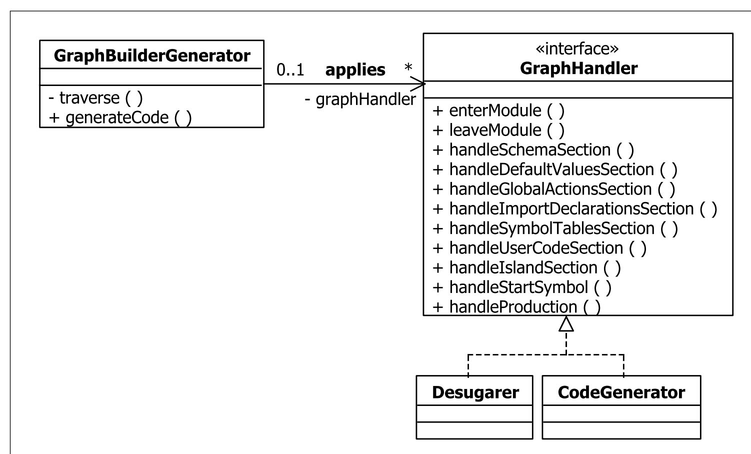


Abbildung 9.9: Die Realisierung der `GraphBuilder-Generator`-Komponente.

Realisiert wird diese Komponente durch die Klasse `GraphBuilderGenerator`, wie sie in Abbildung 9.9 dargestellt ist. Die Aufgabe dieser Klasse besteht im Traversieren des `EDLGraph`en per „depth first"-Strategie. Realisiert wird dies durch die Methode

```
traverse(vertex: EDLVertex, handlers: GraphHandler[]): void
```

Bei dieser Traversierung des dem `EDLGraph`en zugrundeliegenden Syntaxbaums wird ausgehend vom Wurzelknoten des Typs `Definition` die für die `GraphBuilder`-Generierung relevanten Knoten gesucht. Die zu diesen Knoten gehörenden Teilbäume werden vom `GraphBuilderGenerator` nicht weiter traversiert, da sie den registrierten `GraphHandler`n übergeben werden. Diese verarbeiten die entsprechenden Teilbäume verarbeiten, die dabei gegebenenfalls verändert werden können.

Je nach Typ des gefunden Knotens wird für jeden beim `GraphBuilderGenerator` registrierten `GraphHandler` *die entsprechende* `handle()`*-Methode* aufgerufen, die den jeweiligen Knoten übergeben bekommt. Im Falle von `Production`s und `StartSymbol`s gibt es einen weiteren Parameter, über den mitgeteilt wird, ob es sich um eine kontextfreie Regel bzw. Startsymbol handelt. Für Knoten vom Typ `Module` gibt es die zwei Methoden:

```
enterModule(module: Module): void
leaveModule(module: Module): void
```

Erstere wird beim Betreten und letztere beim Verlassen eines entsprechenden Knotens aufgerufen. Dies ist notwendig da ein EDL-Modul einen eigenen Gültigkeitsbereich für beispielsweise Symboltabellendeklarationen darstellt und auf diese Weise z.B. das Rücksetzen von bestimmten Feldern eines `GraphHandler`s ermöglicht wird.

Als Implementationen des `GraphHandler`-Interfaces gibt es:

- den `Desugarer`, der den `EDLGraph`en zur Code-Generierung vorbereitet (siehe Abschnitt 9.3.1) und
- den `CodeGenerator`, der den `GraphBuilder` generiert (siehe Abschnitt 9.3.2).

Die Traversierung wird durch Aufruf der Methode

```
generateCode(graph: EDLGraph, outputPath: String,
    handlers: GraphHandler[]): void
```

angestoßen. Dabei ist `graph` der zu traversierende `EDLGraph`, `outputPath` der Ort an dem der `GraphBuilder` erzeugt werden soll und `handlers` die `GraphHandler`s, die in der durch das Array vorgegebenen Reihenfolge aufgerufen werden.

9.3.1 Desugarer

Die Aufgabe des `Desugarer`s besteht in der Vorbereitung des `EDLGraph`en zur späteren Code-Generierung. Im Einzelnen bedeutet dies, dass

- bei Knoten vom Typ `GlobalAction` ein `PatternMatcher` erzeugt wird, mit dem überprüft werden kann, welche Pattern auf eine Regel passen.
- bei Startsymbolen Sequences, die nur einen Term enthalten, aufgelöst sowie die Indizes der `BodyVariable`-Knoten angepasst werden.
- bei Knoten vom Typ `Production`, die eine Regel repräsentieren, die semantischen Aktionen der zutreffenden Pattern ergänzt und das Default-Mapping ausgeführt werden. Darüber hinaus werden Symboltabellen-Annotationen in Methodenaufrufe umgewandelt, Sequences, die nur einen Term enthalten, aufgelöst sowie die Indizes der `BodyVariable`-Knoten angepasst.

Der das Interface `GraphHandler` implementierende `Desugarer` realisiert die soeben beschriebene Semantik durch die drei Methoden `handleGlobalActionSection()`, `handleStartSymbol()` und `handleProduction()`. Was beim Aufruf der ersten Methode ausgeführt wird, ist in Abschnitt 9.3.1.1 erläutert. Da `handleStartSymbol()` nur Aktionen ausführt, die ebenfalls bei `handleProduction()` durchgeführt werden, wird die Semantik beider Methoden in Abschnitt 9.3.1.2 beschrieben.

9.3.1.1 Verarbeitung von `GlobalAction`-Knoten

Die in EDL definierbaren Pattern beschreiben den Aufbau von Regeln, für die frei definierbare semantische Aktionen ausgeführt werden sollen. Diese Aktionen können entweder vor oder hinter dem Rumpf der beschriebenen Regel stehen. Das Pattern besteht aus einem einelementigen Regelkopf und einem beliebig viele Elemente umfassenden Regelrumpf. Dabei besteht ein Element aus der Wildcard `"_"` oder einem in SDF gültigen Term, die mit einer Multiplizität versehen sein können. Ein Beispiel für ein Pattern ist:

```
pattern [a](2..4) _(*) [b](0..1) -> A #$=0;#
```

Auf dieses Pattern würden die folgenden Regeln passen:

```
rule [a] [a] [b] -> A
rule [a] [a] [a] [c] [b] -> A
rule [a] [a] [a] [a] [a] [b] [a] -> A
```

Sollten mehrere Pattern auf eine Regel passen, so werden die semantischen Aktionen in der Reihenfolge ausgeführt, in der sie im EDL-Modul definiert worden sind. Aus diesem Grund muss für jede Regel *jedes Pattern separat geprüft* werden.

Die Terme bzw. Wildcards in den Pattern und Regeln werden als Strings miteinander verglichen, da so der Vergleich von Teilgraphen umgangen wird. Um die Semantik der Multiplizitäten zu realisieren, liegt die Idee nahe, reguläre Ausdrücke in Java zu verwenden, da sich beispielsweise durch `(\[a\]){2,3}` die Semantik von `[a](2..3)` ausdrücken ließe. Diese Idee funktioniert jedoch aufgrund der Wildcards nicht, die für einen beliebigen Term stehen können. Daher wird jedes Pattern zunächst in einen indeterminierten endlichen Automaten überführt, der im Anschluss in einen determinierten Automat transformiert wird.

Beispiel

Um den Ansatz mit indeterminierten endlichen Automaten zu verdeutlichen, zeigt Abbildung 9.10 den aus dem Pattern

```
pattern [a](2..4) _(*) [b](0..1) -> A #$=0;#
```

erzeugten Automaten.

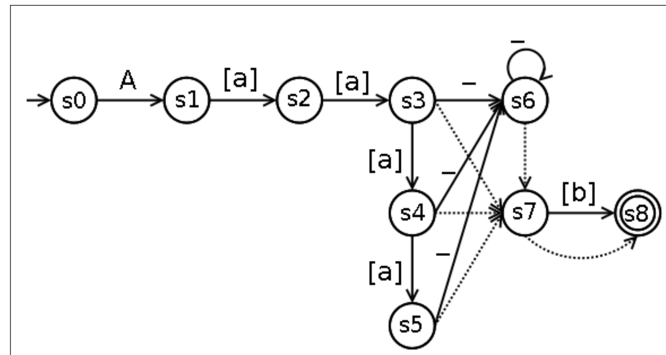


Abbildung 9.10: Der endliche Automat für das Beispiel-Pattern mit gepunkteten Pfeilen als ε-Transitionen.

Der Startzustand ist $s0$, was durch den eingehenden Pfeil ausgedrückt wird. Nur falls der Regelkopf dem String `A` entspricht, geht der Automat in den Folgezustand $s1$ über. Wird nun zweimal der String `[a]` erkannt, gelangt der Automat in den Zustand $s3$. Hier gibt es nun drei verschiedene Transitionen, um fortzufahren:

1. Durch die Transition mit der Wildcard _ kann ein beliebiger Term erkannt werden.
2. Mithilfe der ε-Transition, die durch einen gepunkteten Pfeil dargestellt ist, kann direkt in den Zustand $s7$ gewechselt werden, ohne einen Term zu erkennen. Dies würde bedeutet, das die Wildcard nicht gematcht wurde.
3. Durch die Transition nach $s4$ kann ein weiteres `[a]` erkannt werden.

Für den Zustand $s4$ bestehen analoge Transitionen wie für $s3$. Bei $s5$ gibt es keine weiteren `[a]`-Übergänge, da an diesem Punkt bereits vier `[a]` erkannt wurden.

Im Zustand $s6$ können nun beliebig viele weitere Terme durch Nutzung der Schlinge erkannt werden. Alternativ kann auch direkt in den Zustand $s7$ gewechselt werden, ohne einen Term zu erkennen. Von $s7$ aus kann direkt oder durch die Erkennung eines `[b]` in den Finalzustand $s8$ gewechselt werden.

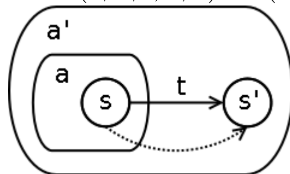
Erzeugung des indeterminierten endlichen Automaten

Die Konstruktion eines indeterminierten endlichen Automaten lässt sich durch die Funktion

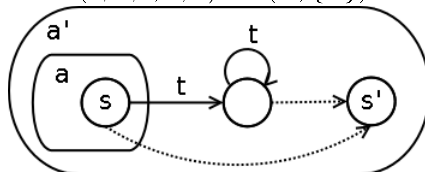
$createAutomaton : Automata \times 2^{States} \times Strings \times \mathbb{N}_0 \times Mult \rightarrow Automata \times 2^{States}$

mit *Automatons* als die Menge aller endlichen Automaten, *States* als die Menge aller Zustände, *Strings* als die Menge aller Strings und *Mult* als $\mathbb{N}_0 \cup \{*\}$ wie folgt definieren:

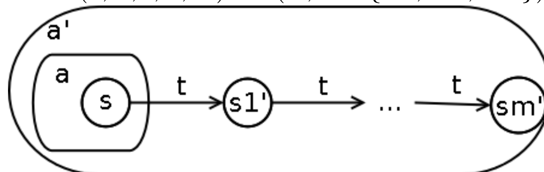
- $createAutomaton(a, S, t, 0, 0) := (a, S)$
- $createAutomaton(a, S, t, 0, 1) := (a', \{s'\})$ mit

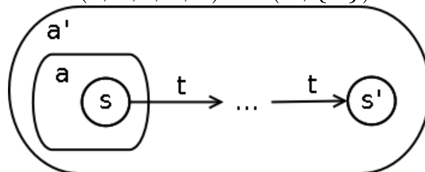
  für alle $s \in S$.
- $createAutomaton(a, S, t, 0, *) := (a', \{s'\})$ mit

  für alle $s \in S$.
- $createAutomaton(a, S, t, 0, m) := (a', S \cup \{s1', \ldots, sm'\})$ mit $m > 1$, $m \neq *$ und

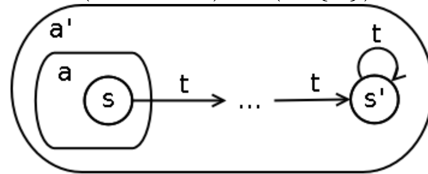
  

  für alle $s \in S$.
- $createAutomaton(a, S, t, n, n) := (a', \{s'\})$ mit $n > 0$ und

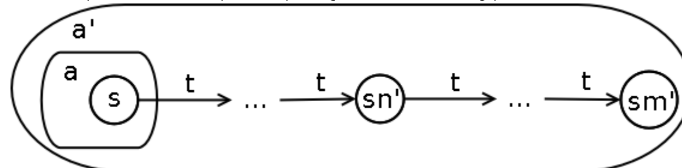
  

  für alle $s \in S$ und mit n t-Transitionen zwischen s und s'.

- *createAutomaton*$(a, S, t, n, *) := (a', \{s'\})$ mit $n > 0$ und

  für alle $s \in S$ und mit n t-Transitionen zwischen s und s'.
- *createAutomaton*$(a, S, t, n, m) := (a', \{sn', \ldots, sm'\})$ mit $0 < n < m$, $m \neq *$ und

  für alle $s \in S$ und mit n t-Transitionen zwischen s und sn' sowie , t-Transitionen zwischen sn' und sm'.

Allgemein lässt sich für ein Pattern

```
pattern t0(n0..m0) ... tl(nl..ml) -> t #...#
```

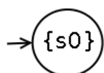
ein indeterminierter endlicher Automat wie folgt konstruieren:

1. *createAutomaton*$(a_0, S_0, t, 1, 1) = (a_1, s_1)$ wobei $S_0 = \{s_0\}$ mit s_0 der initiale Zustand und a_0 ein endlicher Automat, der nur aus s_0 besteht, sind.
2. *createAutomaton*$(a_{i+1}, S_{i+1}, t_i, n_i, m_i) = (a_{i+2}, S_{i+2})\ \forall i \in [0, l]$.
3. Mach alle zuletzt erzeugten Zustände zu finalen Zuständen.

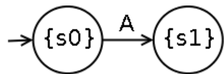
Transformation in einen determinierten endlichen Automaten

Aus dem erstellten indeterminierten endlichen Automaten wird mithilfe der Myhill-Konstruktion [EP08] oder auch Potenzmengenkonstruktion genannt ein determinierter Automat erstellt. Seine Funktionsweise wird anhand des weiter oben aufgeführten Beispiels verdeutlicht. Da jeder Zustand in dem determinierten endlichen Automaten für eine Menge S_i von Zuständen des indeterminierten Automaten steht, wird zur besseren Verständlichkeit S_i mit angegeben.

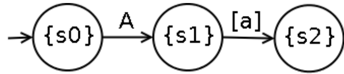
Als neuer Startzustand wird ein Zustand erzeugt, der für alle Startzustände des indeterminierten Automaten steht.



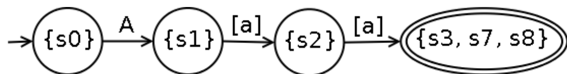
Der Zustand $s0$ hat nur eine ausgehende `A`-Transition, über die der Zustand $s1$ erreichbar ist. Daher wird im determinierten Automaten ein neuer Zustand $\{s1\}$ erzeugt und über eine `A`-Transition mit $\{s0\}$ verbunden.



Analog wird für die aus $s1$ ausgehende `[a]`-Transition verfahren.

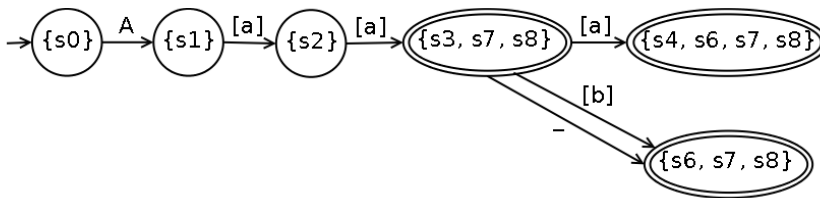


Über die aus $s2$ ausgehende `[a]`-Transition kann der Zustand $s3$ erreicht werden. Da von diesem aus per ε-Übergänge die Zustände $s7$ und $s8$ erreichbar sind, wird im determinierten Automaten der Zustand $\{s3, s7, s8\}$ erzeugt. Er ist ein Finalzustand, da $s8$ final ist.

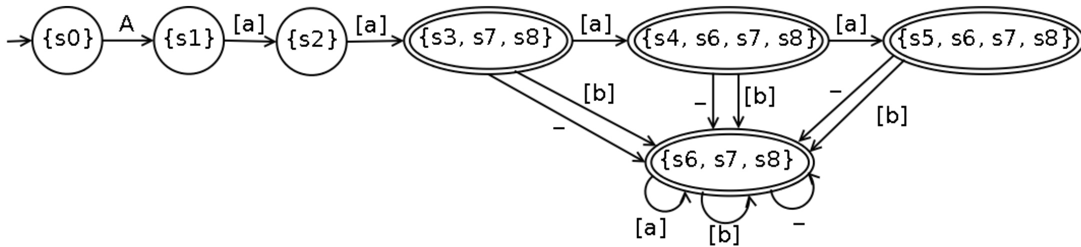


Aus den drei in $\{s3, s7, s8\}$ enthaltenen Zuständen gibt es `[a]`-, `[b]`- und _-Transitionen. Dabei steht _ für eine Wildcard durch die ein beliebiger Term erkannt werden kann. Diese drei verschiedenen Transitionsarten werden nun nacheinander betrachtet.

1. Vom Zustand $s3$ aus kann durch den Term `[a]` der Zustand $s4$ und durch ε-Übergänge auch die Zustände $s7$ sowie $s8$ erreicht werden. `[a]` kann aber auch über die Wildcard erkannt werden, wodurch $s6$ auch erreichbar ist. Da $s7$ und $s8$ keine Transitionen haben, die `[a]` akzeptieren, wird im determinierten Automaten der finale Zustand $\{s4, s6, s7, s8\}$ erzeugt und über eine `[a]`-Transition mit $\{s3, s7, s8\}$ verbunden.
2. Der Term `[b]` kann im Zustand $s3$ nur über die _-Transition erkannt werden, wodurch $s6$ und bedingt durch die ε-Übergänge auch $s7$ und $s8$ erreicht werden können. Im Zustand $s7$ kann ebenfalls `[b]` erkannt werden. Da sich $s8$ bereits in der Menge befindet, verändert sich dies nicht. $s8$ besitzt keine ausgehenden Transitionen, weswegen im determinierten Automaten der finale Zustand $\{s6, s7, s8\}$ erzeugt und über eine `[b]`-Transition mit $\{s3, s7, s8\}$ verbunden wird.
3. Alle von `[a]` und `[b]` verschiedenen Terme können ausschließlich im Zustand $s3$ über die Wildcard-Transition erkannt werden. Dadurch können die Zustände $s6$, $s7$ und $s8$ erreicht werden. Da sich im determinierten Automaten bereits der Zustand $\{s6, s7, s8\}$ befindet, wird dieser durch eine _-Transition mit $\{s3, s7, s8\}$ verbunden. Um die Determiniertheit zu erhalten, darf die soeben erzeugte Transition nur genutzt werden, falls kein anderer Übergang möglich ist.



Wird dieses Verfahren fortgesetzt, so wird schließlich der folgende determinierte Automat erzeugt:



Realisierung

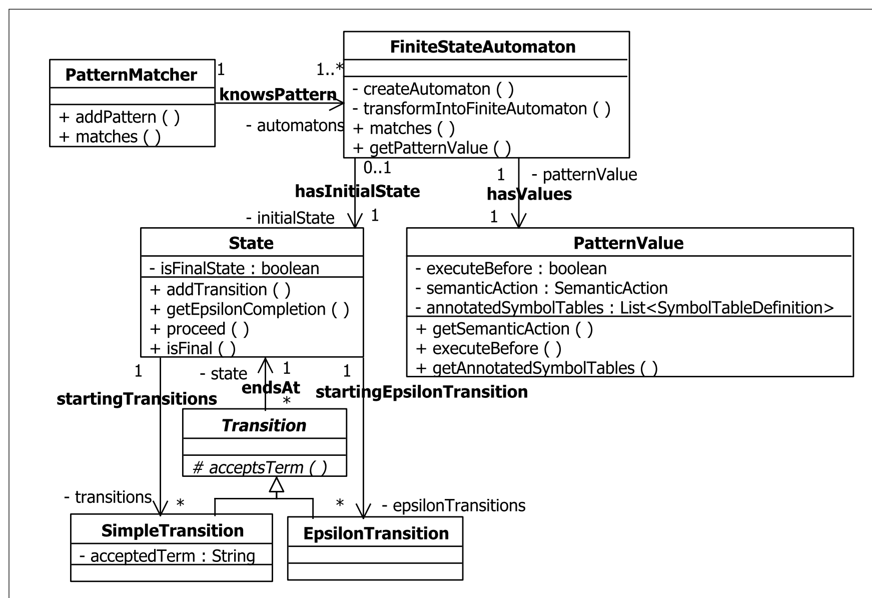


Abbildung 9.11: Der `PatternMatcher`.

Abbildung 9.11 zeigt die *Klasse* `PatternMatcher`, die für die Verarbeitung aller Pattern eines Moduls zuständig ist. Für jeden Knoten vom Typ `Pattern` wird die Methode

    addPattern(pattern: Pattern): void

aufgerufen, welche eine Instanz vom Typ `FiniteStateAutomaton` erzeugt. Die Rei-

henfolge, in der die Pattern hinzugefügt werden, entspricht der, in der später überprüft wird, welche Pattern auf eine Regel passen.

Bei der Instanziierung der Klasse `FiniteStateAutomaton` wird zunächst ausgehend vom `Pattern`-Knoten bestimmt, welche semantischen Aktionen vor oder hinter dem Regelrumpf ausgeführt werden sollen und ob es annotierte Symboltabellen gibt. Diese Werte werden in einem `PatternValue`-Objekt gekapselt, welches durch einen Getter abgefragt werden kann. Im Anschluss wird ein initialer Zustand erzeugt und mithilfe der weiter oben beschriebenen Funktion *createAutomaton*, die durch eine gleichnamige Methode repräsentiert ist, wird zunächst ein endlicher indeterminierter Automat aufgebaut und im Anschluss durch Aufruf von `transformIntoFiniteAutomaton()` in einen determinierten Automaten umgewandelt.

Die Zustände des auf diese Weise erzeugten endlichen Automats werden durch Instanzen der Klasse `State` repräsentiert. Im Feld `isFinalState` wird gespeichert, ob es sich um ein finalen Zustand handelt. Alle ausgehenden echten Transitionen werden im Feld `transitions` und die ausgehenden ε-Transitionen im Feld `epsilonTransitions` vermerkt.

Alle Transitionen, die durch die abstrakte Klasse `Transition` dargestellt werden, besitzen eine Referenz auf den Folgezustand (`state`). `EpsilonTransition` repräsentiert ε-Transitionen. `SimpleTransition` stellt alle Transitionen zwischen zwei verschiedenen Zuständen dar. Der jeweils akzeptierte Term wird als String im Feld `acceptedTerm` gespeichert.

Funktionsweise eines `PatternMatcher`s

Nachdem beim `PatternMatcher` alle Pattern registriert wurden, kann mit der Methode

```
matches(rule: Production): List<PatternValue>
```

überprüft werden, *welche Pattern auf die Regel `rule` passen*. Hierzu wird für jede registrierte `FiniteStateAutomaton`-Instanz überprüft, ob sie mit `rule` matched. Sollte dem so sein, wird über `getPatternValue()` das entsprechende `PatternValue`-Objekt an die Ergebnisliste gehängt.

Die `matches()`-Methode der Klasse `FiniteStateAutomaton` erhält als aktuellen Parameter eine Liste der String-Repräsentationen des Terms im Kopf gefolgt von den Termen im Rumpf der Regel. Beginnend mit dem Startzustand wird die Methode

```
proceed(String input): State
```

der Klasse `State` aufgerufen, die für alle ausgehenden `SimpleTransition`s überprüft,

ob sie `input` akzeptieren. Transitionen die die Wildcard akzeptieren werden als letztes kontrolliert. Sollte eine mögliche Transition gefunden werden, so wird der Folgezustand zurück gegeben. `matches()` überprüft auf diesem Wege für jede String-Repräsentation, ob es für den jeweils aktuellen Zustand einen passenden Übergang gibt. Sollten alle Eingabe-Strings erkannt worden und der letzte Zustand final sein, so liefert `matches()` `true` zurück.

9.3.1.2 Verarbeitung von `Production`- und `StartSymbols`-Knoten

Um mithilfe eines `PatternMatcher`s überprüfen zu können, welche Pattern auf die aktuelle Regel passen, muss bei der Verarbeitung der `Production`-Knoten eine Liste erzeugt werden, die die String-Repräsentationen des Regelkopfs und aller Terme des Regelrumpfs enthält. Bei der hierfür nötigen Traversierung werden alle Variablen eingesammelt, da sie in den semantischen Aktionen der passenden Pattern wiederverwendet werden müssen. Darüber hinaus werden Sequences entfernt, die nur aus einem einzigen Term bestehen.

Entfernung von einelementigen Sequences

Bei der BNF-Transformation werden einelementige Sequences der Form `(`T`)` in T überführt, wobei T für einem beliebigen Term steht (siehe 5.2.3.1). In SDF ist dies problemlos möglich, da die Semantik beider Terme äquivalent ist. Im Kontext von EDL bedeutet dies jedoch, dass eine Sichtbarkeitsebene verschwindet und die semantischen Aktionen eventuell angepasst werden müssen. Um diese Anpassungen zu veranschaulichen, werden sie an dem folgenden Beispiel ausgeführt:

```
rule [a] #$a=1;#
      ( #lift($a); $a=2;#
         [b] #$a=$0; lift($a);#
      )
      -> A #$=$1;#
```

Die semantischen Aktionen werden nun von hinten nach vorne transformiert. In einem ersten Schritt wird überprüft, ob die zu entfernende Sequence von einer Variablen referenziert wird. Im `EDLGraph`en ist dies durch Verfolgen der Kante vom Typ `IsReferencedMaximalTerm` möglich. Im vorliegenden Beispiel geschieht dies durch die Variable `$1` in der semantischen Aktion hinter dem Regelkopf. Die soeben verfolgte Kante wird nun auf `[b]` umgelegt. Hätte eine solche Variable nicht existiert, wäre der

Index bestimmt worden, den die Sequence auf Ebene des Regelrumpfs besitzt und bei Bedarf wäre eine neue Variable erzeugt worden.

Die `lift()`-Operation hinter `[b]` hat die Semantik, dass der Term `([b])` im Regelrumpf den Wert der temporären Variable `$a` besitzt. Daher wird `lift($a);` durch die Zuweisung `$1=$a;` ersetzt.

```
rule [a] #$a=1;#
     ( #lift($a); $a=2;#
        [b] #$a=$0; $1=$a;#
     )
     -> A #$=$1;#
```

Im Anschluss wird in der Zuweisung `$a=$0;` das `$0`, das den einzigen Term `[b]` referenziert, durch `$1` ersetzt.

```
rule [a] #$a=1;#
     ( #lift($a); $a=2;#
        [b] #$a=$1; $1=$a;#
     )
     -> A #$=$1;#
```

Mit dem noch verbliebenen `lift()` muss anders verfahren werden, da zu diesem Zeitpunkt `[b]` noch nicht geparst wurde und somit noch nicht referenziert werden darf. Das Verschieben der Operation hinter `[b]` ist nicht möglich, da in dem übersprungenen Statement der Wert von `$a` auf 2 verändert wird. Daher wird der alte Wert dieser temporären Variablen in dem neuen `$neu` zwischengespeichert und die Zuweisung `$1=$neu;` als erste Aktion hinter `[b]` eingefügt.

```
rule [a] #$a=1;#
     ( #$neu=$a; $a=2;#
        [b] #$1=$neu; $a=$1; $1=$a;#
     )
     -> A #$=$1;#
```

Durch die Reihenfolge der Bearbeitung von hinten nach vorne bliebe auf diesem Wege die Semantik mehrerer `lift()`-Operationen vor `[b]` erhalten. Zum Abschluss kann die Sequence entfernt und die beiden semantischen Aktionsblöcke vor `[b]` konkateniert werden.

```
rule [a] #$a=1; $neu=$a; $a=2;#
     [b] #$1=$neu; $a=$1; $1=$a;#
     -> A #$=$1;#
```

Da solche einelementigen Sequences beliebig geschachtelt sein können, wird immer die innerste Sequence zuerst transformiert.

Ergänzung der semantischen Aktionen von passenden Pattern

Sollten im aktuellen Module Pattern definiert worden sein, so wurde ein Objekt vom Typ `PatternMatcher` erzeugt. Mit seiner Hilfe werden die Pattern bestimmt, die auf die aktuelle Regel passen (siehe Abschnitt 9.3.1.1). Als Ergebnis wird eine Liste von anzuwendenden `PatternValue`-Objekten erzeugt, die in der durch die Liste vorgegebenen Reihenfolge angewendet werden müssen. Dies bedeutet zum einen, dass die Menge der an den Pattern annotierten Symboltabellen mit der Menge an der aktuellen Regel annotierten Symboltabellen vereinigt wird.

Zum anderen bedeutet dies, dass alle semantischen Aktionen der `PatternValue`-Objekte vor bzw. hinter den Regelrumpf kopiert werden. Dies ist notwendig, da falls ein Pattern auf mehrere Regeln passt, die regel-spezifischen `Variable`-Knoten verwendet werden müssen, die bereits bei der initialen String-Repräsentantenerzeugung gesammelt worden sind.

Zum Abschluss werden die semantischen Aktionen hinter dem Regelkopf ans Ende des Regelrumpfs verschoben.

Index-Anpassung der **`BodyVariable`**-Knoten

Bei der BNF-Transformation von kontextfreien Regeln wird zwischen zwei Termen die Option `LAYOUT?` ergänzt. So wird beispielsweise aus der Regel

```
Expr "+" Expr -> Expr
```

die Regel

```
Expr LAYOUT? "+" LAYOUT? Expr -> Expr
```

Damit die in einer semantischen Aktion verwendete Variable wie beispielsweise `$2` auch nach der Transformation noch das zweite `Expr` im Regelrumpf referenziert, muss ihr Index verdoppelt werden. Bei kontextfreien Startsymbolen wird aus der einzig zulässigen `BodyVariable $0` die Variable `$1` da ein initiales `LAYOUT?` bei der Transformation ergänzt wird.

Eine Besonderheit stellen Terme vom Typ `FunctionTerm` dar, weil bei der BNF-Transformation aus

(`$T_0$ ... $T_n$ => $T$`)

im lexikalischen Fall die Regel

(`$T_0$ ... $T_n$ => $T$`) `$T_0$ ... $T_n$ -> T`

und im kontextfreien Fall die Regel

(`$T_0$ ... $T_n$ => $T$`) `LAYOUT? $T_0$ LAYOUT? ... LAYOUT? $T_n$ -> T`

wird[6]. Daher müssen die Indizes zunächst inkrementiert werden, bevor sie im kontextfreien Fall verdoppelt werden.

Anwendung des Default-Mappings

Sollte ein Schema definiert worden sein und es genau eine nicht abstrakte Knotenklasse *Vc* geben, deren Name identisch mit der String-Repräsentation des Regelkopfs ist, wird ein Default-Mapping ausgeführt. Dies bedeutet, dass ein Knoten vom Typ *Vc* erzeugt wird, der als Ergebnis der Regelanwendung zurückgegeben wird. Realisiert wird dies durch die Zuweisung `$=`*Vc*`();`, die als erste Aktion vor dem Regelrumpf ergänzt wird.

Verarbeitung der annotierten Symboltabellen

Zum Abschluss wird für jede annotierte Symboltabelle *table* als erste Aktion vor dem Regelrumpf

table`.push();`

und als letzte Aktion hinter dem Regelrumpf

table`.pop();`

ergänzt.

9.3.2 CodeGenerator

Um die in allen EDL-Modulen angegebenen semantischen Aktionen ausführen zu können, wird eine einzige Java-Datei am durch den Parameter `output` angegebenen Ort generiert. Ihr Name setzt sich aus dem Namen des zu erzeugenden Graphen gefolgt von `Builder.java` zusammen. Diese Klasse ist *von `GraphBuilderBaseImpl` abgeleitet*, welche Methoden enthält, die die Code-Generierung vereinfachen. Nachdem die Code-Generierung abgeschlossen ist, wird die Java-Datei compiliert, so dass sie direkt zum Parsen von Eingabedateien verwendet werden kann.

[6]Das Symbol `...` in den Termen stellt ein Metasymbol dar.

Der Code des generierten `GraphBuilder`s sieht wie folgt aus:

Paket-Deklaration

```
1 package prefix;
```

Listing 9.8: Das Paket-Präfix.

Die Paket-Deklaration hat die in Listing 9.8 gezeigte Form, wobei `prefix` das durch den Parameter `--packagePrefix` festgelegte Paket-Präfix ist.

Import-Deklaration

Im Block der Import-Deklarationen werden zunächst alle Importe vorgenommen, die vom generierten Code benötigt werden. Im Anschluss werden alle Importe vorgenommen, die in den **`import declarations`**-Sektionen der EDL-Module definiert sind.

Klassendefinition

```
1 public class MyGraphBuilder extends GraphBuilderBaseImpl {
2     ...
3 }
```

Listing 9.9: Die Klassendefinition.

Die Klassendefinition des generierten `GraphBuilder`s für einen Graphen mit Namen `MyGraph` hat die in Listing 9.9 dargestellte Form.

Symboltabellen-Deklaration

Zum besseren Verständnis wird die Code-Generierung für die Deklarationen von (nicht-) persistenten Symboltabellen anhand eines Beispiels erklärt.

Listing 9.10 zeigt eine **`symbol tables`**-Sektion, in der in Zeile 2 eine Symboltabelle mit Namen `identifier` definiert wird. In ihr dürfen nur Knoten vom Typ `Identifier` eingefügt werden. Die Symboltabelle `sections` in Zeile 3 ist eine persistente Symboltabelle, da definiert wurde, dass jeder eingefügte Knoten vom Typ `Section` über eine

```
symbol tables
  identifier<Identifier>
  sections<Section-->IsSectionOf>:Module
```

Listing 9.10: Eine exemplarische Symboltabellen-Deklaration.

ausgehende `IsSectionOf`-Kante mit einem den Gültigkeitsbereich repräsentierenden Knoten vom Typ `Module` verbunden werden soll.

```
/**
 * module: Test line: 2 column: 1 length: 22<br>
 * identifier<common.Identifier>
 */
public final SymbolTableStack identifier_0 =
   new SymbolTableStack("identifier", this,
      new VertexClass[]{getVertexClass("common.Identifier")});

/**
 * module: Test line: 3 column: 1 length: 38<br>
 * sections<section.Section-->section.IsSectionOf>:common.Module
 */
public final PersistentSymbolTableStack sections_1 =
   new PersistentSymbolTableStack("sections", this,
      new VertexClass[]{getVertexClass("section.Section")},
      new EdgeClass[]{getEdgeClass("section.IsSectionOf")},
      new EdgeDirection[]{EdgeDirection.OUT},
      getVertexClass("common.Module"));
```

Listing 9.11: Der für die Symboltabellen-Deklaration in Listing 9.10 generierte Java-Code.

Der generierte Code für die in Listing 9.10 deklarierten Symboltabellen ist in Listing 9.11 dargestellt.

Die Zeilen 1 bis 7 zeigen den Code für die Symboltabelle `identifier`. Als Javadoc-Kommentar ist die genaue Positionsangabe der Deklaration zu sehen, aus der dieser Code generiert wurde (Zeile 2). In Zeile 3 wurde die Deklaration dieser Symboltabelle explizit angegeben. Der einfache Name der Knotenklasse `Identifier` wurde durch den qualifizierten Namen `common.Identifier` ersetzt.

Da die Symboltabelle auch nach der Nutzung des generierten `GraphBuilders` noch zugreifbar sein soll, ist die Sichtbarkeit des Felds auf `public` gesetzt worden (Zeile 5). Der Typ des Felds ist `SymbolTableStack`, wie er in Abschnitt 8.6 vorgestellt wurde. Der Name des Felds entspricht dem um `_0` erweiterten Namen, der bei der Symboltabellen-Deklaration vergeben wurde. Die Erweiterung ist notwendig, da sonst die Symbolta-

bellen des Startmoduls von gleichnamigen modul-spezifischen Symboltabellen verdeckt werden könnten.

Der erste aktuelle Parameter des Konstruktoraufrufs in Zeile 6 ist der Name der aktuellen Symboltabelle, wie er vom Nutzer deklariert wurde. Dieser Parameter wird dazu genutzt, um bei möglichen Exceptions die Symboltabelle benennen zu können, die den Fehler verursacht hat. In Zeile 7 wird ein Array erzeugt, das alle für diese Symboltabelle zulässigen Knotenklassen enthält. In diesem Beispiel ist es ausschließlich die Knotenklasse `common.Identifier`, welche über die Methode `getVertexClass()` der Klasse `GraphBuilderBaseImpl` bestimmt wird.

Der Code für die persistente Symboltabelle `sections` ist in den Zeilen 9 bis 18 zu sehen. Für den Konstruktoraufruf wird zusätzlich für jede erlaubte Knotenklasse die Angabe benötigt, durch welche Kantenklasse die Knoten dieses Typs persistiert werden (Zeile 16) und die Richtung der persistierenden Kanten (Zeile 17). Darüber hinaus wird in Zeile 18 der Typ des Knotens festgelegt, der den Gültigkeitsbereich der Elemente widerspiegelt.

Konstruktoren

Für den `GraphBuilder` werden zwei Konstruktoren generiert, die in Listing 9.12 zu sehen sind. Der erste (Zeile 1 bis 3) bekommt das Schema übergeben, für das der Graph erzeugt werden soll. Mithilfe des zweiten Konstruktors in den Zeilen 5 bis 7 kann ein Graph übergeben werden, der durch den Parse-Vorgang erweitert werden soll.

```
1  public MyGraphBuilder(Schema schema) {
2     super("Path/to/ParseTable.tbl", schema);
3  }
4
5  public MyGraphBuilder(Graph graph) {
6     super("Path/to/ParseTable.tbl", graph);
7  }
```

Listing 9.12: Die generierten Konstruktoren.

Island-Deklarationen

Sollten im `EDLGraph`en `Island`-Knoten vorhanden sein, so wird die in Listing 9.13 gezeigte Methode `activateIslandGrammar()` erzeugt. Die Methoden, die in den Zeilen 2 bis 5 aufgerufen werden, sind in `GraphBuilderBaseImpl` implementiert und führen dazu, dass nur die durch diese Pattern definierten Inseln der Eingabe geparst werden.

```
1 public void activateIslandGrammar() {
2     addInclusiceStartPattern("pattern0");
3     addExclusiceStartPattern("pattern1");
4     addInclusiceEndPattern("pattern2");
5     addExclusiceEndPattern("pattern3");
6 }
```

Listing 9.13: Die Umsetzung einer Island-Sektion.

Main-Methode

Da der generierte `GraphBuilder` von der Kommandozeile aus aufgerufen werden kann, um eine Eingabedatei zu parsen, wird eine `main()`-Methode generiert.

```
1 public static void main(String[] args) throws Exception {
2     Schema schema = instantiateSchema(loadSchema("path/to/Schema.tg"));
3     GraphBuilder graphBuilder = new MyGraphBuilder(schema);
4     graphBuilder.activateIslandGrammar();
5     processCommandLineOptions(args, graphBuilder);
6 }
```

Listing 9.14: Die generierte `main()`-Methode.

Listing 9.14 zeigt eine exemplarische `main()`-Methode. Sollte beim `EDLPreprocessor`-Aufruf der Parameter `--schema` gesetzt sein, so wird das angegebene Schema in Zeile 2 zunächst geladen und dann instanziiert. Sollte der Parameter fehlen, so muss die Schemaklasse dem ClassLoader der JVM bekannt sein und Zeile 2 sähe wie in Listing 9.15 gezeigt aus.

```
1 Schema schema = instantiateSchema(getSchemaClass("name.of.SchemaClass"));
```

Listing 9.15: Der generierte Code zur Schema-Instanziierung.

In Zeile 3 wird eine neue Instanz des generierten `GraphBuilders` erzeugt. Sollte im `EDLGraphen` Knoten vom Typ `Island` vorkommen, so wird in Zeile 4 die Verarbeitung der Eingabe auf die erkannten Inseln beschränkt. Schließlich werden in Zeile 5 die aktuellen Parameter verarbeitet und der Parse-Vorgang gestartet.

`UserCode`-Deklarationen

Der nutzerspezifische Java-Code der **`user code`**-Sektionen wird unverändert in den generierten `GraphBuilder` kopiert und mit Kommentaren umschlossen, die ihn als

nutzerspezifischen Code identifizieren. Sollten EDL-spezifische Anweisungen enthalten sein, so werden sie in Methodenaufrufe umgewandelt, die den Java-Code für diese Anweisungen enthält. Wie diese Methoden aussehen, ist weiter unten beschrieben.

`DefaultValues`-Deklarationen

Die Erzeugung von Graphelementen mithilfe von EDL-Ausdrücken führt zum Aufruf der `createVertex()`- bzw. `createEdge()`-Methode der `GraphBuilderBaseImpl`. Erstere vermerkt in der Map `positionsMap` zusätzlich zum erzeugten Knoten v die Position des Lexems der aktuellen Regelanwendung, durch die v erzeugt worden ist. Durch diese Zentralisation der GraphElement-Erzeugung ist es möglich mittels Aufrufen der überladenen `setDefaultValues()`-Methoden default-Werte zu setzen.

```
1 @Override
2 protected void setDefaultValues(Edge edge) {
3     ...
4 }
5 
6 @Override
7 protected void setDefaultValues(Vertex vertex) {
8     ...
9 }
```

Listing 9.16: Die generierten `setDefaultValues()`-Methoden.

Listing 9.16 skizziert die generierten `setDefaultValues()`-Methoden. Die erste in Zeile 1 bis 4 setzt die default-Werte von Kanten und die zweite in Zeile 6 bis 9 die default-Werte von Knoten. Um den Code für die Methodenrümpfe generieren zu können, werden die in den .edl-Dateien definierten Anweisungen zunächst nach Knoten- und Kantenklassen sortiert[7]

```
1 if(edge.isInstanceOf(getEdgeClass("common.EDLEdge"))) {
2     ...
3 }
```

Listing 9.17: Die generierten if-Anweisung für default-Werte.

[7] Wie in Abschnitt 6.2.3.2 beschrieben, ist innerhalb einer **`default values`**-Sektion nur das Setzen von Attributen für einen bestimmten Graphelement-Typ zulässig.

Für alle default-Werte einer Graphelement-Klasse wird genau eine if-Anweisung erzeugt, die den Typ des aktuellen Graphelements überprüft. Für das Setzen der default-Werte von `EDLEdge`-Kanten, sähe das if-Statement wie in Listing 9.17 gezeigt aus.

```
1 setAttribute(edge, "length", positionsMap.get(edge.getAlpha()).getLength();
```

Listing 9.18: Der generierte Code für `EDLEdge.length = length(alpha);`.

Das Setzen eines einzelnen default-Werts wie beispielsweise

`EDLEdge.length = length(alpha);`

führt zur Erzeugung des in Listing 9.18 dargestellten Java-Codes. Sollte anstelle von `alpha` `omega` stehen, so wird anstatt `getAlpha()` der Methodenaufruf `getOmega()` erzeugt. EDL-Ausdrücke auf der rechten Seite der Zuweisung, die keine Positionsangaben beschreiben, werden, wie weiter unten erläutert, übersetzt.

`execute`-Methoden

Damit die für eine Regel spezifizierten semantischen Aktionen korrekt ausgeführt werden, werden `execute()`-Methoden erzeugt, wie sie in Abschnitt 8.8 beschrieben sind. Die `execute()`-Methoden, die den Java-Code zur Ausführung der semantischen Aktionen enthalten, haben einen formalen Parameter namens `currentElement`, welcher das aktuelle `StackElement` referenziert (siehe Abschnitt 8.3).

Code für EDL-Statements

Alle an einer bestimmten Position in einer Regel definierten Statements, werden in dieselbe `execute()`-Methode übersetzt. Jedes einzelne wird dabei mit je einem try-catch-Block umschlossen, sodass im Falle eines Fehlers der Nutzer über das diesen Fehler verursachende EDL-Statement und seine Position im jeweiligen Modul informiert werden kann. So wird beispielsweise das Statement

`$=4;`

in den Java-Code in Listing 9.19 überführt.

```
1 try{
2    currentElement.getParentApplicationOfDefinedRule().setResult(4);
3 }catch(Throwable t){
4    throw new SemanticActionException("$=4;\n"
5       + "module: grammar/Main line: 5 column: 13 length: 4\n"
6       + "at rule: \"4\" -> Four", t);
7 }
```

Listing 9.19: Der generierten Code für das Statement `$=4;`.

Code für EDL-Ausdrücke

Der generierte Java-Code für EDL-Ausdrücke wird mithilfe der Funktion

$generate : Expression \to JavaCode$

definiert, wobei *Expression* die Menge der EDL-Ausdrücke und *JavaCode* ein gültiger Java-Ausdruck ist.

Der für Zuweisungen[8] generierte Java-Code unterscheidet sich je nachdem, wie der Ausdruck auf der linken Seite der Zuweisung aussieht.

- *generate*(`$ = ` *expr*) :=
  `currentElement.getParentApplicationOfDefinedRule()`
  `.setResult(`*generate*(*expr*)`)`
  mit dem EDL-Ausdruck *expr*, der den zugewiesenen Wert erzeugt.
- *generate*(`$`*i* `=` *expr*) :=
  `currentElement.getChild(`*i*`).setResult(`*generate*(*expr*)`)`
  mit *i* dem Index des referenzierten Terms und dem EDL-Ausdruck *expr*, der den zugewiesenen Wert erzeugt.
- *generate*(`$`*a* `=` *expr*) :=
  `currentElement`
  `.setValueOfTemporaryVariable("`*a*`", `*generate*(*expr*)`)`
  mit *a* als Name der temporären Variable und dem EDL-Ausdruck *expr*, der den zugewiesenen Wert erzeugt.
- *generate*(*expr*1`[`*expr*2`] =` *expr*3) :=
  `set_ElementOfList(`*generate*(*expr*1)`,`*generate*(*expr*2)`,`*generate*(*expr*3)`)`
  mit den EDL-Ausdrücken *expr*1, der eine Liste erzeugt, *expr*2, der den Index der Liste erzeugt und *expr*3, der den zugewiesenen Wert erzeugt.
- *generate*(*expr*1`.alpha =` *expr*2) :=
  `((Edge)`*generate*(*expr*1)`).setAlpha(`*generate*(*expr*2)`)`

[8] Wie in Java sind auch in EDL Zuweisungen Ausdrücke.

mit den EDL-Ausdrücken *expr*1, der eine Kante erzeugt und *expr*2, der einen Knoten erzeugt.

- *generate*(*expr*1`.omega = `*expr*2) :=
  `((Edge)`*generate*(*expr*1)`).setOmega(`*generate*(*expr*2)`)`
  mit den EDL-Ausdrücken *expr*1, der eine Kante erzeugt und *expr*2, der einen Knoten erzeugt.
- *generate*(*expr*1`.`*field* `=` *expr*2) :=
  `setAttribute(`*generate*(*expr*1)`, "`*field*`", `*generate*(*expr*2)`)`
  mit *field*, das den Namen des Felds repräsentiert und den EDL-Ausdrücken *expr*1, der ein Objekt erzeugt, sowie *expr*2, der den zugewiesenen Wert erzeugt.
  Die Methode `setAttribute()` setzt das Attribut *field* auf den durch *expr*2 erzeugten Wert, falls durch *expr*1 ein Graph oder ein Graphelement erzeugt wird. Ansonsten wird mittels Reflection das Feld *field* des durch *expr*1 erzeugten Objekts auf den durch *expr*2 erzeugten Wert gesetzt. Sollte ein solches Feld oder Attribut nicht existieren oder einen inkompatiblen Typ besitzen, wird eine Exception geworfen.

Für Literale wird der folgende Java-Code erzeugt:

- *generate*(`null`) :=
  `null`
- *generate*(`true`) :=
  `true`
- *generate*(`false`) :=
  `false`
- *generate*(*i*) :=
  i
  mit *i* eine ganze Zahl oder Fließkommazahl ist.
- *generate*(`"`*aString*`"`) :=
  `"`*aString*`"`
  mit *aString* ein String ist.

Für Variablen wird der folgende Java-Code erzeugt:

- *generate*(`$`) :=
  ```
  currentElement.getParentApplicationOfDefinedRule()
      .getResult()
  ```
- *generate*(`$0`) :=
  ```
  ((ApplicationOfListRule)currentElement)
      .getResultOfLastT1()
  ```

falls `$0` in einer semantischen Aktion hinter dem $T2$ in dem Term `{`$T1$ $T2$`}+` oder `{`$T1$ $T2$`}*` vorkommt (siehe Abschnitt 8.3).

- *generate*(`$`i) :=
    `currentElement.getChild(`i`).getResult()`
  für alle anderen Anwendungen von Variablen, die einen Term im Rumpf einer Regel referenzieren.
- *generate*(`$`a) :=
    `currentElement.getValueOfTemporaryVariable("`a`")`
  mit a als Name der temporären Variable.

Für den Zugriff auf ein Element einer Liste wird der folgende Java-Code erzeugt:

- *generate*(`$`*expr*1`[`*expr*2`]`) :=
    `get_ElementOfList(`*generate*(*expr*1)`,`*generate*(*expr*2)`)`
  mit den EDL-Ausdrücken *expr*1, der eine Liste erzeugt und *expr*2, der den Index der Liste erzeugt.

Für den `"."`-Operator wird der folgende Java-Code erzeugt:

- *generate*(*AEnumDomain*`.`*ENUMCONS*) :=
    `getEnumConstant("`*AEnumDomain*`", "`*ENUMCONS*`")`
  mit *AEnumDomain* der qualifizierte Name einer Enumeration und *ENUMCONS* der Name einer Enumeration-Konstanten.
- *generate*(*expr*1`.alpha`) :=
    `((Edge)`*generate*(*expr*1)`).getAlpha()`
  mit dem EDL-Ausdruck *expr*1, der eine Kante erzeugt.
- *generate*(*expr*1`.omega`) :=
    `((Edge)`*generate*(*expr*1)`).getOmega()`
  mit dem EDL-Ausdruck *expr*1, der eine Kante erzeugt.
- *generate*(*expr*1`.`*field*) :=
    `getAttribute(`*generate*(*expr*1)`, "`*field*`")`
  mit *field*, das den Namen des Felds repräsentiert und dem EDL-Ausdruck *expr*1, der ein Objekt erzeugt.
  Die Methode `getAttribute()` liefert den Wert des Attributs *field* zurück, falls durch *expr*1 ein Graph oder ein Graphelement erzeugt wird. Ansonsten wird mittels Reflection der Wert des Felds *field* des durch *expr*1 erzeugten Objekts zurückgegeben. Sollte ein solches Feld oder Attribut nicht existieren, wird eine Exception geworfen.
- *generate*(*expr*1`.`*methodName*`(`*expr*2`, ...,`*exprn*`)`) :=
    `callMethod(`*generate*(*expr*1)`, "`*methodName*`",`

generate(*expr*2), ..., *generate*(*exprn*))

mit *methodName*, der den Namen der Methode repräsentiert und den EDL-Ausdrücken *expr*1 bis *exprn*, die Objekte erzeugen.

Die Methode `callMethod()` sucht mittels Reflection die Methode *methodName* in der Klasse des ersten aktuellen Parameters, deren Typen der formalen Parameter mit den Klassen des dritten bis letzten aktuellen Parameters des `callMethod()`-Aufrufs kompatibel sind. Sollte eine solche Methode existieren, wird sie aufgerufen und das Ergebnis zurückgegeben. Ansonsten wird eine Exception geworfen.

Für Konstruktoraufrufe wird der folgende Java-Code erzeugt:

- *generate*(*AVertexClass*()) :=

  createVertex("*AVertexClass*",
  currentElement.getParentApplicationOfDefinedRule().
  getPosition())

  wobei *AVertexClass* der eindeutig identifizierende Name einer Knotenklasse ist.
  Die Methode `createVertex()` bekommt den Namen der Knotenklasse übergeben, die instanziiert werden soll. Darüber hinaus erhält sie die Position, die für den neuen Knoten in der `positionsMap` registriert werden soll.
- *generate*(*AnEdgeClass*(*expr*1, *expr*2)) :=

  createEdge("*AnEdgeClass*", *generate*(*expr*1), *generate*(*expr*2))

  wobei *AnEdgeClass* der eindeutig identifizierende Name einer Kantenklasse ist. Die beiden EDL-Ausdrücke *expr*1 und *expr*2 lassen sich zu Knoten oder Listen von Knoten auswerten.
  Die Methode `createEdge()` bekommt den Namen der zu instanziierenden Kantenklasse übergeben. Darüber hinaus erhält sie einen einzelnen oder eine Liste von alpha-Knoten und den bzw. die omega-Knoten.
- *generate*(*ARecordDomain*(*expr*)) :=

  createRecord("*ARecordDomain*", *generate*(*expr*))

  wobei *ARecordDomain* der eindeutig identifizierende Name einer `RecordDomain` und *expr* ein EDL-Ausdruck ist, der sich zu einer Map mit den Werten des zu erstellenden Records auswerten lässt.

Für Methodenaufrufe wird der folgende Java-Code erzeugt:

- *generate*(`graph()`) :=

  `getGraph()`
- *generate*(`file()`) :=

  `currentElement.getNameOfParsedFile()`

- *generate*(*position* `($)`) :=
  `currentElement.getParentApplicationOfDefinedRule().`
  `.get`*position*`()`
  wobei *position* für `offset`, `line`, `column`, `length` oder `lexem` stehen kann.
- *generate*(*position* `($`*i*`)`) :=
  `currentElement.getChild(`*i*`).get`*position*`()`
  wobei *position* für `offset`, `line`, `column`, `length` oder `lexem` stehen kann und *i* den Index des referenzierten Terms angibt.
- *generate*(`list(`*expr*1`, ..., `*exprn*`)`) :=
  `createList(`*generate*(*expr*1)`, ..., `*generate*(*exprn*)`)`
  wobei *expr*1 bis *exprn* EDL-Ausdrücke sind.
- *generate*(`set(`*expr*1`, ..., `*exprn*`)`) :=
  `createSet(`*generate*(*expr*1)`, ..., `*generate*(*exprn*)`)`
  wobei *expr*1 bis *exprn* EDL-Ausdrücke sind.
- *generate*(`map(`*expr*1`, ..., `*exprn*`)`) :=
  `createMap(`*generate*(*expr*1)`, ..., `*generate*(*exprn*)`)`
  wobei *expr*1 bis *exprn* EDL-Ausdrücke sind. *n* muss gerade sein, da es ansonsten einen Schlüssel ohne zugehörigen Wert gibt.
- *generate*(`lift(`*expr*`)`) :=
  `currentElement.setResult(`*generate*(*expr*)`)`
  wobei *expr* ein EDL-Ausdruck ist.
- *generate*(`getWhitespaceBefore($`*i*`)`) :=
  `currentElement.getWhitespaceBefore(`*i*`)`
  wobei *i* den Index des referenzierten Terms angibt.
- *generate*(`getPrefixWhitespace()`) :=
  `currentElement.getChild(0).getResult()`
  Diese Methode darf nur hinter kontextfreien Start-Symbolen *S* genutzt werden. Die bei der BNF-Transformation erzeugte Regel hat die folgende Form:
  `cf(LAYOUT?) cf(`*S*`) cf(LAYOUT) -> <START>`
  Somit ist `getPrefixWhitespace()` äquivalent zu dem Zugriff auf den Term mit Index 0.
- *generate*(`getSuffixWhitespace()`) :=
  `currentElement.getChild(2).getResult()`
  Diese Methode darf nur hinter kontextfreien Start-Symbolen *S* genutzt werden. Die bei der BNF-Transformation erzeugte Regel hat die folgende Form:
  `cf(LAYOUT?) cf(`*S*`) cf(LAYOUT) -> <START>`

Somit ist `getSuffixWhitespace()` äquivalent zu dem Zugriff auf den Term mit Index 2.

- *generate*(*methodName*`(`*expr*1`, ..., `*exprn*`)`) :=
  `callMethod(this, "`*methodName*`",`
  generate(*expr*1)`, ..., `*generate*(*exprn*)`)`
  mit *methodName*, der den Namen der Methode repräsentiert und den EDL-Ausdrücken *expr*1 bis *exprn*, die Objekte erzeugen.

Im Gegensatz zu den bisherigen Verfahren, wird für die Verwendung von Symboltabellen gesonderter Code erzeugt, da die zulässigen Methodenaufrufe bekannt sind:

- *generate*(*symbolTable*) :=
  symbolTable_i
  falls *symbolTable* der Name einer Symboltabelle und *i* die eindeutige Nummer dieser Symboltabellen-Variable ist.
- *generate*(*symbolTable*`.push()`) :=
  symbolTable_i`.push()`
  falls *symbolTable* der Name einer Symboltabelle und *i* die eindeutige Nummer dieser Symboltabellen-Variable ist.
- *generate*(*symbolTable*`.pop()`) :=
  symbolTable_i`.pop()`
  falls *symbolTable* der Name einer Symboltabelle und *i* die eindeutige Nummer dieser Symboltabellen-Variable ist.
- *generate*(*symbolTable*`.declare(`*expr*1`, `*expr*2`)`) :=
  symbolTable_i`.declare(`*generate*(*expr*1)`, (Vertex) `*generate*(*expr*2)`)`
  falls *symbolTable* der Name einer Symboltabelle und *i* die eindeutige Nummer dieser Symboltabellen-Variable ist. *expr*1 und *expr*2 stellen EDL-Ausdrücke dar, wobei letzterer einen Knoten erzeugen muss.
- *generate*(*symbolTable*`.use(`*expr*`)`) :=
  symbolTable_i`.use(`*generate*(*expr*)`,`
  `currentElement.getParentApplicationOfDefinedRule().`
  `getPosition())`
  falls *symbolTable* der Name einer Symboltabelle, *i* die eindeutige Nummer dieser Symboltabellen-Variable und *expr* ein EDL-Ausdruck ist.
- *generate*(*symbolTable*`.useOrDeclare(`*expr*1`, `*expr*2`)`) :=
  symbolTable_i`.useOrDeclare(`*generate*(*expr*1)`, (Vertex) `*generate*(*expr*2)`)`
  falls *symbolTable* der Name einer Symboltabelle und *i* die eindeutige Nummer dieser Symboltabellen-Variable ist. *expr*1 und *expr*2 stellen EDL-Ausdrücke dar, wobei letzterer einen Knoten erzeugen muss.

- *generate*(*symbolTable*`.getTemporaryVertices()`) :=
  symbolTable_i`.getTemporaryVertices()`
  falls *symbolTable* der Name einer Symboltabelle und *i* die eindeutige Nummer dieser Symboltabellen-Variable ist.
- *generate*(*symbolTable*`.namespace`) :=
  symbolTable_i`.getNameSpace()`
  falls *symbolTable* der Name einer Symboltabelle und *i* die eindeutige Nummer dieser Symboltabellen-Variable ist.
- *generate*(*symbolTable*`.namespace =` *expr*) :=
  symbolTable_i`.setNameSpace((Vertex)` *generate*(*expr*))
  falls *symbolTable* der Name einer Symboltabelle, *i* die eindeutige Nummer dieser Symboltabellen-Variable und *expr* ein EDL-Ausdruck ist, der einen Konten erzeugt.

```
1 /**
2  * module: grammar/Test line: 13 column: 0 length: 54
3  */
4 private void userCode_0(StackElement currentElement){
5    ...
6 }
```

Listing 9.20: Die generierte Methode für nutzerspezifischen Java-Code.

Nutzerspezifischer Java-Code wird durch den Aufruf einer neu erzeugten Methode ersetzt, wie sie in Listing 9.20 dargestellt ist. Anhand des Javadoc-Kommentars kann die Position des nutzerspezifischen Codes im EDL-Modul identifiziert werden. Sollte ein return-Statement enthalten sein, so wird anstatt des Rückgabetyps `void` der Typ `Object` genommen. Um den Methodennamen eindeutig zu machen, ist die 0 angehängt und wird für jede weitere Methode dieser Form inkrementiert.

```
1 /**
2  * module: grammar/Test line: 13 column: 0 length: 54
3  */
4 private Object semanticAction_0(StackElement currentElement){
5    return generate(expr);
6 }
```

Listing 9.21: Die generierte Methode für in nutzerspezifischem Java-Code enthaltene EDL-Ausdrücke.

Sollten im nutzerspezifischen Java-Code EDL-Ausdrücke oder -Statements verwendet worden sein, so werden diese ebenfalls durch den Aufruf einer neu erzeugten Methode

ersetzt. Für einen enthalten EDL-Ausdruck sähe die Methode wie in Listing 9.21 dargestellt aus. Im Falle von EDL-Statements wäre der Rückgabetyp `void` und jedes Statement wäre, wie bereits weiter oben beschrieben, mit einem try-catch-Block umschlossen.

10 Beispielanwendungen

Um EDL anwenden zu können, muss zunächst ein TGraph-Schema des zu erstellenden abstrakten Syntaxgraphen (ASG) erstellt werden. Im Anschluss kann die EDL Grammatik erstellt werden, die die Syntax der zu parsenden Eingabe beschreibt und die zum Aufbau des ASG führenden semantischen Aktionen definiert. Aus dieser Grammatik wird ein `GraphBuilder` generiert, mit dessen Hilfe die Eingabe geparst und der ASG aufgebaut werden kann.

Diese Arbeitsschritte werden in diesem Kapitel anhand jeweils einer Grammatik für XML (siehe Abschnitt 10.1) und Java (siehe Abschnitt 10.2) verdeutlicht. Im ersten Fall wird die Nutzung von EDL mittels Kommandozeile und im zweiten Fall aus einem Java-Programm heraus demonstriert[1].

Um die Beispiele nachvollziehen zu können, müssen die Programme `sdf2table`[2] und `dot`[3] von der Kommandozeile aus aufrufbar sein. Darüber hinaus muss JGraLab[4] und strategoxt.jar[5] dem Java-Classpath hinzugefügt werden.

10.1 XML

In diesem ersten Beispiel werden XML-Dateien in einen abstrakten Syntaxgraphen überführt, in dem Referenzen auf andere XML-Elemente durch Kanten repräsentiert werden.

[1]Die Schemas und die Grammatiken befinden sich auf der dieser Masterarbeit beiliegenden CD im Ordner `examples`.
[2]`sdf2table` ist für Windows, Linux und Mac OS unter der URL `https://svn.strategoxt.org/repos/StrategoXT/spoofax-imp/trunk/org.strategoxt.imp.nativebundle/native/` abrufbar.
[3]`dot` ist Teil von `Graphviz` und kann unter `http://www.graphviz.org/` gefunden werden.
[4]JGraLab ist unter `https://github.com/jgralab/jgralab` abrufbar.
[5]Die aktuelle Version der strategoxt.jar ist unter `http://hydra.nixos.org/job/strategoxt-java/strc-java-trunk/build` zu finden.

Schema

Als erster Schritt wird das Schema des abstrakten Syntaxgraphen benötigt, in den die XML-Dateien überführt werden sollen. Hierzu wird das in JGraLab enthaltene XML-Schema verwendet.

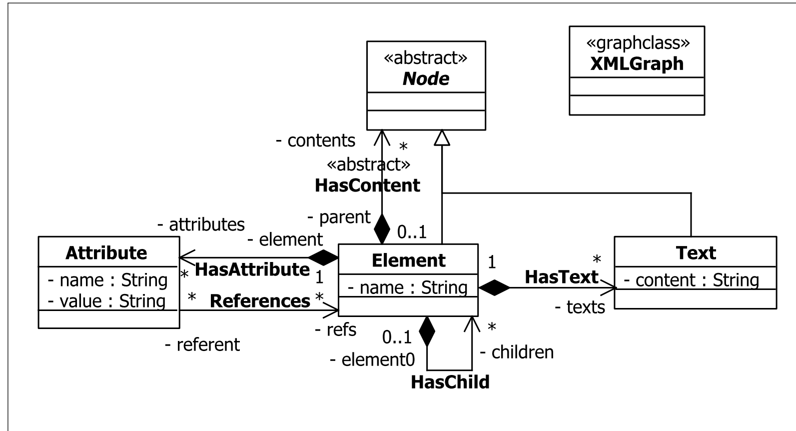


Abbildung 10.1: Das XML-Schema.

Abbildung 10.1 zeigt das Schema eines `XMLGraph`en. Die einzelnen XML-Elemente werden durch Knoten vom Typ `Element` repräsentiert. Der jeweilige Name wird im Attribut `name` persistiert. Die in einem Element geschachtelten XML-Bestandteile werden durch Instanzen des abstrakten Knotentyps `Node` dargestellt und über Kanten vom abstrakten Typ `HasContent` verbunden. Im Falle von XML-Elementen bedeutet dies, dass `Element`-Instanzen über `HasChild`-Kanten angebunden werden. Alle anderen geschachtelten Bestandteile von XML werden durch `Text`-Knoten repräsentiert und über `HasText`-Kanten mit dem Elternelement verbunden.

Die Attribute eines XML-Elements werden durch Instanzen der Knotenklasse `Attribute` dargestellt und über Kanten vom Typ `HasAttribute` angebunden. Der Name wird im Attribut `name` und der Wert im Attribut `value` persistiert. Sollte der Name `idref` oder `idrefs` lauten, so enthält dieses Attribut die Ids der referenzierten XML-Elemente als Wert. Sie werden über `References`-Kanten mit dem referenzierenden `Attribute`-Knoten verbunden.

Grammatik

Die EDL-Grammatik für XML wurde anhand der in der Spezifikation [BPSM+06] enthaltenen EBNF-Regeln erstellt. Wie in Abschnitt 5.2.2 beschrieben, bietet „Stratego/XT“ nur über die Zeichen `\255` bis `\253` eine Unterscheidung von Unicode-Buchstaben und

-Ziffern sowie sonstigen Zeichen. Da XML allerdings auf Unicode-Zeichen definiert wurde, kommt es in einigen Fällen wie beispielsweise bei der Definitionen der zulässigen Namen in XML zu Abweichungen von der Spezifikation. Im Folgenden werden die beiden Module `xml/XMLMain` und `xml/Element` vorgestellt, da sie die für den Aufbau des `XMLGraph`en relevanten semantischen Aktionen beinhalten.

```
1  module xml/XMLMain
2  schema de.uni_koblenz.jgralab.utilities.xml2tg.schema.XMLSchema
3
4  imports xml/Document %% Document
5
6  exports
7    context-free start-symbols Document
```

Listing 10.1: Das Modul `xml/XMLMain`.

Listing 10.1 zeigt das Startmodul der XML-Grammatik. In Zeile 2 wird als zu verwendendes Schema das weiter oben beschriebene XML-Schema festgelegt. Als Startsymbol wird in Zeile 7 `Document` definiert.

```
1  module xml/Element
2  imports ...
3
4  symbol tables
5    id2Element<Element>
6
7  import declarations
8    java.util.Set;
9     ...
10
11 user code{
12   private Set<String> idAttributes = new HashSet<String>();
13
14   public void defineIdAttributes(String... idAttrNames) {
15     for (String name: idAttrNames) {
16       idAttributes.add(name);
17     }
18   }
19
20   private boolean isIdAttribute(String nameOfAttribute){
21     return idAttributes.contains(nameOfAttribute);
22   }
23 }
```

Listing 10.2: Das Modul `xml/Element` (Teil 1).

Das Modul `xml/Element`, mit dessen Hilfe XML-Elemente und seine Attribute erkannt werden können, ist in Listing 10.2 zu sehen. Die Symboltabelle `id2Element` wird in Zeile 5 definiert, welche die Id eines Elements auf den zugehörigen `Element`-Knoten abbildet.

Um entscheiden zu können, in welchem Attribut sich die Id befindet, wird in Zeile 12 eine `HashSet` erzeugt, die die Namen der entsprechenden Attribute enthält. Mithilfe der Methode `defineIdAttributes()` können weitere Namen in die Menge `idAttributes` eingefügt werden (Zeile 14 bis 18). Durch Aufruf von `isIdAttribute()` in den Zeilen 20 bis 22 kann überprüft werden, ob der Name eines Attributs in `idAttributes` enthalten ist. Die für diesen nutzerspezifischen Java-Code benötigten Importe werden in den Zeilen 7 bis 9 getätigt.

```
24 global actions
25   pattern _(1..3) -> Element #$=$0;#
26
27 exports
28   sorts ...
29
30   context-free syntax
31     %% [39]
32     rule EmptyElemTag -> Element
33     rule STag Content ETag -> Element
34       #{
35         @SuppressWarnings("unchecked")
36         List<Node> listOfNodes = (List<Node>) #$1#;
37         for(Node node: listOfNodes){
38           assert node != null;
39           currentElement
40               .setValueOfTemporaryVariable("node",node);
41           if(node.isInstanceOf(Element.VC)){
42             #HasChild($, $node)#;
43           } else {
44             assert node.isInstanceOf(Text.VC);
45             #HasText($, $node)#;
46           }
47         }
48       }#
```

Listing 10.3: Das Modul `xml/Element` (Teil 2).

In Listing 10.3 wird ein leeres Tag (Zeile 32) und ein Tag mit Kindern (Zeile 33) als XML-Element definiert. Der Kommentar in Zeile 31 gibt die Nummer der EBNF-Regel in der

Spezifikation [BPSM+06] an, aus der diese Regeln entstanden sind. Als Ergebnis der Regelanwendungen für `EmptyElemTag` und `STag` wird ein `Element`-Knoten erzeugt. Durch die globale Aktion in Zeile 25 wird dieser Knoten als Ergebnis beider `Element`-Regeln festgelegt.

Die Regelanwendung für `Content` in Zeile 33 liefert als Ergebnis eine Liste von `Node`-Knoten, die den Inhalt dieses XML-Elements repräsentieren. Da je nach Knotentyp ein anderer Typ von Kante erzeugt werden muss, wird in den Zeilen 37 bis 47 über alle Elemente der Liste iteriert. Durch die Anweisung in den Zeilen 39 und 40 wird eine temporäre Variable mit Namen `name` erzeugt, die denselben Knoten wie die gleichnamige Java-Variable referenziert. Im Falle eines `Element`-Knotens wird in Zeile 42 eine `HasChild`-Kante erzeugt und ansonsten in Zeile 45 eine `HasText`-Kante.

```
49    %% [40]
50    rule
51      #$=Element();#
52      "<" Name #$.name=lexem($1);#
53      Attribute #{
54        if (isIdAttribute((String) #$0.name#)) {
55          #id2Element.declare($0.value,$);#
56        }
57        #HasAttribute($,$0);#
58      }# *
59      ">" -> STag
```

Listing 10.4: Das Modul `xml/Element` (Teil 3).

Wie in Listing 10.4 gezeigt, besteht ein Start-Tag aus einer öffnenden spitzen Klammer gefolgt von einem Namen, beliebig vielen Attributen und der schließenden spitzen Klammer. Für jede Anwendung dieser Regel wird in Zeile 51 ein `Element`-Knoten erzeugt. In Zeile 52 wird sein `name`-Attribut auf das durch die syntaktische Variable `Name` erkannte Lexem gesetzt.

Jeder durch die Anwendung der `Attribute`-Regel erzeugte `Attribute`-Knoten wird in Zeile 57 durch eine `HasAttribute`-Kante mit der in dieser Regel erzeugten `Element`-Instanz *e* verbunden. Sollte es sich bei diesem Attribut um die Definition einer Id handeln, so wird *e* unter der definierten Id in der Symboltabelle `id2Element` registriert (Zeile 55). Sollte in der Symboltabelle ein temporärer Knoten unter dieser Id enthalten sein, so wird dieser zunächst in einen `Element`-Knoten umgewandelt und mit dem neu deklarierten Knoten verschmolzen.

```
60    %% [41]
61    rule
62      Name #$.name=lexem($0);#
63      Eq AttValue #$.value=$2;#
64      -> Attribute #{
65        String nameOfAttribute=((String) #$.name#).toLowerCase();
66        if (nameOfAttribute.equals("idref")
67              ||nameOfAttribute.endsWith(":idref")) {
68          #References($,id2Element.use($.name));#
69        } else if (nameOfAttribute.equals("idrefs")
70              ||nameOfAttribute.endsWith(":idrefs")) {
71          for (String ref : nameOfAttribute.split("\\s+")) {
72            currentElement
73                .setValueOfTemporaryVariable("ref",ref);
74            #References($,id2Element.use($ref));#
75          }
76        }
77      }#
```

Listing 10.5: Das Modul `xml/Element` (Teil 4).

Listing 10.5 zeigt die Regel, die den Aufbau eines Attributs definiert. Da der Name dieser Regel identisch mit dem Namen der Knotenklasse `Attribute` ist, wird per Default-Mapping ein Knoten dieses Typs erzeugt und als Ergebnis dieser Regel festgelegt. Der Wert von `name` wird in Zeile 62 auf das durch `Name` erkannte Lexem gesetzt. In Zeile 63 wird `value` auf den von `AttValue` zurückgegebenen Wert gesetzt, welcher aus dem erkannten Lexem ohne die umschließenden `"` besteht.

Sollte das Attribut `idref` heißen, so ist sein Wert die Id eines referenzierten Elements. Für diese Id wird in Zeile 68 in der Symboltabelle `id2Element` nach einem entsprechenden `Element`-Knoten gesucht. Sollte ein solcher Knoten noch nicht registriert sein, so wird ein neuer temporärer Knoten zurückgegeben und unter der nachgefragten Id in der Symboltabelle registriert. Mithilfe einer neuen `References`-Kante wird der referenzierte Knoten mit der aktuellen `Attribute`-Instanz verbunden.

Im Falle eines `idrefs`-Attributs, besteht sein Wert aus einer durch Leerzeichen separierten Sequenz von Ids. Jede einzelne wird zunächst in den Zeilen 72 und 73 der temporären Variable `ref` zugewiesen. Im Anschluss wird in Zeile 74 der referenzierte `Element`-Knoten durch eine `References`-Kante mit der aktuellen `Attribute`-Instanz verbunden.

```
78      %% [42]
79      rule "</" Name ">" -> ETag
80      %% [43]
81      rule
82        #$=list();#
83        CharData #$.add($0);# ?
84        (
85          (Element | Reference | CDSect | PI| Comment)
86          #$.add($0);#
87          CharData #$.add($0);# ?
88        )* -> Content
```

Listing 10.6: Das Modul `xml/Element` (Teil 5).

Die Regel in Zeile 79 des Listings 10.6 definiert das Aussehen eines schließenden Tags. Der Inhalt eines XML-Elements wird durch die Regel in den Zeilen 81 bis 88 beschrieben. Als Ergebnis wird in Zeile 82 eine leere Liste erzeugt. Für jedes durch die syntaktische Variable `Element` erkannte XML-Element wird der zurückgegebene `Element`-Knoten in diese Liste eingefügt. Alle anderen syntaktischen Variablen führen zur Erzeugung von `Text`-Knoten, in deren `content`-Attribut das erkannte Lexem persistiert ist. Sie werden ebenfalls in die zurückzuliefernde Liste eingefügt.

```
89      %% [44]
90      rule
91      #$=Element();#
92      "<" Name #$.name=lexem($1);#
93      Attribute #{
94        if (isIdAttribute((String) #$0.name#)) {
95          #id2Element.declare($0.value,$);#
96        }
97        #HasAttribute($,$0);#
98      }# *
99      "/>" -> EmptyElemTag
```

Listing 10.7: Das Modul `xml/Element` (Teil 6).

Für leere Tags, wie sie in Listing 10.7 definiert sind, werden dieselben semantischen Aktionen wie für öffnende Tags in Listing 10.4 ausgeführt.

Anwendung

Um die Befehle zur Generierung des `GraphBuilder`s nachvollziehen zu können, wird davon ausgegangen, dass im aktuellen Ausführungsort die beiden Ordner `testoutput` und `examples` existieren. Im ersten werden die aus der XML-Grammatik generierten Dateien erzeugt und der zweite enthält einen Ordner namens `xml` der alle Module der XML-Grammatik enthält.

Vorverarbeitung der XML-Grammatik

Durch Aufruf des Befehls

```
java de.uni_koblenz.edl.preprocessor.EDLPreprocessor
    -i ./examples -m xml/XMLMain -o ./testoutput -p testoutput
```

wird die Datei `./examples/xml/XMLMain.edl` und alle von diesem Modul importierten Module geladen und die folgenden vier Dateien in dem Ordner `./testoutput` erstellt:

1. `XMLGraph.def`: Diese Datei enthält den aus der XML-Grammatik extrahierten SDF-Anteil. Aus ihr wurde die Parse-Tabelle erzeugt und wird nun nicht mehr benötigt.
2. `XMLGraph.tbl`: Diese Datei ist die Parse-Tabelle, die aus `XMLGraph.def` generiert wurde. Sie wird für jeden Parse-Vorgang benötigt und muss sich in demselben Verzeichnis wie der generierte `GraphBuilder` befinden.
3. `XMLGraphBuilder.java`: Diese Datei ist der generierte `GraphBuilder` mit dessen Hilfe XML-Dateien geparst werden können. Sein Paket-Präfix wurde auf den für den Parameter `-p` angegeben Wert `testoutput` gesetzt.
4. `XMLGraphBuilder.class`: Diese Datei entstand durch die Compilierung von `XMLGraphBuilder.java`. Sollte diese Datei fehlen, muss sie durch Aufruf von `javac` aus `XMLGraphBuilder.java` compiliert werden.

Parsen einer XML-Datei

Zur Verdeutlichung des Parsens wird die `build.xml` des edl-Projekts genommen, wie sie in Listing 10.8 auszugsweise dargestellt ist.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <project name="edl" basedir="." default="build">
3    <property name="projectname" value="edl" />
4    <property name="main" value="" />
5    ...
6 </project>
```

Listing 10.8: Die `build.xml` des edl-Projekts.

Durch Aufruf von

`java testoutput.XMLGraphBuilder -i ./build.xml -o ./build.tg`

wird die `build.xml` geparst und der erzeugte Graph unter `build.tg` gespeichert. Da das XML-Schema Bestandteil von JGraLab und somit im Classpath enthalten ist, wird die explizite Angabe des zu verwendenden Schemas unnötig. Der für den in Listing 10.8 gezeigten Ausschnitt erstellte Graph ist in Abbildung 10.2 zu sehen.

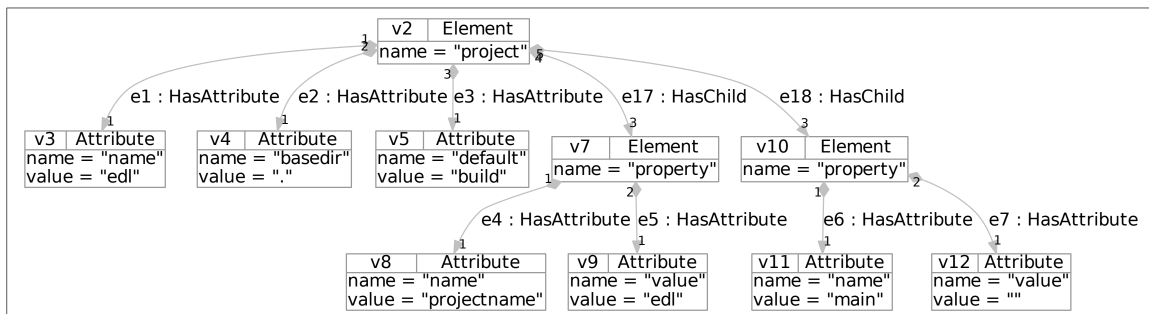


Abbildung 10.2: Der für den in Listing 10.8 gezeigten Ausschnitt erstellte Graph.

Debugging

Sollte bei der Ausführung der semantischen Aktionen in der Grammatik ein Fehler auftreten, so bietet EDL Unterstützung beim Debugging. Um dies zu demonstrieren, wird davon ausgegangen, dass die Erzeugung der Liste in Zeile 82 des Listings 10.6 vergessen worden wäre. Beim Parsen der Eingabe wird nun die folgende Exception geworfen, die die Position der Anweisung ausgibt, die die semantische Aktion ausgelöst hat:

```
de.uni_koblenz.edl.SemanticActionException: $.add($0);
module: xml/Element line: 86 column: 5 length: 10
at rule: CharData? ((Element | Reference |
   CDSect | PI | Comment) CharData?)* -> Content
java.lang.NullPointerException
```

Darüber hinaus besteht die Möglichkeit, sich den Knoten des Parse-Trees anzeigen zu lassen, bei dessen Verarbeitung die Exception geworfen wurde. Durch den Aufruf von

```
java testoutput.XMLGraphBuilder -i ./build.xml -o ./build.tg
    -debug -dot "pdf"
```

wird der Parse-Tree in einer .pdf-Datei dargestellt und der entsprechende Knoten rot dargestellt. Abbildung 10.3 zeigt den relevanten Ausschnitt dieser .pdf-Datei.

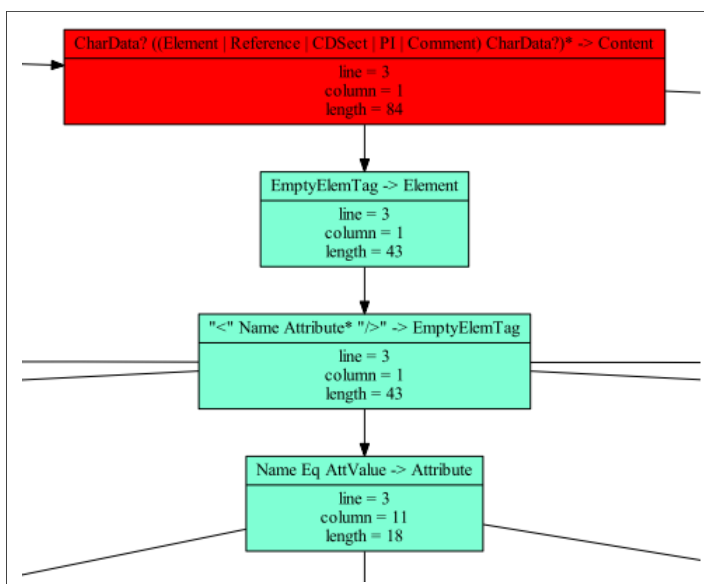


Abbildung 10.3: Der den Fehler verursachende Knoten des Parse-Trees.

10.2 Java

Als zweite exemplarische Anwendung von EDL wird aus den Java-Klassen eines beliebigen Verzeichnisses ein Graph extrahiert, der die Pakete und Klassen, die in ihnen enthaltenen Elemente und die Import-Beziehungen darstellt.

Schema

Das für dieses Beispiel verwendete Java-Schema basiert auf dem Javascanner-Schema[6]. Abbildung 10.4 zeigt das Java-Schema, welches den Aufbau der `JavaGraph`en beschreibt.

[6] Das Javascanner-Schema ist unter `https://svn.uni-koblenz.de/gupro/re-group/trunk/project/javascannerschema` zu finden.

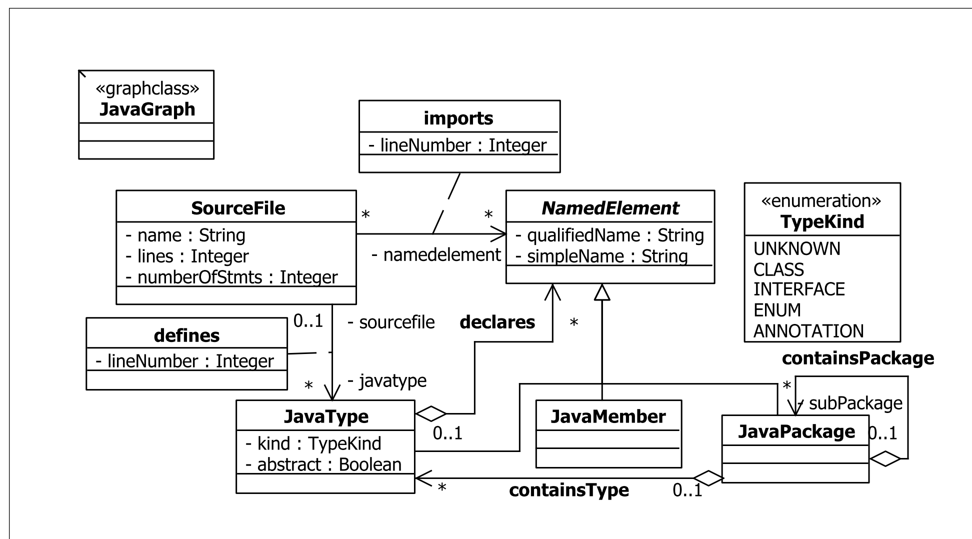


Abbildung 10.4: Das Java-Schema.

Jede geparste .java-Datei wird durch einen `SourceFile`-Knoten repräsentiert. Ihr Name wird im Attribut `name` und die Anzahl der enthaltenen Zeilen im Attribut `lines` persistiert. Die Anzahl der enthaltenen Anweisungen ist in `numberOfStmts` gespeichert.

Jedes in der Paket-Deklaration vorkommende Paket wird durch eine `JavaPackage`-Instanz repräsentiert. Ihre Schachtelung wird durch `ContainsPackage`-Kanten dargestellt.

Für die in einer .java-Datei getätigten Importe wird eine Kante vom Typ `Imports` erzeugt. Die entsprechende Zeilennummer wird im Attribut `lineNumber` persistiert. Da in Java der Import sowohl eines einzelnen Typs als auch aller Typen eines Pakets möglich ist, ist der Omega-Typ von `Imports` die Knotenklasse `NamedElement`, die die gemeinsame Oberklasse von `JavaPackage` und `JavaType` ist.

Die in Java definierten Typ-Arten sind Interfaces, Klassen, Enumerationen und Annotationen. Im `JavaGraph`en werden sie durch Knoten vom Typ `JavaType` repräsentiert. Anhand des Attributs `kind` ist erkennbar, um welche Typ-Art es sich handelt. Der Wert `UNKNOWN` wird für importierte Typen genutzt, die in keiner der geparsten .java-Dateien definiert worden sind. Das Attribut `abstract` vermerkt, ob der repräsentierte Typ abstrakt ist. Ungeschachtelte Typen, die direkt in einer .java-Datei definiert sind, werden über `Defines`-Kanten mit der jeweiligen `SourceFile`-Instanz verbunden. Die Zeile, in der die Definition beginnt, ist im Attribut `lineNumber` gespeichert. Über Kanten vom Typ `ContainsType` wird vermerkt, in welchem Paket der durch einen `JavaType`-Knoten repräsentierte Java-Typ definiert wurde. Die innerhalb eines Java-Typs dekla-

rierten Felder, Methoden und geschachtelten Java-Typen werden durch `JavaMember`-Instanzen dargestellt und über `Declares`-Kanten angebunden.

Grammatik

Die in diesem Beispiel verwendete Java-Grammatik wurde aus einer bereits bestehenden in SDF geschriebenen Grammatik erzeugt[7], die ihrerseits aus der Java-Sprachdefinition [GJSB05] hergeleitet wurde.

```
1  module java/Main
2  schema de.uni_koblenz.jgralab.demo.schema.JavaSchema
3  imports
4    ...
5  symbol tables
6      name2Package<JavaPackage>
7      name2NamedElement<JavaType, JavaMember>
8      name2Member<JavaType<--Declares,
9                  JavaMember<--Declares>:JavaType
10 default values
11     Imports.lineNumber = line(omega);
12     Defines.lineNumber = line(omega);
```

Listing 10.9: Das Modul `java/Main` (Teil 1).

Das Startmodul `java/Main` wurde aufgrund seiner Größe in zwei Listings aufgeteilt. In Listing 10.9 wird zunächst in Zeile 2 das zu verwendende Schema deklariert. In der Symboltabellendeklaration in den Zeilen 5 bis 8 werden drei Symboltabellen definiert:

1. `name2Package` enthält alle Knoten vom Typ `JavaPackage`. Sie werden für die Paket-Deklarationen und Importe der .java-Dateien benötigt.
2. `name2NamedElement` enthält alle Knoten vom Typ `JavaType` und `JavaMember`. Sie werden benötigt, um einen Knoten zu erhalten, der für den importierten Java-Typ, das importierte Feld oder die importierte Methode steht.
3. `name2Member` enthält die Bestandteile eines Java-Typs. Für jeden erkannten Typ wird eine neue Map auf den Symboltabellen-Stack gelegt, deren Namensraum der für diesen Typ erzeugte Knoten ist. Die in diese Map eingefügten `JavaType`- und `JavaMember`-Knoten werden mittels `Declares`-Kanten mit dem den Namensraum darstellenden `JavaType`-Knoten verbunden.

[7] Die verwendete Grammatik ist unter `https://svn.strategoxt.org/repos/StrategoXT/java-front/trunk/syntax/src/languages/java-15` zu finden.

Mithilfe der Default-Werte-Sektion in den Zeilen 10 bis 13 wird für jede neu erzeugte Kante *e* vom Typ `Imports` oder `Defines` das Attribut `lineNumber` auf die Nummer der Anfangszeile des Lexems gesetzt, welches bei der Regelanwendung erkannt wurde, bei der der jeweilige Omega-Knoten von *e* erzeugt worden ist.

```
import declarations
    de.uni_koblenz.edl.parser.Position;
    ...
user code {
    private Vertex defaultPackage;
    {
        defaultPackage=createVertex("JavaPackage", new Position());
        defaultPackage.setAttribute("simpleName","");
        defaultPackage.setAttribute("qualifiedName","");
    }

    private boolean isPackageDeclaration = false;

    private String packagePrefix;

    private int statements = 0;

    @SuppressWarnings("unchecked")
    public void blessAllTemporaryVerticesToJavaTypes(){
        ...
    }
}
exports
  context-free start-symbols CompilationUnit
```

Listing 10.10: Das Modul `java/Main` (Teil 2).

Listing 10.10 zeigt den zweiten Teil des `java/Main`-Moduls. Dieser besteht hauptsächlich aus dem für die semantischen Aktionen benötigten nutzerspezifischen Java-Code. In den Zeilen 19 bis 21 wird ein `JavaPackage`-Knoten erzeugt, der das default-Paket repräsentiert. Es wird zur späteren Verwendung durch das Feld `defaultPackage` referenziert.

Da zur Erkennung des Paket-Präfixes und der Importe von allen Typen eines Pakets die gleiche Regel aufgerufen wird, jedoch nur im ersten Fall die gesamte Paketstruktur durch `JavaPackage`-Knoten und `ContainsPackage`-Kanten abgebildet werden soll, wird die Variable `isPackageDeclaration` in Zeile 24 benötigt. Nur im Falle eines Paket-Präfixes hat sie den Wert `true`. Mithilfe des Felds `packagePrefix` in Zeile 26 wird der

Paket-Präfix gespeichert, damit er bei der Erkennung der in einer .java-Datei definierten Java-Typen zur Verfügung steht. Die globale Variable `statements` (Zeile 28) zählt die in einer .java-Datei vorkommenden Anweisungen.

Nachdem alle Eingabedateien geparst worden sind, können sich in den beiden Symboltabellen `name2Package` und `name2NamedElement` temporäre Knoten befinden. Dies ist der Fall, falls beispielsweise ein Java-Typ importiert worden ist, der in keiner geparsten Datei definiert worden ist. Nach Abschluss des Parsens, können durch Aufruf der Methode `blessAllTemporaryVerticesToJavaTypes()` (Zeile 30 bis 33) alle temporären Knoten der Symboltabellen `name2NamedElement` und `name2Package` in Instanzen vom Typ `JavaType` bzw. `JavaPackage` umgewandelt werden.

In den Zeilen 13 bis 15 werden die für den nutzerspezifischen Java-Code benötigten Importe deklariert. Zum Abschluss wird in Zeile 36 `CompilationUnit` als das Symbol festgelegt, von dem aus die Erkennung der Eingabe gestartet wird.

```
1  rule
2    #{statements = 0;}
3     $ = SourceFile();
4     $.name = file();
5     $packageDecl = {return defaultPackage;};
6     {packagePrefix = "";}
7    #
8    (
9      #{isPackageDeclaration = true;}#
10     PackageDec
11     #{isPackageDeclaration = false;}
12      $packageDecl = $0;
13      {packagePrefix = #$0.qualifiedName#.toString();}
14     #
15   )?
16   ImportDec
17     #Imports($,$0);#
18   *
19   TypeDec
20     #Defines($,$0);
21      ContainsType($packageDecl, $0);
22     #
23   * -> CompilationUnit
24   #$.lines = {return currentElement.getLastLine();};
25    $.numberOfStmts = {return statements;};
26   #
```

Listing 10.11: Die `CompilationUnit`-Regel.

Durch die Anwendung der `CompilationUnit`-Regel in Listing 10.11 beginnt die Erkennung einer .java-Datei. In Zeile 2 wird zunächst das Feld `statements` auf 0 gesetzt. Im Anschluss wird ein neuer `SourceFile`-Knoten erzeugt (Zeile 3), dessen `name`-Attribut auf den Namen der aktuell geparsten Datei gesetzt wird (Zeile 4).

Befindet sich eine .java-Datei im default-Paket, so besitzt sie kein Paket-Präfix. Aus diesem Grund wird in den Zeilen 5 und 6 die temporäre Variable `$packageDecl` auf den das default-Paket repräsentierenden Knoten sowie das Feld `packagePrefix` auf den leeren String gesetzt. Sollte es ein Paket-Präfix geben, so wird für die Anwendung der `PackageDec`-Regel `isPackageDeclaration` auf `true` gesetzt, damit die Paketstruktur im Graphen persistiert wird (siehe Listing 10.12). Der zurückgelieferte `JavaPackage`-Knoten wird in Zeile 12 an die temporäre Variable `$packageDecl` übergeben und das Feld `packagePrefix` bekommt den qualifizierten Namen zugewiesen.

Jeder Knoten, der den durch die Anwendung der `ImportDec`-Regel erkannten Import repräsentiert, wird in Zeile 17 durch eine `Imports`-Kante mit der aktuellen `SourceFile`-Instanz verbunden. Mithilfe der syntaktischen Variable `TypeDec` in Zeile 19 werden die in der .java-Datei enthaltenen Java-Typen geparst. Der dabei erzeugte `JavaType`-Knoten wird über eine `Defines`-Kante mit der aktuellen `SourceFile`- und über eine `ContainsType`-Kante mit der aktuellen `JavaPackage`-Instanz verbunden (Zeilen 20 bis 22).

Nachdem die Eingabedatei vollständig erkannt worden ist, wird das `lines`-Attribut des aktuellen `SourceFile`-Knotens auf die Nummer der letzten erkannten Zeile (Zeile 24) und das `numberOfStmts`-Attribut auf die im Feld `statements` gespeicherte Anzahl an erkannten Anweisungen gesetzt (Zeile 25).

In Listing 10.12 wird der Name eines Pakets als eine durch Punkte separierte nicht-leere Folge von Bezeichnern definiert. Die in Zeile 12 definierte temporäre Variable `$qualName` enthält den qualifizierten Namen. Sie wird mit dem leeren String initialisiert. Die den das Elternpaket repräsentierenden `JavaPackage`-Knoten enthaltende temporäre Variable `$parentPackage` bekommt in Zeile 13 den das default-Paket darstellenden Knoten zugewiesen.

Für jeden erkannten Bezeichner wird im Falle eines Paket-Präfixes der qualifizierte Name um das erkannte Lexem erweitert (Zeile 19). Im Anschluss wird ein neuer `JavaPackage`-Knoten *v* erzeugt und seine Attribute `simpleName` sowie `qualifiedName` auf das durch `Id` erkannte Lexem bzw. auf den qualifizierten Namen gesetzt (Zeile 20 bis 22). In den

```
...
user code {
    private boolean isContainsPackageAlreadyCreated(
                        Vertex parentPackage, Vertex pack) {
        ...
    }
}
exports
  sorts PackageName
  context-free syntax
    rule
      #$qualName="";
       $parentPackage={return defaultPackage;};
      #
      {
        Id
        #{
          if (isPackageDeclaration) {
            #$qualName=$qualName.concat(lexem($0));
             $package=JavaPackage();
             $package.simpleName=lexem($0);
             $package.qualifiedName=$qualName;
             $package=name2Package
                 .useOrDeclare($qualName, $package);
            #
            if (!isContainsPackageAlreadyCreated(
                (Vertex) #$parentPackage#, (Vertex) #$package#)){
              #ContainsPackage($parentPackage, $package);#
            }
            #$parentPackage=$package;#
          }
        }#
        "." #$qualName=$qualName.concat(".");#
      }+ -> PackageName
    ...
```

Listing 10.12: Das Modul `java/names/Main`.

Zeilen 23 bis 24 wird in der Symboltabelle kontrolliert, ob es bereits eine `JavaPackage`-Instanz gibt, die unter dem gleichen qualifizierten Namen registriert wurde. Sollte dem so sein, so wird v gelöscht und der bereits existierende genommen. Sollte unter dem qualifizierten Namen ein temporärer Knoten registriert sein, so wird dieser mit v verschmolzen. In dem verbliebenen Fall, dass noch kein Knoten unter dem qualifizierten Namen registriert worden ist, wird v in die Symboltabelle eingefügt.

Durch den Aufruf der Methode `isContainsPackageAlreadyCreated()` in Zeile 26 und 27 wird überprüft, ob v bereits das durch `$parentPackage` referenzierte Elternpaket besitzt. Sollte dies nicht der Fall sein, so wird v durch eine neue `ContainsPackage`-Kante mit dem aktuellen Elternpaket verbunden. In Zeile 30 wird `$parentPackage` auf v gesetzt.

Für jeden erkannten Punkt wird in Zeile 33 der qualifizierte Name um einen Punkt erweitert.

```
1  module java/packages/ImportDeclarations
2  imports ...
3  global actions
4    pattern "import" _(1) _(0..2) ";" -> ImportDec
5      #$import = lexem($1).replaceAll("\\s+", "");#
6    ...
7  exports
8    sorts ImportDec
9
10   context-free syntax
11     rule "import" TypeName ";" -> ImportDec
12       #$ = name2NamedElement.use($import);
13        {
14           if (((Vertex) #$#).isTemporary()) {
15             #$.qualifiedName = $import;
16              $.kind = TypeKind.UNKNOWN;
17             #
18           }
19        }
20       #
21     rule "import" PackageName "." "*" ";" -> ImportDec
22       #$ = name2Package.use($import);
23        {
24           if (((Vertex) #$#).isTemporary()) {
25             #$.qualifiedName = $import;#
26           }
27        }
28       #
29     ...
```

Listing 10.13: Das Modul `java/packages/ImportDeclarations`.

Listing 10.13 definiert die Syntax von den Importen eines einzelnen Java-Typs (Zeile 11 bis 20) und aller Typen in einem Paket (Zeile 21 bis 28). Für beide Regeln wird durch die globale Aktion in Zeile 5 die temporäre Variable `$import` auf den qualifizierten Namen des importierten Elements gesetzt.

Im Falle eines einzelnen importierten Java-Typs wird zunächst der unter dem qualifizierten Namen `$import` in der Symboltabelle `name2NamedElement` registrierte Knoten zurückgegeben (Zeile 12). Sollte kein solches Element existieren, so wird beim Aufruf von `use()` automatisch ein neuer temporärer Knoten v erzeugt und in die Symboltabelle eingefügt. Die Attribute `qualifiedName` und `kind` von v werden dabei auf den erkannten qualifizierten Namen bzw. `UNKNOWN` gesetzt.

Für den Import aller Typen eines Pakets wird in den Zeilen 21 bis 28 ähnlich verfahren. Es wird dabei allerdings die Symboltabelle `name2Package` genutzt und da der Knoten `JavaPackage` kein Attribut mit Namen `kind` besitzt, wird nur `qualifiedName` gesetzt.

```
1  @Symboltable{name2Member}
2  rule
3    #$ = JavaType();
4     $.kind = TypeKind.INTERFACE;
5     name2Member.namespace = $;
6    #
7    InterfaceDecHead
8    #$.simpleName = $0;
9     $packagePrefix = {return packagePrefix;};
10    {packagePrefix += (packagePrefix.isEmpty()? "" : ".")
11                     + #$.simpleName#;}
12    $.qualifiedName = {return packagePrefix;};
13    name2NamedElement.declare($.qualifiedName, $);
14   #
15   "{" InterfaceMemberDec* "}" -> InterfaceDec
16   #{packagePrefix = #$packagePrefix#.toString();}#
```

Listing 10.14: Die `InterfaceDec`-Regel.

Da für die Erkennung der verschiedenen Java-Typen analoge semantische Aktionen ausgeführt werden, wird nur die Behandlung der Interfaces durch die in Listing 10.14 gezeigte `InterfaceDec`-Regel dargestellt. Für jede Anwendung dieser Regel wird eine neue Map auf den Symboltabellen-Stack `name2Member` erzeugt. In Zeile 3 wird eine neue `JavaType`-Instanz erzeugt und ihr Attribut `kind` auf `INTERFACE` gesetzt (Zeile 4).

Im Anschluss wird sie als Namespace-Repräsentant der aktuellen Map auf `name2Member` festgelegt (Zeile 5).

Der durch Anwendung der `InterfaceDecHead`-Regel in Zeile 7 zurückgegebene Bezeichner des Interfaces wird in Zeile 8 dem Attribut `simpleName` zugewiesen. Das bisherige Paket-Präfix wird in der temporären Variable `$packagePrefix` gespeichert, damit es in den Zeilen 10 und 11 um den Namen des Interfaces erweitert werden kann. Der so gebildete qualifizierte Name wird im Attribut `qualifiedName` persistiert (Zeile 12). Der aktuelle `JavaType`-Knoten wird in Zeile 13 in der Symboltabelle `name2NamedElement` unter seinem qualifizierten Namen registriert. Sollte sich in ihr aufgrund eines Imports bereits ein temporärer Knoten befinden, so wird dieser mit dem aktuellen verschmolzen.

Die durch `InterfaceMemberDec*` erkannten Bestandteile des Interfaces, nutzen ihrerseits den durch das Feld `packagePrefix` referenzierten Wert, um ihre qualifizierten Namen zu bestimmen. Die erzeugten Knoten werden in die Symboltabelle `name2Member` eingefügt und dabei automatisch durch `Declares`-Kanten mit der das aktuelle Interface repräsentierenden `JavaType`-Instanz verbunden. Am Ende der Anwendung der `InterfaceDec`-Regel wird in Zeile 16 das Feld `packagePrefix` auf den vor der Anwendung bestehenden Wert zurückgesetzt.

```
1  module java/statements/Statements
2  imports ...
3  global actions
4    pattern _(*) -> Stmt #{statements++;}#
5
6  exports
7    sorts Stmt
8
9    context-free syntax
10     rule Block       -> Stmt
11     ...
```

Listing 10.15: Das Modul `java/statements/Statements`.

Die Syntax der in Java möglichen Anweisungen sind in den Modulen mit dem Namenspräfix `java/statements/` definiert. Die Bestimmung der Anzahl der in einer .java-Datei enthaltenen Anweisungen wird in dem in Listing 10.15 gezeigten Ausschnitt des `java/statements/Statements`-Moduls exemplarisch gezeigt. Da jede Regel in diesem Modul für ein Statement steht und alle den Kopf `Stmt` haben, genügt die globale Aktion in Zeile 4, durch die das Feld `statements` inkrementiert wird.

Anwendung

Um mithilfe der soeben beschriebenen Grammatik .java-Datein parsen zu können, muss zunächst ein `JavaGraphBuilder` erzeugt werden. Dies geschieht analog zu dem im Abschnitt 10.1 beschriebenen XML-Beispiel durch den Befehl:

```
java de.uni_koblenz.edl.preprocessor.EDLPreprocessor
   -i ./examples -m java/Main -o ./testoutput -p testoutput
```

```
1 Schema schema = GraphBuilderBaseImpl
2      .instantiateSchema(GraphBuilderBaseImpl
3           .loadSchema("./javaschema.tg"));
4 JavaGraphBuilder graphBuilder = new JavaGraphBuilder(schema);
5 File baseDir = new File(".");
6 parseAllFiles(graphBuilder, baseDir);
7 graphBuilder.blessAllTemporaryVerticesToJavaTypes();
8 graphBuilder.getGraph().save("Output.tg");
```

Listing 10.16: Der Aufruf des `JavaGraphBuilders`.

Listing 10.16 demonstriert, wie der erzeugte `JavaGraphBuilder` genutzt werden kann, um alle .java-Dateien eines Verzeichnisses in einen einzigen `JavaGraphen` zu überführen. Zunächst wird in den Zeilen 1 bis 3 eine Instanz des Java-Schemas erzeugt, die in Zeile 4 dem neu erzeugten `JavaGraphBuilder`-Objekt übergeben wird. Durch diesen Konstruktoraufruf wird ein neuer leerer Graph erzeugt und die definierten Symboltabellen erstellt.

Durch den Aufruf der `parseAllFiles()`-Methode in Zeile 6 wird das aktuelle Verzeichnis per Tiefensuche durchsucht. Jede gefunden .java-Datei wird durch Aufruf von

```
graphBuilder.parse(pathToJavaFile);
```

geparst. Da jedes Mal die `parse()`-Methoden desselben `JavaGraphBuilder`-Objekts aufgerufen wird, werden die Symboltabellen und der Graph des letzten Parse-Vorgangs wiederverwendet.

Nachdem alle .java-Dateien geparst worden sind, soll der erzeugte Graph in Zeile 8 gespeichert werden. Da bei diesem Vorgang keine temporären Knoten existieren dürfen, wird in Zeile 7 die Methode `blessAllTemporaryVerticesToJavaTypes()` aufgerufen, die sie in `JavaType`- bzw. `JavaPackage`-Instanzen umwandelt.

11 Zeitmessung

Um den praktischen Einsatz von EDL beurteilen zu können, ist es zum einen notwendig zu wissen, wie viel Zeit zum Parsen benötigt wird. Zum anderen ist die Information wichtig, wie stark die benötigte Zeit im Verhältnis zur Eingabegröße ansteigt. In Abschnitt 11.1 wird zunächst erläutert welche Zeiten gemessen und wie die Messung durchgeführt wird. Da der Aufwand des zum Parsen verwendeten GLR-Algorithmus davon abhängt, wie viele Mehrdeutigkeiten in der Grammatik vorkommen, werden die Zeiten für die beiden bereits in Kapitel 10 vorgestellten Grammatiken gemessen: Die gemessenen Zeiten für die XML-Grammatik werden in Abschnitt 11.2 und die für die Java-Grammatik in Abschnitt 11.3 vorgestellt. Schließlich werden in Abschnitt 11.4 die Ergebnisse der Zeitmessung zusammengefasst.

11.1 Messverfahren

Jeder *Extraktionsvorgang mithilfe eines `GraphBuilders`*, wie er in Abbildung 11.1 dargestellt ist, besteht zunächst aus dem Parsen der Eingabe. Dabei wird ein interner Parse-Forest erzeugt. Im Anschluss wird dieser traversiert, wobei die in der dem genutzten `GraphBuilder` zugrundeliegenden Grammatik definierten semantischen Aktionen ausgeführt werden, um den gewünschten abstrakten Syntaxgraphen zu erhalten.

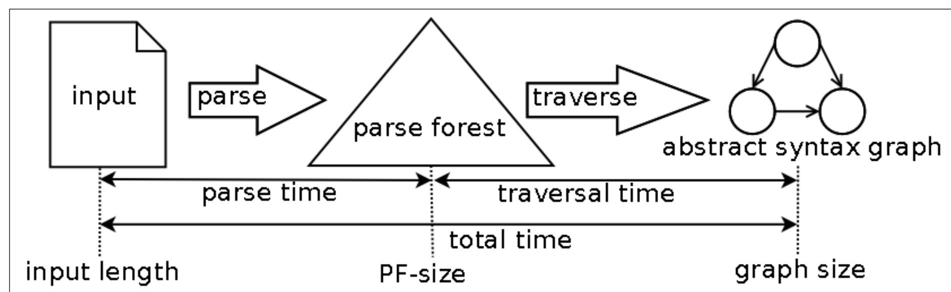


Abbildung 11.1: Der Extraktionsvorgang und die ermittelten Größen.

Anhand des Ablaufs eines Extraktionsvorgangs lassen sich die folgenden drei Zeiten bestimmen:

1. **Parse-Zeit** (in msec): Diese Zeit gibt an, wie lange der in „Stratego/XT“ enthaltene Parser braucht, um aus der Eingabe den internen Parse-Forest aufzubauen.
2. **Traversierungszeit** (in msec): Die Traversierungszeit gibt an, wie lange es dauert, den internen Parse-Forest zu traversieren und die jeweiligen semantischen Aktionen auszuführen.
3. **Gesamtzeit** (in msec): Diese Zeit ist die Summe aus der Parse- und Traversierungszeit.

Um aussagekräftige Ergebnisse zu erhalten, wird jeder Extraktionsvorgang *30-mal* wiederholt und für jeden einzelnen dieser Vorgänge werden die drei Zeiten bestimmt. Im Anschluss wird für jede dieser drei Zeiten die fünf besten und die fünf schlechtesten Werte entfernt und das *arithmetische Mittel über die verbliebenen Werte* berechnet. Durch dieses Verfahren kann es allerdings vorkommen, dass die durchschnittliche Gesamtzeit nicht mehr der Summe aus der durchschnittlichen Parse- und Traversierungszeit entspricht.

Für die gemessenen Zeiten sind die folgenden drei *Bezugsgrößen* bestimmt worden:

1. **Eingabelänge**: Diese Größe gibt die Anzahl der Eingabezeichen an.
2. **Parse-Forest-Größe**: Diese Größe gibt die Anzahl der Knoten im internen Parse-Forest an.
3. **Graphgröße**: Diese Größe ist die Summe der Knoten- und Kantenzahl des erzeugten abstrakten Syntaxgraphen.

11.2 Zeitmessung mithilfe einer XML-Grammatik

Die in Abschnitt 10.1 vorgestellte Grammatik beschreibt die Syntax von XML. Dabei werden alle XML-Elemente, Attribute und die zwischen öffnenden und schließenden Tags enthaltene Texte durch Knoten repräsentiert. Die Schachtelungsstruktur wird durch Kanten ausgedrückt. Als Eingabe werden willkürliche XML-Dateien aus einem realen Kontext genutzt. So repräsentieren beispielsweise eine .xmi-Datei ein in grUML notiertes TGraph-Schema.

Anhand der in der Tabelle 11.1 dargestellten Daten ist zu erkennen, dass bei der verwendeten XML-Grammatik die Größe des internen Parse-Forests das ungefähr *4,5- bis 5-fache* der Eingabelänge beträgt. Dieser Faktor hängt u.a. davon ab, aus wie vielen Elemente die Rümpfe der in der Grammatik definierten Regeln bestehen. Wie die letzten beiden Fälle zeigen, kann der für eine kleinere XML-Datei erzeugte Graph größer ausfallen als der

Eingabedatei	**Eingabelänge**	**PF-Größe**	**Graphgröße**	**∅ Parse-Zeit** (in msec)	**∅ Trav.-Zeit** (in msec)	**∅ Gesamtzeit** (in msec)
CommentTest.xmi	8.896	40.884	481	162	33	195
javascannerschema.xmi	10.369	50.529	693	158	27	187
vwa-db.xmi	14.768	69.490	855	161	31	195
common.xml	18.317	81.805	1.219	294	50	347
jgralab-logo.svg	18.939	80.924	634	269	41	313
OsmSchema.xmi	22.254	105.757	1.416	244	49	298
greqltestschema.xmi	23.075	109.969	1.416	307	64	375
javascannerschema-sketch.emx	36.100	170.177	2.404	589	98	693
grUML-M3.xmi	38.825	186.017	2.615	620	127	737
javascannerschema.emx	45.870	216.070	3.050	693	120	811
greql-evaluator.emx	145.930	690.477	10.094	2.226	513	2.757
EDLSchema.xmi	178.619	869.135	11.971	2.818	593	3.445
map1.osm	181.051	874.646	23.871	3.235	776	4.013
map2.osm	802.006	3.909.373	112.179	14.919	5.039	19.939
EDLSchema.emx	1.193.891	5.612.676	81.828	18.918	5.905	24.989

Tabelle 11.1: Die gemessenen Zeiten unter Verwendung des aus der XML-Grammatik generierten `XMLGraphBuilders`.

aus einer größeren XML-Datei erzeugten. Dies kommt vor, wenn beispielsweise mehr Whitespace-Zeichen, längere Bezeichner oder mehr Zeichen in Attributwerten bzw. zwischen zwei XML-Tags stehen.

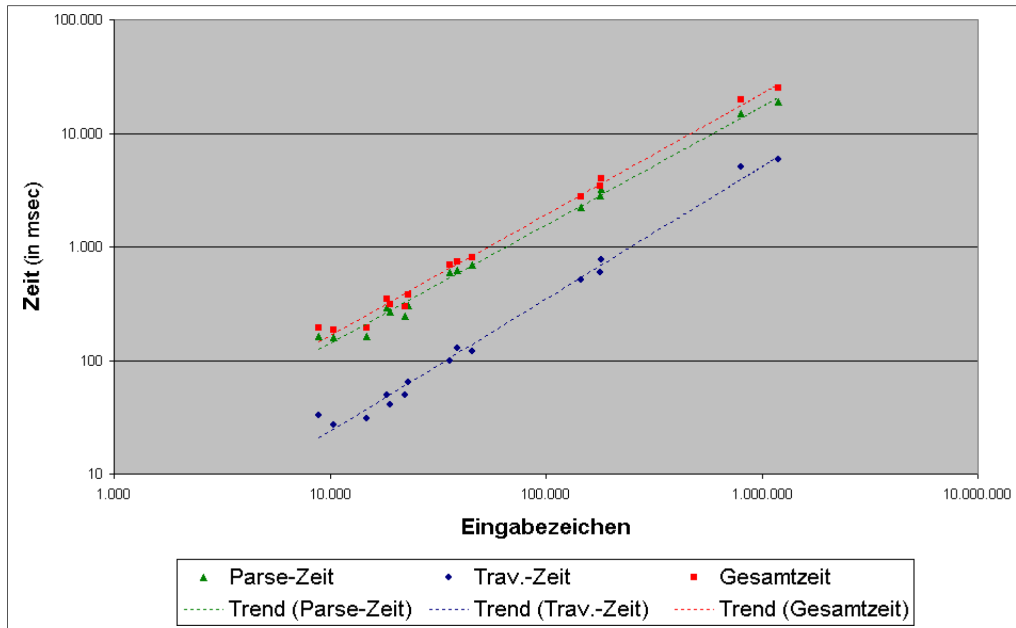


Abbildung 11.2: Die gemessenen Zeiten in Abhängigkeit zur Eingabegröße.

Um die gemessenen Zeiten besser miteinander vergleichen zu können, sind sie in Abbildung 11.2 mittels eines *Punktdiagramms* in Abhängigkeit zur Eingabegröße dargestellt. Um die Übersichtlichkeit zu erhöhen, wurde für beide Achsen eine logarithmische Skalierung verwendet. Die eingezeichneten Trendlinien legen für alle drei gemessenen Zeiten eine *lineare Steigung* nahe und damit einen linearen Zeitaufwand.

11.3 Zeitmessung mithilfe einer Java-Grammatik

Als zweites Beispiel wird der aus der in Abschnitt 10.2 vorgestellten Java-Grammatik generierte `JavaGraphBuilder` zur Zeitmessung genutzt. Diese Grammatik beschreibt die vollständige Java-Syntax. Der extrahierte Graph repräsentiert *alle geparsten .java-Dateien eines realen Projekts* sowie die jeweilige Paketstruktur. Eine einzelne Datei wird durch die definierten und importierten Java-Typen sowie die definierten Felder und Methoden dargestellt. Die in der Tabelle 11.2 dargestellten Werte der Eingabelänge, Parse-Forest-Größe sowie die drei Zeiten sind die *Summen* der für die einzelnen .java-Dateien gemessenen Werte.

Projektname	Dateianzahl	Eingabelänge	PF-Größe	Graphgröße	∅ Parse-Zeit (in msec)	∅ Trav.-Zeit (in msec)	∅ Gesamtzeit (in msec)
javascannerschema	30	125.091	421.560	1.055	1.218	247	1.472
jgstreetmap	23	140.523	491.627	1.416	1.800	289	2.083
rsleditor	105	761.878	2.510.519	5.437	7.713	1.558	9.251
grabaja	181	893.423	2.905.149	4.544	7.636	1.763	9.359
jgralab	2.155	15.409.783	51.967.189	89.497	150.601	28.777	179.490
edl	635	18.474.986	61.926.254	65.724	183.782	32.808	216.569

Tabelle 11.2: Die gemessenen Zeiten unter Verwendung des aus der Java-Grammatik generierten `JavaGraphBuilders`.

Das Verhältnis zwischen der Eingabelänge und der Größe des internen Parse-Forests liegt bei ungefähr 3 bis 3,5. Die Größe des extrahierten Graphen ist nicht nur von der Länge der Eingabe abhängig, sondern auch von der Anzahl der geparsten .java-Dateien sowie der Zahl der importierten Klassen und der deklarierten Felder bzw. Methoden.

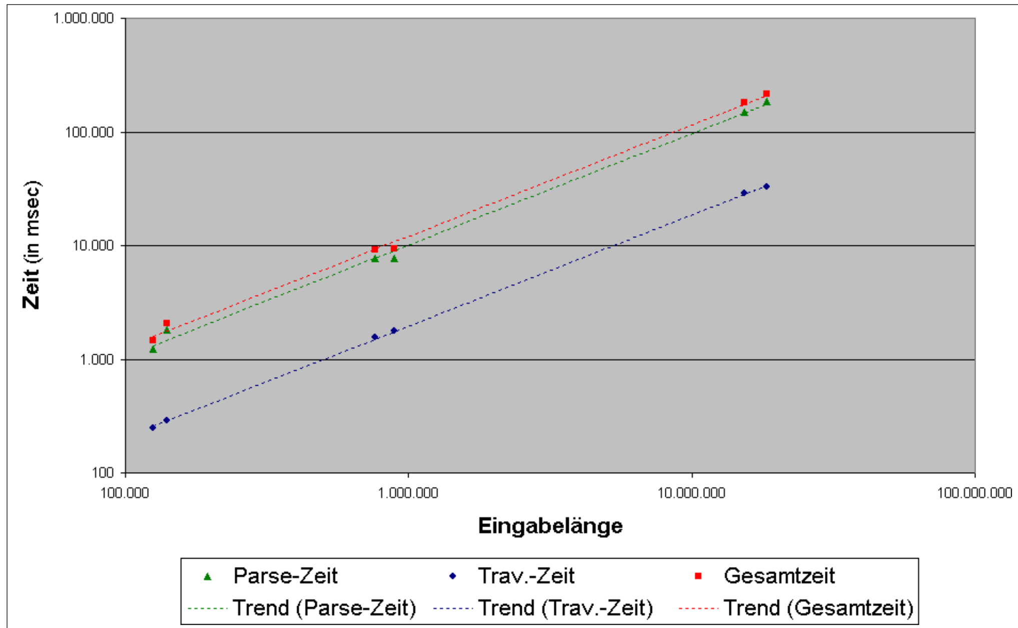


Abbildung 11.3: Die gemessenen Zeiten in Abhängigkeit zur Eingabegröße.

Um die gemessenen Zeiten besser miteinander vergleichen zu können, sind sie in Abbildung 11.3 als Punktdiagramm dargestellt. Zur besseren Übersichtlichkeit wurde für beide Achsen eine logarithmische Skalierung verwendet. Wie anhand der eingezeichneten Trendlinien zu erkennen, liegt auch in diesem Fall anscheinend eine *lineare Abhängigkeit* der benötigten Zeit zur Eingabelänge und somit ein linearer Zeitaufwand vor.

11.4 Zusammenfassung der Ergebnisse

Die anhand der beiden Beispielgrammatiken ermittelten Werte legen eine lineare Abhängigkeit zwischen der Eingabelänge und der zur Extraktion benötigten Zeit nahe. Um aussagekräftigere Ergebnisse zu erhalten, kann im Anschluss an diese Arbeit die Zeit gemessen werden, die die Extraktion eines vollständigen abstrakten Syntaxgraphen für Java benötigen würde. Darüber hinaus könnte untersucht werden, wie sich die semantischen Aktionen und die verwendeten Regeln auf die gemessenen Zeiten auswirken.

Für die Extraktion des Java-Graphen, wie er in Abschnitt 10.2 vorgestellt wurde, sind nur das Paket-Präfix, die Importe sowie die deklarierten Javatypen, Felder und Methoden relevant. Anstatt eine vollständige Java-Grammatik zu nutzen, kann untersucht werden, ob die Verwendung einer nur die benötigten Syntaxabschnitte beschreibende Inselgrammatik eine Zeitersparnis bringt.

12 Zusammenfassung und Ausblick

Im Rahmen dieser Arbeit wurde eine Metasprache namens „Extractor Description Language“ (EDL) entwickelt, mit deren Hilfe die *Syntax* der Eingabe und die *semantischen Aktionen* beschrieben werden können, um einen abstrakten Syntaxgraphen aus einer oder mehreren Eingabedateien zu extrahieren. Im Folgenden werden die Fähigkeiten der EDL zusammengefasst (Abschnitt 12.1), ein Ausblick auf mögliche Weiterentwicklung gegeben (Abschnitt 12.2) und abschließend ein kurzes Fazit gezogen (Abschnitt 12.3).

12.1 Zusammenfassung

TGraphen eignen sich als Datenstruktur für abstrakte Syntaxgraphen (ASG), da sie eine effiziente Analyse und Verarbeitung der repräsentierten Daten zulassen. Mithilfe der in dieser Arbeit entwickelte „Extractor Description Language“ (EDL) ist die Extraktion eines ASG aus einer Eingabe vereinfacht worden.

Dies wird unter anderem durch die Nutzung eines GLR-Parsers erreicht, der mit mehrdeutigen Grammatiken umgehen kann, wodurch die Regeln zur Syntax-Beschreibung für den Menschen besser verständlich ausfallen können. Durch die Notation von Assoziativitäten und Prioritäten können diese Mehrdeutigkeiten aufgelöst und die eindeutige Erkennung der Eingabe erreicht werden. Mithilfe der Modularisierung ist es darüber hinaus möglich, die Regeln zur Syntax-Beschreibung thematisch zu gliedern, wodurch die Grammatik übersichtlicher wird. Da in EDL die Möglichkeit besteht, Inselgrammatiken zu definieren, genügt nun nur noch die Anfertigung der Syntax-Beschreibung für die relevanten Eingabe-Bestandteile.

Durch die Definition von semantischen Aktionen wird der Aufbau des ASG ermöglicht. Es ist dabei möglich, Aktionen sowohl für jede einzelne Regel als auch direkt für eine Menge von Regeln zu definieren, für die diese Aktionen ausgeführt werden sollen. Durch Befehle zum Erzeugen von Graphelementen und Setzen ihrer Attributwerte sowie die Nutzung von Symboltabellen-Stacks, die automatisch im Graphen persistiert werden können, wird die Erzeugung des ASG vereinfacht. Durch die Nutzung einer Java-ähnlichen Syntax ist die Verständlichkeit gegenüber einer völlig neu entworfenen

Syntax erhöht worden. Sollte der vorgegebene Befehlssatz nicht ausreichen, ist es möglich, eine Regel mit beliebigem nutzerspezifischen Java-Code als semantische Aktion zu versehen. Zur Unterstützung des Debuggens wird im Falle einer Exception angegeben, welche semantische Aktion der Grammatik diese ausgelöst hat. Um nachvollziehen zu können, welche Regelanwendung die Exception verursacht hat, können die zur Erkennung der Eingabe benötigten Regelanwendungen in einem sogenannten Parse-Tree visualisiert werden.

12.2 Ausblick

Die in dieser Arbeit entwickelte „Extractor Description Language" bietet die grundlegende Funktionalität zur Extraktion von abstrakten Syntaxgraphen. Um die Nutzung von nutzerspezifischem Java-Code zu minimieren, ist die *Erweiterung der EDL-spezifischen Befehle* um arithmetische Operationen und bedingte Verzweigung sinnvoll. Darüber hinaus wäre ein syntaktisches Konstrukt zur Iteration über die Elemente einer Liste ähnlich der for-each-Schleife in Java nützlich.

Eine weitere hilfreiche Erweiterung wäre die *Verbesserung des Datenflusses* durch vererbte Attribute. Sollte es beispielsweise für eine syntaktische Variable im Regelrumpf mehrere Regeln geben, für die Kind-Knoten erzeugt werden, die mit Kanten verschieden Typs mit dem Eltern-Knoten verbunden werden müssen, ist dies zur Zeit nur über Nutzung des Verschmelzens von Knoten oder einer Fallunterscheidung mithilfe von nutzerspezifischem Code realisierbar. Darüber hinaus wäre ein verbesserter Datenfluss zwischen EDL und nutzerspezifischem Code wünschenswert. So ist es zur Zeit nur umständlich möglich, innerhalb einer EDL-Anweisung auf den Wert einer Iterations-Variablen zuzugreifen, die in einer die EDL-Anweisung umschließenden for-Schleife definiert ist.

Des Weiteren wäre eine *statische Typisierung* nützlich, da so weniger Typ-Casts im nutzerspezifischen Code nötig wären, falls dieser EDL-Ausdrücke enthält. Dies würde darüber hinaus auch eine mögliche *Optimierung der semantischen Aktionen* vereinfachen. Sollte beispielsweise durch eine Repetition wie beispielsweise bei `Element*` eine Liste von Knoten erzeugt werden, die im Anschluss durch Kanten mit einem anderen Knoten verbunden werden, wäre eine mögliche Optimierung, dass die Kantenerzeugung direkt in der Repetition vorgenommen wird, was die Iteration über die Knotenliste erübrigen würde.

Sollte die Eingabe mithilfe der definierten Grammatik nicht parsebar sein, so wird zur Zeit nur eine Exception erzeugt, die dem Nutzer das erste nicht erkennbare Zeichen und seine Position in der Eingabe mitteilt. Der verwendete GLR-Parser bietet eine Schnittstelle, um eine sogenannte *„Error Recovery"* zu realisieren. Dies bedeutet, dass der GLR-

Parser versucht einen Punkt in der Eingabe zu finden, von dem aus er ohne Fehler weiterarbeiten kann. Diese Schnittstelle kann auf die Möglichkeit hin untersucht werden, eine „Error Recovery“ für EDL zu realisieren.

Zur Nutzung von EDL wird ein TGraph-Schema benötigt. Daher könnte es hilfreich sein, aus der erstellten Grammatik solch ein TGraph-Schema automatisch zu generieren, welches dann bei Bedarf nur noch überarbeitet werden müsste.

Der aus einer in EDL geschriebenen Grammatik erzeugte Extraktor dient dem Aufbau eines abstrakten Syntaxgraphen (ASG). Um als Repräsentation des ASG nicht nur ausschließlich auf TGraphen festgelegt zu sein, wäre es wünschenswert, auch andere Technologien zu unterstützen.

12.3 Fazit

Mit der in dieser Arbeit entwickelten „Extractor Description Language“ wird die Erzeugung eines abstrakten Syntaxgraphen vereinfacht, indem das Erstellen eines Extraktors von Hand vermieden werden kann. Des weiteren ist die erstellte Grammatik aufgrund der möglichen Mehrdeutigkeit und der Modularisierung verständlicher und damit wartbarer als vergleichbare Grammatiken wie beispielsweise denen für den Parsergenerator ANTLR. Durch eine Java-ähnliche Syntax und die Notation der semantischen Aktionen direkt in den Regeln wird der Einstieg für Menschen mit Java-Erfahrung vereinfacht.

Literaturverzeichnis

[ANT12] *Why Use ANTLR?* http://www.antlr.org/why.html. Version: 19.01.2012

[BBJ+63] BACKUS, J.W. ; BAUER, F.L. ; J.GREEN ; KATZ, C. ; MCCARTHY, J. ; NAUR, P. ; PERLIS, A.J. ; RUTISHAUSER, H. ; SAMUELSON, K. ; VAUQUOIS, B. ; WEGSTEIN, J.H. ; WIJNGAARDEN, A. van ; WOODGER, M.: Revised Report on the Algorithmic Language ALGOL 60. In: *The Computer Journal* 5 (1963), Nr. 4, 349-367. http://comjnl.oxfordjournals.org/content/5/4/349.full.pdf

[Bis12] *Bison 2.5*. https://www.gnu.org/software/bison/manual/bison.html. Version: 22.01.2012

[BKV07] BRAND, Mark van d. ; KLINT, Paul ; VINJU, Jurgen: *The Syntax Definition Formalism SDF*. http://homepages.cwi.nl/~daybuild/daily-books/syntax/2-sdf/sdf.pdf. Version: 10 2007

[BNF11] *BNF (Backus-Naur-Form)*. http://www.itwissen.info/definition/lexikon/Backus-Naur-Form-BNF.html. Version: 15.11.2011

[Bor91] BORCHERS, J.: Sanftes Re-Engineering kontra Wartungskrise. In: *Online - Zeitschrift für Datenverarbeitung*, 1991, S. 48 – 51

[BPSM+06] BRAY, Tim ; PAOLI, Jean ; SPERBERG-MCQUEEN, C. M. ; MALER, Eve ; YERGEAU, François ; COWAN, John: *Extensible Markup Language (XML) 1.1 (Second Edition)*. http://www.w3.org/TR/2006/REC-xml11-20060816/. Version: September 2006

[Byr92] BYRNE, Eric J.: A conceptual foundation for software re-engineering. In: *International Conference on Software Maintenance*, 1992

[Dah95] DAHM, Peter: *PDL Eine Sprache zur Beschreibung grapherzeugender Parser*, Universität Koblenz-Landau, Diplomarbeit, 1995

[Ebe11] EBERT, Jürgen: *Vorlesung: Effiziente Graphenalgorithmen*. http://www.uni-koblenz-landau.de/koblenz/fb4/institute/IST/AGEbert/teaching/wise1112/graphenalgorithmen. Version: 2011

[EP08] ERK, Katrin ; PRIESE, Lutz: *Theoretische Informatik. Eine umfassende Einführung*. 3., erweiterte Aufl. Heidelberg : Springer, 2008 (eXamen.press). – ISBN 978–3–540–76319–2

[For02] FORD, Bryan: *Packrat Parsing: a Practical Linear-Time Algorithm with Backtracking*, Massachusetts Institute of Technology, Diplomarbeit, September 2002. http://pdos.csail.mit.edu/~baford/packrat/thesis/thesis.pdf

[GJSB05] GOSLING, James ; JOY, Bill ; STEELE, Guy ; BRACHA, Gilad: *Java™Language Specification, The (3rd Edition)*. 3. Addison-Wesley Professional, 2005 http://docs.oracle.com/javase/specs/jls/se5.0/jls3.pdf. – ISBN 0321246780

[GLR12] *Comparison of parser generators*. https://secure.wikimedia.org/wikipedia/en/wiki/Comparison_of_parser_generators#General_context-free.2C_conjunctive_or_boolean_languages. Version: 21.01.2012

[HK86] HEERING, Jan ; KLINT, Paul: A Syntax Definition Formalism. In: *ESPRIT"86: Results and Achievements*, North-Holland, 1986, S. 619–630

[Kah06] KAHLE, Steffen: *JGraLab: Konzeption, Entwurf und Implementierung einer Java-Klassenbibliothek für TGraphen*, Universität Koblenz-Landau, Campus Koblenz, Diplomarbeit, Juni 2006

[Mar06] MARCHEWKA, Katrin: *Entwurf und Definition der Graphanfragesprache GReQL 2*, Universität Koblenz-Landau, Campus Koblenz, Diplomarbeit, September 2006

[Meg06] MEGACZ, Adam: *Scannerless Boolean Parsing*. http://www.cs.berkeley.edu/~megacz/research/papers/megacz,adam-sbp.a.scannerless.boolean.parser.pdf. Version: 2006

[Met12a] *ASF+SDF*. http://www.meta-environment.org/Meta-Environment/ASF%2bSDF. Version: 22.01.2012

[Met12b] *The Future of ASF+SDF and The Meta-Environment*. http://meta-environment.blogspot.com/2010/01/future-of-asfsdf-and-meta-environment.html. Version: 22.01.2012

[Met12c] *SDF (Syntax Definition Formalism)*. http://www.meta-environment.org/Meta-Environment/SDF. Version: 22.01.2012

[PF11] PARR, Terence ; FISHER, Kathleen: LL(*): the foundation of the ANTLR parser generator. In: *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. New York, NY, USA : ACM, 2011 (PLDI '11). – ISBN 978–1–4503–0663–8, 425–436

[Ras12a] *Rascal explained for ASF+SDF programmers*. http://tutor.rascal-mpl.org/Courses/CompareWithOtherParadigms/ASFPlusSDF/ASFPlusSDF.html. Version: 22.01.2012

[Ras12b] *Welcome to Rascal*. http://www.rascal-mpl.org/. Version: 22.01.2012

[Rus09] *Russell's Paradox*. http://plato.stanford.edu/entries/russell-paradox/. Version: 2009

[SJ10] SCOTT, Elizabeth ; JOHNSTONE, Adrian: GLL Parsing. In: *Electronic Notes in Theoretical Computer Science* 253 (2010), Nr. 7, 177 - 189. http://dx.doi.org/10.1016/j.entcs.2010.08.041. – DOI 10.1016/j.entcs.2010.08.041. – ISSN 1571–0661. – <ce:title>Proceedings of the Ninth Workshop on Language Descriptions Tools and Applications (LDTA 2009)</ce:title>

[SLA08] SETHI, Ravi ; LAM, Monica S. ; AHO, Alfred V.: *Compiler. Prinzipien, Techniken und Tools (Pearson Studium): Prinzipien, Techniken und Werkzeuge*. 2. Auflage. PEARSON STUDIUM, 2008 http://www.worldcat.org/isbn/3827370973. – ISBN 3827370973

[Spi12] SPIEWAK, Daniel: *gll-combinators*. https://github.com/djspiewak/gll-combinators. Version: January 2012

[Sta96] STANDARD, E. S. S. ; EBNF (Hrsg.): Information Technology - Syntactic Metalanguage - Extended BNF / International Organisation for Standardisation. Version: 12 1996. http://standards.iso.org/ittf/PubliclyAvailableStandards/s026153_ISO_IEC_14977_1996(E).zip. 1996 (14977). – Forschungsbericht

[str12a] *Stratego/XT Manual*. http://releases.strategoxt.org/strategoxt-manual/unstable/manual/one-page/index.html. Version: 05.02.2012

[Str12b] *Stratego/XT*. http://strategoxt.org/. Version: 22.01.2012

[Tom86] TOMITA, Masaru: *Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems*. Norwell, MA, USA : Kluwer Academic Publishers, 1986. – ISBN 0–89838–202–5

[ult12] *Parser generator (UltraGram)*. `http://www.ust-solutions.com/ultragram.aspx`. Version: 22.01.2012

[Vis97] VISSER, Eelco: *Syntax definition for language prototyping*. Ponsen & Looijen, 1997. – I–VIII, 1–383 S. – ISBN 978–90–74795–75–3

Anhang

Anhang A

EDL-Schema

In diesem Kapitel wird das gesamte EDL-Schema nach Paketen unterteilt gezeigt. Knotenklassen, die EDL-spezifischen Code repräsentieren, sind gelb eingefärbt. Die Navigationsrichtung an den Assoziationen sind im Rahmen dieses Schemas als Leserichtung zu verstehen. Der Supertyp t einer Kantenklasse e wird durch die Notation $e : t$ ausgedrückt. Fehlt $: t$, so hat die Kantenklasse keinen im Schema explizit modellierten Supertyp.

common

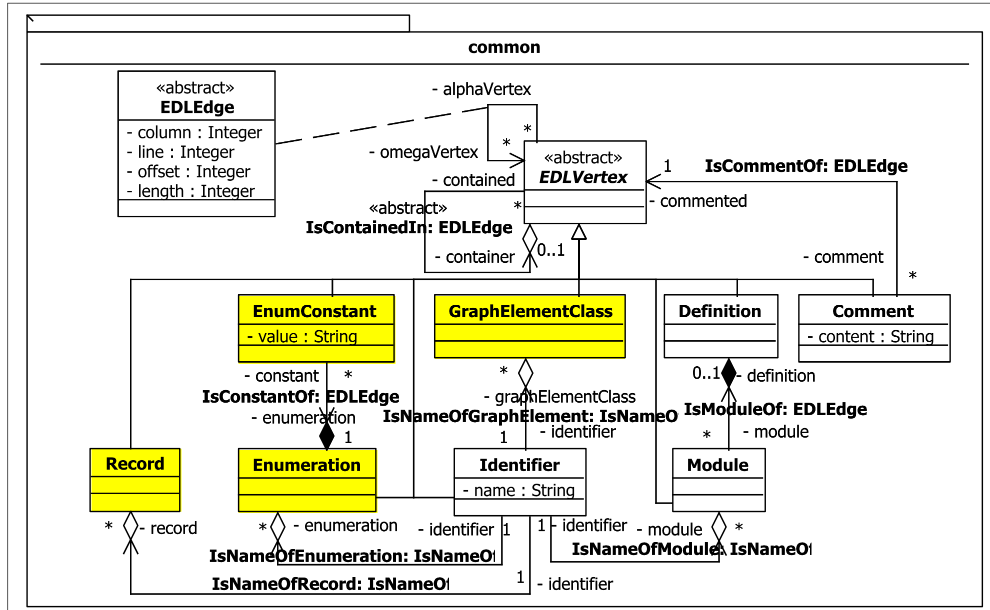


Abbildung A.1: Der Inhalt des `common`-Pakets (Teil 1).

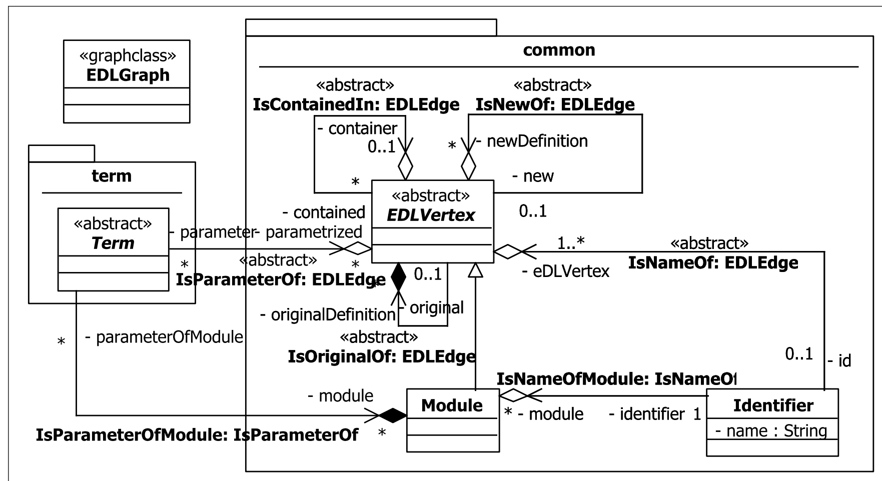


Abbildung A.2: Der Inhalt des `common`-Pakets (Teil 2).

`section`

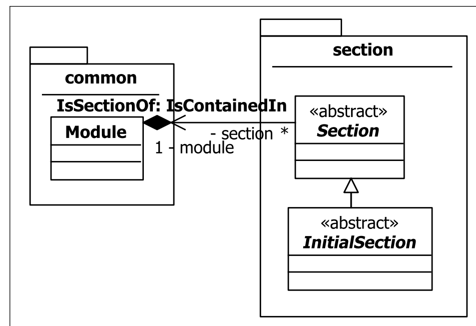


Abbildung A.3: Die beiden `Section`-Kategorien.

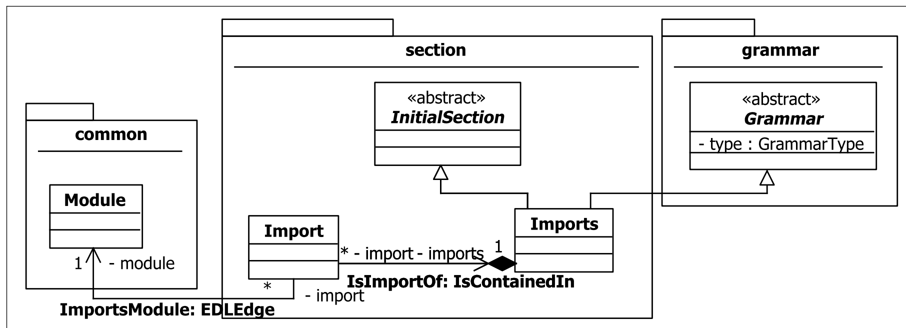


Abbildung A.4: Die `Imports`-Sektion.

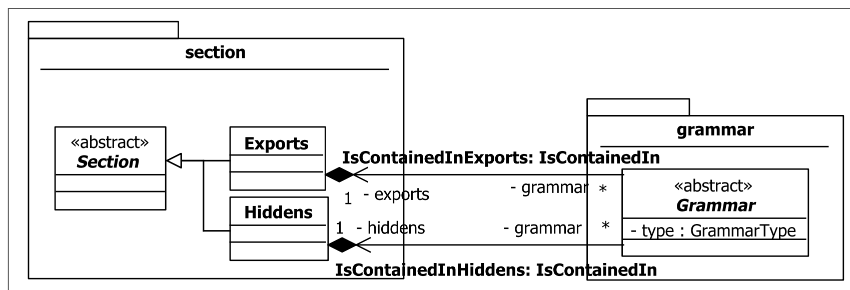


Abbildung A.5: Die `Exports`- und `Hiddens`-Sektion.

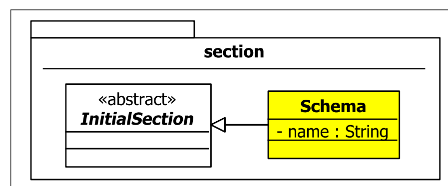


Abbildung A.6: Die `Schema`-Sektion.

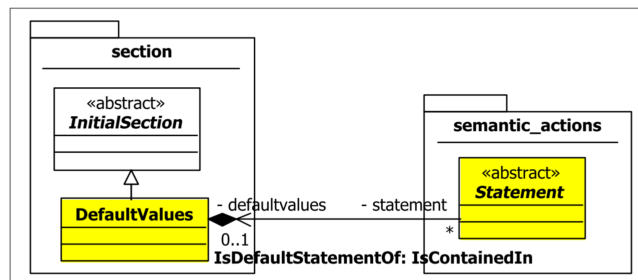


Abbildung A.7: Die `DefaultValues`-Sektion.

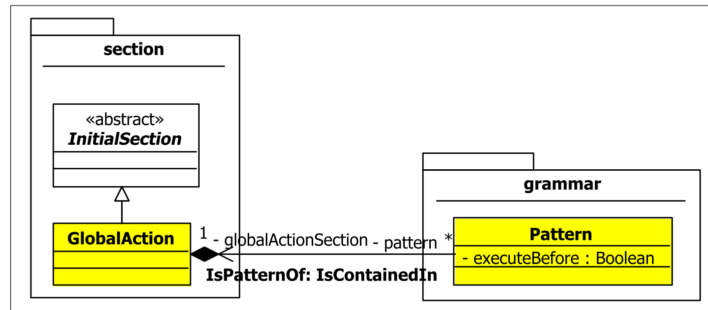


Abbildung A.8: Die `GlobalAction`-Sektion.

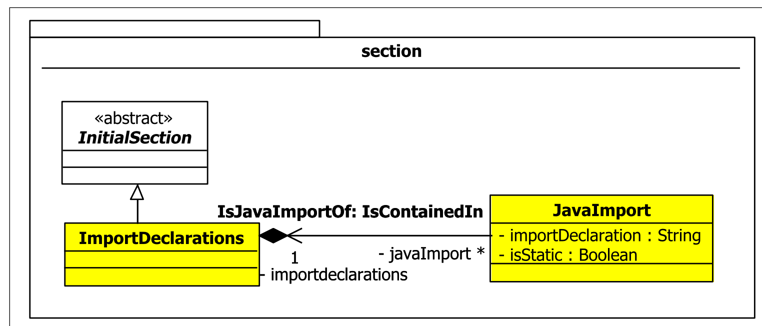


Abbildung A.9: Die `ImportDeclarations`-Sektion.

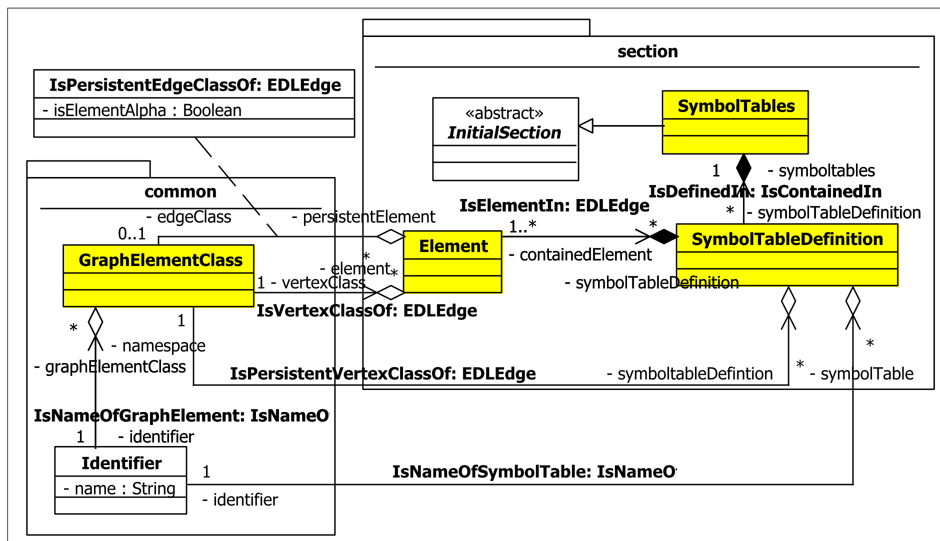


Abbildung A.10: Die `SymbolTables`-Sektion.

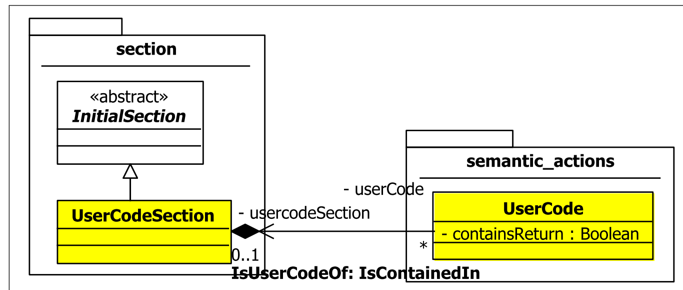


Abbildung A.11: Die `UserCode`-Sektion.

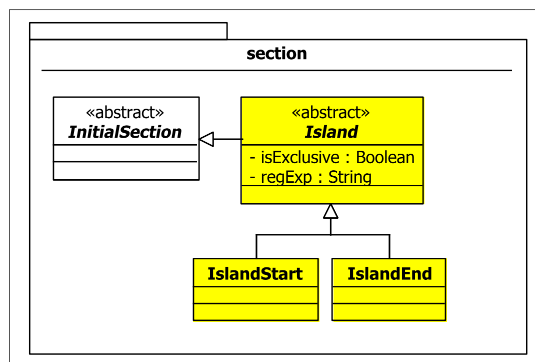


Abbildung A.12: Die `Island`-Sektion.

grammar

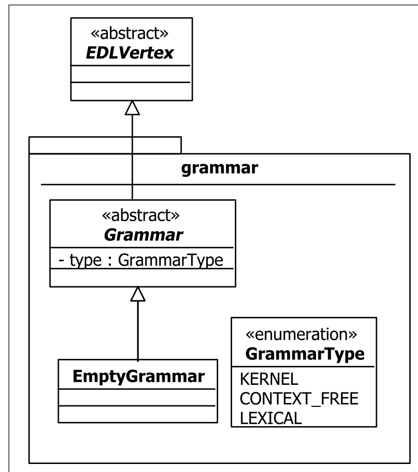


Abbildung A.13: Die leere Grammatik.

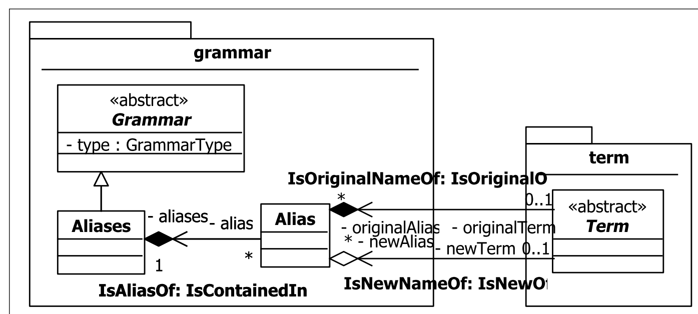


Abbildung A.14: Die Alias-Definition.

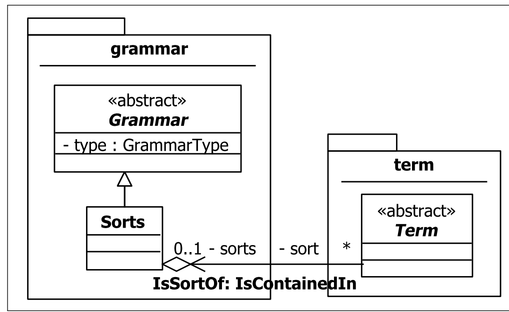


Abbildung A.15: Die Sorten-Deklaration.

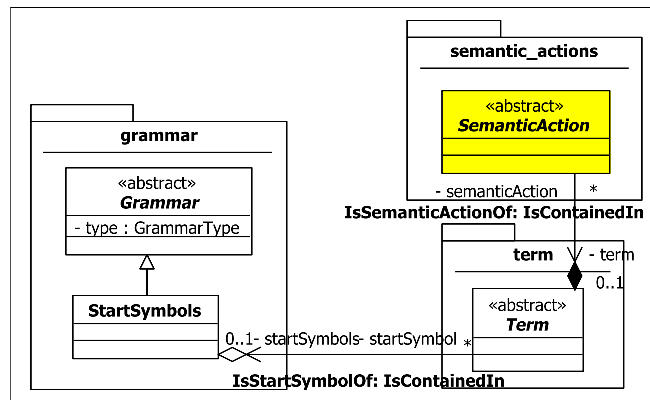


Abbildung A.16: Die Start-Symbol-Deklaration.

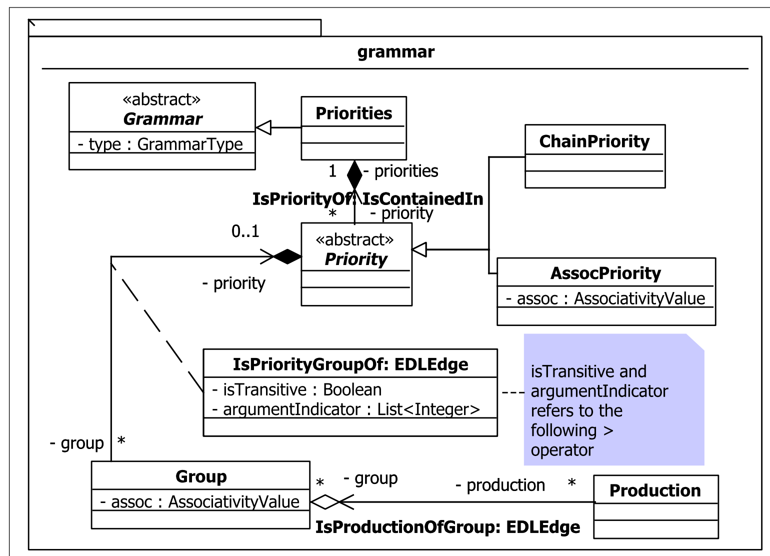


Abbildung A.17: Die Prioritäten.

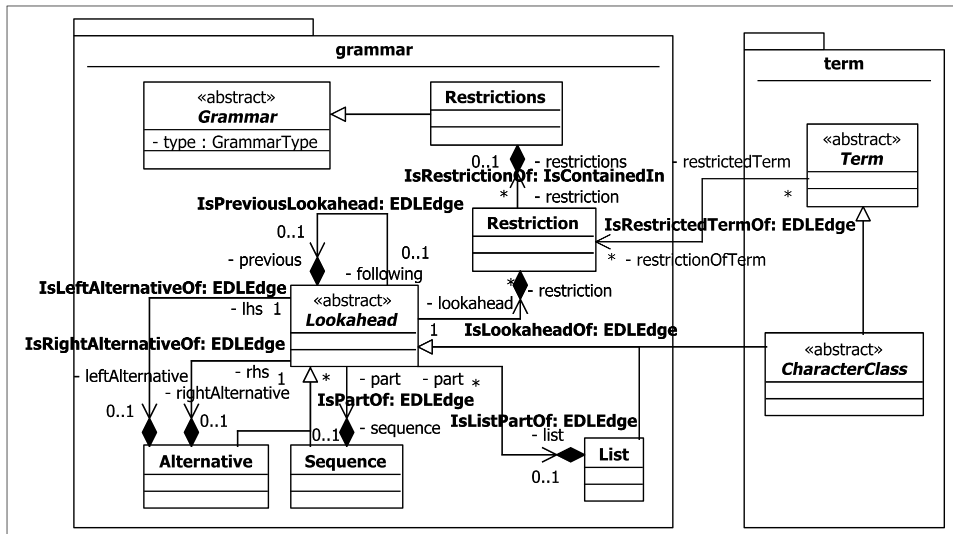


Abbildung A.18: Die Restriktionen.

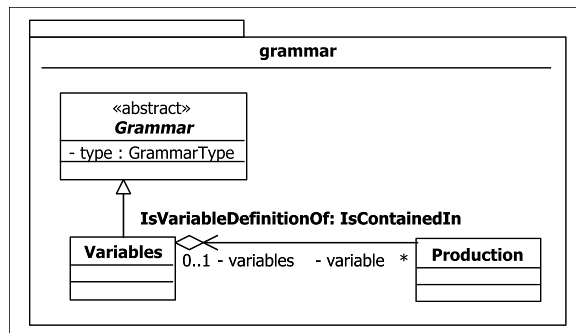


Abbildung A.19: Die Variablen-Deklaration.

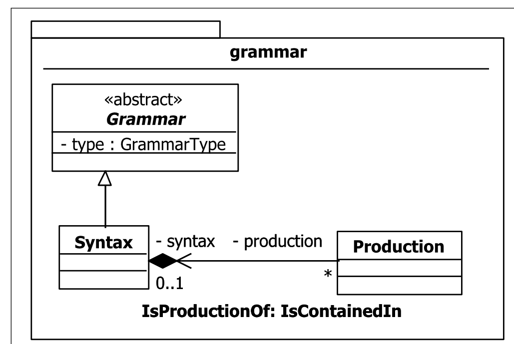


Abbildung A.20: Die Syntax-Deklaration.

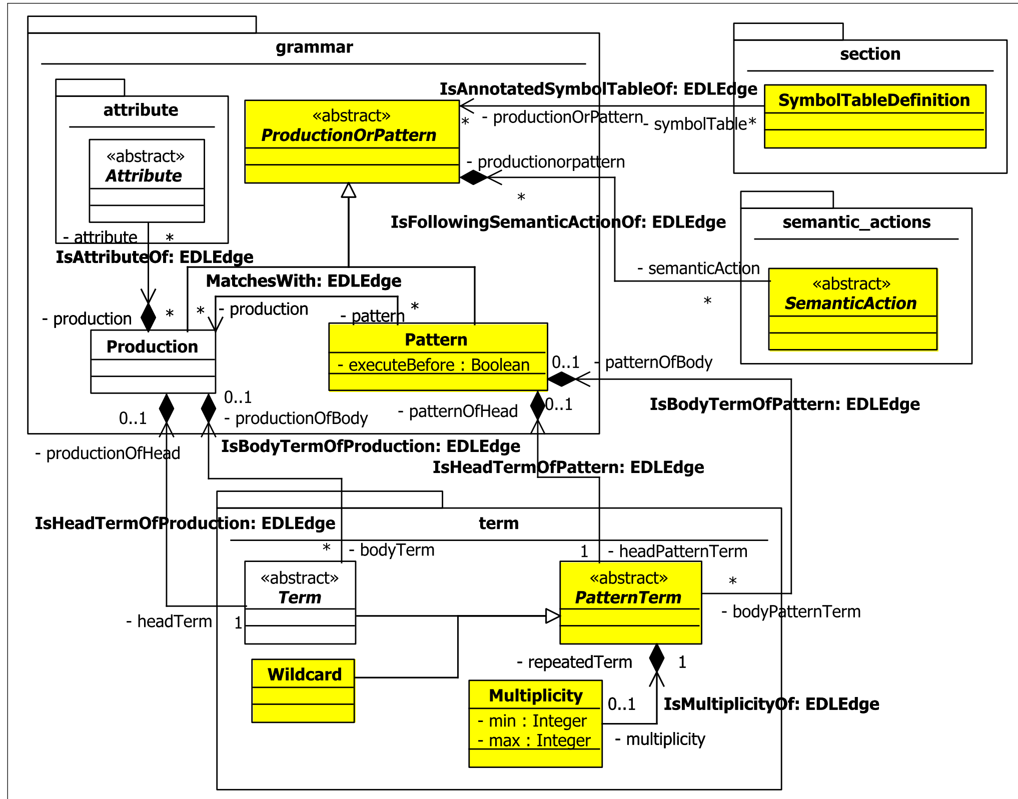


Abbildung A.21: Die Regel-Deklaration.

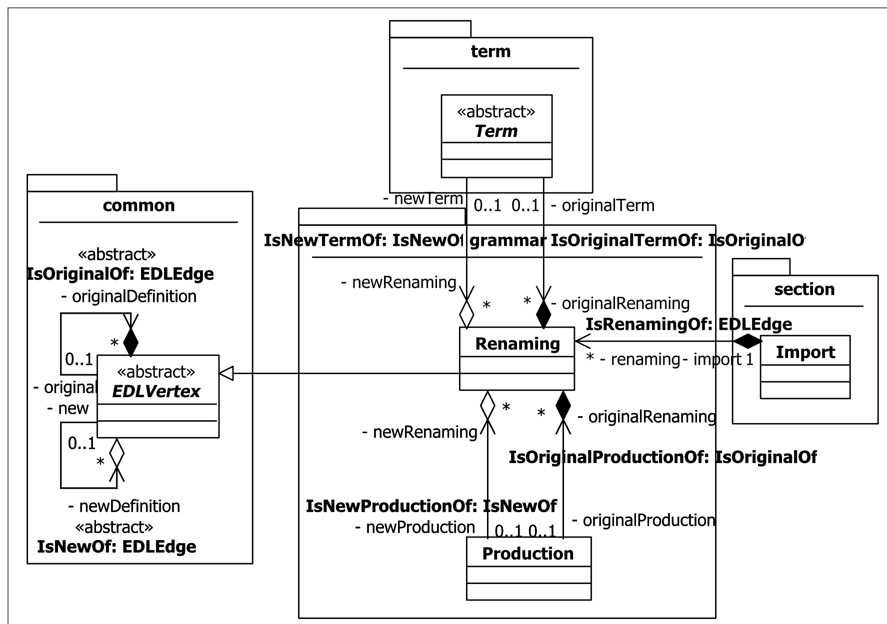


Abbildung A.22: Das Renaming.

grammar.attribute

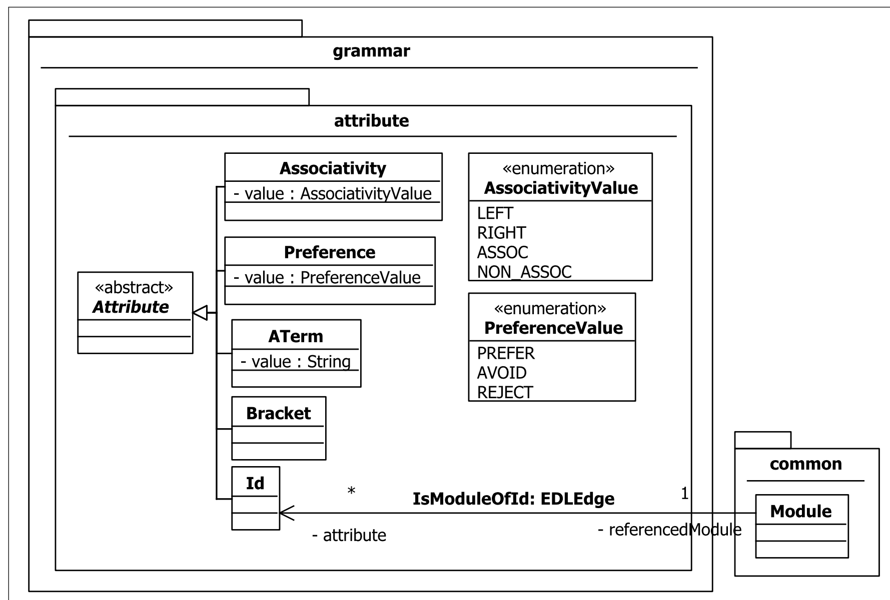


Abbildung A.23: Die Attribute einer Regel.

term

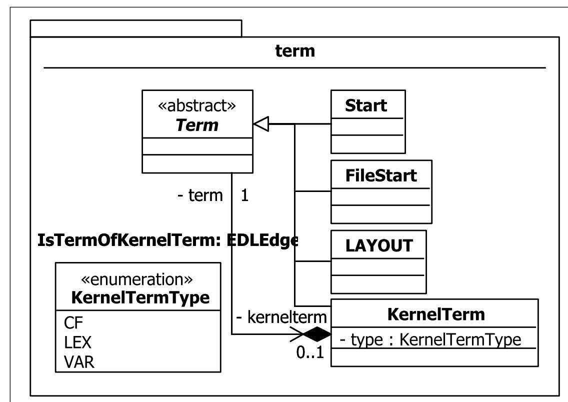


Abbildung A.24: Die Basis-Terme.

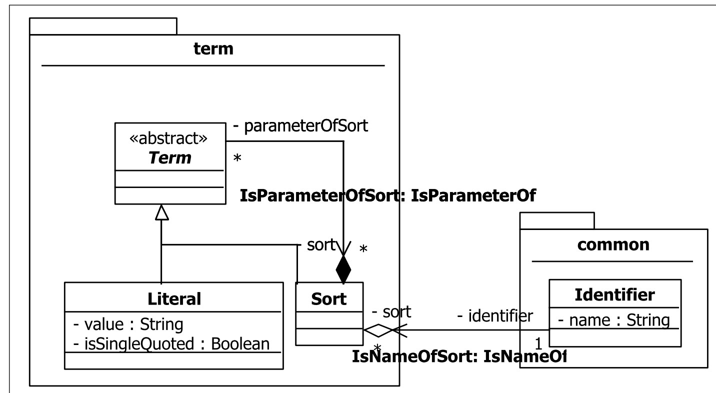


Abbildung A.25: Die Literale und Sorten.

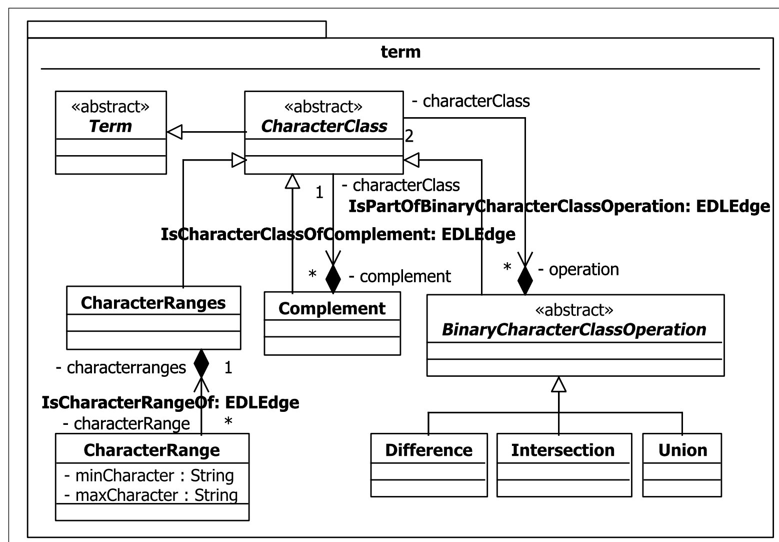


Abbildung A.26: Die `CharacterClass`.

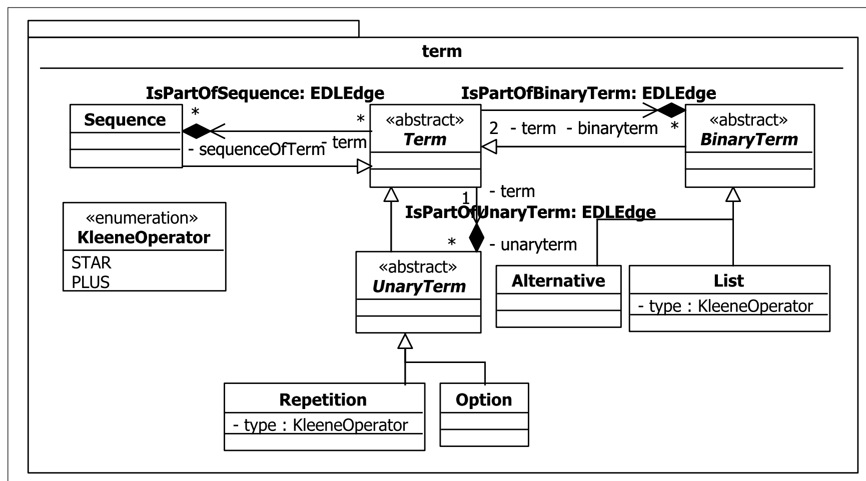


Abbildung A.27: Die regulären Terme.

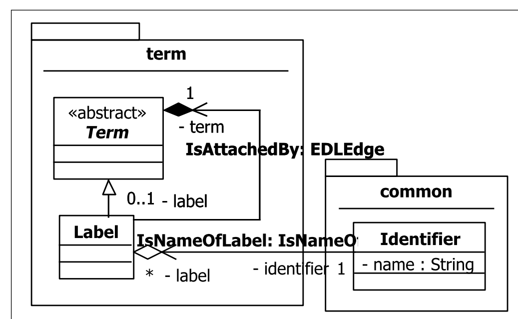


Abbildung A.28: Die Label.

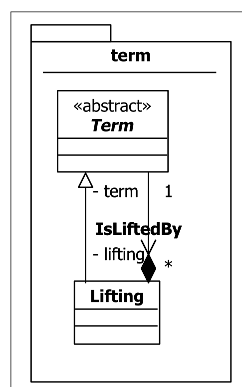


Abbildung A.29: Das Lifting.

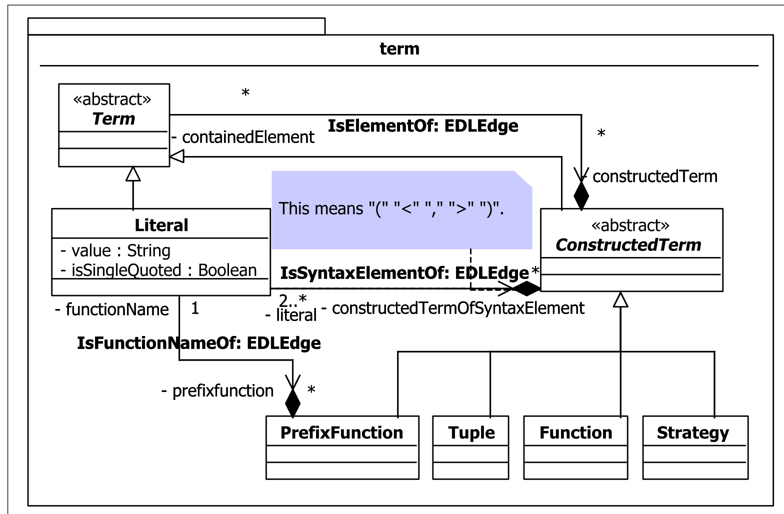


Abbildung A.30: Die restlichen Terme.

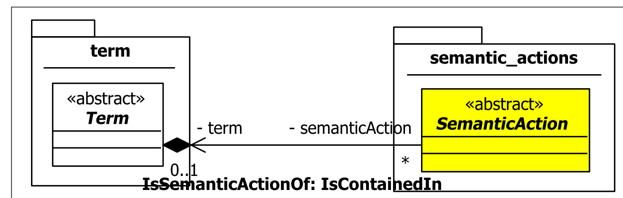


Abbildung A.31: Semantische Aktionen und Terme.

semantic_actions

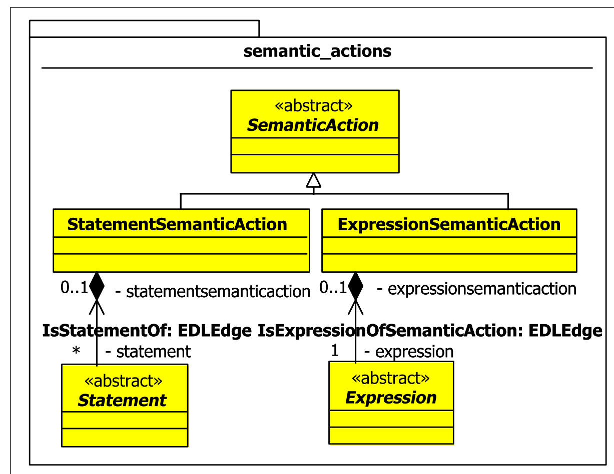


Abbildung A.32: Die semantischen Aktionen.

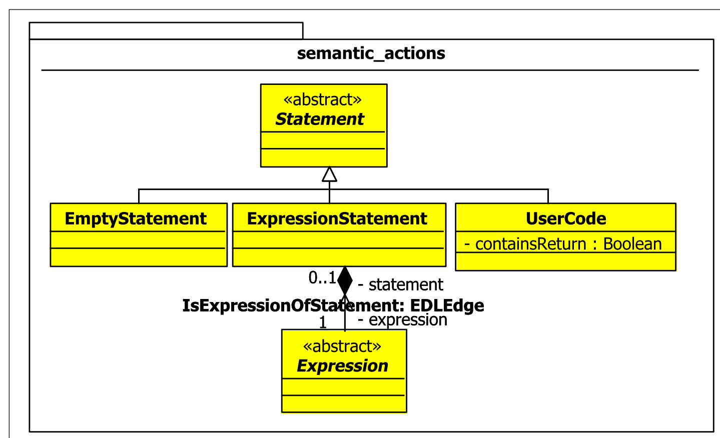


Abbildung A.33: Die Statements.

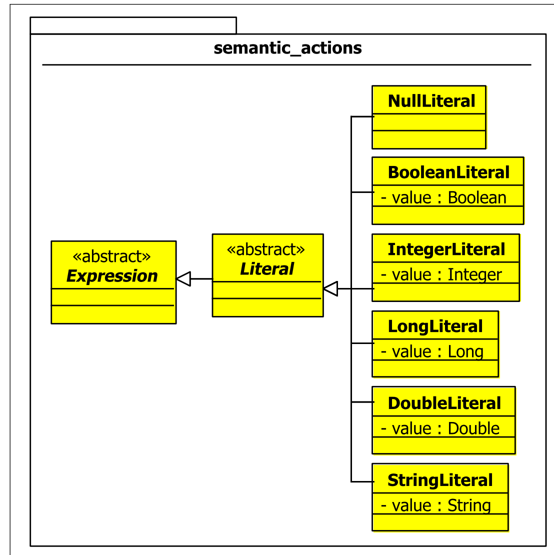


Abbildung A.34: Die Literale (Teil 1).

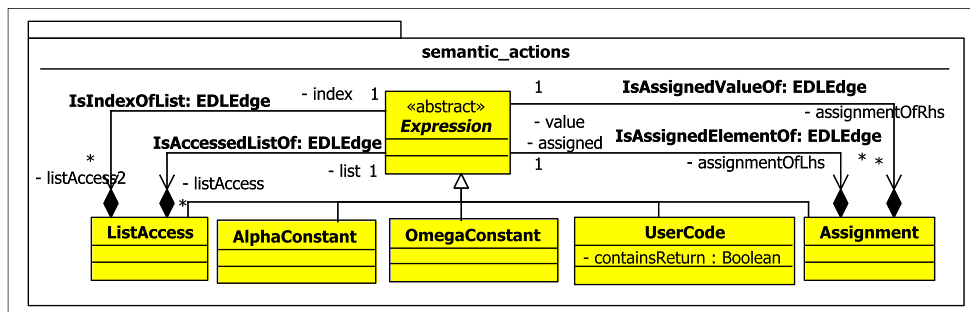


Abbildung A.35: Die Literale (Teil 2), die Zuweisung und der Zugriff auf Listenelemente.

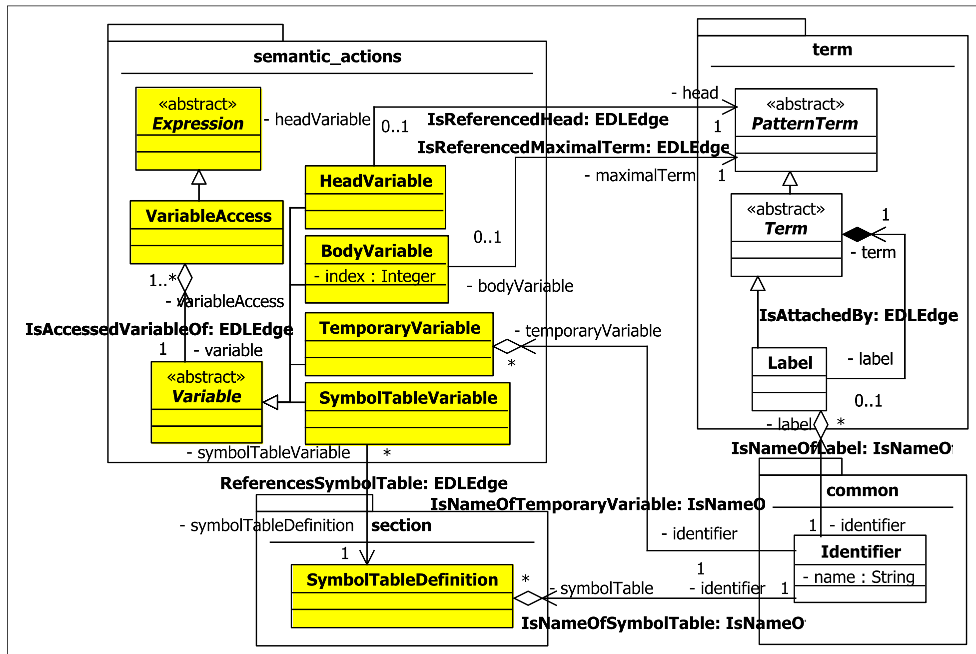


Abbildung A.36: Die Variablen.

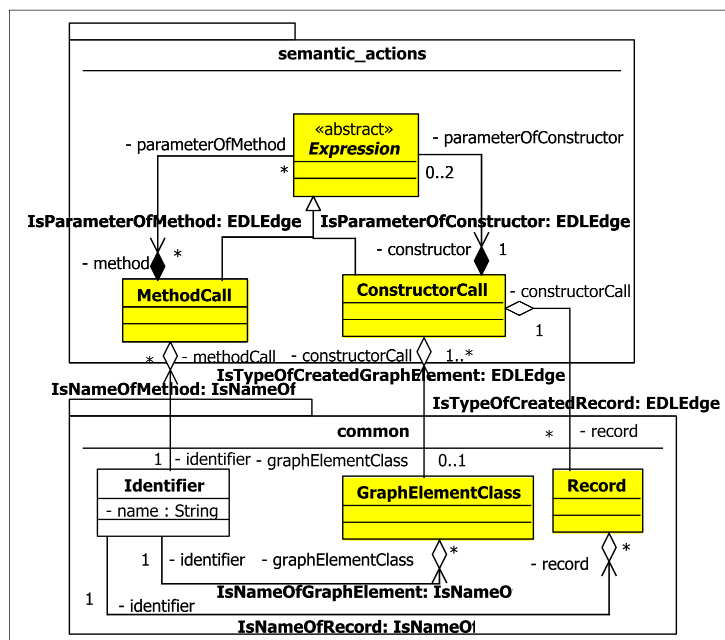


Abbildung A.37: Die Methoden- und Konstruktoraufrufe.

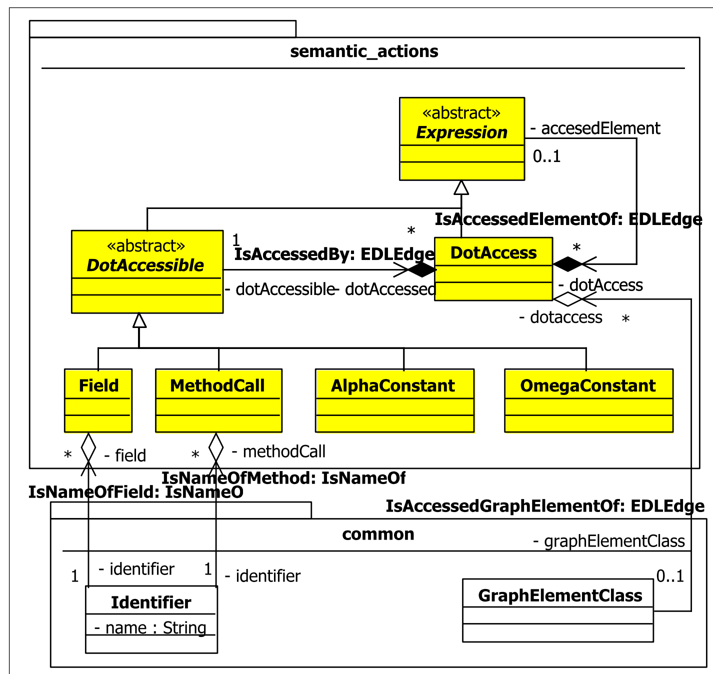


Abbildung A.38: Der Zugriff per "."-Operator.

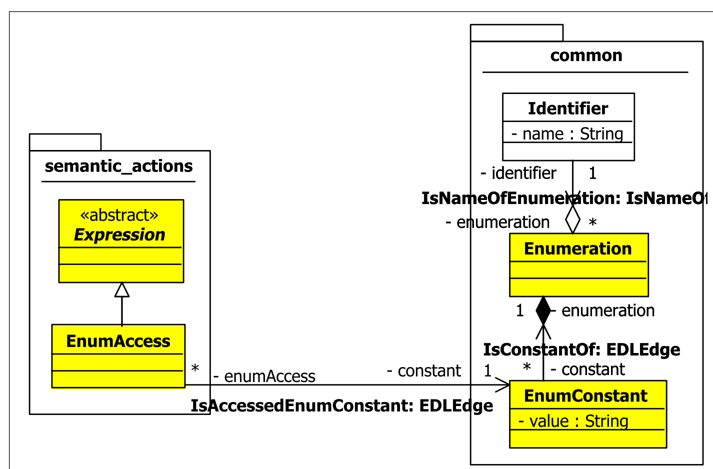


Abbildung A.39: Die Nutzung von Enumerationen.

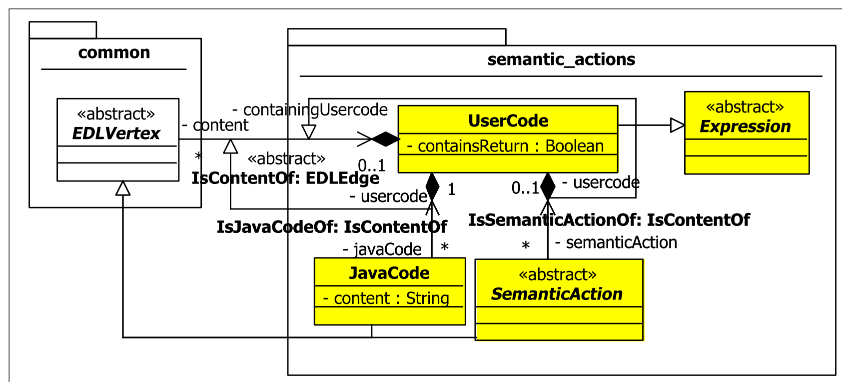


Abbildung A.40: Der nutzerspezifische Code.

Anhang B

Inhalt der CD

Auf der beiliegenden CD befinden sich die folgenden Ordner:

`source` enthält den kompletten Quellcode der „Extractor Description Language“ (EDL) als Eclipse-Export.

`lib` enthält die `edl.jar` sowie die zur Ausführung von EDL benötigten externen Bibliotheken und Programme.

`time_measurement` enthält das Java-Programm, mit dem die Zeitmessung in Kapitel 11 durchgeführt wurde sowie die dafür benötigten Extraktoren. Darüber hinaus befindet sich in diesem Verzeichnis die Auflistung der einzelnen gemessenen Zeiten.

`example` enthält die in Kapitel 10 beschriebenen Grammatiken für XML und Java.

`thesis` enthält diese Masterarbeit als pdf und die Tex-Dateien, aus denen sie generiert wurde.

Abbildungsverzeichnis

1.1 Software Reengineering nach Byrne. Quelle: [Byr92] 1

2.1 Eine TGraph-Instanz zur Repräsentation eines XML-Elements. 5
2.2 Das Schema des Graphen aus Abbildung 2.1. 6

3.1 Schematische Darstellung eines LL-Parsers. Quelle: [SLA08] 34
3.2 Schematische Darstellung des GLL-Recognizers 37
3.3 Schematische Darstellung eines LR-Parsers. Quelle: [SLA08] 55
3.4 Schematische Darstellung eines GLR Parsers 58

5.1 Das Parsen mithilfe von „Stratego/XT". 111
5.2 Der interne Parse-Forest und die verwendeten Regeln für die Eingabe 3. . 118
5.3 Das ITreeBuilder-Interface und seine Implementationen. 119
5.4 Die Verwendung des ITreeBuilders im Parse-Prozess. 120
5.5 Ausschnitt aus dem Parse-Forest für die Eingabe 3. 123
5.6 Der ParseTree für die Eingabe 3. 124
5.7 Der Parse-Forest für die Eingabe `3+4*3`. 125

6.1 Die Graphklasse und die Top-Level-Knoten und -Kanten. 128
6.2 Die Repräsentation einer Variablen-Deklaration. 129
6.3 Die Repräsentation einiger weiterer Statements. 130
6.4 Die Repräsentation einer if-Verzweigung. 131
6.5 Die Repräsentation von Labels und ihrer Verwendung. 131
6.6 Die Repräsentation von while- und do-while-Schleifen. 132

7.1 Der Datenfluss im vollständigen System. 161
7.2 Die Vorverarbeitung der EDL-Grammatik. 162
7.3 Die für die Vorverarbeitung zuständigen Komponenten. 163
7.4 Die Erzeugung eines Graphen aus der Eingabedatei. 164
7.5 Die für das Parsing der Eingabedatei zuständigen Komponenten. 164

8.1 Die `Parser`-Komponente. 167

8.2 Der erzeugte interne Parse-Forest für die Eingabe "50 €". 172
8.3 Die Repräsentation einer Regel. 192
8.4 Der Stack. 197
8.5 Der interne Parse-Forest zur Erkennung eines T_1-Elements. 205
8.6 Der interne Parse-Forest zur Erkennung von $T_1T_2T_1T_2T_1$. 205
8.7 Der Stack in Schritt 6. 206
8.8 Die Realisierung der TreeTraverser-Komponente. 207
8.9 Die Realisierung des Debuggings. 210
8.10 Die grafische Repräsentation des internen Parse-Forests aus Abbildung 8.2 im *Verbose-Modus*. 214
8.11 Die grafische Repräsentation des internen Parse-Forests aus Abbildung 8.2 im *Default-Modus*. 215
8.12 Der Symboltabellen-Stack. 215

9.1 Die EDL-Processor-Komponente. 227
9.2 Die Arbeitsweise des EDL2EDLGraphs. 229
9.3 Die Herleitung des Beispiels in Listing 9.5. 234
9.4 Der Inhalt des ***common***-Pakets (Ausschnitt). 235
9.5 Beispiel zur Verdeutlichung der Baumstruktur. 237
9.6 Die Repräsentation der Variablen. 237
9.7 Der EDLGraph der semantischen Aktion aus Listing 9.5. 239
9.8 Die Realisierung der EDL2EDLGraph-Komponente. 240
9.9 Die Realisierung der GraphBuilder-Generator-Komponente. 244
9.10 Der endliche Automat für das Beispiel-Pattern mit gepunkteten Pfeilen als ε-Transitionen. 247
9.11 Der PatternMatcher. 251

10.1 Das XML-Schema. 272
10.2 Der für den in Listing 10.8 gezeigten Ausschnitt erstellte Graph. 279
10.3 Der den Fehler verursachende Knoten des Parse-Trees. 280
10.4 Das Java-Schema. 281

11.1 Der Extraktionsvorgang und die ermittelten Größen. 291
11.2 Die gemessenen Zeiten in Abhängigkeit zur Eingabegröße. 294
11.3 Die gemessenen Zeiten in Abhängigkeit zur Eingabegröße. 296

A.1 Der Inhalt des common-Pakets (Teil 1). III
A.2 Der Inhalt des common-Pakets (Teil 2). IV
A.3 Die beiden Section-Kategorien. IV

A.4 Die `Imports`-Sektion. V
A.5 Die `Exports`- und `Hiddens`-Sektion. V
A.6 Die `Schema`-Sektion. V
A.7 Die `DefaultValues`-Sektion. V
A.8 Die `GlobalAction`-Sektion. VI
A.9 Die `ImportDeclarations`-Sektion. VI
A.10 Die `SymbolTables`-Sektion. VI
A.11 Die `UserCode`-Sektion. VII
A.12 Die `Island`-Sektion. VII
A.13 Die leere Grammatik. VIII
A.14 Die Alias-Definition. VIII
A.15 Die Sorten-Deklaration. IX
A.16 Die Start-Symbol-Deklaration. IX
A.17 Die Prioritäten. IX
A.18 Die Restriktionen. X
A.19 Die Variablen-Deklaration. X
A.20 Die Syntax-Deklaration. X
A.21 Die Regel-Deklaration. XI
A.22 Das Renaming. XI
A.23 Die Attribute einer Regel. XII
A.24 Die Basis-Terme. XII
A.25 Die Literale und Sorten. XIII
A.26 Die `CharacterClass`. XIII
A.27 Die regulären Terme. XIV
A.28 Die Label. XIV
A.29 Das Lifting. XIV
A.30 Die restlichen Terme. XV
A.31 Semantische Aktionen und Terme. XV
A.32 Die semantischen Aktionen. XVI
A.33 Die Statements. XVI
A.34 Die Literale (Teil 1). XVII
A.35 Die Literale (Teil 2), die Zuweisung und der Zugriff auf Listenelemente. . XVII
A.36 Die Variablen. XVIII
A.37 Die Methoden- und Konstruktoraufrufe. XVIII
A.38 Der Zugriff per `"."`-Operator. XIX
A.39 Die Nutzung von Enumerationen. XIX
A.40 Der nutzerspezifische Code. XX

Tabellenverzeichnis

3.1 Die vordefinierten Regelattribute in SDF. 27
3.2 Die aus der Grammatik G_{bindig} erzeugte LL-Parse-Tabelle. 33
3.3 Der LL-Parse-Vorgang für die Eingabe 101 mithilfe der Parse-Tabelle 3.2. . 34
3.4 Die aus der Grammatik G_{bindig} erzeugte LR-Parse-Tabelle. 56
3.5 Der LR-Parse-Vorgang für die Eingabe 101 mithilfe der Parse-Tabelle 3.4. . 56
3.6 Die Parse-Tabelle. Quelle: [Tom86] 62

5.1 Die GLR-Parsergeneratoren. 108
5.2 Die GLL-Parsergeneratoren. 109

6.1 Die Sichtbarkeitsebenen einer Beispielregel 144
6.2 Die in EDL definierten Funktionen zum Zugriff auf Positionsangaben und Lexeme. 154

11.1 Die gemessenen Zeiten unter Verwendung des aus der XML-Grammatik generierten `XMLGraphBuilders`. 293
11.2 Die gemessenen Zeiten unter Verwendung des aus der Java-Grammatik generierten `JavaGraphBuilders`. 295

Listings

3.1 Ein Wort einer einfache Beispielsprache 12
3.2 BNF-Definition der einfachen Beispielsprache 14
3.3 EBNF-Definition der einfachen Beispielsprache 15
3.4 SDF-Definition von Whitespace 19
3.5 SDF-Definition von Identifier 20
3.6 SDF-Definition von Number 20
3.7 SDF-Definition von Expression 21
3.8 SDF-Definition von Program 22
3.9 Ein Beispiel für die Verwendung von formalen Parametern 23
3.10 Ein Beispiel für die Verwendung von Argumentenlisten in Priorities (Quelle: [BKV07]) 26
3.11 Die für den rekursiven Abstieg der N-Regeln benötigte Methode. 35
3.12 Die für die Regeln R_0 bis R_n mit Kopf A erzeugte Methode. 36
3.13 Der für das Terminal a erzeugte Code. 36
3.14 Der für die syntaktische Variable Z erzeugte Code. 36
3.15 Der erzeugte Coderumpf. 40
3.16 Die Codeerzeugung einer Regel mit eindeutigem Rumpf. 42
3.17 Die Codeerzeugung einer Regel mit mehrdeutigem Rumpf. 42
3.18 Die Codeerzeugung für die Regel S der Beispielgrammatik. 43
3.19 Die Codeerzeugung einer leeren Alternative. 43
3.20 Die Codeerzeugung einer Alternative mit initialem Terminal. 44
3.21 Die Codeerzeugung einer Alternative mit initialem Nonterminal. 44
3.22 Die Codeerzeugung für die erste Alternative Sa der Beispielgrammatik. . 44
3.23 Die Codeerzeugung für die zweite Alternative a der Beispielgrammatik. . 45
3.24 Die Codeerzeugung eines folgenden Terminals. 45
3.25 Die Codeerzeugung eines folgenden Nonterminals. 45
3.26 Der erzeugte Code für das Terminal a der Beispielgrammatik. 46
3.27 Der Code des Recognizers der Beispielgrammatik. 46
3.28 Die *create*-Methode. Quelle: [SJ10] 52
3.29 Die *add*-Methode. Quelle: [SJ10] 53

3.30 Die *pop*-Methode. Quelle: [SJ10] . . . 53
3.31 Die *test*-Methode. Quelle: [SJ10] . . . 54
3.32 Die `parse`-Methode. Quelle: [Tom86] . . . 88
3.33 Die `parseLetter`-Methode. Quelle: [Tom86] . . . 89
3.34 Die `act`-Methode. Quelle: [Tom86] . . . 90
3.35 Die `reduce`-Methode im Standardfall. Quelle: [Tom86] . . . 92
3.36 Die `reduce`-Methode im Falle von Mehrdeutigkeit. Quelle: [Tom86] . . . 94
3.37 Die `reduce`-Methode im Falle eines wiederverwendeten Zustandes. Quelle: [Tom86] . . . 96
3.38 Die ε`reduce`-Methode. Quelle: [Tom86] . . . 97
3.39 Die `shift`-Methode. Quelle: [Tom86] . . . 98

5.1 Die SDF-Definition von Expression. . . . 112

6.1 Das EDL-Modul `Main`. . . . 132
6.2 Das EDL-Modul `Blocks`. . . . 134
6.3 Das EDL-Modul `FieldDeclarations`. . . . 135
6.4 Das EDL-Modul `LocalVariableDeclarations`. . . . 136
6.5 Das EDL-Modul `Statements` (Teil 1). . . . 137
6.6 Das EDL-Modul `Statements` (Teil 2). . . . 138
6.7 Das EDL-Modul `Statements` (Teil 3). . . . 139
6.8 Das EDL-Modul `Statements` (Teil 4). . . . 140
6.9 Der zu parsende Java-Block. . . . 140
6.10 Das EDL-Modul `Comments`. . . . 147
6.11 Das EDL-Modul `LocalVariableDeclarations` (erweiterter Auszug aus Listing 6.4). . . . 148
6.12 Java-Code mit Kommentar. . . . 149
6.13 Der **`default values`**-Block aus Listing 6.1. . . . 155
6.14 Die Deklaration von Symboltabellen. . . . 156
6.15 Assoziation von Symboltabellen mit einer Regel (Auszug aus Listing 6.2). 157
6.16 Die Definition einer Inselgrammatik zum Parsen des Javascript-Anteils einer HTML-Seite. . . . 160

8.1 Die Beispielgrammatik. . . . 169
8.2 Die aus der Beispielgrammatik generierten BNF-Regeln. . . . 170
8.3 Die Methode traverse(). . . . 209
8.4 Die String-Repräsentation des internen Parse-Forests aus Abbildung 8.2 im *Verbose-Modus*. . . . 212

8.5 Die String-Repräsentation des internen Parse-Forests aus Abbildung 8.2 im *Default-Modus*. 212
8.6 Das Beispielmodul `Number`. 219
8.7 Die aus dem Beispielmodul `Number` generierten BNF-Regeln. 220
8.8 Die generierte `execute()`-Methode. 221
8.9 Die generierte `execute_Rule269()`-Methode. 222
8.10 Die generierte `execute_Rule269_Position0()`-Methode. 223
8.11 Die generierte `execute_Rule269_Term0()`-Methode. 223
8.12 Die generierte `execute_Rule269_Term0_Rule267()`-Methode. 224
8.13 Die generierte `execute_Rule269_Term0_Rule267_Position1()`-Methode. 225

9.1 Das Modul `grammar/Module`. 230
9.2 Das Modul `grammar/sections/Main`. 231
9.3 Das Modul `grammar/sections/SchemaSection`. 231
9.4 Das Modul `grammar/terms/Term`. 232
9.5 Der zu erkennende nutzerspezifische Java-Code. 233
9.6 Das Modul `grammar/semantic-actions/UserCode`. 233
9.7 Die zur Verdeutlichung der Variablenrepräsentation benutzte semantische Aktion. 238
9.8 Das Paket-Präfix. 257
9.9 Die Klassendefinition. 257
9.10 Eine exemplarische Symboltabellen-Deklaration. 258
9.11 Der für die Symboltabellen-Deklaration in Listing 9.10 generierte Java-Code. 258
9.12 Die generierten Konstruktoren. 259
9.13 Die Umsetzung einer Island-Sektion. 260
9.14 Die generierte `main()`-Methode. 260
9.15 Der generierte Code zur Schema-Instanziierung. 260
9.16 Die generierten `setDefaultValues()`-Methoden. 261
9.17 Die generierten if-Anweisung für default-Werte. 261
9.18 Der generierte Code für `EDLEdge.length = length(alpha);`. 262
9.19 Der generierten Code für das Statement `$=4;`. 263
9.20 Die generierte Methode für nutzerspezifischen Java-Code. 269
9.21 Die generierte Methode für in nutzerspezifischem Java-Code enthaltene EDL-Ausdrücke. 269

10.1 Das Modul `xml/XMLMain`. 273
10.2 Das Modul `xml/Element` (Teil 1). 273

10.3 Das Modul `xml/Element` (Teil 2). 274
10.4 Das Modul `xml/Element` (Teil 3). 275
10.5 Das Modul `xml/Element` (Teil 4). 276
10.6 Das Modul `xml/Element` (Teil 5). 277
10.7 Das Modul `xml/Element` (Teil 6). 277
10.8 Die `build.xml` des edl-Projekts. 279
10.9 Das Modul `java/Main` (Teil 1). 282
10.10 Das Modul `java/Main` (Teil 2). 283
10.11 Die `CompilationUnit`-Regel. 284
10.12 Das Modul `java/names/Main`. 286
10.13 Das Modul `java/packages/ImportDeclarations`. 287
10.14 Die `InterfaceDec`-Regel. 288
10.15 Das Modul `java/statements/Statements`. 289
10.16 Der Aufruf des `JavaGraphBuilder`s. 290

Index

A

Ableitung 11
ACTION-Tabelle 55
Alias-Bezeichner 146
aliases 25
Alphabet 9
Alternative 30, 115, 145
Anforderungen 101
angeordnet 5
ANTLR 107
Asfix2TreeBuilder 120
Attribut 27
– assoc 27
– avoid 27
– bracket 27
– left 27
– non-assoc 27
– prefer 27
– reject 27
– right 27
Attributbezeichner 7
attributiert 6
Attributierung 7
Attributwert 7, 153

B

Backus-Naur-Form 13, 113
beschränkt 12
BNF . . . . . *siehe* Backus-Naur-Form
BNF-Transformation 113
Bottom-Up 54

C

CharacterClass 29, 145
Chomsky-Hierarchie 11
CodeGenerator 256
column() 154

D

Debugging 104, 210, 279
declare() 158
Default-Werte 202, 261
default values 155
Desugarer 245

E

EBNF . . . . . *siehe* Erweiterte Backus-Naur-Form
EDL . . . . . *siehe* Extractor Description Language
EDL-Schema 235
EDL2EDLGraph 228, 239
EDLGraph 228, 236
EDLParser 167
EDLPreprocessor 161, 228
Erweiterte Backus-Naur-Form 15
Exports-Sektion 24
Extractor Description Language 127

F

file() 154
FIRST 32
FOLLOW 33
FunctionTerm 30, 115, 145

G

gerichtet 5
getPrefixWhitespace() 149
getSuffixWhitespace() 149
getTemporaryVertices() 158
getWhitespaceBefore() 149
GLL 37, 109
global actions 151
GLR 108
GOTO-Tabelle 55
Grammatik 10
graph-structured Stack 58
GraphBuilder 161, 167, 218
GraphBuilderGenerator 244
grUML 6
GSS . . . . *siehe* graph-structured Stack

H

Hiddens-Sektion 24

I

Import-Sektion 23
Inselgrammatik 102, 160, 217, 259
interne Regelrepräsentation 116
interner Parse-Forest 117, 209
Inzidenzabbildung 7
ITreeBuilder 207

J

Java-Grammatik 282
JavaGraph 280
Javascanner-Schema 280
JGraLab 6

K

Kante 7
Knoten 7
Kommentar 24
Konkatenation 10
kontextfrei 12
kontextsensitiv 12

L

Label 30, 115
LALR 55
LAYOUT 31, 149
length() 154
lexem() 154
lift() 146
Lifting 30
line() 154
Linksableitung 11
List 30, 115, 145
Literal 28, 114, 145
LL(*) 31
LL(k) 31
LR(k) 54

M

Mapping
– Default 103, 149, 256
– Regel-spezifisch 103, 151
– Variablen-spezifisch 103
maximaler Term 114, 144
Mehrdeutigkeit 11
Metasprachen 12
Modul 22

Modularisierbarkeit . . . 102
Multiplizität . . . 150

O

offset() . . . 154
Option . . . 29, 114, 145

P

Packrat Parser . . . 107
Parse-Forest . . . 59
Parse-Tabelle . . . 33, 55, 113
Parsergeneratoren . . . 107
Pattern . . . 151, 246
PatternMatcher . . . 252
Pfad . . . 8
PrefixFunction . . . 30
priorities . . . 26

R

rechtslinear . . . 12
reduce . . . 56
Reduce-Reduce-Konflikt . . . 57
Regel . . . 10, 253
Regelkopf . . . 10
Regelrumpf . . . 10
Regeltransformation . . . 113
rekursiver Abstieg . . . 35
Repetition . . . 29, 115, 145
restrictions . . . 26
Rule . . . 192
RuleType . . . 193

S

Schema . . . 6, 103
Schematyp . . . 7
SDF . . *siehe* Syntax Definition Formalism
Sdf2Table . . . 113
SDFGenerator . . . 243
semantische Aktion . . . 102, 143
– global . . . 150, 246
– nutzerspezifisch . . . 104, 159, 260, 269
– Pattern-spezifisch . . . 151, 246
– Regel-spezifisch . . . 151
– Schema-spezifisch . . . 151
Sequence . . . 29, 114, 145, 253
shift . . . 56
Shift-Reduce-Konflikt . . . 57
Sichtbarkeitsebene . . . 144, 199
Software-Reengineering . . . 1
Sort . . . *siehe* Type
sorts . . . 25
Sprache . . . 10
Sprachklasse . . . 11
Stack . . . 197
StackElement . . . 197
start-symbols . . . 25
Startsymbol . . . 10, 114, 253
Stratego/XT . . . 108, 110
Symboltabelle . . . 103, 155, 215, 256, 257
– persistent . . . 156, 216
Symboltabellen Stack . . . 155, 215
symbol tables . . . 156
syntaktische Variable . . . 10
syntax . . . 25
Syntax Definition Formalism . . . 19, 110

T

temporäre Variable . . . 146, 200
Term . . . 28
Terminal . . . 10
TGraph . . . 5, 7, 103
Top-Down . . . 31
TreeTraverser . . . 192, 207
Tuple . . . 30, 115, 145

Typbezeichner 7
Type 28, 145
typisiert 6

U

Unicode 112
use() 158
useOrDeclare() 158

W

Wert eines Terms 145
Wort 9

X

XML-Grammatik 272
XML-Schema 272
XMLGraph 272